On the Practical Computational Power of Finite Precision RNNs for Language Recognition

Gail Weiss
Technion, Israel

Yoav Goldberg
Bar-Ilan University, Israel

Eran Yahav
Technion, Israel

{sgailw,yahave}@cs.technion.ac.il
yogo@cs.biu.ac.il

Abstract

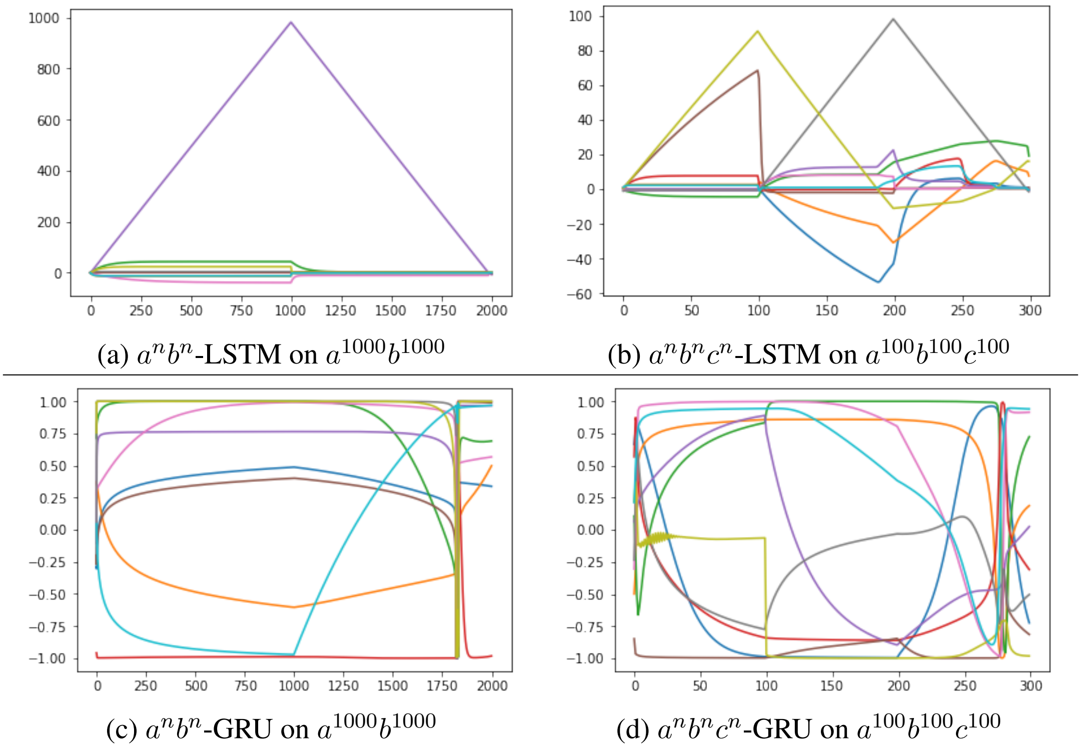
While Recurrent Neural Networks (RNNs) are famously known to be Turing complete, this relies on infinite precision in the states and unbounded computation time. We consider the case of RNNs with finite precision whose computation time is linear in the input length. Under these limitations, we show that different RNN variants have different computational power. In particular, we show that the LSTM and the Elman-RNN with ReLU activation are strictly stronger than the RNN with a squashing activation and the GRU. This is achieved because LSTMs and ReLU-RNNs can easily implement counting behavior. We show empirically that the LSTM does indeed learn to effectively use the counting mechanism.

1 Introduction

Recurrent Neural Network (RNNs) emerge as very strong learners of sequential data. A famous result by Siegelmann and Sontag (1992; 1994), and its extension in (Siegelmann, 1999), demonstrates that an Elman-RNN (Elman, 1990) with a sigmoid activation function, rational weights and infinite precision states can simulate a Turing-machine in real-time, making RNNs Turing-complete. Recently, Chen et al (2017) extended the result to the ReLU activation function. However, these constructions (a) assume reading the entire input into the RNN state and only then performing the computation, using *unbounded time*; and (b) rely on having *infinite precision* in the network states. As argued by Chen et al (2017), this is not the model of RNN computation used in NLP applications. Instead, RNNs are often used by feeding an input sequence into the RNN one item at a time, each immediately returning a state-vector that corresponds to a prefix of the sequence and which can be passed as input for a subsequent feed-forward prediction network operating in constant time. The amount of tape used by a Turing machine under this restriction is linear in the input length, reducing its power to recognition of context-sensitive language. More importantly, computation is often performed on GPUs with 32bit floating point computation, and there is increasing evidence that competitive performance can be achieved also for quantized networks with 4-bit weights or fixed-point arithmetics (Hubara et al., 2016). The construction of (Siegelmann, 1999) implements pushing 0 into a binary stack by the operation $g \leftarrow g/4 + 1/4$. This allows pushing roughly 15 zeros before reaching the limit of the 32bit floating point precision. Finally, RNN solutions that rely on carefully orchestrated mathematical constructions are unlikely to be found using backpropagation-based training.

In this work we restrict ourselves to *input-bound recurrent neural networks with finite-precision states (IBFP-RNN)*, trained using back-propagation. This class of networks is likely to coincide with the networks one can expect to obtain when training RNNs for NLP applications. An IBFP Elman-RNN is finite state. But what about other RNN variants? In particular, we consider the Elman RNN (SRNN) (Elman, 1990) with squashing and with ReLU activations, the Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) and the Gated Recurrent Unit (GRU) (Cho et al., 2014; Chung et al., 2014).

The common wisdom is that the LSTM and GRU introduce additional *gating components* that handle the vanishing gradients problem of training SRNNs, thus stabilizing training and making it more robust. The LSTM and GRU are often considered as almost equivalent variants of each other.



(a) a^nb^n-LSTM on $a^{1000}b^{1000}$

(b) $a^nb^nc^n$-LSTM on $a^{100}b^{100}c^{100}$

(c) a^nb^n-GRU on $a^{1000}b^{1000}$

(d) $a^nb^nc^n$-GRU on $a^{100}b^{100}c^{100}$

Figure 1: Activations — c for LSTM and h for GRU — for networks trained on a^nb^n and $a^nb^nc^n$. The LSTM has clearly learned to use an explicit counting mechanism, in contrast with the GRU.

We show that in the input-bound, finite-precision case, there is a real difference between the computational capacities of the LSTM and the GRU: the LSTM can easily perform unbounded counting, while the GRU (and the SRNN) cannot. This makes the LSTM a variant of a k-counter machine (Fischer et al., 1968), while the GRU remains finite-state. Interestingly, the SRNN with ReLU activation followed by an MLP classifier also has power similar to a k-counter machine.

These results suggest there is a class of formal languages that can be recognized by LSTMs but not by GRUs. In section 5, we demonstrate that for at least two such languages, the LSTM manages to learn the desired concept classes using back-propagation, while using the hypothesized control structure. Figure 1 shows the activations of 10-d LSTM and GRU trained to recognize the languages a^nb^n and $a^nb^nc^n$. It is clear that the LSTM learned to dedicate specific dimensions for counting, in contrast to the GRU.[1]

2 The RNN Models

An RNN is a parameterized function R that takes as input an input vector x_t and a state vector h_{t-1} and returns a state vector h_t:

$$h_t = R(x_t, h_{t-1}) \tag{1}$$

The RNN is applied to a sequence $x_1, ..., x_n$ by starting with an initial vector h_0 (often the 0 vector) and applying R repeatedly according to equation (1). Let Σ be an input vocabulary (alphabet), and assume a mapping E from every vocabulary item to a vector x (achieved through a 1-hot encoding, an embedding layer, or some other means). Let $RNN(x_1, ..., x_n)$ denote the state vector h resulting from the application of R to the sequence $E(x_1), ..., E(x_n)$. An *RNN recognizer* (or *RNN acceptor*) has an additional function f mapping states h to $0, 1$. Typically, f is a log-linear classifier or multi-layer perceptron. We say that *an RNN recognizes a language* $L \subseteq \Sigma^*$ if $f(RNN(w))$ returns 1 for all and only words $w = x_1, ..., x_n \in L$.

Elman-RNN (SRNN) In the Elman-RNN (Elman, 1990), also called the Simple RNN (SRNN), the function R takes the form of an affine transform followed by a tanh nonlinearity:

[1] Is the ability to perform unbounded counting relevant to "real world" NLP tasks? In some cases it might be. For example, processing linearized parse trees (Vinyals et al., 2015; Choe and Charniak, 2016; Aharoni and Goldberg, 2017) requires counting brackets and nesting levels. Indeed, previous works that process linearized parse trees report using LSTMs and not GRUs for this purpose. Our work here suggests that this may not be a coincidence.

$$h_t = \tanh(Wx_t + Uh_{t-1} + b) \tag{2}$$

Elman-RNNs are known to be at-least finite-state. Siegelmann (1996) proved that the tanh can be replaced by any other squashing function without sacrificing computational power.

IRNN The IRNN model, explored by (Le et al., 2015), replaces the tanh activation with a non-squashing ReLU:

$$h_t = max(0, (Wx_t + Uh_{t-1} + b)) \tag{3}$$

The computational power of such RNNs (given infinite precision) is explored in (Chen et al., 2017).

Gated Recurrent Unit (GRU) In the GRU (Cho et al., 2014), the function R incorporates a *gating mechanism*, taking the form:

$$\begin{aligned}
z_t &= \sigma(W^z x_t + U^z h_{t-1} + b^z) \quad &(4)\\
r_t &= \sigma(W^r x_t + U^r h_{t-1} + b^r) \quad &(5)\\
\tilde{h}_t &= \tanh(W^h x_t + U^h(r_t \circ h_{t-1}) + b^h) &(6)\\
h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \quad &(7)
\end{aligned}$$

Where σ is the sigmoid function and $\circ$ is the Hadamard product (element-wise product).

Long Short Term Memory (LSTM) In the LSTM (Hochreiter and Schmidhuber, 1997), R uses a different gating component configuration:

$$\begin{aligned}
f_t &= \sigma(W^f x_t + U^f h_{t-1} + b^f) \quad &(8)\\
i_t &= \sigma(W^i x_t + U^i h_{t-1} + b^i) \quad &(9)\\
o_t &= \sigma(W^o x_t + U^o h_{t-1} + b^o) \quad &(10)\\
\tilde{c}_t &= \tanh(W^c x_t + U^c h_{t-1} + b^c) \quad &(11)\\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \quad &(12)\\
h_t &= o_t \circ g(c_t) \quad &(13)
\end{aligned}$$

where g can be either tanh or the identity.

Equivalences The GRU and LSTM are at least as strong as the SRNN: by setting the gates of the GRU to $z_t = 0$ and $r_t = 1$ we obtain the SRNN computation. Similarly by setting the LSTM gates to $i_t = 1$,$o_t = 1$, and $f_t = 0$. This is easily achieved by setting the matrices W and U to 0, and the biases b to the (constant) desired gate values.

Thus, all the above RNNs can recognize finite-state languages.

3 Power of Counting

Power beyond finite state can be obtained by introducing counters. Counting languages and k-counter machines are discussed in depth in (Fischer et al., 1968). When unbounded computation is allowed, a 2-counter machine has Turing power. However, for computation bound by input length (real-time) there is a more interesting hierarchy. In particular, real-time counting languages cut across the traditional Chomsky hierarchy: real-time k-counter machines can recognize at least one context-free language ($a^n b^n$), and at least one context-sensitive one ($a^n b^n c^n$). However, they *cannot* recognize the context free language given by the grammar $S \rightarrow x|aSa|bSb$ (palindromes).

SKCM For our purposes, we consider a simplified variant of k-counter machines (SKCM). A counter is a device which can be incremented by a fixed amount (INC), decremented by a fixed amount (DEC) or compared to 0 (COMP0). Informally,[2] an SKCM is a finite-state automaton extended with k counters, where at each step of the computation each counter can be incremented, decremented or ignored in an input-dependent way, and state-transitions and accept/reject decisions can inspect the counters' states using COMP0. The results for the three languages discussed above hold for the SKCM variant as well, with proofs provided in Appendix A.

4 RNNs as SKCMs

In what follows, we consider the effect on the state-update equations on a single dimension, $h_t[j]$. We omit the index $[j]$ for readability.

LSTM The LSTM acts as an SKCM by designating k dimensions of the memory cell c_t as counters. In non-counting steps, set $i_t = 0, f_t = 1$ through equations (8-9). In counting steps, the counter direction (+1 or -1) is set in $\tilde{c}_t$ (equation 11) based on the input x_t and state h_{t-1}. The counting itself is performed in equation (12), after setting $i_t = f_t = 1$. The counter can be reset to 0 by setting $i_t = f_t = 0$.

Finally, the counter values are exposed through $h_t = o_t g(c_t)$, making it trivial to compare the counter's value to 0.[3]

[2] Formal definition is given in Appendix A.

[3] Some further remarks on the LSTM: LSTM supports both increment and decrement in a single dimension. The counting dimensions in c_t are exposed through a function

We note that this implementation of the SKCM operations is achieved by saturating the activations to their boundaries, making it relatively easy to reach and maintain in practice.

SRNN The finite-precision SRNN cannot designate unbounded counting dimensions.

The SRNN update equation is:

$$h_t = \tanh(Wx + Uh_{t-1} + b)$$

$$h_t[i] = \tanh(\sum_{j=1}^{d_x} W_{ij}x[j] + \sum_{j=1}^{d_h} U_{ij}h_{t-1}[j] + b[i])$$

By properly setting U and W, one can get certain dimensions of h to update according to the value of x, by $h_t[i] = \tanh(h_{t-1}[i] + w_i x + b[i])$. However, this counting behavior is within a $\tanh$ activation. Theoretically, this means unbounded counting cannot be achieved without infinite precision. Practically, this makes the counting behavior inherently unstable, and bounded to a relatively narrow region. While the network could adapt to set w to be small enough such that counting works for the needed range seen in training without overflowing the $\tanh$, attempting to count to larger n will quickly leave this safe region and diverge.

IRNN Finite-precision IRNNs can perform unbounded counting conditioned on input symbols. This requires representing each counter as two dimensions, and implementing Inc as incrementing one dimension, Dec as incrementing the other, and Comp0 as comparing their difference to 0. Indeed, Appendix A in (Chen et al., 2017) provides concrete IRNNs for recognizing the languages $a^n b^n$ and $a^n b^n c^n$. This makes IBFP-RNN with ReLU activation more powerful than IBFP-RNN with a squashing activation. Practically, ReLU-activated RNNs are known to be notoriously hard to train because of the exploding gradient problem.

GRU Finite-precision GRUs cannot implement unbounded counting on a given dimension. The $\tanh$ in equation (6) combined with the interpolation (tying z_t and $1 - z_t$) in equation (7) restricts the range of values in h to between -1 and 1, precluding unbounded counting with finite precision. Practically, the GRU can learn to count up to some bound m seen in training, but will not generalize well beyond that.[4] Moreover, simulating forms of counting behavior in equation (7) require consistently setting the gates z_t, r_t and the proposal $\tilde{h}_t$ to precise, non-saturated values, making it much harder to find and maintain stable solutions.

Summary We show that LSTM and IRNN can implement unbounded counting in dedicated counting dimensions, while the GRU and SRNN cannot. This makes the LSTM and IRNN at least as strong as SKCMs, and strictly stronger than the SRNN and the GRU.[5]

5 Experimental Results

Can the LSTM indeed learn to behave as a k-counter machine when trained using backpropagation? We show empirically that:

1. LSTMs can be trained to recognize $a^n b^n$ and $a^n b^n c^n$.

2. These LSTMs generalize to much higher n than seen in the training set (though not infinitely so).

3. The trained LSTM learn to use the per-dimension counting mechanism.

4. The GRU can also be trained to recognize $a^n b^n$ and $a^n b^n c^n$, but they do not have clear counting dimensions, and they generalize to much smaller n than the LSTMs, often failing to generalize correctly even for n within their training domain.

g. For both $g(x) = x$ and $g(x) = \tanh(x)$, it is trivial to do compare 0. Another operation of interest is comparing two counters (for example, checking the difference between them). This cannot be reliably achieved with $g(x) = \tanh(x)$, due to the non-linearity and saturation properties of the $\tanh$ function, but is possible in the $g(x) = x$ case. LSTM can also easily set the value of a counter to 0 in one step. The ability to set the counter to 0 gives slightly more power for real-time recognition, as discussed by Fischer et al. (1968).

Relation to known architectural variants: Adding peephole connections (Gers and Schmidhuber, 2000) essentially sets $g(x) = x$ and allows comparing counters in a stable way. Coupling the input and the forget gates ($i_t = 1 - f_t$) (Greff et al., 2017) removes the single-dimension unbounded counting ability, as discussed for the GRU.

[4] One such mechanism could be to divide a given dimension by $k > 1$ at each symbol encounter, by setting $z_t = 1/k$ and $\tilde{h}_t = 0$. Note that the inverse operation would not be implementable, and counting down would have to be realized with a second counter.

[5] One can argue that other counting mechanisms—involving several dimensions—are also possible. Intuitively, such mechanisms cannot be trained to perform unbounded counting based on a finite sample as the model has no means of generalizing the counting behavior to dimensions beyond those seen in training. We discuss this more in depth in Appendix B, where we also prove that an SRNN cannot represent a binary counter.

5. Trained LSTM networks outperform trained GRU networks on random test sets for the languages a^nb^n and $a^nb^nc^n$.

Similar empirical observations regarding the ability of the LSTM to learn to recognize a^nb^n and $a^nb^nc^n$ are described also in (Gers and Schmidhuber, 2001).

We train 10-dimension, 1-layer LSTM and GRU networks to recognize a^nb^n and $a^nb^nc^n$. For a^nb^n the training samples went up to $n = 100$ and for $a^nb^nc^n$ up to $n = 50$.[6]

Results On a^nb^n, the LSTM generalizes well up to $n = 256$, after which it accumulates a deviation making it reject a^nb^n but recognize a^nb^{n+1} for a while, until the deviation grows.[7] The GRU does not capture the desired concept even within its training domain: accepting a^nb^{n+1} for $n > 38$, and also accepting a^nb^{n+2} for $n > 97$. It stops accepting a^nb^n for $n > 198$.

On $a^nb^nc^n$ the LSTM recognizes well until $n = 100$. It then starts accepting also $a^nb^{n+1}c^n$. At $n > 120$ it stops accepting $a^nb^nc^n$ and switches to accepting $a^nb^{n+1}c^n$, until at some point the deviation grows. The GRU accepts already $a^9b^{10}c^{12}$, and stops accepting $a^nb^nc^n$ for $n > 63$.

Figure 1a plots the activations of the 10 dimensions of the a^nb^n-LSTM for the input $a^{1000}b^{1000}$. While the LSTM misclassifies this example, the use of the counting mechanism is clear. Figure 1b plots the activation for the $a^nb^nc^n$ LSTM on $a^{100}b^{100}c^{100}$. Here, again, the two counting dimensions are clearly identified—indicating the LSTM learned the canonical 2-counter solution—although the slightly-imprecise counting also starts to show. In contrast, Figures 1c and 1d show the state values of the GRU-networks. The GRU behavior is much less interpretable than the LSTM. In the a^nb^n case, some dimensions may be performing counting within a bounded range, but move to erratic behavior at around $t = 1750$ (the network starts to misclassify on sequences much shorter than that). The $a^nb^nc^n$ state dynamics are even less interpretable.

Finally, we created 1000-sample test sets for each of the languages. For a^nb^n we used words with the form $a^{n+i}b^{n+j}$ where $n \in \text{rand}(0, 200)$ and $i, j \in \text{rand}(-2, 2)$, and for $a^nb^nc^n$ we use words of the form $a^{n+i}b^{n+j}c^{n+k}$ where $n \in \text{rand}(0, 150)$ and $i, j, k \in \text{rand}(-2, 2)$. The LSTM's accuracy was 100% and 98.6% on a^nb^n and $a^nb^nc^n$ respectively, as opposed to the GRU's 87.0% and 86.9%, also respectively.

All of this empirically supports our result, showing that IBFP-LSTMs can not only theoretically implement "unbounded" counters, but also learn to do so in practice (although not perfectly), while IBFP-GRUs do not manage to learn proper counting behavior, even when allowing floating point computations.

6 Conclusions

We show that the IBFP-LSTM can model a real-time SKCM, both in theory and in practice. This makes it more powerful than the IBFP-SRNN and the IBFP-GRU, which cannot implement unbounded counting and are hence restricted to recognizing regular languages. The IBFP-IRNN can also perform input-dependent counting, and is thus more powerful than the IBFP-SRNN.

We note that in addition to theoretical distinctions between architectures, it is important to consider also the practicality of different solutions: how easy it is for a given architecture to discover and maintain a stable behavior in practice. We leave further exploration of this question for future work.

Acknowledgments

The research leading to the results presented in this paper is supported by the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 (PRIME), The Israeli Science Foundation (grant number 1555/15), and The Allen Institute for Artificial Intelligence.

[6] Implementation in DyNet, using the SGD Optimizer. Positive examples are generated by sampling n in the desired range. For negative examples we sample 2 or 3 n values independently, and ensuring at least one of them differs from the others. We dedicate a portion of the examples as the dev set, and train up to 100% dev set accuracy.

[7] These fluctuations occur as the networks do not fully saturate their gates, meaning the LSTM implements an imperfect counter that accumulates small deviations during computation, e.g.: increasing the counting dimension by 0.99 but decreasing only by 0.98. Despite this, we see that the its solution remains much more robust than that found by the GRU — the LSTM has learned the essence of the counting based solution, but its implementation is imprecise.

Appendix

A Simplified K-Counter Machines

We use a simplified variant of the k-counter machines (SKCM) defined in (Fischer et al., 1968), which has no autonomous states and makes classification decisions based on a combination of its current state and counter values. This variant consumes input sequences on a symbol by symbol basis, updating at each step its state and its counters, the latter of which may be manipulated by increment, decrement, zero, or no-ops alone, and observed only by checking equivalence to zero. To define the transitions of this model its accepting configurations, we will introduce the following notations:

Notations We define $z : \mathbb{Z}^k \to \{0,1\}^k$ as follows: for every $n \in \mathbb{Z}^k$, for every $1 \leq i \leq k$, $z(n)_i = 0$ iff $n_i = 0$ (this function masks a set of integers such that only their zero-ness is observed). For a vector of operations, $o \in \{-1,+1,\times 0,\times 1\}^k$, we denote by $o(n)$ the pointwise application of the operations to the vector $n \in \mathbb{Z}^k$, e.g. for $o = (+1,\times 0,\times 1)$, $o((5,2,3)) = (6,0,3)$.

We now define the model. An *SKCM* is a tuple $M = \langle \Sigma, Q, q_o, k, \delta, u, F\rangle$ containing:

1. A finite input alphabet Σ
2. A finite state set Q
3. An initial state $q_0 \in Q$
4. $k \in \mathbb{N}$, the number of counters
5. A state transition function
$$\delta : Q \times \Sigma \times \{0,1\}^k \to Q$$
6. A counter update function[8]
$$u : \Sigma \to \{-1,+1,\times 0,\times 1\}^k$$
7. A set of accepting masked[9] configurations
$$F \subseteq Q \times \{0,1\}^k$$

The set of *configurations* of an SKCM is the set $C = Q \times \mathbb{Z}^k$, and the initial configuration is $c_0 = (q_0, \bar{0})$ (i.e., the counters are initiated to zero). The transitions of an SKCM are as follows: given a configuration $c_t = (q, n)$ ($n \in \mathbb{Z}^k$) and input $w_t \in \Sigma$, the next configuration of the SKCM is $c_{t+1} = (\delta(q, w_t, z(n)), u(w_t)(n))$.

The language recognized by a k-counter machine is the set of words w for which the machine reaches an accepting configuration — a configuration $c = (q, n)$ for which $(q, z(n)) \in F$.

Note that while the counters can and are increased to various non-zero values, the transition function δ and the accept/reject classification of the configurations observe only their zero-ness.

A.1 Computational Power of SKCMs

We show that the SKCM model can recognize the context-free and context-sensitive languages a^nb^n and $a^nb^nc^n$, but not the context free language of palindromes, meaning its computational power differs from the language classes defined in the Chomsky hierarchy. Similar proofs appear in (Fischer et al., 1968) for their variant of the k-counter machine.

a^nb^n: We define the following SKCM over the alphabet $\{a, b\}$:

1. $Q = \{q_a, q_b, q_r\}$
2. $q_0 = q_a$
3. $k = 1$
4. $u(a) = +1,\ u(b) = -1$
5. for any $z \in \{0,1\}$:
$\delta(q_a, a, z) = q_a,\quad \delta(q_a, b, z) = q_b,$
$\delta(q_b, a, z) = q_r,\quad \delta(q_b, b, z) = q_b$
$\delta(q_r, a, z) = q_r,\quad \delta(q_r, b, z) = q_r$
6. $C = \{(q_b, 0)\}$

The state q_r is a rejecting sink state, and the states q_a and q_b keep track of whether the sequence is currently in the "a" or "b" phase. If an a is seen after moving to the b phase, the machine moves to (and stays in) the rejecting state. The counter is increased on input a and decreased on input b, and the machine accepts only sequences that reach the state q_b with counter value zero, i.e., that have increased and decreased the counter an equal number of times, without switching from b to a. It follows easily that this machine recognizes exactly the language a^nb^n.

$a^nb^nc^n$: We define the following SKCM over the alphabet $\{a, b\}$. As its state transition function ignores the counter values, we use the shorthand $\delta(q, \sigma)$ for $\delta(q, \sigma, z)$, for all $z \in \{0,1\}^2$.

1. $Q = \{q_a, q_b, q_c, q_r\}$

[8] We note that in this definition, the counter update function depends only on the input symbol. In practice we see that the LSTM is not limited in this way, and can also update according to some state-input combinations — as can be seen when it it is taught, for instance, the language a^nba^n We do not explore this here however, leaving a more complete characterization of the learnable models to future work.

[9] i.e., counters are observed only by zero-ness.

2. $q_0 = q_a$
3. $k = 2$
4. $u(a) = (+1, \emptyset)$,
   $u(b) = (-1, +1)$,
   $u(c) = (\emptyset, -1)$
5. for any $z \in \{0, 1\}$:
   $\delta(q_a, a) = q_a$, $\delta(q_a, b) = q_b$, $\delta(q_a, c) = q_r$,
   $\delta(q_b, a) = q_r$, $\delta(q_b, b) = q_b$, $\delta(q_b, c) = q_c$,
   $\delta(q_c, a) = q_r$, $\delta(q_c, b) = q_r$, $\delta(q_c, c) = q_c$,
   $\delta(q_r, a) = q_r$, $\delta(q_r, b) = q_r$, $\delta(q_r, c) = q_r$
6. $C = \{(q_c, 0, 0)\}$

By similar reasoning as that for $a^n b^n$, we see that this machine recognizes exactly the language $a^n b^n c^n$. We note that this construction can be extended to build an SKCM for any language of the sort $a_1^n a_2^n ... a_m^n$, using $k = m - 1$ counters and $k + 1$ states.

Palindromes: We prove that no SKCM can recognize the language of palindromes defined over the alphabet $\{a, b, x\}$ by the grammar $S \to x|aSa|bSb$. The intuition is that in order to correctly recognize this language in an one-way setting, one must be able to reach a unique configuration for every possible input sequence over $\{a, b\}$ (requiring an exponential number of reachable configurations), whereas for any SKCM, the number of reachable configurations is always polynomial in the input length.[10]

Let M be an SKCM with k counters. As its counters are only manipulated by steps of 1 or resets, the maximum and minimum values that each counter can attain on any input $w \in \Sigma^*$ are $+|w|$ and $-|w|$, and in particular the total number of possible values a counter could reach at the end of input w is $2|w| + 1$. This means that the total number of possible configurations M could reach on input of length n is $c(n) = |Q| \cdot (2n + 1)^k$.

$c(n)$ is polynomial in n, and so there exists a value m for which the number of input sequences of length m over $\{a, b\}$ — 2^m — is greater than $c(m)$. It follows by the pigeonhole principle that there exist two input sequences $w_1 \neq w_2 \in \{a, b\}^m$ for which M reaches the same configuration. This means that for any suffix $w \in \Sigma^*$, and in particular for $w = x \cdot w_1^{-1}$ where w_1^{-1} is the reverse of w_1, M classifies $w_1 \cdot w$ and $w_2 \cdot w$ identically—despite the fact that $w_1 \cdot x \cdot w_1^{-1}$ is in the language and $w_2 \cdot x \cdot w_1^{-1}$ is not. This means that M necessarily does not recognize this palindrome language, and ultimately that no such M exists.

Note that this proof can be easily generalized to any palindrome grammar over 2 or more characters, with or without a clear 'midpoint' marker.

B Impossibility of Counting in Binary

While we have seen that the SRNN and GRU cannot allocate individual counting dimensions, the question remains whether they can count using a more elaborate mechanism, perhaps over several dimensions. We show here that one such mechanism — a binary counter — is not implementable in the SRNN.

For the purposes of this discussion, we first define a binary counter in an RNN.

Binary Interpretation In an RNN with hidden state values in the range $(-1, 1)$, the *binary interpretation* of a sequence of dimensions $d_1, ..., d_n$ of its hidden state is the binary number obtained by replacing each positive hidden value in the sequence with a '1' and each negative value with a '0'. For instance: the binary interpretation of the dimensions 3,0,1 in the hidden state vector $(0.5, -0.1, 0.3, 0.8)$ is 110, i.e., 6.

Binary Counting We say that the dimensions $d_1, d_2, ..., d_n$ in an RNN's hidden state implement a *binary counter* in the RNN if, in every transition, their binary interpretation either increases, decreases, resets to 0, or doesn't change.[11]

A similar pair of definitions can be made for state values in the range $(0, 1)$.

We first note intuitively that an SRNN would not generalize binary counting to a counter with dimensions beyond those seen in training — as it would have no reason to learn the 'carry' behavior between the untrained dimensions. We prove further that we cannot reasonably implement such counters regardless.

We now present a proof sketch that a single-layer SRNN with hidden size $n \geq 3$ cannot implement an n-dimensional binary counter that will consistently increase on one of its input symbols. After this, we will prove that even with helper

[10] This will hold even if the counter update function can rely on any state-input combination.

[11] We note that the SKCMs presented here are more restricted in their relation between counter action and transition, but prefer here to give a general definition. Our proof will be relevant even within the restrictions.

dimensions, we cannot implement a counter that will consistently increase on one input token and decrease on another — as we might want in order to classify the language of all words w for which $\#_a(w) = \#_b(w)$.[12]

Consistently Increasing Counter: The proof relies on the linearity of the affine transform $Wx + Uh + b$, and the fact that 'carry' is a non-linear operation. We work with state values in the range $(-1, 1)$, but the proof can easily be adapted to $(0, 1)$ by rewriting h as $h' + 0.5$, where $h' = h - 0.5$ is a vector with values in the range $(-0.5, 0.5)$.

Suppose we have a single-layer SRNN with hidden size $n = 3$, such that its entire hidden state represents a binary counter that increases every time it receives the input symbol a. We denote by x_a the embedding of a, and assume w.l.o.g. that the hidden state dimensions are ordered from MSB to LSB, e.g. the hidden state vector $(1, 1, -1)$ represents the number 110=6.

Recall that the binary interpretation of the hidden state relies only on the signs of its values. We use p and n to denote 'some' positive or negative value, respectively. Then the number 6 can be represented by any state vector (p, p, n).

Recall also that the SRNN state transition is

$$h_t = \tanh(Wx_t + Uh_{t-1} + b)$$

and consider the state vectors $(-1, 1, 1)$ and $(1, -1, -1)$, which represent 3 and 4 respectively. Denoting $\tilde{b} = Wx_a + b$, we find that the constants U and $\tilde{b}$ must satisfy:

$$\begin{aligned} tanh(U(-1, 1, 1) + \tilde{b}) &= (p, n, n) \\ tanh(U(1, -1, -1) + \tilde{b}) &= (p, n, p) \end{aligned}$$

As tanh is sign-preserving, this simplifies to:

$$\begin{aligned} U(-1, 1, 1) = (p, n, n) - \tilde{b} \\ U(1, -1, -1) = (p, n, p) - \tilde{b} \end{aligned}$$

Noting the linearity of matrix multiplication and that $(1, -1, -1) = -(-1, 1, 1)$, we obtain:

$$U(-1, 1, 1) = U(-(1, -1, -1)) = -U(1, -1, -1)$$

$$(p, n, n) - \tilde{b} = \tilde{b} - (p, n, p)$$

i.e. for some assignment to each p and n, $2\tilde{b} = (p, n, n) + (p, n, p)$, and in particular $\tilde{b}[1] < 0$.

Similarly, for $(-1, -1, 1)$ and $(1, 1, -1)$, we obtain

$$\begin{aligned} U(-1, -1, 1) = (n, p, n) - \tilde{b} \\ U(1, 1, -1) = (p, p, p) - \tilde{b} \end{aligned}$$

i.e.

$$(n, p, n) - \tilde{b} = \tilde{b} - (p, p, p)$$

or $2\tilde{b} = (p, p, p) + (n, p, n)$, and in particular that $\tilde{b}[1] > 0$, leading to a contradiction and proving that such an SRNN cannot exist. The argument trivially extends to $n > 3$ (by padding from the MSB).

We note that this proof does not extend to the case where additional, non counting dimensions are added to the RNN — at least not without further assumptions, such as the assumption that the counter behave correctly for *all* values of these dimensions, reachable and unreachable. One may argue then that, with enough dimensions, it could be possible to implement a consistently increasing binary counter on a *subset* of the SRNN's state.[13] We now show a counting mechanism that cannot be implemented even with such 'helper' dimensions.

Bi-Directional Counter: We show that for $n \geq 3$, no SRNN can implement an n-dimensional binary counter that increases for one token, σ_{up}, and decreases for another, σ_{down}. As before, we show the proof explicitly for $n = 3$, and note that it can be simply expanded to any $n > 3$ by padding.

Assume by contradiction we have such an SRNN, with $m \geq 3$ dimensions, and assume w.l.o.g. that a counter is encoded along the first 3 of these. We use the shorthand $(v_1, v_2, v3)c$ to show the values of the counter dimensions explicitly while abstracting the remaining state dimensions, e.g. we write the hidden state $(-0.5, 0.1, 1, 1, 1)$ as $(-0.5, 0.1, 1)c$ where $c = (1, 1)$.

Let x_{up} and x_{down} be the embeddings of σ_{up} and σ_{down}, and as before denote $b_{up} = Wx_{up} + b$ and $b_{down} = Wx_{down} + b$. Then for some reachable state $h_1 \in \mathbb{R}$ where the counter value is 1 (e.g., the state reached on the input sequence σ_{up}[14])), we find that the constants U, b_{down}, and

[12] Of course a counter could also be 'decreased' by incrementing a parallel, 'negative' counter, and implementing compare-to-zero as a comparison between these two. As intuitively no RNN could generalize binary counting behavior to dimensions not used in training, this approach could quickly find both counters outside of their learned range even on a sequence where the difference between them is never larger than in training.

[13] (By storing processing information on the additional, 'helper' dimensions)

[14] (Or whichever appropriate sequence if the counter is not initiated to zero.)

b_{up} must satisfy:

$$tanh(Uh_1 + b_{up}) = (n, p, n)c_1$$
$$tanh(Uh_1 + b_{down}) = (n, n, n)c_2$$

(i.e., σ_{up} increases the counter and updates the additional dimensions to the values c_1, while σ_{down} decreases and updates to c_2.) Removing the sign-preserving function tanh we obtain the constraints

$$Uh_1 + b_{up} = (n, p, n)\text{sign}(c_1)$$
$$Uh_1 + b_{down} = (n, n, n)\text{sign}(c_2)$$

i.e. $(b_{up} - b_{down})[0 : 2] = (n, p, n) - (n, n, n)$, and in particular $(b_{up} - b_{down})[1] > 0$. Now consider a reachable state h_3 for which the counter value is 3. Similarly to before, we now obtain

$$Uh_3 + b_{up} = (p, n, n)\text{sign}(c_3)$$
$$Uh_3 + b_{down} = (n, p, n)\text{sign}(c_4)$$

from which we get $(b_{up} - b_{down})[0 : 2] = (p, n, n) - (n, p, n)$, and in particular $(b_{up} - b_{down})[1] < 0$, a contradiction to the previous statement. Again we conclude that no such SRNN can exist.

References

Roee Aharoni and Yoav Goldberg. 2017. Towards string-to-tree neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 132–140, Vancouver, Canada. Association for Computational Linguistics.

Yining Chen, Sorcha Gilroy, Kevin Knight, and Jonathan May. 2017. Recurrent neural networks as weighted language recognizers. *CoRR*, abs/1711.05408.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, 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.

Do Kook Choe and Eugene Charniak. 2016. Parsing as language modeling. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2331–2336, Austin, Texas. Association for Computational Linguistics.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555 [cs]*.

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

Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. 1968. Counter machines and counter languages. *Mathematical systems theory*, 2(3):265–283.

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

F. A. Gers and J. Schmidhuber. 2000. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194 vol.3.

K. Greff, R. K. Srivastava, J. Koutnk, B. R. Steunebrink, and J. Schmidhuber. 2017. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232.

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

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4107–4115. Curran Associates, Inc.

Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. 2015. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *arXiv:1504.00941 [cs]*.

Hava Siegelmann. 1999. *Neural Networks and Analog Computation: Beyond the Turing Limit*, 1 edition. Birkhäuser Basel.

Hava T. Siegelmann. 1996. Recurrent neural networks and finite automata. *Computational Intelligence*, 12:567–574.

Hava T. Siegelmann and Eduardo D. Sontag. 1992. On the computational power of neural nets. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992.*, pages 440–449.

Hava T. Siegelmann and Eduardo D. Sontag. 1994. Analog computation via neural networks. *Theor. Comput. Sci.*, 131(2):331–360.

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 2773–2781, Cambridge, MA, USA. MIT Press.