LiLT volume 16, issue (1) July 2018

Can Recurrent Neural Networks Learn Nested Recursion?

JEAN-PHILIPPE BERNARDY, *Centre for Linguistic Theory and Studies in Probability (CLASP), Department of Philosophy, Linguistics and Theory of Science, University of Gothenburg.*

Abstract

Context-free grammars (CFG) were one of the first formal tools used to model natural languages, and they remain relevant today as the basis of several frameworks. A key ingredient of CFG is the presence of *nested recursion*.

In this paper, we investigate experimentally the capability of several recurrent neural networks (RNNs) to learn nested recursion. More precisely, we measure an upper bound of their capability to do so, by simplifying the task to learning a generalized Dyck language, namely one composed of matching parentheses of various kinds. To do so, we present the RNNs with a set of random strings having a given maximum nesting depth and test its ability to predict the kind of closing parenthesis when facing deeper nested strings. We report mixed results: when generalizing to deeper nesting levels, the accuracy of standard RNNs is significantly higher than random, but still far from perfect. Additionally, we propose some non-standard stack-based models which can approach perfect accuracy, at the cost of robustness.

1 Introduction

In many settings, Recurrent Neural Networks (RNNs) act as generative language models. That is, given a prefix, they predict the likelihood of the next symbol. This probability is expressed as a continuous function of parameters (typically millions), which can be optimized (or trained) using gradient descent. By definition, RNNs are constructed by repeating a simple component, often called a cell, each cell taking care of one time-step. Additionally each cell passes some information to the cell at the next timestep. This information can be understood as an internal (so called "hidden") state, which can be used by the RNN to represent the input seen so far. RNNs can additionally be used for a variety of tasks such as input classification (e.g. sentiment analysis) or translation tasks.

The key ingredient to well-performing RNNs is to have each cell update the state by using a linear combination of the previous state and a new value. This linearity means that the current timestep depends linearly on symbols which occur much earlier in the input, and, in turn, the gradients of the parameters are also linearly dependent on the error in the output. RNNs need non-linear components as well, which allows them to model complex (non-linear) relations between inputs and outputs. Popular RNNs functioning on these principles include the long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997), and the gated recurrent unit (GRU) (Cho et al., 2014).

Thanks to their versatility, relative ease of training and ability to model long-term dependencies, RNNs have become the leading tool for natural language processing. For example, Karpathy et al. (2015) and Karpathy (2016) show that RNNs can model Shakespeare plays, Wikipedia articles, algebraic geometry papers and modules of the Linux source code. While experts in either domain will quickly recognize the generated samples as nonsense, they nonetheless broadly exhibit the syntactic structure that one would expect, both for small-scale (words, function names) and large-scale elements (e.g. stanzas, complex C-code statements).

Yet understanding how *precisely* they learn how to generalize is an open question. Even experienced computational linguists use words such as "amazing" or even "magic" to describe them, betraying that it remains mysterious how, simply by performing arithmetic operations, the RNN can effectively mimic human linguistic production (Karpathy, 2016). Unfortunately this combination of poor understanding and enthusiasm may lead less experienced researchers into believing that the capabilities of LSTM RNNs are limitless, and that, with enough data, they can model any language you throw at them. In particular, because Karpathy et al. shows that their models can generate text which appears to adhere to the rules of a context-free grammar, one might draw the conclusion that LSTM RNNs are in fact suitable to learn

context-free languages from a set of real-world examples.

But can they? Unfortunately, the generated samples only show display shallow recursion depths, so one cannot conclude that they model nested recursion – which is one of the defining features of the venerable Context-Free Grammars (CFG). (Whether or not nested recursion is a fundamental feature of natural languages has been the subject of a fierce debate. While we do not wish to take sides in that debate, we do acknowledge the importance of recursion when modeling syntax.) In this paper we measure an upper bound on the capability of various RNNs to learn nested recursion, by focusing on a simpler problem than that of learning context-free languages. More precisely, we investigate experimentally the ability of RNNs to learn a nested recursion in a language composed of balanced parentheses of various kinds. Thus, the results presented here apply directly to formal languages rather than natural languages.

The present work is part of an effort in the computational linguistic community to understand the capabilities of neural networks. We discuss previous work, including that of Linzen et al. (2016), in section 6. We test the LSTM, GRU, and a special purpose recurrent unit with a stack-like state (hereafter referred to as RUSS), and combinations thereof (section 2). We test the ability to learn both the plain Dyck language (with a kind pair of parentheses, see section 4) and a generalization of it, with several kinds of parentheses (section 5) — the precise definition of the language can be found in section 3.

2 Models

We experiment with LSTMs, GRUs and RUSSs, sometimes in combination. In this section we give the precise definition of each of the components.

2.1 LSTM

We use the variant of the LSTM RNN defined by the following equations:

$$
\begin{aligned}
v_t &= h_{t-1} \diamond x_t \\
f_t &= \sigma(W_f v_t + b_f) \\
i_t &= \sigma(W_i v_t + b_i) \\
o_t &= \sigma(W_o v_t + b_i) \\
\tilde{c}_t &= \sigma(W_c v_t + b_c) \\
c_t &= (f_t \odot c_{t-1}) + (i_t \odot \tilde{c}_t) \\
h_t &= (o_t \odot \mathsf{tanh}(c_t))
\end{aligned}
$$

In the above, the $\diamond$ operator denotes vector concatenation. The vector x_t is the input at time t and W and b are trainable parameters. The so-called hidden state h_t both serves as the output of the RNN and is passed to the next time-step. The dimension of each vector h_t is an hyperparameter and is called the

number of units of the LSTM, which we note by n in the following. The c_t vector, whose size equals the number of units, acts as the long-term memory of the RNN. Indeed at each timestep c_t is updated according to a linear combination of its previous value and a new vector $\tilde{c}_t$, according to the forget (f_t) and input (i_t) gates. This linear updates mean that backpropagating gradients are also linear functions. Thus, if the forget and input gates are properly set, a memory can persist as long as necessary in the time dimension, for the purpose of gradient descent.

Assuming that the size of every input vector x_t is m, then a plain LSTM has $4(n+m)n+4n$ scalar parameters. Thus, the number of parameters is a quadratic function of the unit size. This quadratic relation is what puts a bound on n. Indeed, the more scalar parameters a model has, the more it is consuming resource and the more it is prone to overfitting.

2.2 GRU

The gated recurrent unit (GRU) is another popular component of RNNs, invented by Cho et al. (2014). We use here the variant of the GRU defined by the following equations:

$$
\begin{aligned}
v_t &= h_{t-1} \diamond x_t \\
z_t &= \sigma(W_z v_t) \\
r_t &= \sigma(W_r v_t) \\
\tilde{h}_t &= \mathsf{tanh}(W_h((r_t \odot h_{t-1}) \diamond x_t)) \\
h_t &= ((1-z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

The GRU does not feature a specific memory state: the so-called hidden state h_t serves both as internal state (memory) and output. The number of parameters is $3(n+m)n$, which is less than the LSTM, but still quadratic in the number of units n.

2.3 RUSS

The creative contribution of this paper is a recurrent unit with a stack-like state hereafter abbreviated to RUSS, inspired by Grefenstette et al. (2015), which is defined as follows:

$$
\begin{aligned}
v_t &= s_{t-1}[0] \diamond x_t \\
a_t &= \mathsf{softmax}(W_a v_t + b_a) \\
s_t &= a_t \cdot [[x_t] \diamond s_{t-1}[0..d-2], s_{t-1}[1..d-1] \diamond [0], s_{t-1}] \\
h_t &= s_t[0]
\end{aligned}
$$

The internal state of this RNN is s_t, which can be understood as the memory, and is manipulated as a stack. It is a tensor of dimensions $d \times n$, where

d is the maximum depth of the stack. The expression $s_t[0]$ refers to the top of the stack at time-step t. We have a control gate (a_t) which drives the actions on the stack. The value of a_t is determined by learnable parameters W_a and b_a, together with the input x_t and the top of the stack at the previous time step. To compute the value of the stack at the current time step s_t, we combine linearly all the possible modifications of the stack (push, pop or no action) according to a_t — this linear combination is a dot product of such modifications with a_t.

The number of scalar parameters of the RUSS is $6m + 3$. Importantly, this number is *independent* of the depth of the stack. Consequently, when the depth of the stack is increased, the total memory size is increased as well; but the number of parameters remain the same. Further, the parameters themselves do not depend on the size of the stack. Consequently it is even possible to change the stack size between runs. (Say, train it using a small stack size and use it to evaluate new inputs using a bigger stack size.)

2.4 RNN configurations

In our experiments we use LSTM, GRU and RUSS in 11 different configurations, as follows.

1. one GRU layer with 40 units
2. one LSTM layer with 40 units
3. one GRU layer with 160 units
4. one LSTM layer with 160 units
5. one RUSS layer
6. two GRU layers with 40 units each
7. two LSTM layers with 40 units each
8. one LSTM layer followed by a GRU layer, with 40 units each
9. one GRU layer followed by a LSTM layer, with 40 units each
10. one LSTM layer followed by a RUSS layer, with 40 units each
11. one GRU layer followed by a RUSS layer, with 40 units each

The configurations 1 to 5 are used to assess the performance of each layer independently. In particular, testing the RUSS on its own is intended to confirm at a specific-purpose RNN can do the nested recursion task. The other configurations are meant to indicate how the components would perform in a complex RNN. When two layers are used, the output of the first layers is fed as input to the second layer, as is customary in multi-layer RNNs.

In each case, the input characters are translated to a vector of size 12 by means of a standard embedding layer.[1] Likewise, the output of either config-

[1] In preliminary experiments, we observed that using arbitrary embeddings instead of one-hot vectors impacted the accuracy positively in a statistically significant way (but without impact on

uration is fed to a dense layer with softmax activation, predicting the probability of the next character. Additionally, a dropout of 10% is applied equally on the input (x_t) and on the hidden state (h_t) of LSTMs and GRUs but not on the linear memory (c_t). No dropout is applied to RUSS layers. The loss is computed by summing the cross-entropy between predictions and examples in the test set, for each character. The parameters are optimized by using the Adam optimizer (Kingma and Ba, 2014) with a learning rate of 10^{-4} (a procedure which is standard at the time of writing).

3 Generalized-Dyck Language

Let us define the language D_P as the set of strings generated by the following context-free rules, dependent on a set P of matching parenthesis pairs.

$$\begin{aligned} S &::= E\# \\ E &::= oEc && \text{for every } (o,c) \in P \\ E &::= EE \\ E &::= \varepsilon \end{aligned}$$

Let us further call D_P^n the subset of D_P composed of strings of length $n+1$ (adding one to account for the termination character (#)), and $D_P^{(\leq n)}$ the the subset of D composed of strings of length $n+1$ or less.

$$\begin{aligned} D_P^n &= \{x \in D_P \mid \mathsf{length}(x) = n+1\} \\ D_P^{\leq n} &= \{x \in D_P \mid \mathsf{length}(x) \leq n+1\} \end{aligned}$$

4 Learning $D_{\{(,)\}}$

We start by checking that RNNs can learn recursion in a Dyck language with a single pair of parentheses. Thus we let P be $\{(,)\}$.

In this experiment, we train RNNs on strings in $D_P^{\leq 14}$ and test it on all strings in $D_P^{\leq 18}$. The training set is a subset of 512 strings picked at random in $D_P^{\leq 14}$. ($\#D_P^{\leq 14} = 625$ and $\#D_P^{\leq 18} = 6917$.) While it is customary to select a large training set and a small test set, we do the opposite here because 1. the language is small enough to test in its entirety and 2. we want to train our model on a data set which does not cover the whole language.

Recall that RNNs predict the probability of any given character, knowing the previous characters in the string. However, in a Dyck language with a single pair of possible parentheses, the next character is only restricted in the situation when all opening parentheses have been closed — in which case

the general trends). Because using embeddings has little to no impact on the complexity of the models, we stuck to this choice in the reported experiments.

only an opening parenthesis is possible. Thus, for testing our models, we only check that the model correctly predicts that, after being fed a complete string, no closing parenthesis can follow. Testing this condition suffices to check that the model would not generate an incorrect string of size 19 or longer.

4.1 Results

For this experiment, all our RNN configurations are able to reach an accuracy greater than 99%, in less than 5 epochs (the exact numbers vary from run to run, but the reported numbers are conservative).

4.2 Interpretation

One could be quick and conclude from this experiment alone that any of the tested RNN is able to learn nested recursion. However, we must realize that $D_{\{(,)\}}$ is an extremely simple language to model. If one has access to a counting device[2] one can model $D_{\{(,)\}}$ by simply using a counter initialized at 0, incrementing it on closing parentheses and decrementing on closing parentheses. When it is equal to 0, only an opening parenthesis is acceptable. It is hard to conclude anything about the suitability of RNNs to model most commonly-used language from this experiment — indeed any non-trivial context-free language will require matching several kinds of symbols, and thus will necessitate, instead of a counter, a full-fledged push-down automaton.

Furthermore, we can dismiss by a reduction process any experiment which would check that the language $a^n b^n$ can be learned: it is a subset of $D_{\{(,)\}}$ and thus can also be modeled using a simple counter. Likewise, from knowing how to learn $a^n b^n c^n$ using RNNs, not much insight can be gained on whether RNNs can learn context-sensitive languages in general.

5 Learning $D_{\{((,)),(+,-),([,]),(<,>),(\{,\})\}}$

Thus we set out to train RNNs on a language with several kinds of parentheses — a task which requires memory proportional to the depth of recursion. We choose P such that symbols in P are unique and $\#P = 5$. (Thus say $P = \{((,)),(+,-),([,]),(<,>),(\{,\})\}$). We train each RNN in our repertoire to predict, as before, the probability of any given character knowing the previous characters in the string. We note right away that one can never approach 100% accuracy for this task. Indeed, consider the prefix:

```
( ( [ (
```

The next character can either be a closing parenthesis of the right type (`]`) or any of the opening parentheses. Thus in this situation there are six possible choices, equally valid. In situations where no closing parenthesis is possible,

[2] Presumably all customary RNNs can count, as they are after all composed of units which can be activated at arbitrary floating point values, and updated using a linear combination of activations at the previous time-step.

one can instead expect the "end of string" character (#) instead. Thus the number of valid choices is always 6, and a perfect strategy would yield an accuracy of about 17%. In contrast, a random strategy would pick among all 11 possible characters and have an accuracy of about 9%.

In sum, the accuracy of predicting the next character is not a clear measure of a generative model to recognize this language. Thus we measure instead the accuracy of predicting the next character *assuming that it is a closing parenthesis in the test set*. Consequently, a perfect predictor will yield 100% accuracy (because only one kind of closing parenthesis is acceptable in any given situation), while a random strategy would yield about 20% accuracy (random choice of any closing parenthesis; and otherwise assuming that the end of string symbol does not introduce confusion.)

For each index i in the test input ($w \in X$), if w_i is a closing parenthesis then we test the prediction of the model. We count a success if the model assigns the highest probability to the expected kind of closing parenthesis and a failure otherwise. We will report the frequencies of success in function of i for each model and task. By reporting results in function of the time-index i, we are able to detect differences in modeling power which a compound measure would not reveal.

The training set that we use for this experiments is composed of 512×200 strings picked at random in $D_P^{\leq 10}$. ($\#D_P^{10} = 131250$). Thus our training set is

$$T = \mathsf{Sample}(512 \times 200, D_P^{\leq 10})$$

We train our models for 100 epochs; yielding 100 versions of the parameters values, and keep those which yield best accuracy on the test data. Indeed, after a while, models may tune-in to the training data, at the expense of validation accuracy (a phenomenon called "over-fitting"). Thus our method gives enough time for most models to converge (see specific remarks below), while minimizing potential over-fitting effects.

5.1 Subtask A: Long-Term Dependency

For this subtask, the test set is composed of 512×20 strings picked at random in the set generated by sentences of the form: $s_1 s_2$ or $o s_1 s_2 c$, where s_1 and s_2 are picked randomly among strings of size 8, and (o, c) is a matching parentheses pair. Formally:

$$\begin{aligned} X = \mathsf{Sample}(512 \times 20, \{ o s_1 s_2 c \,|(o,c) \in P, \\ s_1 \in \mathsf{Sample}(1000, D_P^8), \\ s_2 \in \mathsf{Sample}(1000, D_P^8)\}) \end{aligned}$$

This task is designed to check that the RNN can capture long-term de-

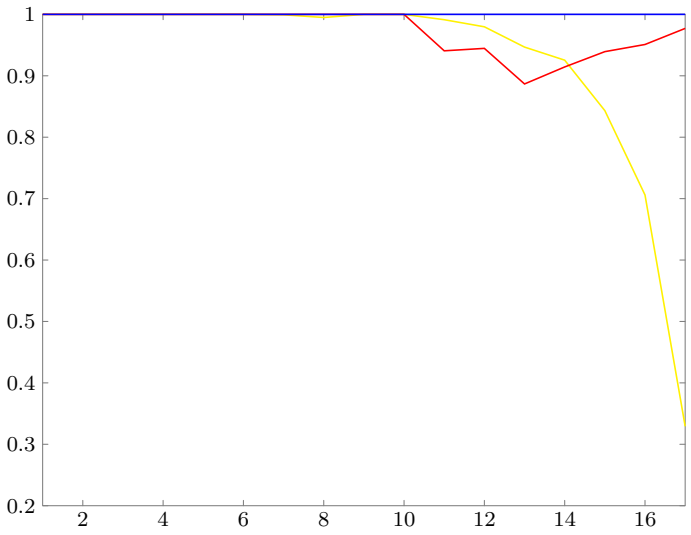

FIGURE 1 Results for long-distance dependency task for LSTM (red), GRU (yellow) and RUSS (blue).

pendencies, by generalizing to longer strings. Indeed, while both s_1 and s_2 are probably in the training set, the complete string is not: its total length is 1+8+8+1 = 18. Furthermore, correct predictions for the last character require correctly matching it with either the first character of the input or the character at position 9. Performing this matching could be done with counting alone, but *only if the model optimizes for guessing the last character*. On the contrary, in our experiment, models are trained to predict any character, so one can expect that to predict any closing parenthesis character the model has to remember every unclosed parenthesis seen so far. Unfortunately, one never knows how a model really behaves before measuring it, so we report the accuracy for every character (and perform further experiments described in the next section).

Results

We report only the results for one run, because there is no qualitative difference from run to run. The results are shown in figure 1. The RUSS performs expected given its design, with no error whatsoever. The LSTM shows near perfect accuracy for all known strings (up to length 10). For longer strings its accuracy dips slightly, but remains excellent, never going below 88%. The GRU performs better than the LSTM up to position 14, but then suffers a sharp drop in performance. It ends up completely failing the test at the position which really matters for this task (the last one), with an accuracy hardly

above that of a random strategy.

5.2 Subtask B: New Depths

For this subtask, we keep the same training sets, but the test set is composed of sentences of the form: o^*Sc^*, where S is picked randomly in D_P^{10}, o^* is a random sequence of 5 opening parentheses and c^* is the corresponding closing sequence. Formally:

$$X = \mathsf{Sample}(512 \times 20, \{o^* s_1 c^* \mid (o^*, c^*) \in (P^5)^T, \\ s_1 \in \mathsf{Sample}(1000, D_P^{10})\})$$

Similarly to the previous task, while s_1 is likely to be found in the training set, the complete input is guaranteed not to be. This test is engineered to check that the model correctly generalizes to strings with deeper nesting.

Results

For every run, a new test set is generated. When the results vary significantly from run to run, we report the results for several runs, thereby giving an indication of the variability.[3] The results are reported in figures 2 to 7. We want to avoid biasing the reader towards our interpretation of the results as far as possible, and thus we will give our interpretation only at the end of the section.

Again we break down the accuracy per position of closing parentheses. The predictive power at different positions correspond to different modeling capabilities. First, keep in mind that there are no closing parentheses in the test set for indices smaller than 6, and thus we do not report any result for those positions. Second, the positions 14 and above always have a closing parenthesis. Indeed at position 14 we have the closing parenthesis corresponding to the opening one at position 5. Thus:

- Positions 6 to 10 correspond to depths seen in the input set.
- At positions 10 to 13 the prefixes are not seen in the training set, but the closing parenthesis matches with a parenthesis which is at a position at 6 or greater. Thus, in principle, to predict those it is not necessary to take into account the prefix before position 6, even though it constitutes a factor of confusion.
- Parentheses at positions 14 and greater can only be predicted by considering the nesting levels which are unseen in the training set.

[3] Many neural network models are trained on a fixed dataset, for example harvested from a natural language corpus. The convention is to split this fixed data set in three part (dev/test/train), so that while constructing the model, the test set is never used. This means that hyper parameters cannot be specifically adjusted to perform best for this specific test set. In our case, because new data is generated for every run, it is impossible to adjust the models to a specific test test.

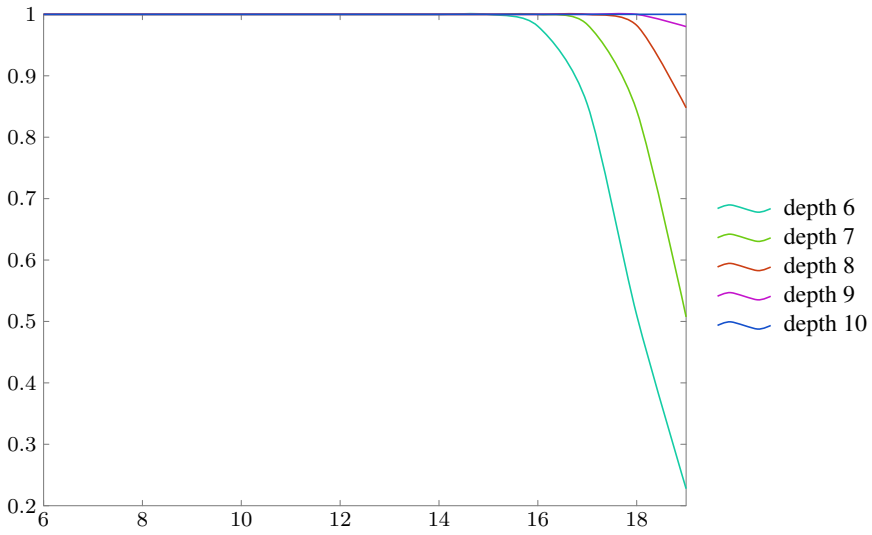

FIGURE 2 Results for a single RUSS layer.

We first report that the RUSS, which was after all specially designed for the task, perfectly generalizes to deeper recursion (see fig 2). We tested the RUSS with several depths for the stack. This set of tests confirmed that by limiting the maximum stack depth, one prevents accurate prediction of the last character in strings with nesting that is too deep. Additionally, we confirmed experimentally that a depth of 10 is enough to accurately model our test set. In later experiments, this depth was used in other configurations which involve the RUSS.

For the remaining RNN configurations, we ran the experiment five times and report the results for each run, thereby giving an indication of the variability of results. In fig. 3 we show the results for a single layer of GRU or LSTM. The GRU shows performance significantly above the random strategy, but with rapid decline as distance from the corresponding opening parenthesis increases. For the longest such distance, its performance is hardly better than that of a random guess. We observed that the GRU converges quickly: its best performance is often obtained after less than 20 epochs.

The LSTM also shows performance significantly above the random strategy, around 60% accuracy for index 15, where it performs the worst. One observes an expected drop in performance for depths which are not seen in the test data. However, the LSTM is able to make more accurate predictions for indices which are close to the end of the string (close to 90% accuracy).

To obtain the results shown in figure 4, we have increased the number

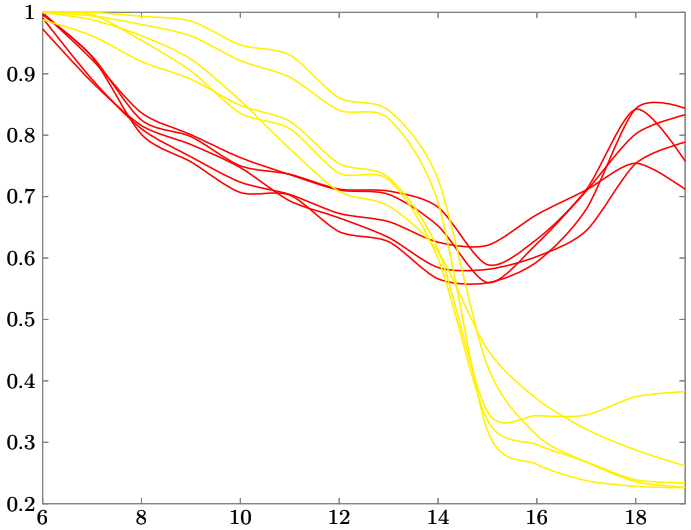

FIGURE 3 Results for LSTM, 40 units (red) and GRU (yellow)

of units to 160. In this situation the models reach their peak performance about as quickly as before. The LSTM displays a smoother accuracy curve through time-steps, but the general characteristic of dipping around position 15 remains. By and large, the same can be said about the GRU. The main difference in this situation is that it does not perform as well for early positions. Consequently the LSTM dominates the GRU across the board.

In figure 5 we show the results for a double layers of either GRU or LSTM, each with 40 units. The double LSTM model converges after around 70 epochs. It exhibits roughly the same performance as the single LSTM with the same number. The effect of better accuracy for the last few positions is amplified. The double GRU model converges around 50 epochs, and exhibits roughly the same pattern as the single-layer GRU RNN, with better accuracy, but an even sharper drop around position 14.

In figure 6 we show the results for models combining one GRU and one LSTM, with 40 units. These models generally exhibit behavior similar to that of their first layer alone, but with their specific characteristics (positive or negative) somewhat smoothed out.

In figure 7 we show the results of combining the RUSS with either GRU or LSTM. In these configurations the RUSS is provided with data exhibiting more complex patterns. Thus they correspond more to how the RUSS would perform for not so simple languages, where an LSTM or RNN would be needed to pre-process the data before it can be consumed by the RUSS.

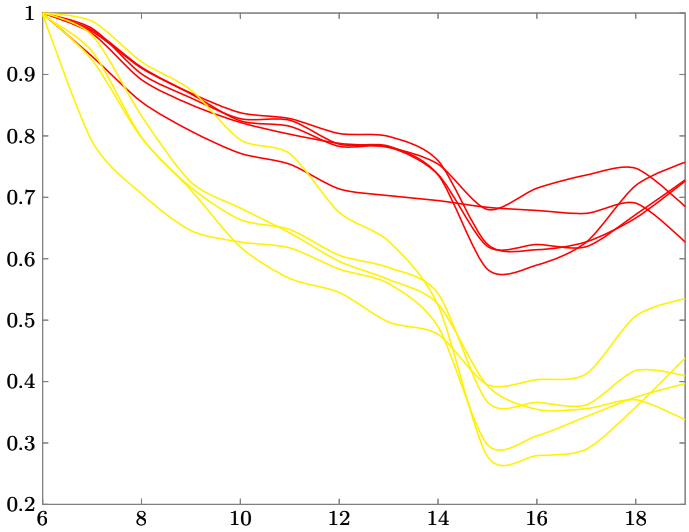

FIGURE 4 Results for LSTM, 160 units (red) and GRU (yellow)

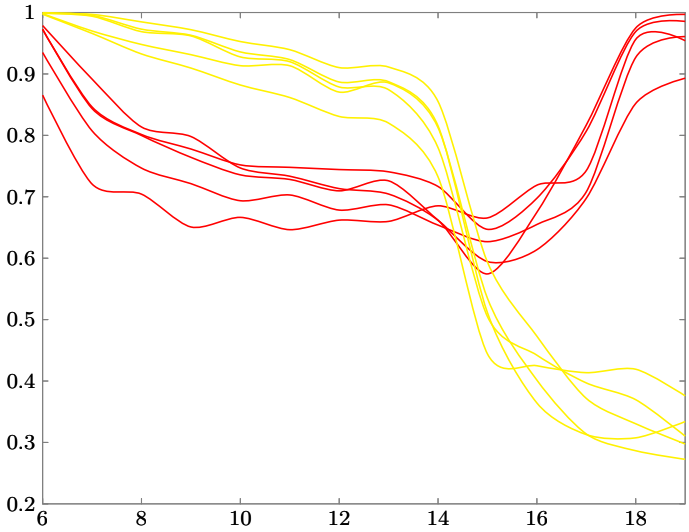

FIGURE 5 Results for two layers of LSTM (red) and two layers of GRU (yellow)

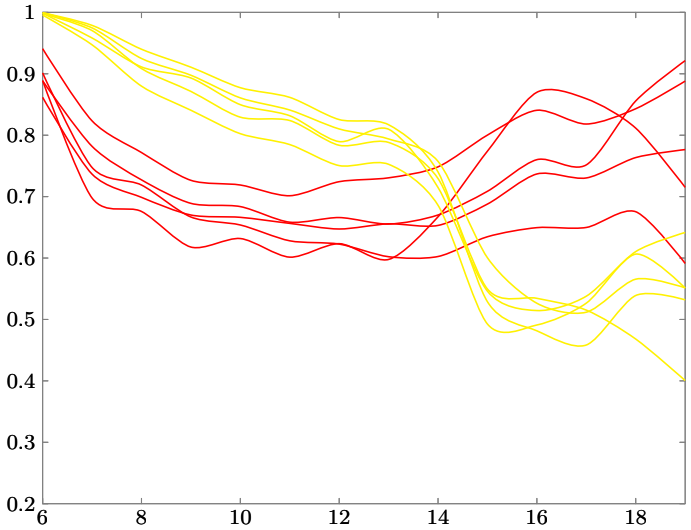

FIGURE 6 Results for LSTM followed by GRU (red) and GRU followed by LSTM (yellow)

In these configurations, the model was often still improving even after 100 epochs.

Adding the RUSS to the LSTM improves its accuracy both for short prefixes and for positions close to the end of the string. However it sometimes worsens slightly its accuracy for positions 14 to 15. The GRU and RUSS combined model displays great variability from run to run, ranging from perfect accuracy to not much better than the single GRU.

From these results, we note the following salient points:

1. The single LSTM exhibits the unusual feature of better accuracy for the longest distances than for mid-range ones. Our interpretation is that the LSTM learned to specially match the outermost parentheses. How is that possible? First, remember that the LSTM can develop a counting mechanism (section 4), which can thus predict when the closing parenthesis corresponding to the outermost opening parenthesis is expected. Assuming that such a mechanism is available, it suffices to combine it is with another mechanism to remember the first character of the input to correctly predict the last closing parenthesis.
2. Layering two RNNs of the same kind tended to yield small improvements and reinforce the general trends exhibited by their single-layer counterparts.
3. Increasing 4-fold the number of units tended to yield small improve-

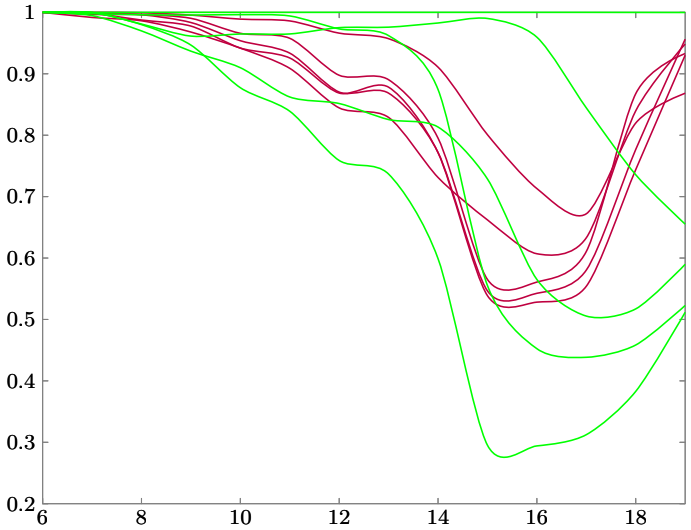

FIGURE 7 Results for LSTM+RUSS (purple) and GRU+RUSS (green)

ments, and smooth out the general trends exhibited by their single-layer counterparts.

4. The behavior of a heterogeneously layered RNN seems to be primarily following the behavior of its first layer. However, the RUSS can have a significant beneficial impact on the behavior of the compound, when employed as a second layer.

6 Related work

Das et al. (1992) proposed the first neural network architecture with a differentiable stack, and used it to learn context-free language. However they considered only very simple synthetic languages: $a^n b^n$, $a^n b^n c b^m a^m$, $a^{n+m} b^n c^m$ and an undefined "parenthesis" language (which they report is easier to learn than $a^n b^n$). Such languages can be modeled by counting only, thus, as explained in section 4 we dismiss them as unrepresentative of the context-free (or worse context-sensitive) language classes for the purposes of modeling using RNNs. Gers and Schmidhuber (2001) showed that the LSTM could learn such simple synthetic languages. (The special stack structure is not needed; the general-purpose LSTM works well already.) This counting behavior has been studied in depth for various RNNs architectures by Weiss et al. (2018).

Elman (1991) studied the learnability of a context-free language with more structure, involving central embedding and thus long-distance dependencies. However he did not train the network to generalise from shallow to

deeper levels of recursion. This limited focus persists in the followup work of Cartling (2008).

Our main source of inspiration for the RUSS is the work of Grefenstette et al. (2015), who use a differentiable stack for transduction. Even though language modeling is not explicit in their work, their transduction uses an RNN and thus has implicit language modeling aspects. The main methodological difference from this work is that they do not attempt to generalize to deeper nested recursion. The main technical difference is that we have simplified the differentiable stack and packaged it as a reusable RNN layer.

Joulin and Mikolov (2015) also use stacks for modeling purposes. Their domain of applications is algorithmic patterns, such as binary addition and memorization of a string. However their stack is different from ours, in that all the units in their stack state are controlled independently from each other. In effect we (and Grefenstette et al.) have one stack with n units, whereas they have n stacks of 1 unit each. A consequence of their choice is that the number of scalar parameters is larger by a factor of n. In other words, the model has the possibility (and the burden) to learn lots of different push/pop patterns.

The modeling capabilities of the LSTM in natural language have been investigated by Linzen et al. (2016), Bernardy and Lappin (2017). Specifically, they focus on the *capability* of the LSTM to model long-distance dependencies, by measuring the accuracy of the LSTM when predicting agreement of verb with subject, in English. Their test did not measure the amount of nesting, but only the number of intervening nouns with mismatching features (which they call attractors). They found that for no attractors, the LSTM scored near perfectly, but for 5 or more attractors, it did not perform much better than the baseline. The good accuracy for less than 5 attractors was corroborated by Gulordava et al. (2018), for other languages, including Italian, Hebrew and Russian.

Yogatama et al. (2018) have investigated the effect of various memory architectures, including stacks, on the long-distance dependency task, for up to 5 attractors. They found that the best-performing model uses a stack with the ability to perform several `pop` operations per time step.

Unfortunately none of the above tests can be used to conclude significantly that RNNs can learn nested hierarchical structures. Indeed, in natural data, the relationship between nesting depth and number of attractor is exponential. Consequently, the nesting depth grows on average logarithmically with the nesting depth. In turn, by limiting their experiment to a small fixed number of attractors, one cannot draw any conclusion on nested recursion from their results, despite what the title used by Gulordava et al. (2018) suggests.

Furthermore, even when considering the effect of only the number of attractors on predicting agreement, English data will be influenced by semantic factors (multiple syntactic interpretations are possible if semantics do not dis-

ambiguate). Additionally, in English the syntactic marker of nesting are not always present. In contrast, here we have established conclusively that, in the case of our synthetic language, the LSTM suffers from no confusion by intervening symbols with mismatching features. Indeed, in our experiment (see sec. 5.1) the accuracy of predicting the closing character is very high, regardless of the distance to the corresponding opening parenthesis. (And, in most cases, intervening characters would be composed of attractors). One can however not say the same thing about the GRU, which fails to generalise to unseen longer strings.

7 Conclusion

7.1 Learnability of depth recursion

We have found out that, by and large, the LSTM is capable of generalization to new depths, with a respectable, yet not nearly perfect accuracy. On the contrary, the GRU, as defined in sec. 2.2, is not suitable for such generalization. While the accuracy of the LSTM may be suitable for some tasks, with an error rate approaching 40% on nesting levels unseen in the training set, it cannot be considered to reliably model CFGs. Thus, if one desires higher accuracy on predicting agreement in unseen nesting levels, special-purpose components such as the RUSS must be employed.

7.2 Suitability of RNN variants

The discrepancy between LSTM and GRU goes contrary to neural-network folklore, which propagates the idea that they are roughly interchangeable components. Indeed, many tests of RNNs focus on some compound measure of accuracy or perplexity, which is roughly equivalent for LSTM or GRU, even in our tests. But such a compound measure hides variation in behavior through time-steps. By focusing precisely on this variation, our test methodology reveals sharp differences in the behavior of various RNNs and their combinations.

In sum, our experiment suggests that using the GRU is a better idea if only a short-distance generalization is desired, whereas the LSTM is preferable for learning dependencies which need to generalize over long distances and new depths. The GRU could also be a sensible choice if little training time is available: it tends to find its optimal parameters more quickly than the LSTM. Unfortunately, at the time of writing, the state-of-the-art analysis of neural models is mostly experimental and thus we cannot offer a theoretical explanation for the difference between LSTM and GRU, but we can make educated guesses. The difference could be explained by either 1. the larger number of parameters of the LSTM or 2. the architectural differences, mainly having a separate memory and output. Guess (1.) is refuted by our

experiments (because increasing the number of parameters does not make a qualitative difference). Guess (2.) is compatible with our experiments, but corroborating evidence would be necessary before accepting it as fact.

Besides, we note that the popular technique of layering several instances of the same type of RNN did not prove fruitful for our main experiment: it only yielded incremental improvements. More interesting behaviors were obtained by combining different kinds of RNN. In particular, incorporating special-purpose RNN layers such as the RUSS can have beneficial effects.

7.3 Deep recursion in natural language

Karlsson (2007) has observed that deep recursion does not occur in practice in natural language input. Therefore, it is possible to train LSTM RNNs on all nesting levels which do occur in practice — and thus the RUSS is not a necessary component in such applications. It is intriguing that the LSTM thus appears to be a better model of human ability than computer models with explicit stacks, such as the RUSS or the traditional push-down automata. This observation is compatible with the hypothesis of Karlsson (2010), namely that the limits on recursion depth in natural language is caused by memory limitations. Indeed, the LSTM has finite memory while the memory of the RUSS can be expanded dynamically (section 2.3). Regardless, we are careful not to claim any similarity of structure between the LSTM and any human organ; such claims would be outside our domain of expertise.

Relevance of this study In the light of the limited depth of recursion found in natural language corpora, one can wonder if this study is relevant to such contexts? One area where generalization to unseen depths can be useful is argumentative dialogue. Indeed, in a such a dialogue one can quote a previous interaction and embed it in one's own production, thereby increasing nesting depth. Dealing with this situation is critical if the system is intended to cope with adversarial users — who can at any point push the system to its limits.

Another area where nested recursion occurs is in works which are bound by formal rules and whose author is using external tools[4] as unbounded memory. Examples would include logical propositions, program source code, and perhaps even legal texts and works in certain musical genres.

7.4 Applications of RUSS

Because of its potentially unbounded memory, the RUSS cannot be recommended if one desires to model human language production and recognition. Regardless, it has several characteristics which may make it suitable for certain practical applications. First, its input-output structure is the same as that of standard RNNs. Consequently, it can be combined with other RNNs in the

[4] classically pen and paper, but computers in a modern setting

usual way, namely by layering them. The RUSS is thus a component which can be easily incorporated in any RNN-based model. Second, experiments indicate that even for tasks which do not require generalization to arbitrary depth, a combination of RUSS and LSTM performs well. Third, many applications may want to go beyond basic human capabilities and thus desire depth generalization, for example in the domains listed in the previous subsection.

Acknowledgments The research reported in this paper was supported by a grant from the Swedish Research Council (VR project 2014-39) for the establishment of the Centre for Linguistic Theory and Studies in Probability (CLASP) at the University of Gothenburg.

We thank Samuel Bowman, Shalom Lappin, Aleksandre Maskharashvili and Charalambos Themistocleous and anonymous reviewers for useful feedback on earlier versions of this paper. Additionally the experimental setup was informed by discussions with Shalom Lappin.

References

Bernardy, Jean-Philippe and Shalom Lappin. 2017. Using deep neural networks to learn syntactic agreement. *Linguistic Issues In Language Technology* 15(2):15.

Cartling, Bo. 2008. On the implicit acquisition of a context-free grammar by a simple recurrent neural network. *Neurocomputing* 71(7):1527–1537.

Cho, Kyunghyun, Bart van Merriënboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Doha, Qatar: Association for Computational Linguistics.

Das, Sreerupa, 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. Indiana University*, page 14.

Elman, Jeffrey L. 1991. Distributed representations, simple recurrent networks, and grammatical structure. *Machine learning* 7(2-3):195–225.

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

Grefenstette, Edward, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 1828–1836. Cambridge, MA, USA: MIT Press.

Gulordava, Kristina, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. 2018. Colorless green recurrent networks dream hierarchically. *arXiv preprint arXiv:1803.11138* .

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

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

Karlsson, Fred. 2007. Constraints on multiple center-embedding of clauses. *Journal of Linguistics* 43(2):365–392.

Karlsson, Fred. 2010. Working memory constraints on multiple center-embedding. In *Proceedings of the Cognitive Science Society*, vol. 32.

Karpathy, Andrej. 2016. The unreasonable effectiveness of recurrent neural networks http://karpathy.github.io/2015/05/21/rnn-effectiveness.

Karpathy, Andrej, Justin Johnson, and Li Fei-Fei. 2015. Visualizing and understanding recurrent networks. In *Proceedings of ICLR 2015*.

Kingma, Diederik P. and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980.

Linzen, Tal, Emmanuel Dupoux, and Yoav Golberg. 2016. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association of Computational Linguistics* 4:521–535.

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

Yogatama, Dani, Yishu Miao, Gabor Melis, Wang Ling, Adhiguna Kuncoro, Chris Dyer, and Phil Blunsom. 2018. Memory architectures in recurrent neural network language models. *Proc. ICLR* .