On the Practical Ability of Recurrent Neural Networks to Recognize Hierarchical Languages

Satwik Bhattamishra[♠] **Kabir Ahuja**[♢*] **Navin Goyal**[♠]
[♠] Microsoft Research India
[♢] Udaan.com
{t-satbh,navingo}@microsoft.com
kabir.ahuja@udaan.com

Abstract

While recurrent models have been effective in NLP tasks, their performance on context-free languages (CFLs) has been found to be quite weak. Given that CFLs are believed to capture important phenomena such as hierarchical structure in natural languages, this discrepancy in performance calls for an explanation. We study the performance of recurrent models on Dyck-n languages, a particularly important and well-studied class of CFLs. We find that while recurrent models generalize nearly perfectly if the lengths of the training and test strings are from the same range, they perform poorly if the test strings are longer. At the same time, we observe that recurrent models are expressive enough to recognize Dyck words of arbitrary lengths in finite precision if their depths are bounded. Hence, we evaluate our models on samples generated from Dyck languages with bounded depth and find that they are indeed able to generalize to much higher lengths. Since natural language datasets have nested dependencies of bounded depth, this may help explain why they perform well in modeling hierarchical dependencies in natural language data despite prior works indicating poor generalization performance on Dyck languages. We perform probing studies to support our results and provide comparisons with Transformers.

1 Introduction

Recurrent models (RNNs and more specifically LSTMs) have been used extensively across several NLP tasks such as machine translation (Sutskever et al., 2014), language modeling (Melis et al., 2017) and question answering (Seo et al., 2016). Natural languages involve phenomena such as hierarchical and long-distance dependencies. Although RNNs are known to be Turing-complete (Siegelmann and Sontag, 1992) given unbounded precision, their practical ability to model such phenomena remains unclear.

Recently, several works (Weiss et al., 2018; Sennhauser and Berwick, 2018; Skachkova et al., 2018) have attempted to understand the capabilities of LSTMs by empirically analyzing them on different types of formal languages. Natural languages, for the most part, can be modeled by context-free languages (Jger and Rogers, 2012) and their hierarchical structure has been emphasized by Chomsky (2002). Thus studying the capabilities of RNNs in recognizing context-free languages (CFLs) can shed light on how well they can model hierarchical structures. An important family of context-free languages is the Dyck-n language[1].

Previous works (Suzgun et al., 2019a; Suzgun et al., 2019c; Yu et al., 2019) showed that LSTMs achieve limited generalization performance on recognizing Dyck-2. This prompted the development of memory-augmented variants (Joulin and Mikolov, 2015; Suzgun et al., 2019c) of LSTMs which generalize well on Dyck languages but are notoriously hard to train and fail to perform well on practical NLP tasks (Shen et al., 2019). On the other hand, despite the limited performance of LSTMs on Dyck languages, several studies (Gulordava et al., 2018; Tran et al., 2018) have found that LSTMs perform well in modeling hierarchical structure in natural language data. In this work, we take a step towards bridging this gap.

[*]This research was conducted during the author's internship at Microsoft Research.

[1]Informally, the Chomsky–Schützenberger representation theorem (1959) asserts that Dyck-n languages for $n \geq 1$ capture the complexity of context-free languages in a precise sense.

Proceedings of the 28th International Conference on Computational Linguistics, pages 1481–1494
Barcelona, Spain (Online), December 8-13, 2020

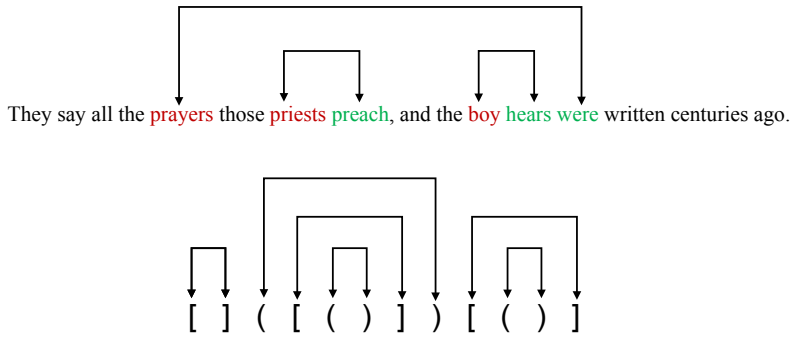


Figure 1: Nested dependencies in English sentences and in Dyck-2.

Our Contributions. We investigate the ability of recurrent models to learn and generalize on Dyck languages. We first evaluate the ability of LSTMs to recognize randomly sampled Dyck-n sequences and find, in contrast to prior works (Suzgun et al., 2019a; Suzgun et al., 2019c), that they can generalize nearly perfectly when the test samples are within the same lengths as seen during training. Similar to prior works, when evaluated on randomly generated samples of higher lengths we observe limited performance. Dyck languages and (deterministic) CFLs can be recognized by (deterministic) pushdown automata (PDA). We construct an RNN that directly simulates a PDA given unbounded precision. A key observation is that the higher the depth of the stack the higher is the required precision. This implies that fixed precision RNNs are expressive enough to recognize strings of arbitrary lengths if the required depth of the stack is bounded. Based on this observation, we test the hypothesis whether LSTMs can generalize to higher lengths if the depth of the inputs in the training and test set is bounded by the same value. In the bounded depth setting, LSTMs are able to generalize to much higher lengths compared to the lengths used during training. Given that natural languages in practical settings also contain nested dependencies of bounded depths (Gibson, 1991; McElree, 2001), this may help explain why LSTMs perform well in modeling natural language corpora containing nested dependencies. We then assess the generalization performance of the model across higher depths and find that although LSTMs can generalize to a certain extent, their performance degrades gradually with increasing depths. Since Transformer (Vaswani et al., 2017) is also a dominant model in NLP (Devlin et al., 2019), we include it in our experiments. To our knowledge, prior works have not empirically analyzed Transformers on formal languages, particularly context-free languages. We further conduct robustness experiments and probing studies to support our results.

2 Preliminaries and Motivation

The language Dyck-n parameterized by $n \geq 1$ consists of well-formed words from n types of parentheses. Its derivation rules are: $S \rightarrow (_i S)_i,\ S \rightarrow SS,\ S \rightarrow \epsilon$ where $i \in \{1, \ldots, n\}$. Dyck-n is context-free for every n, Dyck-2 being crucial among them, since all Dyck-n for $n > 2$ can be reduced to Dyck-2 (Suzgun et al., 2019a). For words in Dyck-n, the required depth of the stack in its underlying PDA is the maximum number of unbalanced parentheses in a prefix. For instance, the word ([( )]) [ ] in Dyck-2 has maximum depth of 3 corrsponding to the prefix ([(.

RNNs. RNNs are defined by the update rule $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$, where the function f could be a feedforward network, $\boldsymbol{h}_t$ is the model's memory vector usually referred to as the hidden state vector and $\boldsymbol{x}_t$ denotes the input vector at the t-th step. In practice, f is usually a single layer feedforward network with tanh or ReLU activation. To mitigate the vanishing gradients problem, LSTMs (Hochreiter and Schmidhuber, 1997), a variant of RNNs with additional gates, is most commonly used in practice. In our experiments, we will primarily work with LSTMs.

2.1 Expressiveness Results

Proposition 2.1. *Any Deterministic Pushdown Automaton can be simulated by an RNN with ReLU activation.*

We provide a proof by construction for the above result in Appendix B by using the Cantor-set like

Language	Model	Vanilla (Randomly Sampled)		Bounded Depth		
		Bin-1A Accuracy [2, 50]↑	Bin-2A Accuracy [52, 100]↑	Bin-1B Accuracy [2, 50]↑	Bin-2B Accuracy [52, 100]↑	Bin-3B Accuracy [102, 150]↑
Dyck-2	**LSTM**	**99.5**	**75.1**	**99.9**	**99.6**	**96.0**
	Transformer	95.1	21.8	**99.9**	92.1	36.3
Dyck-3	**LSTM**	**97.3**	**54.0**	**99.7**	**96.3**	**89.5**
	Transformer	87.7	26.2	90.1	48.9	6.4
Dyck-4	**LSTM**	**97.8**	**50.7**	**99.9**	**95.1**	**87.0**
	Transformer	92.7	36.6	94.4	49.3	5.6

Table 1: The performance of neural models on considered Dyck languages. The reported scores are obtained by averaging the accuracies of 5 best performing hyperparameter configurations of each model. All models are trained on inputs with length in [2,50] and evaluated on validation sets. Bin-2A contains higher lengths without any restriction on depth. In Bin-1B, Bin-2B and Bin-3B, the test inputs have their depths upper bounded by 10.

encoding scheme as introduced in Siegelmann and Sontag (1992) to emulate stack operations. The above result was first obtained by Korsky and Berwick (2019) somewhat indirectly using the Chomsky and Schützenberger (1959) theorem. The above Proposition implies that RNNs can recognize any deterministic CFL given unbounded precision. In the construction, the higher the required depth of the stack the higher the required precision. This implies that fixed precision RNNs are expressive enough to recognize strings of arbitrary lengths if the required depth of the stack is bounded. This can also be gleaned from the construction of Korsky and Berwick (2019) with some additional work[2].

3 Experiments

Setup. In our experiments, we consider three Languages, namely Dyck-2, Dyck-3 and Dyck-4. For each of the three languages, we generate 3 different types of training and validation sets. In the first case, we generate 10k strings for the training set within lengths [2, 50] and generate two validation sets each containing 1k strings within lengths [2, 50] and [52, 100] respectively without any restriction on the depth of the Dyck words. Our second setting also resembles the previous dataset in terms of lengths of the strings in the training and validation sets. However, in this case, we restrict all the strings to have depths in the range [1, 10]. This is to test the generalization ability across lengths when the depth is bounded. In the third case, we test the generalization ability across depths, when the lengths in train and validation sets are in the same interval [2,100]. Along with LSTMs, we also report the performance of Transformers (as used in GPT (Radford et al., 2018)) on each task since it is also a dominant model in NLP

Tasks. We train and evaluate our models on the Next Character Prediction Task (NCP) introduced in Gers and Schmidhuber (2001) and used in Suzgun et al. (2019a) and Suzgun et al. (2019c). Similar to an LM setup, the model is only presented with positive samples from a given language. In NCP, for a sequence of symbols $s_1, s_2, \ldots, s_n$, the model is presented with the sequence $s_1, s_2, \ldots, s_i$ at the i^{th} step and the goal of the model is to predict the set of valid characters for the $(i+1)^{th}$ step, which can be represented as a k-hot vector. The model assigns a probability to each character in the vocabulary corresponding to its validity in the next step, which is achieved by applying sigmoid activation over the unnormalized scores predicted through its output layer. Following Suzgun et al. (2019b) and Suzgun et al. (2019a), we use mean squared error between the predicted probabilities and k-hot labels as the loss function. During inference, we use a threshold of 0.5 to obtain the final prediction. The models prediction is considered to be correct if and only if its output at every step is correct. The accuracy of the model over test samples is the fraction of total samples which are predicted correctly. Note that, this is a relatively stringent metric as a correct prediction is obtained only when the model's output is correct at every step as opposed to standard classification tasks. Refer to Bhattamishra et al. (2020) for a discussion on the choice of character prediction task and its relation with other tasks such as standard classification and language modeling. Details of the dataset and parameters relevant for reproducibility can be found in section C in Appendix.

[2]Since our construction, although novel, is not critical for the inferences, it has been moved to the appendix.

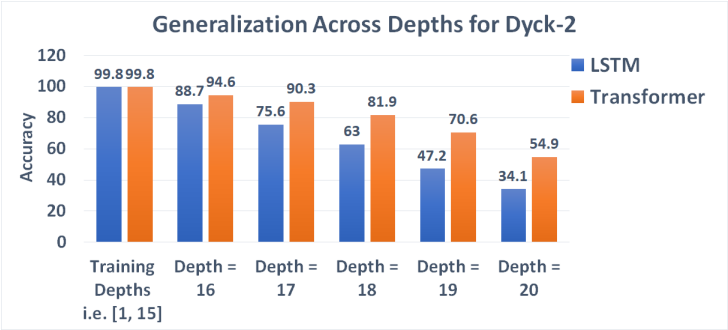


Figure 2: Generalization of LSTMs and Transformers on higher depths. The lengths of strings in the training set and all validation sets were fixed to lie between 2 to 100.

We have made our source code available at `https://github.com/satwik77/RNNs-Context-Free`.

3.1 Results

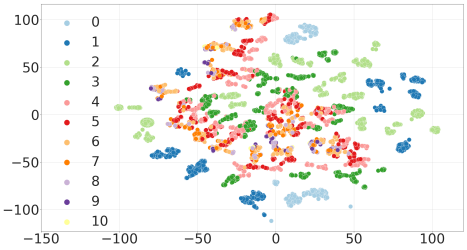
Table 1 shows the performance of LSTMs and Transformers on the considered languages. When inputs are randomly sampled in a given range of lengths, LSTMs can generalize well on the validation set containing inputs of the same lengths as seen during training (Bin-1A)[3], for all considered Dycks. However, for higher lengths (Bin-2A), it struggles to generalize on these languages. For the case when the depth is bounded, LSTMs generalize very well to much higher lengths (Bin 1B, 2B, and 3B). Transformers, on the other hand, fail to generalize to higher lengths in either case, which might be attributed to the fact that at test time, it receives positional encodings that it was not trained on. To investigate the generalization ability of models across increasing depths, we trained the models up to depth 15 and evaluated on 5 validation sets with an incremental increase in depth in each set (refer to Figure 2). We found that the models were able to generalize up to a certain extent but their performance degraded gradually as we increase the depth. However, Transformers performed relatively better as compared to LSTMs.

Probing. We also conducted probing experiments on LSTMs to better understand how they perform these particular tasks. We first attempt to extract the depth of the underlying stack from the cell state of an LSTM model trained to recognize Dyck-2. We found that a single layer feedforward network was easily able to extract the depth with perfect (100%) accuracy and generalize to unseen data. Figure 3a shows a visualization of t-SNE projection of the hidden state labeled by their corresponding depths. Additionally, we also try to extract the stack elements from the hidden states of the network. For Dyck-2 samples within lengths [2,50] and depths [1,10], along with training LSTM for the NCP task, we co-train auxiliary classifiers to predict each element of the stack up to depth 10, i.e. the hidden state of the LSTM that is used to predict the next set of valid characters is now also utilized in parallel to predict (by supplying 10 separate linear layers for each element) the elements of the stack. We find that not only was the model able to predict the elements in a validation set from the same distribution, it was also able to extract the elements for sequences of higher lengths ([52,150]) on which it was not trained on (see Figure 3b). This further provides evidence to show that LSTMs are able to robustly generalize to inputs of higher lengths when their depths are bounded. We also conducted a few additional robustness experiments to ensure that the model does not overfit on training distribution. Details of probing tasks as well as additional robustness experiments can be found in the Appendix.

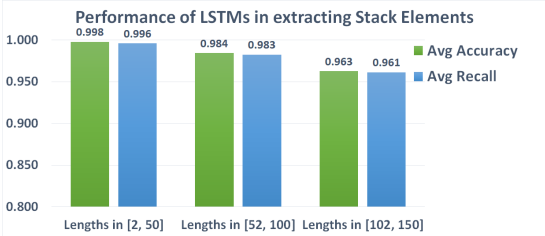
4 Discussion

LSTMs and Transformers have been effective on language modeling tasks. In practice, during language modeling on a natural language corpus, the entire input is fed sequentially to the LSTM. Hence, the length of the input processed by the LSTM is bound to be large, requiring it to model a number of

[3]This result is in disagreement with the results reported in Suzgun et al. (2019a) and Suzgun et al. (2019c). The possible discrepancy could be due to more extensive hyperparameter tuning in our experiments. We found this holds even while tuning within the same constraints as mentioned in their setup.



(a) Visualization of t-SNE Projections of the hidden states obtained from a pre-trained LSTM for different Dyck-2 substrings, colored by their depths.



(b) Accuracies and recalls obtained on extracting top-10 elements (averaged over them) of the stack corresponding to different Dyck-2 strings using the hidden state vector of the LSTM.

Figure 3

nested dependencies. Prior works in psycholinguistics (Gibson, 1991; McElree, 2001) have pointed out that given limited working memory of humans, natural language as used in practice should have nested dependencies of bounded depth. Some works (Jin et al., 2018; Noji et al., 2016) have even sought to build parsers for natural language with depth-bounded PCFGs. Our generalization results for LSTMs on depth-bounded CFGs could help explain why they perform well on modeling hierarchical dependencies in natural language datasets. For Transformer, although it did not generalize to higher lengths, but in practice Transformers (as used in BERT and GPT) process inputs in a fixed-length context window. Our results indicate that it does not have trouble in generalizing when the train and validation sets contain inputs of the same lengths.

Our experiments also demonstrate that the limiting factor in LSTMs is in generalizing to higher depths as opposed to its memory-augmented variants. The exact mechanism with which trained LSTMs perform the task is not entirely clear. The limited performance of LSTMs could be due to precision issues or unstable stack encoding mechanism (Stogin et al., 2020). However, given that natural language datasets are likely to have nested dependencies of bounded depth, this limitation may not play a significant role and may help explain why prior works (Gulordava et al., 2018; Tran et al., 2018) have found LSTMs to perform well in modeling hierarchical structure on natural language datasets.

Acknowledgements

We thank the anonymous reviewers for their constructive comments and suggestions. We would also like to thank our colleagues at Microsoft Research and Michael Hahn for their valuable feedback and helpful discussions.

References

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the ability and limitations of transformers to recognize formal languages.

Noam Chomsky and Marcel P Schützenberger. 1959. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 118–161. Elsevier.

Noam Chomsky. 2002. *Syntactic structures*. Walter de Gruyter.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June. Association for Computational Linguistics.

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

Edward Albert Fletcher Gibson. 1991. *A computational theory of human linguistic processing: Memory limitations and processing breakdown*. Ph.D. thesis, Carnegie Mellon University Pittsburgh, PA.

Kristina Gulordava, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. 2018. Colorless green recurrent networks dream hierarchically. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1195–1205, New Orleans, Louisiana, June. Association for Computational Linguistics.

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

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

Lifeng Jin, Finale Doshi-Velez, Timothy Miller, William Schuler, and Lane Schwartz. 2018. Unsupervised grammar induction with depth-bounded pcfg. *Transactions of the Association for Computational Linguistics*, 6:211–224.

Armand Joulin and Tomas Mikolov. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198.

Gerhard Jger and James Rogers. 2012. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Samuel A Korsky and Robert C Berwick. 2019. On the computational power of rnns. *arXiv preprint arXiv:1906.06349*.

Brian McElree. 2001. Working memory and focal attention. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 27(3):817.

Gábor Melis, Chris Dyer, and Phil Blunsom. 2017. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*.

Hiroshi Noji, Yusuke Miyao, and Mark Johnson. 2016. Using left-corner parsing to encode universal structural constraints in grammar induction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 33–43.

Jorge Prez, Javier Marinkovi, and Pablo Barcel. 2019. On the Turing completeness of modern neural network architectures. In *International Conference on Learning Representations*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Luzi Sennhauser and Robert Berwick. 2018. Evaluating the ability of LSTMs to learn context-free grammars. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 115–124, Brussels, Belgium, November. Association for Computational Linguistics.

Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. 2016. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*.

Yikang Shen, Shawn Tan, Arian Hosseini, Zhouhan Lin, Alessandro Sordoni, and Aaron C Courville. 2019. Ordered memory. In H. Wallach, H. Larochelle, A. Beygelzimer, F. AlcheBuc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 5037–5048. Curran Associates, Inc.

Hava T Siegelmann and Eduardo D Sontag. 1992. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449. ACM.

Natalia Skachkova, Thomas Trost, and Dietrich Klakow. 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, November. Association for Computational Linguistics.

John Stogin, Ankur Mali, and C Lee Giles. 2020. Provably stable interpretable encodings of context free grammars in rnns with a differentiable stack.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Mirac Suzgun, Yonatan Belinkov, Stuart Shieber, and Sebastian Gehrmann. 2019a. LSTM networks can perform dynamic counting. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 44–54, Florence, August. Association for Computational Linguistics.

Mirac Suzgun, Yonatan Belinkov, and Stuart M. Shieber. 2019b. On evaluating the generalization of LSTM models in formal languages. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 277–286.

Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M Shieber. 2019c. Memory-augmented recurrent neural networks can learn generalized dyck languages. *arXiv preprint arXiv:1911.03329*.

Ke Tran, Arianna Bisazza, and Christof Monz. 2018. The importance of being recurrent for modeling hierarchical structure. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4731–4736, Brussels, Belgium, October-November. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia, July. Association for Computational Linguistics.

Xiang Yu, Ngoc Thang Vu, and Jonas Kuhn. 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, August. Association for Computational Linguistics.

A Preliminaries

Definition A.1 (Deterministic Pushdown Automata (Hopcroft et al., 2006))**.** A DPDA is a 7-tuple $\langle \Sigma, Q, \Gamma, q_0, Z_0, \delta, F \rangle$ with

1. A finite alphabet Σ
2. A finite set of states Q
3. A finite stack alphabet
4. An initial state q_0
5. An initial stack symbol Z_0
6. $F \subseteq Q$ set of accepting states
7. A state transition function
$$\delta : \Sigma \times Q \times \Gamma \to (q, \gamma)$$
The output of δ is a finite set of pairs (q, γ), where q is a new state and γ is the string of stack symbols that replaces the top of the stack. If X is at the top of the stack, then $\gamma = \epsilon$ indicates that the stack is popped, $\gamma = X$ denotes that the stack is unchanged and if $\gamma = YZ$, then X is replaced by Z, and Y is pushed onto the stack.

A machine processes an input string $x \in \Sigma^*$ one token at a time. For each input token, the machine looks at the current input, state and top of the stack to make a transition into a new state and update its stack. The machine can also take empty string as input and make a transition based on the stack and its current state. The machine halts after reading the input and a string is accepted if the state after reading the complete input is a final state.

A.1 Cantor-set Encodings

In our construction, we will make use of Cantor-set like encodings as introduced in (Siegelmann and Sontag, 1992). The Cantor-set like encodings in base-4 provides us a means to encode a stack of values 0s and 1s and easily apply stack operations like push, pop and top to update them. Let υ_s denote the encoding of a stack. The contents of the stack can be viewed as a rational number $\frac{p}{4^q}$ where $0 < p < 4^q$. The i-th element from the top of the stack can be seen as the i-th element to the right of the decimal point the in a base-4 expansion. A 0 stored in the stacked is associated with a 1 in the expansion while a 1 stored in the stack is associated with 3. Hence, only numbers of the special form $\Sigma_{i=1}^{t} \frac{a_i}{4^i}$ where $a_i \in \{1, 3\}$ will appear in the encoding. For inputs of the form $I \in \{0, 1\}$, the standard stack operations can be applied by simple affine operations. For instance, push(I) operation can be obtained by $\upsilon_s \mapsto \frac{1}{4}\upsilon_s + \frac{1}{2}I + \frac{1}{4}$ and the pop(I) can be obtained by $\upsilon_s \mapsto 4\upsilon_s - 2I - 1$. The top of the stack can be obtained by $\sigma(4\upsilon_s - 2)$ which will be 1 if the top of the stack is 1 else 0. The emptiness of a stack can be checked by $\sigma(\upsilon_s)$ which will be 1 if the stack is nonempty or else it will be 0.

B Construction

We will make use of some intermediate notions to describe our construction. We will use these multiple times in our construction. Particularly, Lemma B.1 will be used to combine the information of the state vector, input and the symbol at the top of the stack. Lemma B.2 and Lemma B.3 will be used to implement the state transition and decisions related to stack operations.

For the feed-forward networks we use the activation as in (Siegelmann and Sontag, 1992), namely the saturated linear sigmoid activation function:

$$\sigma(x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } 0 \leq x \leq 1, \\ 1 & \text{if } x > 1. \end{cases} \tag{1}$$

Note that, we can easily work with the standard ReLU activation via $\sigma(x) = \mathsf{ReLU}(x) - \mathsf{ReLU}(x-1)$.

Consider two sets Φ and Ψ (such as the set of states Q and the set of inputs Σ), and consider the one-hot representations of their elements $\phi \in \Phi$ and $\psi \in \Psi$ as $\boldsymbol{\phi} \in \mathbb{Q}^{|\Phi|}$ and $\boldsymbol{\psi} \in \mathbb{Q}^{|\Psi|}$ respectively. We use $(\phi, \psi) \in \mathbb{Q}^{|\Phi|\times|\Psi|}$ to denote a unique one-hot encoding for each pair of ϕ and ψ. More specifically, consider the enumerations $\pi_\Phi : \Phi \to \{1, 2, \ldots, |\Phi|\}$ and $\pi_\Psi : \Psi \to \{1, 2, \ldots, |\Psi|\}$. Then given two elements $\phi \in \Phi$ and $\psi \in \Psi$, the vector $(\phi, \psi) \in \mathbb{Q}^{|\Phi|\times|\Psi|}$ will have a 1 in position $(\pi_\Psi(\psi)-1)|\Phi|+\pi_\Phi(\phi)$ and a 0 in every other position. We now prove that given a vector $[\boldsymbol{\phi}, \boldsymbol{\psi}]$ containing concatenation of one-hot representations of the elements ϕ and ψ, there exists a single-layer feedforward network that can produce the vector (ϕ, ψ).

Lemma B.1. *There exists a function $O(\boldsymbol{x}) : \mathbb{Q}^{|\Phi|+|\Psi|} \to \mathbb{Q}^{|\Phi||\Psi|}$ of the form $\sigma(\boldsymbol{x}\mathbf{W} + \boldsymbol{b})$ such that,*

$$O([\boldsymbol{\phi}, \boldsymbol{\psi}]) = [(\phi, \psi)]$$

Proof. Let $\mathbf{A}_i$ for $i \in \{1, \ldots, |\Phi|\}$ denote a matrices of dimensions $|\Psi| \times |\Phi|$ such that $\mathbf{A}_i$ has 1s in its i-th row and 0 everywhere else. For any one-hot vector $\boldsymbol{\phi}$, note that $\boldsymbol{\phi}\mathbf{A}_i = \mathbf{1}$ if $i = \pi_\Phi(\phi)$ or else it is $\mathbf{0}$. Thus, consider the transformation,

$$\boldsymbol{t}_{(\phi,\psi)} = [\, \boldsymbol{\phi} + \boldsymbol{\psi}\mathbf{A}_1, \boldsymbol{\phi} + \boldsymbol{\psi}\mathbf{A}_2, \ldots, \boldsymbol{\phi} + \boldsymbol{\psi}\mathbf{A}_{|\Psi|}]$$

Note that, the vector $\boldsymbol{t}_{(\phi,\psi)}$ has a value 2 exactly at the position $(\pi_\Psi(\psi) - 1)|\Phi| + \pi_\Phi(\phi)$ and it is either 0 or 1 at the rest of the positions. Hence by making use of bias vectors, it is easy to obtain $[(\phi, \psi)]$,

$$\sigma(\boldsymbol{t}_{(\phi,\psi)} - \mathbf{1}) = [(\phi, \psi)]$$

which is what we wanted to show.

□

As an example, consider two sets Φ and Ψ such that $|\Phi| = 3$ and $|\Psi| = 2$. Consider two elements $\phi \in \Phi$ and $\psi \in \Psi$ such that $\pi_\Phi(\phi) = 3$ and $\pi_\Psi(\psi) = 2$. Hence, this implies the corresponding one-hot vectors $\boldsymbol{\phi} = [0, 0, 1]$ and $\boldsymbol{\psi} = [0, 1]$. According to the construction above, the weight matrix will be,

$$\boldsymbol{W} = \left[\begin{array}{cccccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right],$$

Thus by construction, the output of the feedforward network $\sigma([\boldsymbol{\phi}, \boldsymbol{\psi}]\mathbf{W} - \mathbf{1}) = [0, 0, 0, 0, 0, 1]$. A similar technique was employed by (Prez et al., 2019) in their Turing completeness result for Transformer.

We will describe another technical lemma that we will make use of to implement our transition functions and other such mappings. Consider two sets Φ and Ψ (such as the set of states Q and the set of inputs Σ or set of stack symbols Γ), and consider the one-hot representations of their elements $\phi \in \Phi$ and $\psi \in \Psi$ as $\boldsymbol{\phi} \in \mathbb{Q}^{|\Phi|}$ and $\boldsymbol{\psi} \in \mathbb{Q}^{|\Psi|}$ respectively. Let $\delta : \Phi \times \Psi \to \Phi$ be any transition function that takes elements of two sets as input and produces an element of one of the sets. Let $[(\phi, \psi)] \in \mathbb{Q}^{|\Phi|\times|\Psi|}$ denote a unique one-hot encoding for each pair of ϕ and ψ as defined earlier. We demonstrate that given $[(\phi_1, \psi)]$ as input, there exists a single layer feedforward network that produces the vector $\boldsymbol{\phi}_2$ if $\delta(\phi_1, \psi) = \phi_2$.

Lemma B.2. *There exists a function $O(\boldsymbol{x}) : \mathbb{Q}^{|\Phi||\Psi|} \to \mathbb{Q}^{|\Phi|}$ of the form $\sigma(\boldsymbol{x}\mathbf{W} + \boldsymbol{b})$ such that,*

$$O([(\phi_1, \psi)]) = \boldsymbol{\phi}_2$$

Proof. This can easily implemented using a linear transformation. Consider a matrix $\mathbf{A} \in \mathbb{Q}^{|\Phi||\Psi|\times|\Phi|}$. Given two inputs $\phi_i, \phi_k \in \Phi$ and $\psi_j \in \Psi$ such that $\delta(\phi_i, \psi_j) = \phi_k$, the row $(\pi_\Psi(\psi_j) - 1)|\Phi| + \pi_\Phi(\phi_i)$ of the matrix $\mathbf{A}$ will be the one-hot vector corresponding to ϕ_k, that is, $\mathbf{A}_{(\pi_\Psi(\psi_j)-1)|\Phi|+\pi_\Phi(\phi_i),:} = \boldsymbol{\phi}_k$.

□

Similarly, a mapping $\theta : \Phi \times \Psi \to \{0,1\}^n$ can also be implemented using linear transformation using a transformation matrix $\mathbf{A} \in \mathbb{Q}^{|\Phi||\Psi| \times n}$ where each row of the matrix $\mathbf{A}$ will be the corresponding mapping $\{0,1\}^n$ for the pair of (ϕ, ψ) corresponding to that row.

Lemma B.3. *There exists a function $O(\boldsymbol{x}) : \mathbb{Q}^{|\Phi||\Psi|} \to \{0,1\}^n$ of the form $\sigma(\boldsymbol{x}\mathbf{W} + \boldsymbol{b})$ such that,*

$$O([(\phi, \psi)]) = [\{0,1\}^n]$$

Proof is similar to proof of Lemma B.2.

Proposition B.4. *For any Deterministic Pushdown Automaton, there exists an RNN that can simulate it.*

Proof. The construction is straightforward and follows by induction. We will show that at the t-th timestep, given that the model has information about the state and a representation of the stack, the model can compute the next state and update the stack representation based on the input. More formally, given a sequence $x_1, x_2, \ldots, x_n \in \Sigma^*$, consider that the hidden state vector at the t-th timestep is $\boldsymbol{h}_t = [\boldsymbol{q}_t, \boldsymbol{\omega}_t]$ where $\boldsymbol{q}_t \in \mathbb{Q}^{|\Sigma|}$ is a one-hot encoding of the state vector and $\boldsymbol{\omega}_t \in \mathbb{Q}^{|\Gamma|}$ is a representation of the stack based on the cantor-set like encoding. Then, given an input $x_{t+1} \in \Sigma$, we will show how the network can compute $\boldsymbol{h}_{t+1} = [\boldsymbol{q}_{t+1}, \boldsymbol{\omega}_{t+1}]$. After reading the whole input, a sequence is accepted if $\boldsymbol{q}_n$ is in the set of final states F or else it is rejected.

Our construction will use a 5-layer feed forward network that takes as input the vectors $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ at each timestep and produces the vector $\boldsymbol{h}_t$. The vectors $\boldsymbol{h}_t \in \mathbb{Q}^{|Q|+|\Gamma|}$ will have two subvectors of size $|Q|$ and $|\Gamma|$ containing a one-hot representation of the state of the underlying automaton and a representation of the stack encoded in the Cantor-set representation respectively. For each input symbol $x \in \Sigma$, its corresponding input vector $\boldsymbol{x} \in \mathbb{Q}^{|\Sigma|}$ will be a one-hot vector. If the underlying stack takes empty string as input at particular step, the RNN will take a special symbol as input which also have a unique one-hot representation similar to other input symbols.

As opposed to the construction of Siegelmann and Sontag (1992), which only takes 0s and 1s as input and use a scalar to encode a stack of 0s and 1s, we will encode one-hot representation of stack symbols in vectors of size $|\Gamma|$. The push and pop operations will always be in the form of one-hot vectors and this will ensure that retrieving the top element provides a one-hot encoding of the stack symbol.

Details of the construction. At each timestep, the model will receive the hidden state vector $\boldsymbol{h}_{t-1} = [\boldsymbol{q}_{t-1}, \boldsymbol{\omega}_{t-1}]$ and the input vector $\boldsymbol{x}_t$ as input. At timestep 0, the hidden state vector will be $\boldsymbol{h}_0 = [\boldsymbol{q}_0, \boldsymbol{\omega}_0]$ containing the one-hot representation of the initial state and stack encoding containing Z_0. For instance consider $|Q| = 3$ and $|\Gamma| = 4$. If the one-hot encoding of q_0 is $\boldsymbol{q}_0 = [1, 0, 0]$ and one-hot encoding of Z_0 is $\boldsymbol{z}_0 = [1, 0, 0, 0]$, then $\boldsymbol{\omega}_0 = \frac{1}{4}\mathbf{0} + \frac{1}{2}[1, 0, 0, 0] + \frac{\mathbf{1}}{4} = [3/4, 1/4, 1/4, 1/4]$. That is, the vector $\boldsymbol{z}_0$ is pushed to the empty stack using the Cantor-set encoding method described in section A.1. Hence, the vector $\boldsymbol{h}_0 = [1, 0, 0, 3/4, 1/4, 1/4, 1/4]$.

The first layer of the feedforward network $\sigma(\mathbf{W}_h \boldsymbol{h}_{t-1} + \mathbf{W}_x \boldsymbol{x}_t + \boldsymbol{b})$ will produce the vector $\boldsymbol{h}^{(1)}_{t-1} = [(q_{t-1}, x_t), \boldsymbol{\tau}^{top}_{t-1}, \boldsymbol{\omega}_{t-1}]$, where $\boldsymbol{\tau}^{top}_{t-1} \in \mathbb{Q}^{|\Gamma|}$ denotes a one-hot vector representation of the symbol at the top of our stack representation and the subvector $(q_{t-1}, x_t) \in \mathbb{Q}^{|Q| \times |\Sigma|}$ is a unique one-hot vector for each pair of state q and input x. Thus, the vector $\boldsymbol{h}^{(1)}_{t-1}$ is of dimension $|Q|.|\Sigma| + 2|\Gamma|$. The vector (q_{t-1}, x_t) can be obtained by using Lemma B.1 where $\Phi = Q$ and $\Psi = \Sigma$. The vector corresponding to the symbol at the top of the stack can be easily obtained using the top operation ($\sigma(\mathbf{4}\boldsymbol{\omega}_{t-1} - \mathbf{2})$) defined in section A.1.

In the second layer, we will use Lemma B.1 again to obtain a unique one-hot vector for each combination of the state, input and stack symbol. The output of the second layer of the feedforward network will be of the form $\boldsymbol{h}^{(2)}_{t-1} = [(q_{t-1}, x_t, \boldsymbol{\tau}^{top}_{t-1}), \boldsymbol{\tau}^{top}_{t-1}, \boldsymbol{\omega}_{t-1}]$, where the subvector $(q_{t-1}, x_t, \boldsymbol{\tau}^{top}_{t-1})$ is a unique one-hot encoding for each combination of the state $q \in Q$, input $x \in \Sigma$ and a stack symbol $\tau \in \Gamma$. Hence, the vector $\boldsymbol{h}^{(2)}_{t-1}$ will be of the dimension $|Q|.|\Sigma|.|\Gamma| + 2|\Gamma|$. Since we already had the vector (q_{t-1}, x_t), and the vector $\boldsymbol{\tau}^{top}_{t-1}$, the vector $(q_{t-1}, x_t, \boldsymbol{\tau}^{top}_{t-1})$ can be obtained using Lemma B.1 by considering $\Phi = Q \times \Sigma$ and $\Psi = \Gamma$. The primary idea is that the vector $(q_{t-1}, x_t, \boldsymbol{\tau}^{top}_{t-1})$ provides us with all the necessary information required to implement the further steps and produce $\boldsymbol{q}_t$ and $\boldsymbol{\omega}_t$.

We can use the vector $(q_{t-1}, x_t, \boldsymbol{\tau}_{t-1}^{top})$ to directly map to $\boldsymbol{q}_t$ using a simple linear transformation. To obtain $\boldsymbol{\omega}_t$, we will produce all three candidate stack representations corresponding to push, pop and no-operation. That is, from $(q_{t-1}, x_t, \boldsymbol{\tau}_{t-1}^{top})$ we will obtain $\boldsymbol{\omega}_t^{push}$, $\boldsymbol{\omega}_t^{pop}$ and $\boldsymbol{\omega}_t^{no-op}$. Along with that, using linear transformation we will obtain three control signals $\boldsymbol{c}_{push}, \boldsymbol{c}_{pop}$ and $\boldsymbol{c}_{no-op}$. To obtain the final stack representation $\boldsymbol{\omega}_t$, we will implement the following operation

$$\boldsymbol{\omega}_t = \boldsymbol{c}_{push}.\boldsymbol{\omega}_t^{push} + \boldsymbol{c}_{pop}.\boldsymbol{\omega}_t^{pop} + \boldsymbol{c}_{no-op}.\boldsymbol{\omega}_t^{no-op}. \tag{2}$$

We will implement the above steps using an additional three layers of feedforward network and thus we will obtain $\boldsymbol{h}_t = [\boldsymbol{q}_t, \boldsymbol{\omega}_t]$.

The third layer of the feedforward network will produce the vector

$$\boldsymbol{h}_{t-1}^{(3)} = [\boldsymbol{q}_t, \boldsymbol{\omega}_t, \boldsymbol{\omega}_t^{pop}, \boldsymbol{\omega}_t^{no-op}, \boldsymbol{\tau}_{t-1}^{push}, \boldsymbol{c}_{push}, \boldsymbol{c}_{pop}, \boldsymbol{c}_{no-op}]$$

The vector $\boldsymbol{q}_t$ in $\boldsymbol{h}_{t-1}^{(3)}$ can be obtained using Lemma B.2 given the vector $(q_{t-1}, x_t, \boldsymbol{\tau}_{t-1}^{top})$ in $\boldsymbol{h}_{t-1}^{(2)}$. The vector $\boldsymbol{\omega}_t^{pop}$ via the Cantor set encoding method using the transformation $\mathbf{4}\boldsymbol{\omega}_t - \mathbf{2}\boldsymbol{\tau}_{t-1}^{top} - \mathbf{1}$ over $\boldsymbol{h}_{t-1}^{(2)}$. The vector $\boldsymbol{\omega}_t^{no-op}$ can be obtained using the Identity Transformation. The vector $\boldsymbol{\tau}_{t-1}^{push}$ can be obtained using Lemma B.2. If for a given transition the stack operation is not a push operation then $\boldsymbol{\tau}_{t-1}^{push} = \mathbf{0}$. The vector $\boldsymbol{c}_{push} = \mathbf{1}$ if the current stack operation is Push and it is $\mathbf{0}$ otherwise. Similarly, the vectors $\boldsymbol{c}_{pop}$ and $\boldsymbol{c}_{no-op}$ are $\mathbf{1}$ if the current operation is Pop or No-operation respectively and are $\mathbf{0}$ otherwise. The vectors $\boldsymbol{c}_{push}, \boldsymbol{c}_{pop},$ and $\boldsymbol{c}_{no-op}$ can be obtained using Lemma B.3. During any timestep, only one of them is $\mathbf{1}$ vector and rest of them are zero vectors. The vectors $\boldsymbol{c}_{op}$s can be seen as control signals. That is, the candidate stack representation will be used if its corresponding control signal is $\mathbf{1}$ or else the candidate stack representation will be transformed to zero vector.

In the fourth layer, the feedforward network will produce the following vector,

$$\boldsymbol{h}_{t-1}^{(4)} = [\boldsymbol{q}_t, \boldsymbol{\omega}_t^{pop}.\boldsymbol{c}_{pop}, \boldsymbol{\omega}_t^{no-op}.\boldsymbol{c}_{no-op}, \boldsymbol{\omega}_t^{push}.\boldsymbol{c}_{push}]$$

For any operation denoted by $op \in \{push, pop, no-op\}$, the vector

$$\boldsymbol{\omega}_t^{op}.\boldsymbol{c}_{op} = \begin{cases} \boldsymbol{\omega}_t^{op} & \text{if } \boldsymbol{c}_{op} = \mathbf{1}, \\ \mathbf{0} & \text{if } \boldsymbol{c}_{op} = \mathbf{0}. \end{cases} \tag{3}$$

Since we are using first order RNNs and hence multiplicative operations are not directly possible. To implement the operation in equation 3 via linear transformations with saturation linear sigmoid activation, first note that $0 \leq \boldsymbol{\omega}_t \leq 1$ and thus $\sigma(\boldsymbol{\omega}_t - \mathbf{1}) = \mathbf{0}$. Using that we can implement the operation in equation 3 by,

$$\sigma(\boldsymbol{\omega}_t^{op} + \boldsymbol{c}_{op} - \mathbf{1}) = \begin{cases} \boldsymbol{\omega}_t^{op} & \text{if } \boldsymbol{c}_{op} = \mathbf{1}, \\ \mathbf{0} & \text{if } \boldsymbol{c}_{op} = \mathbf{0}. \end{cases} \tag{4}$$

In the fourth layer of the feedforward network we obtain the candidate stack representation for Push operation via linear operations as described in section A.1 along with adding $\boldsymbol{c}_{push}$ and adding $-\mathbf{1}$ via bias vectors. For the candidate stack representations of Pop and No-op operations, we simply add their control vectors and subtract by $\mathbf{1}$ via bias vectors. In the vector $\boldsymbol{h}_{t-1}^{(4)}$, only one of the stack representation will have nonzero values and the other representations will be zero vectors depending on the control signals $\boldsymbol{c}_{push}, \boldsymbol{c}_{pop},$ and $\boldsymbol{c}_{no-op}$.

The fifth layer will simply sum the three candidate stack vectors along with their control signals to obtain,

$$\boldsymbol{h}_{t-1}^{(5)} = [\boldsymbol{q}_t, \boldsymbol{\omega}_t^{pop}.\boldsymbol{c}_{pop} + \boldsymbol{\omega}_t^{no-op}.\boldsymbol{c}_{no-op} + \boldsymbol{\omega}_t^{push}.\boldsymbol{c}_{push}]$$

which is equal to,

$$[\boldsymbol{q}_t, \boldsymbol{\omega}_t] = \boldsymbol{h}_t$$

Dataset	Training Data			Test Data					
	Size	Length Range	Depth Range	Size per Bin	Bin-2 Length Range	Bin-2 Depth Range	Length Increments	Depth Increments	Number of Bins
Unbounded Length and Depth	10000	[2, 50]	Unrestricted	1000	[52, 100]	Unrestricted	50	NA	2
Bounded Depth	10000	[2, 50]	[1, 10]	1000	[52, 100]	[1, 10]	50	0	3
Bounded Length	10000	[2, 100]	[1, 15]	1000	[2, 100]	[16, 16]	0	1	6

Table 2: Statistics of different datasets used in the experiments. Note that the distribution of the first bin is always defined by the training set, hence for test set we report the statistics from the second bin. The depth and length bounds for the other bins can be obtained by considering the lower bound of a bin i as one plus the upper bound of the previous bin $i-1$ and obtaining the upper bounds by adding the Length and Depth Increments to the previous bin's upper bounds.

Hyperparameter	Bounds
Hidden Size	[4, 256]
Heads	[1, 4]
Number of Layers	[1, 2]
Learning Rate	[1e-2, 1e-3]
Position Encoding	[True, False]

Table 3: Different hyperparameters and the minimum and maximum values considered for each of them. Note that certain parameters like Heads and Position Encodings are only relevant for Transformer based models and not for LSTMs. Note that, for Dyck-2 we also found them to generalize to Bin-1A with bounds mentioned in Suzgun et al. (2019a) (such as with hidden size 8).

which is what we wanted to show.

□

The above construction is a direct simulation of Deterministic Pushdown Automaton via RNNs in real-time. The construction of Siegelmann and Sontag (1992) first takes all the inputs and then takes further processing time. However, in practice, RNNs process the inputs and produce outputs in real-time.

Note that, the dependence on precision is primarily determined by the depth of the stack. That is, as the number of elements in the stack keep increasing, the stack representation gets exponentially smaller due to the Cantor-set encoding scheme. If for a set of input strings, the depths are bounded, then for some finite precision, an RNN based on the above construction will be able to recognize strings of arbitrary lengths.

C Experimental Details

We run our experiments on three Context Free Languages namely Dyck-2, Dyck-3 and Dyck-4 for LSTM and Transformer based models. Three separate datasets were generated for each language, to run the ablations, details of which are given in Table 2. For each of these experiments we do extensive hyperparameter tuning before reporting the final results. Table 3 provides different hyperparameters considered in our experiments and their bounds. All in all, this resulted in about 56 different settings for each dataset for RNNs and about 144 settings for Transformers. While reporting the final scores we take an average of the accuracies corresponding the top-5 hyperparameter settings.

All of our models were trained using RMSProp Optimizer with a smoothing constant α of 0.99. For each language and its corresponding datasets, we use a batch size of 32 and train for 100 epochs. In case an accuracy of 0.99 is acheived for all of the bins before completing 100 epochs we stop the training

Language	Model	Validation Set 1 $p = 0.5, q = 0.25$	Validation Set 2 $p = 0.4, q = 0.35$	Validation Set 3 $p = 0.6, q = 0.15$
Dyck-2	**LSTM**	99.5	99.6	99.1
	Transformer	95.1	94.7	94.4
Dyck-3	**LSTM**	97.3	98.8	96.5
	Transformer	87.7	87.8	87.3
Dyck-4	**LSTM**	97.8	96.7	94.9
	Transformer	92.7	89.8	90.9

Table 4: The performance of neural models on considered Dyck languages for data generated from three different distributions. Validation Set 1 was constructed from the same distribution used to generate the training data, while the other two were generated from different distributions. All the validation sets had strings with the lengths in the interval [2, 50] and there was no restriction kept on the depth of these strings.

process at that point. We run all of our experiments on 4 Nvidia Tesla P100 GPUs each containing 16GB memory.

Probing Details For designing a probe for extracting the depth of the underlying stack of Dyck-2 substrings from the cell states of pretrained LSTMs, we used a single hidden layer Feed-Forward network. The hidden size of the network was kept 32 and it was trained using Adam Optimizer (Kingma and Ba, 2014) with a batch size of 200. The accuracy on the validation set was computed by only considering the predictions to be correct for a sequence if it predicted the correct depth at every step of that sequence. In the second set of experiments that aimed to predict the elements of the stack, we trained the LSTM model on the NCP task along with an auxillary loss for predicting the top-10 elements of the stack. For the computation of the auxillary loss, we added 10 parallel linear layers on the top of the LSTM's output with i-th linear layer tasked to predict if the i-th element of the stack was i) a round opening bracket or ii) a square opening bracket or iii) If no element was present at that position. For each of these 10 linear layers we compute Cross Entropy Loss which are then averaged to obtain the auxillary loss. The final loss is computed as:

$$L = L_{NCP} + \lambda L_{aux} \tag{5}$$

where L_{NCP} is the loss obtained from the next character prediction task and L_{aux} is the auxillary loss just described and we use $\lambda = \frac{1}{20}$ in our experiments. The stack extraction auxillary task is evaluated by computing Accuracy and Recall metrics for each stack element. The accuracy for the i-th element is computed by considering if the model can predict the i-th element correctly at each step of a sequence. Since there will be a fewer cases for smaller lengths containing elements at higher depths, we also report Recall for each i-th stack element, where we only consider if the model can correctly predict the sequences containing at least one occurrence of depth i.

D Robustness Experiments

To ensure that our results didn't overfit on the training distribution we did some robustness experiments to check efficacy of the considered neural models. As a reminder, the PCFG for Dyck-n languages is given by the following derivation rules:

$$S \rightarrow ({}_i S)_i, \text{ with probability } p \tag{6}$$

$$S \rightarrow SS, \text{ with probabililty } q \tag{7}$$

$$S \rightarrow \epsilon, \text{ with probbaility } 1 - (p + q) \tag{8}$$

$$\tag{9}$$

For the experiments described in the main paper we used $p = 0.5$ and $q = 0.25$. To check the generalization ability of our models we checked whether a model trained with the strings generated using these values can generalize on Dyck words obtained from a different distribution. Table 4 shows the accuracies obtained by LSTMs and Transformers on data generated from different distributions. It can be

observed from the results that the performance of the models for all languages remain more or less the same across different distributions.

A model trained on the Next Character Prediction task can also be used to generate strings of the language it was trained on by starting from an empty string and then exhaustively iterating over all possible valid characters predicted by the model. We used this idea to check if a pretrained LSTM model can indeed generate all possible Dyck-2 strings upto a certain length (since the number of possible strings will grow exponentially with increasing lengths). For a maximum length of 10, there exists a total of 1619 valid Dyck-2 strings. When we used a pretrained model to exhaustively generate the valid strings, we observed that it produced exactly those 1619 strings, no more and no less.

E Additional Results

The depth generalization results for the neural models on Dyck-3 and Dyck-4, are given in Figure 4. Similar to Dyck-2, here also we see a gradual drop in performance as we move to the higher depths but Transformers perform relatively better than LSTMs.

(a) (b)

Figure 4: Generalization of LSTMs and Transformers on higher depths for (a) Dyck-3 and (b) Dyck-4. The lengths of strings in the training set and all validation sets were fixed to lie between 2 to 100.