Second-order RNNs: Resurgence of Recurrence

Stanford CS224N Custom Project

Sundararajan Renganathan and Shreya Singh
Department of Computer Science
Stanford University
{rsundar, ssingh16}@stanford.edu

Abstract

[1][2] This project explores input-dependent hidden-to-hidden transitions for RNNs with the goal of developing robust language models for natural language and the m-bounded Dyck-k language. Our contributions are two-fold: (1) a special purpose RNN with input dependent transitions that allows the hidden state to function as a LIFO-queue to predict the closing bracket for the m-bounded Dyck-k language and (2) a novel extension of mLSTM [1] to capture more complex input-dependent behavior by using the insights from modelling the Dyck language. We further evaluate variations of our models on character and word level modelling tasks and present various qualitative and quantitative findings.

1 Key Information to include

- Mentor: John Hewitt

2 Introduction

Language modelling is the art of predicting the probability of occurrence of a token (character or word) in a particular sequence. Over the course of advancement in language modelling, many different forms of language models have emerged - like the n-gram language model, window-based neural language model, RNNs, LSTMs and GRUs. Although each of these models are significantly different from each other and overcome the shortcomings faced by the previous version of state-of-the-art model, the fundamental principle behind their architectures is consistent. Language models accomplish their goal of predicting the next token of a sequence by conditioning on a window of n previously occurring tokens. The size of the window, however, could range from only a subset of the previously occurring sequence to the entire previous sequence.

Recurrent Neural Networks (RNNs) are one such category of language models which make use of the entire previous sequence of tokens to predict the next token. RNNs are powerful density estimators which can capture long contexts to make predictions. The two main parameters of an RNN are : (1) the input token (x_t) which is fed into the model at each time step t of the sequence and (2) the hidden state vector (h_t) which summarizes the past inputs. The recursive nature of RNN comes into play while updating the hidden state vector h_t using a non-linear combination of the current input x_t and the previous hidden state vector h_{t-1} as

$$h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)) \tag{1}$$

where $W^{(hh)}$ and $W^{(hx)}$ are the weight matrices corresponding to the hidden-to-hidden transition and input-to-hidden transition respectively.

[1] We would like to go with Option 2 grading for a formative feedback.
[2] As discussed we would like to explore the possibility of submitting our work at Rep4NLP workshop @ACL

The hidden state vector h_t for each time step t is multiplied by a weight matrix $W^{(S)}$ and run through a softmax over the corpus to predict the probability distribution over the next sequence element ($\hat{y}_t$):

$$\hat{y}_t = softmax(W^{(S)} h_t) \tag{2}$$

RNNs have been known to suffer from the vanishing gradient problem while training long sequences [2], but advances such as long short-term memory architecture [3] which incorporate a gating mechanism to selectively remember/forget the historical sequence information.

In this project, our primary goal is to investigate the hidden-to-hidden transition weight matrix ($W^{(hh)}$) as specified in Eq. 1 by making it input-dependent. Input-dependent hidden-to-hidden transition weight makes the RNNs more robust against making mistakes when an unexpected input is encountered. It also makes the RNN less likely to be trapped in a bad numerical state for making future predictions as the hidden state now depends on the input of each time-step t [1]. Input-dependent transitions also capture complex input-dependent behavior which makes the models more expressive and robust to surprising inputs. This increased robustness also allows the learning to occur in equivalent, if not lesser number epochs and parameters and results in better performance as shown through our experiments. That being said, input dependent transitions add significantly to the complexity and number of parameters of the model as now, instead of having a single hidden-to-hidden weight matrix (W_{hh}), we have one at each time step. This is seen in previous approaches like the tensor RNN [1] which contains a separately learned transition matrix (W_{hh}) for each input (x_t).

We further bisect our goal of using input-dependent hidden-to-hidden transitions for modelling two different variants of language - Natural language dataset from the Penn Tree Bank [4] and Context-free m-bounded Dyck-k language [5]. For the m-bounded Dyck-k language we propose a special purpose RNN called the Dyck-RNN whose hidden state acts as a stack whose push and pop behavior is facilitated by two static weight matrices (W_1 and W_2). Dyck-RNN helps in our task of predicting the correct closing bracket given an incomplete sequence of parentheses. We further leverage the learning and intuition of using input dependent matrix choice formulation for the Dyck language to introduce a novel architecture for modelling Natural Language called Multi-Matrix Long Short-Term Memory (mmLSTM). mmLSTM is an extension of the Multiplicative Long Short-Term Memory architecture (mLSTM) [1] which is a shared-parameter approximation to tensor RNN and uses a factorized hidden-to-hidden transition matrix. In mmLSTM, instead of having a separate transition matrix (W_{hh}) for each input, a set of matrices ($W_{1..i}$) is shared among the inputs. Further, input-dependent weighted average of these matrices is taken to build the hidden-to-hidden weight matrix (W_{hh}).

Further, we extensively evaluate our two novel architectures on their respective language datasets (Dyck and Penn Tree Bank). Our experimental setup introduces the evaluation metrics for both the architectures and later delves into the details of comparing mmLSTM for character and word-level language modelling tasks against well-established baselines. We also evaluate Dyck-RNN and several natural language models for predicting the correct closing bracket. We further conduct extensive qualitative analysis of our model and publish our findings.

The rest of the paper is organized as follows: In Section 3 we discuss the methods and findings of several related literature. Section 4 discusses our proposed architectures in detail and our main empirical intuitions. In section 5 we discuss our data, experimental setup, experimental metrics and results in detail followed by an extensive qualitative analysis in Section 6. We further conclude and provide future directions in Section 7.

3 Related Work

In our project, we are proposing two novel architectures and so, in this section, we would be discussing some recent literature revolving around both of them. In [6], the authors have introduced a novel RNN variant (mRNN) that uses multiplicative connections to allow the current input token to determine the hidden-to-hidden weight matrix. mRNN was able to surpass the performance of the (then) state-of-the-art model model for character-level language modelling and was able to generate text that exhibited a significant amount of interesting and high-level linguistic structure. However, although mRNNs have improved significantly over vanilla RNNs for character-level language modelling tasks, they fall short against the most popular LSTM architectures [7]. This is because the standard RNN

units in an mRNN suffer from the vanishing gradient problem and suffer from the difficulty of retaining long-term sequence context.

An improvement over the mRNN architecture emerged in the form of multiplicative LSTM for Sequence Modelling (mLSTM) [1]. mLSTM combines the vanilla Long Short Term Memory architecture with Multiplicative RNNs (mRNNs) and is powered by the ability to have a different recurrent transition function for each input of the training sample, just like mRNNs. The main contribution of the mLSTM architecture is to combine the flexible input-dependent transitions of mRNNs with the long time lag and information control of LSTMs. This would make mLSTMs more robust to surprising inputs while also preserving the desirable information flow control properties of vanilla LSTMs. One of the evident drawbacks of the mLSTM architecture is that it has not been evaluated on word-level language modelling tasks as they have solely looked at character level modelling in the evaluations. Also, it would be worthwhile to try out non-linear input dependent transition functions - like using a neural network based W_{hh} representation depending on the input token.

In [8] the authors propose a simple structural design called 'multiplicative integration' which uses hadamard product instead of addition while combining the contributions from input and hidden units. The result of this modification changes the RNN from first order to second order [9]. The effect of multiplication results in a gating type structure and allows potential for greater expressiveness without an increase in the number of parameters. Multiplicative Integration can be integrated into many popular RNN models and achieves state-of-the-art performance on 11 different datasets of varying sizes and scales. Multiplicative integration drives the intuition of how having two or more instances of multiplicative matrices can serve as a gating mechanism and add more expressiveness in the model.

Coming onto the context-free grammars (CGGs), RNNs have been used to learn context-free grammars through the generalized Dyck language. In [10], the authors explore the use of attention mechanisms within the seq2seq framework to learn the Dyck language. The attention mechanism provides some improvement regarding the generalizability of the models with respect to the depth of recursion but still cannot completely generalize over the recursion depth. However, they perform better than other models on the closing bracket tagging task - a task which we evaluate Dyck-RNN on. in [5], the authors introduce m-bounded Dyck-k, a family of formal languages characterized by a hierarchical structure and bounded memory requirements and demonstrate that LSTMs can effectively model such a hierarchical language by turning a part of its hidden state into a stack.

4 Approach

This section provides details about the three novel architectures that we have developed - Dyck-RNN, mmRNN and mmLSTM. Note that mmLSTM is an extension of mmRNN and both have same core architecture. We also discuss some of the baselines architectures which we use to compare our models against.

4.1 Baselines

We compare the performance of mmRNN and mmLSTM against some well-established baselines which don't have input-dependent hidden-to-hidden transitions - like Vanilla RNN and Vanilla LSTM [3]. Among the set of models which do have input-dependent hidden-to-hidden transitions, we pick mRNN [6] and mLSTM [1] architectures explained in Section 3 as our baselines. We compare our models against these baselines for character and word-level language modelling tasks on the PTB dataset. For Dyck language modelling, we don't have any well-established architectures where the hidden state acts like a stack. Hence, we train our set of natural language models over the m-bounded Dyck-2 language language and evaluate their performance for the correct closing bracket prediction task. All the baseline models have been implemented from scratch in PyTorch.

4.2 Dyck-RNN

Dyck-RNN is a special purpose RNN whose hidden state acts as a stack by using two matrices W_1 and W_2 with 1's on the subdiagonal and superdiagonal respectively and 0's everywhere else. W_1, when multiplied by the hidden state shifts the elements down by 1 whereas multiplying by W_2 shifts

the elements up by 1. Each parenthesis is embedded into a 1-dimensional space and these embeddings are non-trainable. A sigmoid (gating) function is applied to the linearly scaled input embedding as in Eq. 3. This gate ($g^{(t)}$) controls whether to perform the push operation or the pop operation i.e. it chooses between W_1 and W_2. Here $W_{proj} \in R^{\text{input embed size} \times 1}$, $g^{(t)} \in R$, $W_1 \in R^{\text{hidden size} \times \text{hidden size}}$, $W_1 \in R^{\text{hidden size} \times \text{hidden size}}$.

$$g^{(t)} = sigmoid(W_{proj} x_t) \tag{3}$$

$$W^{hh(t)} = g^{(t)} \times W_1 + (1 - g^{(t)}) \times W_2 \tag{4}$$

$$h^{(t)} = W^{hh(t)} h^{(t-1)} + g^{(t)} \times W^{hx} x^{(t)} \tag{5}$$

The hidden state at each timestep t is calculated as in Eq. 5. Note that we do not apply a non-linear transformation to the hidden state at each timestep. Finally, after the entire input sequence has been passed through Dyck-RNN, we inspect the top element of the stack (last hidden state vector $t = n$), apply an affine transformation and pass it through a softmax function to get a probability distribution over all k closing brackets. The bracket with the highest softmax probability is the predicted closing bracket and we backpropagate the cross-entropy loss. We predict all the closing brackets during train and test time through this approach.

4.3 Multi Matrix RNN

Inspired by the input dependent matrix choice formulation in Dyck-RNN, we propose the following natural language model which we dub Multi matrix RNN, or **mmRNN**. In mmRNN, we have $W \in R^{4 \times \text{hidden size} \times \text{hidden size}}$ which is a 3-way tensor consisting of 4 choice matrices of dimension $R^{\text{hidden size} \times \text{hidden size}}$. The ideology is similar to that of Dyck-RNN where we have two push and pop matrices W_1 and W_2. We ran our experiments with different number of choice matrices and arrived at the conclusion of having 4 choice matrices through empirical analysis. We then construct a key vector $v^{key} \in R^4$ that is initialized to give equal weight to all the 4 choice matrices in tensor W. At each time step, v^{key} is updated to be a linear combination of the concatenation of key vector and hidden state vector of the previous time step $(t-1)$ and input embedding of the current time step t. We then take a weighted average of all the choice matrices in W through the choice vector v^{key} as shown in Eq. 7. The hidden state vector for each time step t is then calculated as shown in Eq. 8 Here, $W^{proj} \in R^{4 \times (4 + \text{hidden size} + \text{input embed size})}$. We also implemented multi-layer versions of mmRNNs where the input to subsequent layers is the hidden state of the previous layers.

$$v^{key(t)} = softmax(W^{proj} \, [v^{key(t-1)}; h^{(t-1)}; x^{(t)}]) \tag{6}$$

$$W^{hh(t)} = \sum_{i=1}^{4} v^{key(t)}[i] \times W[i] \tag{7}$$

$$h^{(t)} = tanh(W^{hh(t)} h^{(t-1)} + W^{hx} x^{(t)}) \tag{8}$$

For all the natural language models described above, we pass the hidden state at each timestep through a softmax to produce the predictions. We use teacher-forcing to train these networks.

4.4 Multi Matrix LSTM

Multi Matrix LSTM or **mmLSTM** is a hybrid architecture that combines the matrix choice forumulation of mmRNN with the gating framework of LSTMs. The core architecture is the same, however, we introduce three gates - input gate i, forget gate f and output gate o that have recurrent and feed-forward connections just like vanilla LSTMs. The difference is that the matrices controlling the transitions for input, forget and output gates are themselves input dependent. The equations governing mmLSTMs can be found below. We also implemented multi-layer versions of mmLSTMs where the input to subsequent layers is the hidden state of the previous layers.

$$v_{\hat{c}}^{key(t)} = softmax(W_{\hat{c}}^{proj}\ [v_{\hat{c}}^{key(t-1)}; h^{(t-1)}; x^{(t)}]) \tag{9}$$

$$W_{\hat{c}}^{hh(t)} = \sum_{j=1}^{4} v_{\hat{c}}^{key(t)}[j] \times W_{\hat{c}}[j] \tag{10}$$

$$v_{i}^{key(t)} = softmax(W_{i}^{proj}\ [v_{i}^{key(t-1)}; h^{(t-1)}; x^{(t)}]) \tag{11}$$

$$W_{i}^{hh(t)} = \sum_{j=1}^{4} v_{i}^{key(t)}[j] \times W_{i}[j] \tag{12}$$

$$v_{f}^{key(t)} = softmax(W_{f}^{proj}\ [v_{f}^{key(t-1)}; h^{(t-1)}; x^{(t)}]) \tag{13}$$

$$W_{f}^{hh(t)} = \sum_{j=1}^{4} v_{f}^{key(t)}[j] \times W_{f}[j] \tag{14}$$

$$v_{o}^{key(t)} = softmax(W_{o}^{proj}\ [v_{o}^{key(t-1)}; h^{(t-1)}; x^{(t)}]) \tag{15}$$

$$W_{o}^{hh(t)} = \sum_{j=1}^{4} v_{o}^{key(t)}[j] \times W_{o}[j] \tag{16}$$

$$i^{(t)} = sigmoid(W_{i}^{hh(t)} h^{(t-1)} + W_{i}^{hx} x^{(t)}) \tag{17}$$

$$f^{(t)} = sigmoid(W_{f}^{hh(t)} h^{(t-1)} + W_{f}^{hx} x^{(t)}) \tag{18}$$

$$o^{(t)} = sigmoid(W_{o}^{hh(t)} h^{(t-1)} + W_{o}^{hx} x^{(t)}) \tag{19}$$

$$\hat{c}^{(t)} = tanh(W_{\hat{c}}^{hh(t)} h^{(t-1)} + W_{\hat{c}}^{hx} x^{(t)}) \tag{20}$$

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)} \tag{21}$$

$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)}) \tag{22}$$

5 Experiments

5.1 Data

We use two datasets in our project:

- For natural language modelling, we will use the Penn Treebank (PTB) dataset which is composed of 2499 stories selected from a set of 98,732 stories which appeared in the Wall Street Journal (WSJ) over a three year period. This dataset has been annotated significantly with labels like part-of-speech tags (among other things) but for our task of character-level language modelling, we won't be using these annotations.
- For the m-bounded Dyck-k language modelling (Dyck-RNN), we have an artificial dataset that our project mentor John Hewitt has generated and shared with us. This dataset contains complete sequences (i.e. sequences of parentheses in the m-bounded Dyck-k language for which each opening bracket has a corresponding closing bracket) for $m \in \{4, 6, 8\}$ and $k = 2$. For each value of m and k, this dataset contains 24,000 complete input sequences, each of which has a length of the order of 100's of characters (parentheses).

5.2 Evaluation method

For the character-level natural language modelling task, we measure bits per character[3] (BPC) on the test set. For the word-level modelling task, we simply compute the average cross entropy loss on the test set and use this for our comparisons. We compare mmRNNs and mmLSTMs against all our baselines.

For closing bracket prediction, we compute the *worst-case long-distance prediction accuracy (WCPA)* [5] for each $m \in \{4, 6, 8\}$ and $k = 2$ (and report it as percentage points) where a prediction is correct if at least 80% probability mass is assigned to the correct bracket.

[3] BPC is the average cross entropy over characters where the logarithms are taken in base 2.

Model Type	Model Config	Layers	Train Time (mins)	Epochs to converge
Dyck model, $m \in \{4, 6, 8\}$	h = m	1	3, 6 and 10	4, 2 and 1
Vanilla RNN	h = 2032	1	59	26
Vanilla RNN	h = 1194	2	58	23
Vanilla LSTM	h = 991	1	43	18
Vanilla LSTM	h = 589	2	57	22
mRNN	h = m = 1972	1	51	7
mLSTM	h = m = 977	1	53	9
mmRNN	h = 1038	1	107	20
mmRNN	h = 696	2	247	22
mmLSTM	h = 512	1	108	20
mmLSTM	h = 345	2	252	25
Vanilla RNN	h = 133	1	9	29
Vanilla LSTM	h = 125	1	10	33
Vanilla LSTM	h = 115	2	16	47
mmRNN	h = 128	1	11	24
mmLSTM	h = 112	1	20	33
mmLSTM	h = 97	2	59	50

Table 1: Experimental Details (Dyck, Character and Word models respectively)

5.3 Experimental details

We train the natural language models on the Tesla M60 GPU. For the character-level and word-level language modelling task on PTB dataset, the learning rate was set to 0.001. For all our models, we use the ADAM optimizer and clip gradients to 3.5. We set the maximum number of epochs to 200 and use early stopping with a patience of 5. We have tried to keep the number of parameters approximately the same for each model. We use a batch size of 128 for all the natural language experiments. We apply a dropout of 0.25 to the hidden state before passing it through the softmax at each timestep so as to help models generalize better. This dropout is applied to all the models described above in order to have a fair comparison. For the character level modelling task, the emdedding size is 128 while the word embedding size is 64. Other experimental details can be found in Table 1.

We optimized our implementations of mRNN, mLSTM, mmRNN and mmLSTM to allow us to run bigger models with millions of parameters and this can be seen in comparable training times with respect to vanilla RNNs and vanilla LSTMs which have been optimized for GPUs. We made these optimizations by noticing, from our earlier implementations, that the most time and memory consuming task was the construction of $W^{hh(t)}$ matrices for the entire batch (i.e. of size $batch\_size \times hidden\_size \times hidden\_size$). Note that this is not the question of simple broadcasting of a fixed $W^{hh(t)}$ because each element in the batch would have a different (input-dependent) $W^{hh(t)}$. However, we realized that we can replace the operations in mmRNNs and mmLSTMs with equivalent operations where we don't have to construct these large matrices. This optimization allowed us to run models which are more than 10 times the size of our earlier models.

We used a batch size of 512 for modelling Dyck-RNN. Dyck-RNN was trained on the CPU with a learning rate of 0.01 and training stops when the dev loss goes below 10^{-5}.

5.4 Results

Table 2 records the WCPA, BPC and the average cross entropy loss scores for Dyck-RNN and natural language models.

Across both character and word level models, we can make the following observations: we see that LSTM based models like Vanilla LSTM, mLSTM and mmLSTM perform better than their RNN counterparts, which is expected. We see that the single layer mmLSTM performs better than single layer vanilla LSTM and it converges in approximately the same number of epochs as the latter. We also see that single layer and double layer mmRNNs perform better than their vanilla RNN counterparts. This shows us that having input dependent transitions allows for the model to generalize

Model Type	Number of parameters	Metric	Metric score
Dyck model	5	WCPA	**100**, for all $m \in \{4, 6, 8\}$
Vanilla RNN	4.5M	BPC	1.419
Vanilla RNN (2 layers)	4.5M	BPC	1.369
mRNN	4.5M	BPC	1.460
mLSTM	4.5M	BPC	1.450
mmRNN	4.5M	BPC	1.361
mmRNN (2 layers)	4.5M	BPC	1.333
Vanilla LSTM	4.5M	BPC	1.308
mmLSTM	4.5M	BPC	**1.301**
Vanilla LSTM (2 layers)	4.5M	BPC	**1.283**
mmLSTM (2 layers)	4.5M	BPC	1.288
Vanilla RNN	2M	Average CE loss	4.847
mmRNN	2M	Average CE loss	4.822
Vanilla LSTM	2M	Average CE loss	4.816
mmLSTM	2M	Average CE loss	**4.794**
Vanilla LSTM (2 layers)	2M	Average CE loss	**4.770**
mmLSTM (2 layers)	2M	Average CE loss	4.818

Table 2: Experimental Results (Dyck, Character and Word models respectively)

better to unseen inputs and thereby outperform popular and ubiquitous architectures like LSTMs even for millions of parameters. However, we also see that 2-layer mmLSTMs are not able to beat 2-layer LSTMs and we hypothesize that it is because the number of hidden states in 2-layer mmLSTMs are too small. It could also be because having input dependent transitions in the second layer (where the input in this case is basically the hidden state of the previous layer) is not necessary. Investigating the poor performance of multi-layer mmLSTMs is left to future work.

For the Dyck-RNN, we achieve 100% on WCPA, which is a strict metric. This high score shows that the model effectively learns stack-like behavior by using only m hidden units. We also implemented our natural language models (RNNs, LSTMs, mmRNNs and mmLSTMs) for the Dyck language to investigate if natural language models could model hierarchical language. On evaluation, these models failed miserably and none could achieve a WCPA score of above 50%.

6 Analysis

6.1 Generalizing from partial sentences

To test out our original hypothesis that having input dependent transitions can help models achieve better generalization, we run the following experiment based on the character level language modelling task: we cut off sentences during the training phase and use these partial sentences to train the model. The validation/dev set still contains complete sentences and this set is used to pick the best model by having early stopping with a patience of 5. The test set contains complete sentences too as we want to see how models trained on curtailed sentences generalize to full sentences. We do this experiment only for the vanilla LSTM and mmLSTM models because as per the results from Section 5, these two are the best performing models. The results can be seen in Table 3. A sentence cutoff of 25 means that during training we cut off sentences at 25 characters (including whitespaces). We can see that for three different cutoffs (25, 50 and 75), the two-layer mmLSTM model generalizes better than all the other models, including the two-layer vanilla LSTM. This gives an indication that having input dependent transitions can lead to better generalization to unseen settings.

6.2 Unfreezing static embeddings in Dyck-RNN

To see if our Dyck model can learn if the embedding isn't statically defined by hand, we unfreeze the embeddings and train on the same closing bracket prediction task. This means that the gradients now pass through the embeddings as well. In this case, we see that for a 1-dimensional embedding (initialized randomly), the model is unable to achieve good performance on the WCPA metric.

Model Type	Number of parameters	Sentence cutoff	Test BPC
Vanilla LSTM	4.5M	25	1.591
Vanilla LSTM (2 layers)	4.5M	25	1.582
mmLSTM	4.5M	25	1.580
mmLSTM (2 layers)	4.5M	25	**1.577**
Vanilla LSTM	4.5M	50	1.427
Vanilla LSTM (2 layers)	4.5M	50	1.416
mmLSTM	4.5M	50	1.423
mmLSTM (2 layers)	4.5M	50	**1.411**
Vanilla LSTM	4.5M	75	1.369
Vanilla LSTM (2 layers)	4.5M	75	1.355
mmLSTM	4.5M	75	1.367
mmLSTM (2 layers)	4.5M	75	**1.353**

Table 3: Training on partial sentences

However, on increasing the embedding dimension to 2 and increasing the hidden units from m to $2 \times m$, we find that the model is able to converge and achieve perfect scores on the WCPA metric for all $m \in \{4, 6, 8\}$.

7 Conclusion

In this project, we explored input dependent transitions for recurrent neural networks. Our novel architectures for doing input dependent transitions is able to perform as well as (and sometimes better than) ubiquitous architectures like RNNs and LSTMs on character and word-level language modelling tasks. We further evaluate variations of our models on character and word level modelling tasks and present various qualitative and quantitative findings.

References

[1] Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

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

[4] Mary Ann Marcinkiewicz Ann Taylor Mitchell P. Marcus, Beatrice Santorini. Treebank-3, 1999.

[5] Anonymous. Lstms can provably capture bounded hierarchical structure by building a stack.

[6] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1017–1024, 2011.

[7] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.

[8] Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Russ R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Advances in neural information processing systems*, pages 2856–2864, 2016.

[9] Mark W Goudreau, C Lee Giles, Srimat T Chakradhar, and Dong Chen. First-order versus second-order single-layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(3):511–513, 1994.

[10] Xiang Yu, Ngoc Thang Vu, and Jonas Kuhn. 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, 2019.