Sequential neural networks as automata

William Merrill

Advisor: Dana Angluin
Robert Frank

Department of Computer Science
Department of Linguistics
Yale University

This thesis is submitted for the degree of
Bachelor of Science

April 2019

Acknowledgements

First of all, thanks to my advisors Dana Angluin and Robert Frank for their frequent productive discussions and detailed edits of my drafts. I appreciate having advisors who are as enthusiastic about my senior project as I am. Additional thanks goes to:

- All the members of Computational Linguistics at Yale[1]
- 2018-2019 Linguistics Senior Seminar (Anelisa, James, Jay, Jisu, Magda, Noah, Rose, Hadas, Raffaella)
- Dragomir Radev's 2018 Advanced NLP Seminar
- Vidur Joshi, for suggesting the experiment in Section 4.2 during my visit to the Allen Institute for Artificial Intelligence
- Carl-Gustav Werner, for his runic LaTeX package [28]

Any remaining errors are my own.

[1] `http://clay.yale.edu/`.

Abstract

In recent years, neural network architectures for sequence modeling have been applied with great success to a variety of NLP tasks. What neural networks provide in performance, however, they lack in interpretability and theoretical motivation. This work attempts to explain the types of computation that neural models are able to perform by relating neural architectures to automata-theoretic classes from classical theoretical computer science.

My first contribution is to develop a notion of what it means for a real-time, finite-precision neural network to accept a language. A measure of the effective memory capacity for neural networks models follows from this definition. This can be used to derive upper bounds on the generative capacity of any neural network. I also further characterize the classes of languages acceptable by LSTMs, convolutional networks, and other architectures by relating them to automata. In particular, I prove that the LSTMs have equivalent generative capacity to counter machines, and I relate convolutional networks to the subregular hierarchy.

Overall, this work attempts to increase our understanding and ability to interpret neural network models through the lens of theory. The theoretical insights I develop help explain neural network computation, as well as the relationship between neural networks and natural-language grammar.

Table of contents

List of theorems

Chapter 1

Introduction

1.1 Background

In recent years, neural networks have achieved tremendous success on a variety of natural language processing (NLP) tasks. Neural networks employ continuous distributed representations of linguistic data, which contrasts with classical discrete methods. For example, Mikolov et al. [19] developed word2vec, a neural-network method for building vectors that effectively encode the meanings of words. This contrasts with classical approaches that represent lexical semantics as discrete expressions in the lambda calculus.

While neural approaches have shown impressive performance empirically on a variety of tasks, one of the downsides of the distributed representations which they utilize is that they are hard to interpret. In particular, it is hard to tell what kinds of computation a model is capable of, and when a model is working, it is hard to tell what it is doing.

This work aims to address such issues of interpretability by relating sequential neural networks to forms of computation that are more well understood. In theoretical computer science, the computational capacities of many different kinds of automata formalisms are clearly established. Moreover, the Chomsky hierarchy links natural language to such automata-theoretic languages [4]. Thus, relating neural networks to automata both yields insight into what general forms of computation such models can perform, as well as how such computation relates to natural-language grammar.

Recent work has investigated what kinds of automata-theoretic computations various types of neural networks can simulate. Weiss et al. [27] propose a connection between long short-term memory networks (LSTMs) and counter automata. They sketch a proof of how an LSTM can simulate a simplified variant of a counter machine, and then demonstrate that this counting ability constitutes a real computational difference between LSTMs and gated recurrent units (GRUs). Peng et al. [21], on the other hand, describe a connection between

the gating mechanisms of several neural network architectures and weighted finite-state acceptors (WFSAs).

I start by formalizing what it means for a real-time, bounded precision neural network to accept a formal language. I show how this definition leads to a measure of complexity that is generalizable to any neural sequence model. I use this theory to derive computation upper bounds and automata-theoretic characterizations for several different kinds of recurrent neural networks (Chapter 2), as well as other kinds of neural sequence models like convolutional networks and transformers (Chapter 3). Overall, this leads to a fairly complete automata-theoretic characterization of sequential neural networks. Chapter 4 reports empirical findings that test predictions of the theory developed in the preceding chapters.

In Chapter 5, I introduce the rational recurrences developed by Peng et al. [21], and I show that the simplified counter machines of Weiss et al. [27] are in fact rationally recurrent. This unifies two separate automata-theoretic analyses of recurrent neural networks that exist in the literature.

Finally, in Chapter 6, I further discuss the implications of my results for the connection between neural networks and natural language. In particular, this chapter focuses on the relationship between counter machines and context-free grammars, whether counter languages are semi-linear, and state complexity requirements for different syntactic constructions in natural language. The appendices provide additional technical results about the counter machines (Appendix A) and linearly separable expressions (Appendix B).

Overall, this work provides insight about the types of problems that several neural network architectures are able to compute through the lens of formal language theory. In so doing, I also obtain results relating these modes of computation to the computational mechanisms underlying natural language.

1.2 Introducing the asymptotic analysis

To investigate the capacities of different neural network architectures, we need to first define what it means for a neural network to accept a language. It is important to get this definition right. With unbounded computation time and arbitrary real-valued precision, even a simple recurrent network (SRN) becomes Turing complete [25]. Thus, to construct a theory that is both realistic and predicts differences between architectures, we need to capture the following constraints [27]:

1. *Real-time*: The network receives one input per step of computation.

2. *Bounded precision*: The value of each unit in the network is representable by $O(\log n)$ bits.

Informally, I will define a *neural sequence acceptor* as a network which reads variable-length sequence of characters and returns the probability that the input sequence is a valid sentence in some formal language. More precisely, we can write:

Definition 1.2.1 (Neural sequence acceptor)**.** Let $\mathbf{X}_{n\times l}$ be a matrix representation of an n-length sentence where each row $\mathbf{x}_t$ is a one-hot vector over an alphabet Σ with cardinality l. A neural sequence acceptor $\hat{\mathbb{1}}$ is a family of functions parameterized by weights θ. For each θ and n, the function $\hat{\mathbb{1}}^\theta$ takes the form

$$\hat{\mathbb{1}}^\theta : \mathbf{X} \mapsto p \in (0,1).$$

In this definition, $\hat{\mathbb{1}}$ corresponds to a general architecture like an LSTM, whereas $\hat{\mathbb{1}}^\theta$ corresponds to a specific network, such as an LSTM with weights that have been learned from data.

In order to get an acceptance decision from this kind of network, we will consider what happens as the magnitude of its parameters gets very large. Under these asymptotic conditions, the internal connections of the network become discrete, and the probabilistic output approaches the indicator function of some language. Figure 1.1 illustrates how taking this limit discretizes the network. I formalize this idea as *asymptotic acceptance*:

Definition 1.2.2 (Asymptotic acceptance)**.** Let L be a language with indicator function $\mathbb{1}_L$. A neural sequence acceptor $\hat{\mathbb{1}}$ with weights θ asymptotically accepts L if

$$\lim_{N\to\infty} \hat{\mathbb{1}}^{N\theta} = \mathbb{1}_L.$$

Note that the limit of $\hat{\mathbb{1}}^{N\theta}$ represents the function which $\hat{\mathbb{1}}^{N\theta}$ converges to pointwise.[1]

By formulating acceptance asymptotically, we restrict the amount of precision that each unit in the network can encode. From another point of view, the output of each squashing function acts like a bit, which forces all the representations in the network to be more discrete. This prevents complex fractal representations that rely on infinite precision. We will see later that, for every architecture that I consider, this definition ensures that every unit in the network is representable in $O(\log n)$ bits on sequences of length n. I do not make the claim that real neural networks do not utilize intermediate values of squashing functions at all: in fact, we know that they can. However, I argue that this definition still seems to make good

[1] `https://en.wikipedia.org/wiki/Pointwise_convergence`.

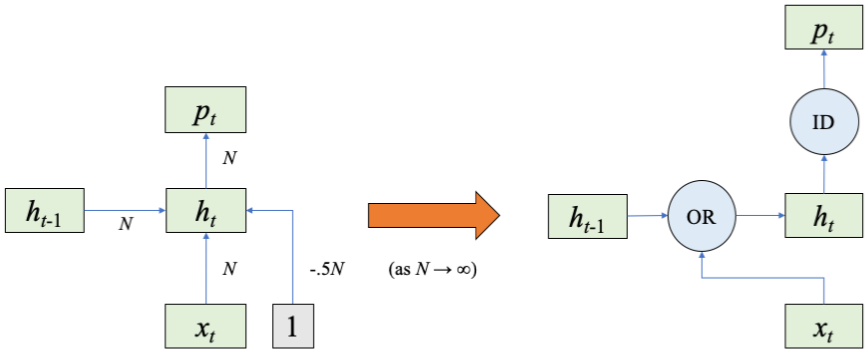


Fig. 1.1 With sigmoid activations, the network on the left accepts a sequence of bits if and only if $x_t = 1$ for some t. On the right is the discrete computation graph that the network approaches asymptotically.

predictions about the classes of languages that networks can accept because it enforces a precision bound. In Chapter 4, I show some empirical studies that support this claim.

An immediate consequence of this definition of acceptance is that, if a neural network asymptotically accepts a language, we can pick weights for the network such that it will correctly decide every string in that language up to an arbitrary length. Formally, we can state this as follows:

Theorem 1.2.1 (Arbitary approximation)**.** *Let $\hat{\mathbb{1}}$ be a neural sequence acceptor for L. For all m, there exist parameters θ_m such that, for all strings $\mathbf{x}_1, .., \mathbf{x}_n$ (represented by the matrix $\mathbf{X}$) with $n < m$,*

$$\left[\hat{\mathbb{1}}^{\theta_m}(\mathbf{X})\right] = \mathbb{1}_L(\mathbf{X}).$$

Proof. Consider a string $\mathbf{X}$. By the definition of asymptotic acceptance, we can pick $M_{\mathbf{X}}$ such that, for all $N \geq M_{\mathbf{X}}$,

$$\left|\hat{\mathbb{1}}^{N\theta}(\mathbf{X}) - \mathbb{1}_L(\mathbf{X})\right| < \frac{1}{2}, \tag{1.1}$$

which means that

$$\left[\hat{\mathbb{1}}^{N\theta}(\mathbf{X})\right] = \mathbb{1}_L(\mathbf{X}). \tag{1.2}$$

Now, observe that the set of all strings $\mathbf{X}$ with length less than m will always be finite. This means we can pick θ_m just by taking

$$\theta_m = \max_{\mathbf{X}} M_{\mathbf{X}}\theta. \tag{1.3}$$

□

Considering a network's asymptotic behavior also gives us a notion of the memory capacity of the network. Weiss et al. [27] illustrate how the additive cell state update of an LSTM gives it more effective memory than the squashed state of an SRN or GRU for solving counting tasks. I generalize this concept of memory capacity as *state complexity*. Informally, the state complexity of a hidden state node within a network represents the number of values that the node can achieve as a function of the sequence length n. For example, the LSTM cell state will have $O(n^k)$ state complexity (2.2.1), whereas the state of other recurrent networks has $O(1)$ (2.1.1).

Generally, we can define a hidden state similarly to a sequence acceptor, except that the output is an n-length sequence of vectors:

Definition 1.2.3 (Hidden state)**.** A k-length hidden state matrix $\mathbf{H}$ is a family of functions parameterized by weights θ. For each θ, the function $\mathbf{H}^\theta$ takes the form

$$\mathbf{H}^\theta : \mathbf{X}_{n\times l} \mapsto \mathbf{V}_{n\times k}. \tag{1.4}$$

I write $\mathbf{h}_t^\theta$ to mean the t-th row of $\mathbf{H}^\theta$. That is, for each θ and $1 \leq t \leq n$, $\mathbf{h}_t^\theta$ is a function

$$\mathbf{h}_t^\theta : \mathbf{X} \mapsto \mathbf{v}_t \in \mathbb{R}^k. \tag{1.5}$$

Often, a neural sequence acceptor can be written as a function of an intermediate hidden state. For example, the activations of the recurrent layer act as a hidden state in an LSTM acceptor or language model. In recurrent architectures, the value of the hidden state is a function of the preceding prefix of characters, but with convolution or attention, it can depend on characters occuring after index t. Next, I define the *configuration set* of such a hidden state:

Definition 1.2.4 (Configuration set)**.** Let E_l denote the set of l-length one-hot vectors. For all n, the configuration set of a hidden state $\mathbf{h}_n$ with weights θ is given by

$$M(\mathbf{h}_n^\theta) = \left\{ \lim_{N\to\infty} \mathbf{h}_n^{N\theta}(\mathbf{x}_1, .., \mathbf{x}_n) \mid \mathbf{x}_1, .., \mathbf{x}_n \in E_l \right\}.$$

Definition 1.2.5 (Fixed state complexity)**.** For a hidden state $\mathbf{h}_n$ with weights θ, the fixed state complexity with respect to θ is given by

$$\bowtie(\mathbf{h}_n^\theta) = \left|M(\mathbf{h}_n^\theta)\right|.$$

Definition 1.2.6 (General state complexity)**.** The state complexity of hidden state $\mathbf{h}_n$ is

$$\bowtie(\mathbf{h}_n) = \max_\theta \bowtie(\mathbf{h}_n^\theta).$$

To illustrate these definitions, consider a simplified memory mechanism based on an LSTM cell. The architecture is parameterized by a vector $\theta \in \mathbb{R}^2$. At each time step, the network reads a bit x_t and computes

$$f_t = \sigma(\theta_1 x_t) \tag{1.6}$$

$$i_t = \sigma(\theta_2 x_t) \tag{1.7}$$

$$h_t = f_t h_{t-1} + i_t. \tag{1.8}$$

When we set $\theta^+ = \langle 1, 1\rangle$, h_t asymptotically computes the sum of the preceding inputs. Because this sum can evaluate to any integer between 0 and n, $h_n^{\theta^+}$ has a fixed state complexity of

$$\bowtie\left(h_n^{\theta^+}\right) = O(n). \tag{1.9}$$

However, when we use parameters $\theta^z = \langle 0, 1\rangle$, we lose the recurrence and instead get a network where $h_t = x_t$ asymptotically. This means that the fixed state complexity is

$$\bowtie\left(h_n^{\theta^z}\right) = O(1). \tag{1.10}$$

Finally, the general state complexity for the hidden state h_n is the maximum fixed complexity, which in this case is $O(n)$.

For any neural network hidden state, the state complexity is at most exponential in the sequence length n (1.2.2). This means that the value of the hidden unit can be encoded in $O(n)$ bits. Moreover, for every specific architecture that I go on to consider, I observe that each fixed-length state vector has at most $O(n^k)$ state complexity, or, equivalently, can be represented in $O(\log n)$ bits. Architectures that utilize exponential state complexity, such as the transformer, do so by using a variable-length hidden state. We can also apply state complexity to a growing hidden state, with the only difference being that the state matrix $\mathbf{H}$ (1.2.3) becomes a sequence of variably sized objects.

Now, I prove the general bound on state complexity:

Theorem 1.2.2 (General bound on state complexity)**.** *Let* $\mathbf{h}_n$ *be a neural network hidden state. For any length n, it holds that*

$$\bowtie(\mathbf{h}_n) = 2^{O(n)}.$$

Proof. The number of configurations of $\mathbf{h}_n$ cannot be more than the number of distinct inputs to the network. By construction, each $\mathbf{x}_t$ is a one-hot vector over the alphabet Σ. Thus, the state complexity is bounded according to

$$\bowtie(\mathbf{h}_n) \leq |\Sigma|^n = 2^{O(n)}.$$

□

Equipped with the concept of asymptotic acceptance, we can go on to consider what classes of languages different neural networks can accept. State complexity will also prove useful for analyzing the computational capacities of different architectures. The theory that emerges from these tools will help us to better understand and interpret the computational processes underlying neural sequence models.

Chapter 2

Recurrent neural networks

In this chapter, I consider the relationship between automata and three kinds of recurrent neural networks (RNNs). As already mentioned, RNNs are Turing-complete [25] under an unconstrained definition of acceptance. This classical reduction relies on two very strong assumptions about RNN computation [27]. First, the number of recurrent computations must be unbounded in the length of the input, whereas, in practice, RNNs are almost always trained in a real-time fashion. Second, it relies heavily on infinite precision of the network's logits. Restricting computation to be real-time and have bounded precision severely constrains the class of formal languages that an RNN can accept. Thus, it is not unreasonable to ask whether a real-time, bounded precision RNN has the capacity to accept a certain language.

I will introduce the SRN, GRU, and LSTM and reason about what kind of automata computations they can perform. This will allow me to derive upper and lower bounds on the types of languages they can accept.

2.1 SRNs

The SRN, or Elman network, is the simplest type of recurrent neural network. We make the hidden layer recurrent by simply including the output at the previous time step in a standard affine transformation [6]. This can be written as

$$\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b}). \tag{2.1}$$

A well-known problem with SRNs is that they struggle with long-distance dependencies. One explanation of this is the vanishing gradient problem, which motivated the development of more sophisticated architectures like LSTMs [12]. Intuitively, another shortcoming of SRNs is that, in some sense, they have less state than an LSTM. This is because, while both

architectures have a fixed number of hidden units, the SRN units remain between 0 and 1, whereas the value of each LSTM cell can grow unboundedly [27]. The notion of asymptotic acceptance allows us to formalize this intuition. In particular, it turns out that an SRN only has a finite number of asymptotic configurations:

Theorem 2.1.1 (SRN state complexity)**.** *For any length n, the SRN cell state* $\mathbf{h}_n \in \mathbb{R}^k$ *has state complexity*

$$\bowtie(\mathbf{h}_n) \leq 2^k = O(1).$$

Corollary 2.1.1.1. *Let* $L(\text{SRN})$ *denote the languages acceptable by an SRN, and* RL *the regular languages. Then,*

$$L(\text{SRN}) \subseteq \text{RL}.$$

Proof. For every t, each unit of $\mathbf{h}_t$ will be the output of a tanh. In the limit, it can achieve either -1 or 1. Thus, for the full vector, the number of configurations less than or equal to 2^k. □

Interestingly, we can also show the other direction of containment, which yields the following result:

Theorem 2.1.2 (SRN characterization)**.**

$$L(\text{SRN}) = \text{RL}.$$

Proof. We already know that any SRN-acceptable language is regular. Now we will show that any language acceptable by a finite-state machine is SRN-acceptable. To do this, we need to asymptotically compute a representation of the machine's state in $\mathbf{h}_t$. I will do this by storing the value of the following predicate at each time step:

$$\eth_t(i,\alpha) \iff q_{t-1}(i) \wedge x_t = \alpha \tag{2.2}$$

where $q_t(i)$ is true if the machine is in state q_i at time t. Assuming h_t asymptotically computes $\eth_t$, we can compute an acceptance decision in the final layer according to the linearly separable formula (B.1.3) given by

$$a_t \iff \bigvee_{i \in F} \bigvee_{\langle j,\alpha \rangle \in \delta^{-1}(i)} \eth_t(j,\alpha). \tag{2.3}$$

Now, all that remains to be shown is how to compute $\eth_t$ at each time step. By rewriting q_{t-1} in terms of the previous $\eth_{t-1}$ values, we can get the following recurrence:

$$\eth_t(i,\alpha) \iff x_t = \alpha \wedge \bigvee_{\langle i,\alpha\rangle \in \delta^{-1}(i)} \eth_t(i,\alpha). \tag{2.4}$$

Since this formula is linearly separable in $x_t \| \eth_{t-1}$ (B.1.4), we can compute it in a single neural network layer from x_t and h_{t-1}. Now, we just need to worry about the base case: in other words, we need to ensure that transitions out of the initial state work out correctly at the first time step. We can do this by adding a new memory unit f_t to h_t which is always rewritten to have value 1. Thus, if $f_{t-1} = 0$, we can be sure we are in the initial time step. For each transition out of the initial state q_0, we add $f_{t-1} = 0$ as an additional term in the disjunction to get

$$\eth_t(0,\alpha) \iff x_t = \alpha \wedge \big(f_{t-1} = 0 \vee \bigvee_{\langle i,\alpha\rangle \in \delta^{-1}(0)} \eth_t(i,\alpha)\big). \tag{2.5}$$

This equation is still linearly separable (B.1.4) and guarantees that the initial step will be computed correctly.

□

Thus, we find that SRN-acceptable languages are equivalent to the regular languages. This is quite a diminished characterization compared to their Turing completeness under an unrestricted definition of acceptance [25]. We will see that LSTMs, on the other hand, are strictly more powerful than the regular languages.

2.2 LSTMs

An LSTM is a recurrent neural network with a complex gating mechanism that determines how information from one time step is passed to the next. Originally, this gating mechanism was designed to remedy the vanishing gradient problem in simple recurrent networks, or, equivalently, to make it easier for the network to remember long-term dependencies [12]. Due to strong empirical performance on many language tasks, LSTMs have become the canonical model for NLP.

Interestingly, Weiss et al. [27] suggest that another way to understand the success of the LSTM architecture is that they are expressive enough to accept simplified counter languages. They point out that this constitutes a real difference between the LSTM and the GRU, whose update equations do not allow it to operate as a counter machine.

I am to further investigate the connection between counter machines and LSTMs. In particular, I will derive upper bounds on what kinds of computation LSTMs can perform.

Together with Weiss et al. [27]'s arguments, this suggests that the generative capacity of LSTMs is essentially equivalent to some subclass of the counter languages.

To start, I will introduce the recurrent update equations for the LSTM:

Definition 2.2.1 (LSTM layer)**.**

$$\mathbf{i}_t = \sigma(\mathbf{W}^i \mathbf{x}_t + \mathbf{U}^i \mathbf{h}_{t-1} + \mathbf{b}^i) \tag{2.6}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}^f \mathbf{x}_t + \mathbf{U}^f \mathbf{h}_{t-1} + \mathbf{b}^f) \tag{2.7}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}^o \mathbf{x}_t + \mathbf{U}^o \mathbf{h}_{t-1} + \mathbf{b}^o) \tag{2.8}$$

$$\tilde{\mathbf{c}}_\mathbf{t} = \tanh(\mathbf{W}^c \mathbf{x}_t + \mathbf{U}^c \mathbf{h}_{t-1} + \mathbf{b}^c) \tag{2.9}$$

$$\mathbf{c}_t = \mathbf{i}_t \odot \mathbf{c}_{t-1} + \mathbf{f}_t \odot \tilde{\mathbf{c}}_\mathbf{t} \tag{2.10}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot f(\mathbf{c}_t). \tag{2.11}$$

In the last equation, we can let f be either the identity or tanh [27], although tanh is more standard in practice. The vector $\mathbf{h}_t$ is the output that is received by the next layer, and $\mathbf{c}_t$ is an unexposed memory vector that I will refer to as the cell state. Both of these vectors are copied and fed into the layer computation at the next time step.

Theorem 2.2.1 (LSTM state complexity)**.** *The LSTM cell state $\mathbf{c}_n \in \mathbb{R}^k$ has a state complexity of*

$$\bowtie(\mathbf{c}_n) = O(n^k).$$

Proof. At each time step t, we know that $\mathbf{i}_t, \mathbf{f}_t, \mathbf{o}_t \in \{0,1\}^k$ and $\tilde{\mathbf{c}}_\mathbf{t} \in \{-1,1\}^k$. This allows us to rewrite the elementwise recurrent update as

$$[\mathbf{c}_t]_i = [\mathbf{i}_t]_i [\mathbf{c}_t]_i + [\mathbf{f}_t]_i [\tilde{\mathbf{c}}_\mathbf{t}]_i. \tag{2.12}$$

$$\therefore [\mathbf{c}_t]_i = a[\mathbf{c}_t]_i + b \tag{2.13}$$

where $a \in \{0,1\}$ and $b \in \{-1,0,1\}$.

Define S_t to be the set of values that $[\mathbf{c}_t]_i$ can achieve. Observe that, at each time step, two new values appear in S_t that were not in S_{t-1}:

$$(\arg\min_{s \in S_{t-1}} s) - 1 \tag{2.14}$$

and

$$(\arg\max_{s \in S_{t-1}} s) + 1. \tag{2.15}$$

It follows that

$$|S_t| = 2 + |S_{t-1}| \tag{2.16}$$

$$\implies |S_n| = O(n). \tag{2.17}$$

Therefore, for all k units of the cell state, we have

$$\bowtie(\mathbf{c}_n) \leq |S_n|^k = O(n^k). \tag{2.18}$$

□

Additionally, analyzing the asymptotic configurations of an LSTM allows us to derive an upper bound on its expressive power:

Theorem 2.2.2 (LSTM upper bound)**.** *Let* CL *be the counter languages* (A.1.6)*. Then,*

$$L(\text{LSTM}) \subseteq \text{CL}.$$

Proof. The machine that we construct in Theorem 2.2.1 takes the form of a general counter machine whose counter and state update functions are constrained to be linearly separable. This implies that any LSTM-acceptable language is acceptable by a general counter machine. □

Theorem 2.2.2 constitutes a very tight upper bound on the expressiveness of LSTM computation. It implies that LSTMs are not powerful enough to model even simple context-free languages like $w\#w^R$.

Weiss et al. [27] argue for a lower bound on LSTM computation: namely that

$$\text{SCL} \subseteq L(\text{LSTM}) \tag{2.19}$$

where SCL is their class of simplified counter languages (A.2.2). Combining these results, we see that the expressiveness of the LSTM falls somewhere between the general counter machine and a simplified variant. This is good theoretical evidence that counting is a good way to understand the dynamics of an LSTM cell.

2.3 GRUs

The GRU is a popular gated recurrent neural network architecture that is in many ways similar to the LSTM [3]. Rather than having both an include and forget gate, the GRU utilizes a single gate which, along with its complement, modulates both the ability to include and to forget:

Definition 2.3.1 (GRU layer)**.**

$$\mathbf{z}_t = \sigma(\mathbf{W}^z \mathbf{x}_t + \mathbf{U}^z \mathbf{h}_{t-1} + \mathbf{b}^z) \tag{2.20}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}^r \mathbf{x}_t + \mathbf{U}^r \mathbf{h}_{t-1} + \mathbf{b}^r) \tag{2.21}$$

$$\mathbf{u}_t = \tanh\left(\mathbf{W}^u \mathbf{x}_t + \mathbf{U}^u (\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}^u\right) \tag{2.22}$$

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \mathbf{u}_t. \tag{2.23}$$

Weiss et al. [27] show empirically and analytically how this architectural difference prevents a GRU from similating a counter machine like an LSTM can. Similarly, my theory predicts that the GRU has strictly less state complexity than the LSTM:

Theorem 2.3.1 (GRU state complexity)**.** *The hidden state of a GRU has a state complexity of*

$$\bowtie(\mathbf{h}_n) = O(1).$$

Proof. As with the LSTM, the range of z_t converges to $\{0, 1\}$. Thus, we have two possibilities for each value of $[\mathbf{h}_t]_i$: either $[\mathbf{h}_{t-1}]_i$ or $[\mathbf{u}_t]_i$. Let S_t be the set of values that $[\mathbf{h}_{t-1}]_i$ can attain. We can write

$$S_t = S_{t-1} \cup \{-1, 1\}. \tag{2.24}$$

This implies that there are only three possible values for each logit: -1, 0, or 1. Thus, the number of state configurations of $\mathbf{h}_n$ is

$$\bowtie(\mathbf{h}_n) \leq 3^k = O(1). \tag{2.25}$$

□

Building on this result, we can show that the class of GRU-acceptable languages is exactly the regular languages:

Theorem 2.3.2 (GRU characterization)**.**

$$L(\text{GRU}) = \text{RL}.$$

Proof. By Theorem 2.3.1, the regular languages are an upper bound on the generative capacity of GRUs. On the other hand, I will demonstrate that any regular language is acceptable by some GRU. This implies that the classes are equivalent.

We can simulate a finite-state machine using an $\eth$ construction similar to the one in Theorem 2.1.2. For more detail, refer to that proof. I start by defining

$$\eth_t(i,\alpha) \iff q_{t-1}(i) \wedge x_t = \alpha. \tag{2.26}$$

In the recurrent case, we can rewrite this recursively in terms of $\eth_{t-1}$:

$$\eth_t(i,\alpha) \iff x_t = \alpha \wedge \bigvee_{\langle j,\beta\rangle \in \delta^{-1}(i)} \eth_{t-1}(j,\beta). \tag{2.27}$$

This formula is linearly separable in $x_t \| \eth_{t-1}$ (B.1.4). Therefore, we can store $\eth_t$ in our hidden state h_t and recurrently compute its update. The base case can be handled similarly to in Theorem 2.1.2. Then, in a final feedforward layer, we can compute whether we are in an accepting state from the value of $\eth_t$:

$$a_t \iff \bigvee_{i \in F} \bigvee_{\langle j,\beta,\rangle \in \delta^{-1}(i)} \eth_t(j,\beta). \tag{2.28}$$

This gives us a way to simulate any finite-state machine.

□

2.4 Summary

Synthesizing all of these results, we get the following complexity hierarchy:

$$\text{RL} = L(\text{SRN}) = L(\text{GRU}) \subseteq \text{SCL} \subseteq L(\text{LSTM}) \subseteq \text{CL}. \tag{2.29}$$

Most recurrent architectures seem to be essentially finite-state, whereas the LSTM is more powerful than a finite-state machine.

Chapter 3

Other neural sequence models

While recurrent networks are very well established within the field of NLP, it is also possible to use other architectures for sequence modeling and transduction tasks, such as convolutional networks and transformers [26]. Convolutional networks tend to be used for the specific task of modeling subword information, whereas transformers, although originally developed for machine translation [26], have been applied to a variety of tasks. Using the asymptotic analysis developed in Section 1.2, we can also reason about what kinds of computation these models are capable of.

3.1 Convolutional networks

While convolutional networks were originally developed with other purposes in mind [15], they can be use to process variable-length sequences. One popular application of this is to build character-level representations of words [14]. Another example of this is the capsule network architecture of Zhao et al. [30], which utilizes a convolutional layer as an initial feature extractor over a word-level sequence. Thus, we can ask about what kinds of formal languages convolutional networks can accept.

For recurrent networks, we defined output at the last time step as an acceptance decision for the whole sequence. This approach is problematic for convolutional networks because, due to the lack of recurrent connections, it would ignore all computation besides the last time step. Therefore, we should redefine our acceptance criterion.

There are a variety of ways by which we can reduce our vector of convolutional output to a scalar acceptance value. Treating the values as fuzzy bits, we could take a logical operation like AND or OR. Another possibility is to take a majority vote between bits, or add a simple one-bit RNN. A more realistic approach is to use max-over-time pooling [14] to collapse away the time dimension, and then use a final layer to produce an acceptance decision. Because

this resembles practically viable models, I choose to adopt it. The following architecture is based off of Kim et al. [14]:

Definition 3.1.1 (Convolutional language acceptor)**.**

$$\mathbf{h}_t = \tanh\left(\mathbf{W}^h(\mathbf{x}_{t-k}\|..\|\mathbf{x}_{t+k}) + \mathbf{b}^h\right) \tag{3.1}$$

$$\mathbf{h}_+ = \text{maxpool}(\mathbf{H}) \tag{3.2}$$

$$a = \sigma(\mathbf{W}^a\mathbf{h}_+ + \mathbf{b}^a). \tag{3.3}$$

In this model, the initial k-convolutional layer (3.1) produces a vector-valued sequence of outputs. Then, we collapse the time series of representations to a summary for the whole sequence by taking the maximum value of each filter over all the time steps (3.2). Once we have this representation, we add a single feedforward layer to produce an acceptance decision (3.3).

This convolutional architecture is substantially computationally weaker than an LSTM. Right away, we can see that $L(\text{CNN}) \subseteq \text{RL}$. This is because the state vectors $\mathbf{h}_t$ must have finite state. In fact, it turns out that there are simple regular languages that are provably beyond the capacity of a convolutional neural network. Thus, the subset relation is strict.

Theorem 3.1.1 (CNN upper bound)**.**

$$L(\text{CNN}) \subset \text{RL}.$$

Proof. By contradiction. Consider the language a^*ba^*. Assume we can write a network with window size k that accepts any string with exactly one b and reject any other string. Consider a string with two bs at indices i and j where $j - i > 2k + 1$. Then, there are no columns in the network which receive both $\mathbf{x}_i$ and $\mathbf{x}_j$ as input. Observe that the value of $\mathbf{h}_+$ determines whether the network accepts. When we replace one b with an a, the value of $\mathbf{h}_+$ remains the same after pooling, but we get a string with exactly one b. This means that the network should accept, which is a contradiction. □

Thus, to arrive at a characterization of what convolutional sequence acceptors can do, we should move to subregular classes of languages. In particular, we will consider the strictly local languages [23], which can be defined as follows:

Definition 3.1.2 (Strictly k-local grammar)**.** A strictly k-local grammar over an alphabet Σ is a set of constraints S where each $s \in S$ takes the form

$$s \in \left(\Sigma \cup \{\#\}\right)^k$$

where # is a padding symbol for the start and end of sentences.

Definition 3.1.3 (Strictly local acceptance)**.** A strictly k-local grammar S accepts a string σ if, at each index i,

$$\sigma_i \sigma_{i+1} .. \sigma_{i+k-1} \in S.$$

Definition 3.1.4 (SL_k)**.** SL_k is the set of all languages acceptable by a strictly k-local grammar.

The SL_k hierarchy is inherently related to the types of computation that a convolutional sequence acceptor can perform. In particular, we can state this as follows:

Theorem 3.1.2 (Strictly local CNNs)**.** *A k-convolutional network can asymptotically accept any strictly $2k+1$-local language.*

Corollary 3.1.2.1 (CNN lower bound)**.**

$$\mathrm{SL} \subseteq L(\mathrm{CNN}).$$

Proof. In the convolutional layer (3.1), each filter will identify whether a particular invalid $2k+1$-gram is matched. This condition is a conjunction of one-hot terms, which means we can easily construct a transformation that comes out to 1 if a particular invalid sequence was matched, and -1 otherwise.

Next, the pooling layer (3.2) collapses the filter values at each time step. A pooled filter will be positive if and only if the invalid sequence it is detecting was matched somewhere in the sequence.

Finally, we can compute acceptance (3.3) by checking whether any invalid filter was matched at all. To do this, we sum the filters and use sigmoid as a threshold at $-K$ where K is the number of invalid sequences. If any filter was matched, then the sum will exceed $-K$, and we reject. Otherwise, we accept. □

Interestingly, the tier-based strictly local languages have been proposed as a computational model for natural language phonological grammar [11]. Tier-based strictly local languages are very similar to strictly local languages, except that the local patterns can target characters in a specific tier of the vocabulary (e.g., vowels) instead of applying to the full string. In the field of NLP, convolutional networks have been used to model character-level information within words [14]. Theorem 3.1.2 provides a theoretical explanation for this: convolutional networks pick up on strictly local dependencies that are similar to those employed by natural-language phonology. While a single convolutional layer would be unable to extract tiers from

a sentence, it is conceivable that a more complex architecture which stacks convolutional or recurrent layers could simulate this behavior.

3.2 Attention

An attention function can be defined as a mapping from a query vector and a sequence of paired key-value vectors to a weighted combination of a sequence of values [1, 5, 18]. This output is meant to incorporate the values whose keys are relevant to the query.

Definition 3.2.1 (Dot-product attention)**.** For a query $\mathbf{q} \in \mathbb{R}^l$, matrix of key vectors $\mathbf{K} \in \mathbb{R}^{nl}$, and matrix of value vectors $\mathbf{V} \in \mathbb{R}^{nd}$, dot-product attention is given by

$$\text{attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{q}\mathbf{K}^T)\mathbf{V}.$$

This function gives us a vector of similarities between the query $\mathbf{q}$ and each key vector in $\mathbf{K}$. The vector that is outputted is a sum of the value vectors in $\mathbf{V}$ weighted by the similarity of the corresponding keys to the query. In practice, the dot product $\mathbf{q}\mathbf{K}^T$ is often scaled by the square root of the length of the query vector [26]. However, this is only done to improve optimization and has no effect on expressiveness. Therefore, I choose to consider the unscaled version.

We might wonder what the attention mechanism computes in the asymptotic case where $\mathbf{q}$ is parameterized by a scalar N that is approaching infinity. Interestingly, it turns out that computing attention reduces the mean of the values which maximize the similarity between their keys and the query:

Theorem 3.2.1 (Asymptotic attention)**.** *Let $t_1, .., t_m$ be the subsequence of time steps that maximize $\mathbf{q}\mathbf{k}_t$.* [1] *Asymptotically, attention computes*

$$\lim_{N\to\infty} \text{attention}\left(\mathbf{q}\left(N\right), \mathbf{K}, \mathbf{V}\right) = \frac{1}{m}\sum_{i=1}^{m} \lim_{N\to\infty}\left(\mathbf{v}_{t_i}\right).$$

Corollary 3.2.1.1 (Asymptotic attention with unique maximum)**.** *If $f : \mathbf{v}_t \mapsto \mathbf{q}\mathbf{k}_t$ has a unique maximum, then attention asymptotically computes*

$$\lim_{N\to\infty} \text{attention}\left(\mathbf{q}\left(N\right), \mathbf{K}, \mathbf{V}\right) = \lim_{N\to\infty} \arg\max_{\mathbf{v}_t} \mathbf{q}\mathbf{k}_t.$$

[1] To be precise, we can define a maximum over the similarity scores according to the order given by

$$f > g \iff \lim_{N\to\infty} \frac{f(N)}{g(N)} > 1. \tag{3.4}$$

Proof. Observe that, asymptotically, $\text{softmax}(\mathbf{u})$ approaches a function

$$\lim_{N\to\infty} \text{softmax}(N\mathbf{u})_t = \begin{cases} \frac{1}{m} & \text{if } u_t = \max(\mathbf{u}) \\ 0 & \text{otherwise,} \end{cases} \tag{3.5}$$

where m is the number of indices t that maximize u_t. Thus, the output of an attention mechanism reduces to the sum

$$\sum_{i=1}^{m} \frac{1}{m} \lim_{N\to\infty} (\mathbf{v}_{t_i}). \tag{3.6}$$

□

Attention mechanisms were originally used in sequence-to-sequence (seq2seq) networks as a way of modeling alignment in the context of machine translation [1]. At each time step, the decoder attends over the output of the encoder to produce a vector. Because I am concerned with language acceptance instead of sequence transduction, I will consider a variant of the seq2seq architecture that produces an output sequence of length 1. We can define such a model as follows:

Definition 3.2.2 (Attention layer)**.** Consider an encoder network which produces a sequence of vectors $v_1, .., v_n$ where the union of the asymptotic configuration sets for each v_t is finite. We attend over the encoded sequence by computing

$$\mathbf{q}_t = \mathbf{W}^q \mathbf{v}_t \tag{3.7}$$

$$\mathbf{h}_t = \text{attention}(\mathbf{q}_t, \mathbf{V}_t, \mathbf{V}_t). \tag{3.8}$$

In this model, $\mathbf{h}_n$ represents a summary of the relevant information in the prefix $\mathbf{v}_1, .., \mathbf{v}_n$. The query that is used to attend is a simple linear transformation of the final encoder state.

In addition to modeling alignment, another advantage of adding an attention mechanism to a recurrent network is that it introduces additional memory to a bounded-state model. The polynomial state complexity of the LSTM architecture means that it is impossible for LSTMs to copy or reverse arbitrary strings. Therefore, the additional memory provided by attention is essential for sequence transductions tasks like machine translation (2.2.1). To formalize this intuition, we can show that attending over a sequence of encoded vectors gives a model an exponential number of possible states.

Theorem 3.2.2 (Encoder state complexity)**.**

$$\bowtie(\mathbf{V}_n) = 2^{\Theta(n)}.$$

Proof. By the general upper bound on state complexity (1.2.2), we know that $\bowtie(\mathbf{V}_n) = 2^{O(n)}$. So, we just need to show the lower bound. This follows straightforwardly: first, we pick weights θ in the encoder such that there are two possible outputs at each $\mathbf{v}_t$, and the computation at each time step is independent. Thus, $\bowtie(\mathbf{v}_t^\theta) = 2$ for all t. Since the values at each time step are independent, we can write

$$\bowtie(\mathbf{V}_n^\theta) = \bowtie(\mathbf{v}_1^\theta) \cdot .. \cdot \bowtie(\mathbf{v}_n^\theta) = 2^n, \tag{3.9}$$

and in general

$$\bowtie(\mathbf{V}_n) = 2^{\Omega(n)}. \tag{3.10}$$

□

So, by converting the state of the model to a sequence of vectors $\mathbf{V}_n$ instead of a single vector $\mathbf{v}_n$, attention gives a model exponential state complexity. A natural follow-up question is whether this additional complexity is preserved in the attention vector $\mathbf{h}_n$. Attending over $\mathbf{V}_n$ does not preserve exponential state complexity. Instead, we get a linear state-bounded summary of the model's state:

Theorem 3.2.3 (Attention state complexity)**.**

$$\bowtie(\mathbf{h}_n) = O(n).$$

Proof. By Theorem 3.2.1, we know that

$$\lim_{N\to\infty} \mathbf{h}_n = \frac{1}{m} \sum_{i=1}^{m} \lim_{N\to\infty} (\mathbf{v}_{t_i}). \tag{3.11}$$

Now, let's consider how many configurations $\mathbf{h}_n$ can achieve. By construction, there is a finite set S containing all possible configurations of each $\mathbf{v}_t$. When we compute the mean of these values to get $\mathbf{h}_n$, observe that it does not matter what order the values occur in. All that matters is the number of times each distinct element of S occurs. This observation lets us bound the number of configurations of $\mathbf{h}_n$ by

$$|S|m \leq |S|n = O(n). \tag{3.12}$$

□

With minimal assumptions, we can show a more restrictive bound: namely, that the complexity of the attention vector comes out to be finite.

Theorem 3.2.4 (Attention state complexity with unique maximum)**.** *If* $f : \mathbf{v}_t \mapsto \mathbf{q}_n\mathbf{k}_t$ *has a unique maximum, then*

$$\bowtie(\mathbf{h}_n) = O(1).$$

Proof. If $\mathbf{q}_n\mathbf{k}_t$ has a unique maximum, then attention returns the $\mathbf{v}_t$ which maximizes $\mathbf{q}_n\mathbf{k}_t$ (3.2.1.1). By construction, there is a finite set S which contains all the values that any $\mathbf{v}_t$ can achieve. Thus, the $\mathbf{v}_t$ which is returned by attention has

$$\bowtie(\mathbf{h}_n) \leq |S| = O(1). \tag{3.13}$$

□

Theorem 3.2.5 (Attention state complexity with ReLU activations)**.** *If each* $\mathbf{v}_t \in \{0,\infty\}^k$*, then*

$$\bowtie(\mathbf{h}_n) = O(1).$$

Proof. By Theorem 3.2.1, we know that attention computes

$$\lim_{N\to\infty} \mathbf{h}_n = \frac{1}{m}\sum_{i=1}^{m} \lim_{N\to\infty} (\mathbf{v}_{t_i}). \tag{3.14}$$

This sum evaluates to a vector in $\{0,\infty\}^k$, which means that

$$\bowtie(\mathbf{h}_n) \leq 2^k = O(1). \tag{3.15}$$

□

This second theorem applies if the sequence $\mathbf{v}_1,..,\mathbf{v}_n$ is computed as the output of a ReLU. A similar result applies if it is computed as the output of an unsquashed linear transformation (3.21).

3.3 Transformers

Transformers are a new sequence model designed around the concept of neural attention [22, 26]. Due to their inherent uninterpretability and strong performance, analyzing the power of transformers is an interesting question. The general results about attention from Section 3.2 will prove useful for doing this.

The transformer architecture developed by Vaswani et al. [26] is motivated by the claim that "attention is all you need". In other words, their model replaces the recurrent connections of a classical seq2seq encoder with self-attention [24]. At each time step, a self-attention

layer predicts a key, query, and value. The output of the layer at time n is computed by attending with query $\mathbf{q}_n$ over the keys and values at all other time steps. The transformer architecture utilizes several different instantiations of self-attention heads in parallel, and then concatenates the outputted vectors. This *multihead* self-attention allows the network to search for different features over different parts of the sequence:

Definition 3.3.1 (Multihead self-attention)**.** Given a raw query $\mathbf{q}'$, raw keys $\mathbf{K}'$, and raw values $\mathbf{V}'$, multihead attention is given by

$$\mathbf{a}_i = \text{attention}(\mathbf{W}^{q_i}\mathbf{q}', \mathbf{W}^{V_i}\mathbf{K}', \mathbf{W}^{V_i}\mathbf{V}') \tag{3.16}$$

$$\text{multihead}(\mathbf{q}', \mathbf{K}', \mathbf{V}') = \mathbf{a}_1 \| .. \| \mathbf{a}_d. \tag{3.17}$$

The network proposed by Vaswani et al. [26] uses an encoder with multihead self-attention and a self-attention decoder that also attends over the output of the encoder. Further work has also developed a simplified architecture with a self-attention encoder and a feedforward decoder [17, 22]. Radford et al. [22] use this simplified transformer architecture for joint training of language modeling and text classification tasks. Due to the similarity of language modeling to language acceptance, I will use the variant of Radford et al. [22].

Definition 3.3.2 (Transformer layer)**.**

$$\mathbf{q}'_t \| \mathbf{k}'_t \| \mathbf{v}'_t = \mathbf{W}^x \mathbf{x}_t \tag{3.18}$$

$$\mathbf{h}_t = \sigma\big(\mathbf{W}^h \text{multihead}(\mathbf{q}'_t, \mathbf{K}'_t, \mathbf{V}'_t)\big). \tag{3.19}$$

One key difference between this model and the model of Radford et al. [22] is that the multihead attention here is not masked. This is because unmasked attention trivially solves the language modeling task, whereas it does not solve language acceptance. Therefore, I do not make this additional restriction.

Because of the presence of attention (3.2.2), the transformer state has complexity

$$\bowtie(\mathbf{V}_n) = 2^{\Theta(n)}. \tag{3.20}$$

Similarly, because each $\mathbf{v}_t \in \{-\infty, 0, \infty\}^k$, we know that, analogously to Theorem 3.2.5,

$$\bowtie(\mathbf{h}_n) = O(1). \tag{3.21}$$

Despite the transformer's exponential state complexity, it cannot accept every language acceptable by an LSTM. It has been documented that transformers have difficulty learning

positional dependencies without the augmentation of special positional encodings [26]. In the asymptotic case, Vaswani et al. [26]'s positional encodings fail because they rely on periodic functions which will eventually repeat for long enough strings. The positional invariance of the transformer motivates the following proof:

Theorem 3.3.1 (Relation to regular languages)**.**

$$\text{RL} \nsubseteq L(\text{Trans}).$$

Proof. By contradiction. Consider the language ab^*. Assume we can accept a string $x = ab^{n-1}$ for some n. We can swap the positions of a and some arbitrary b to produce a string y on which the state of h_n will be unchanged. Then, we will accept $y \notin L$, which is a contradiction. □

3.4 Stack recurrent networks

One way to make an RNN closer to a context-free grammar is to construct a differentiable pushdown automaton [9, 10]. This is done by defining a stack data structure that is differentiable, and then training a controller network that manipulates the stack as well as producing output. Because the vectors popped from the stack are differentiable with respect to the sequence of vectors that have been pushed onto it, we can use back-propagation to compute all the partial derivatives in the network's computation graph.

The technical details of the differentiable stack architecture are quite complicated. At a high level, the stack implements an interface

$$\langle \mathbf{v}_t, u_t, d_t \rangle \mapsto \mathbf{r}_t, \tag{3.22}$$

where:

1. $\mathbf{v}_{t+1} \in \mathbb{R}^k$ is a vector to be pushed onto the stack matrix $\mathbf{S}_t \in \mathbb{R}^{tk}$
2. $d_{t+1} \in (0,1)$ is the amount of mass that should be popped from the top of $\mathbf{S}_t$
3. $u_{t+1} \in (0,1)$ is the weight with which v should be added to the top of the stack
4. $\mathbf{r}_{t+1} \in \mathbb{R}^k$ is a vector summary for the top of the new stack $\mathbf{S}_{t+1}$

The controller network receives $\mathbf{r}_{t-1}$ and $\mathbf{x}_t$ as input and predicts $\mathbf{v}_t$, u_t, and d_t, which are then used to manipulate the stack. Refer to Hao et al. [10] for a more detailed introduction.

Hao et al. [10] show how a stack RNN can effectively solve a variety of formal language tasks. Additionally, the structured memory mechanism allows for interpretability of the algorithm that a stack RNN is learning [10]. Yogatama et al. [29] use a multipop variant of the same neural stack architecture to achieve state-of-the-art performance on language modeling. It is an open question what other tasks stack neural networks can prove practically viable for. To facilitate further work on this question, I have released a public PyTorch [20] implementation of the stack neural network architecture.[2]

We can abstract away from the specifics of the stack implementation while trying to analyze its computational power. In particular, I will derive its state complexity by considering a simple controller that only ever pushes to its stack. We'll see that, even in this simple case, the stack's state complexity exceeds that of an LSTM. The reasoning here is similar to the argument I presented for attention mechanisms (3.2.2).

Theorem 3.4.1 (Neural stack state complexity)**.** *Let* $\mathbf{S}_n \in \mathbb{R}^{nk}$ *be a neural stack with a feedforward controller. Then,*

$$\bowtie(\mathbf{S}_n) = 2^{\Theta(n)}.$$

Proof. By the general state complexity bound (1.2.2), we know that $\bowtie(\mathbf{S}_n) = 2^{O(n)}$. Thus, we just need to show that $\bowtie(\mathbf{S}_n) = 2^{\Omega(n)}$. The stack at time step n is a matrix $\mathbf{S}_n \in \mathbb{R}^{nk}$ where each row corresponds to a vector that has been pushed on at each time step. Consider the subset of configurations that we reach by only pushing. Since the vector that is pushed onto the stack at time t is a function of $\mathbf{x}_t$ only, it has some finite number of configurations greater than 1. Thus, for all n rows of the matrix, the number of configurations will be $2^{\Omega(n)}$. □

This result show how stack neural networks have representational power beyond that of LSTMs. However, it should be noted that this increased representational power is not necessarily a good thing: perhaps it makes learning more difficult.

3.5 Summary

A one-layer convolutional network is strictly less powerful than the regular languages, and thus also strictly less powerful than all the variants of RNNs. We saw however, that these networks are powerful enough to model strictly local patterns like those occurring in natural-language phonology, which suggests that they have a level of expressiveness that is well-suited for building character-level representations.

[2] `https://github.com/viking-sudo-rm/StackNN`.

Under the asymptotic analysis, attention, transformers, and the differentiable stack data structure all have a state complexity beyond that of RNNs. In the case of attention, the sequence of values that is attended over introduces exponential state complexity into the model, but the complexity of the summary vector produced by attending over such a sequence is finite.

Chapter 4

Empirical results

My analysis of the expressiveness of neural network architectures assumes that the languages that are asymptotically acceptable (1.2.2) by an architecture are a good model for the languages that that architecture can model in practice. This idea relies on the assumption that strategies which use the continuous values in the middle of a squashing function are impractical. One reason for this is that real-world neural networks are implemented on finite-precision machines, so continuous strategies are doomed to fail once the precision bound is exceeded. We can look to empirical evidence to verify that asymptotic acceptance and state complexity are meaningful notions for describing the behavior of neural networks.

4.1 Validating state complexity

Empirical results serve as a sanity check that state complexity reflects the memory that neural models have in practice. I use two formal language tasks to investigate this. The first one is modeling the language $a^n b^n$, which requires a linear number of states. Second, reversing a string ($w \mapsto w^R$) requires exponentially many states because we must remember an n-length prefix. In Figure 4.1, we see that the theoretical measure of state complexity agrees with the

Architecture	Complexity	$a^n b^n$	$w \mapsto w^R$
SRN	$O(1)$	No	No
GRU	$O(1)$	No	No
LSTM	$O(n^k)$	**Yes**	No
LSTM-Attn	$2^{\Theta(n)}$	**Yes**	**Yes**
StackNN	$2^{\Theta(n)}$	**Yes**	**Yes**

Fig. 4.1 Ability of several neural network architectures to learn formal language tasks with known state size requirements.

Architecture	$n = 32$	$n = 128$
Sigmoid LSTM	96	91
ReLU LSTM	95	55

Fig. 4.2 Accuracy on $\{(a|b)^* \mid \#(a) > \#(b)\}$ with a single hidden layer of 4 units.

ability of different architectures to perform these tasks. Architectures with less than linear state complexity cannot model the counter language, and models with less than exponential state complexity cannot reverse a string.

4.2 State complexity of ReLU LSTMs

Observe that, if we replace the sigmoid activations in the LSTM with ReLU activations, the state complexity of the LSTM would collapse to finite state.[1] This provides another testable hypothesis: the ReLU LSTM should should fail if the strings get long enough, whereas the sigmoid LSTM will always be able to solve the task by counting. Conventional wisdom, on the other hand, would predict that there is generally little difference between using a sigmoid or ReLU activation function [15]. I ran an experiment to test the validity of the theoretical prediction using the following simple counter language:

$$\{(a|b)^* \mid \#(a) > \#(b)\}. \tag{4.1}$$

The results of the experiment are recorded in Figure 4.2.[2] I report test accuracy after 10 epochs of training a model with 4 hidden units. For short strings ($n = 32$), both models converge to high accuracy. However, for longer strings ($n = 128$), the ReLU LSTM fails, whereas the standard sigmoid LSTM gets close to 100% accuracy.

To clarify why a finite-state model with 4 hidden units should be able to solve the counting problem for strings of length 32, we can estimate the maximum difference between the number of a's and the number of b's that should occur in any string. With a continuous random walk, this difference is bounded by $\sqrt{2n \log\log n}$ with high probability by the Law of Iterated Logarithms. We can use this as a rough upper bound on the number of states that a model should need to solve the task. With $n = 32$, this quantity is approximately 9. Since $9 < 2^4$, it makes sense that the finite-state model should be able to solve the problem well.

[1] Thanks to Vidur Joshi for pointing this out.

[2] `https://github.com/viking-sudo-rm/nn-automata`.

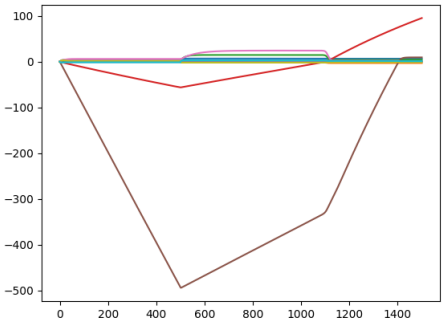


Fig. 4.3 Hidden units of an LSTM language model on $a^{500}b^{1000}$.

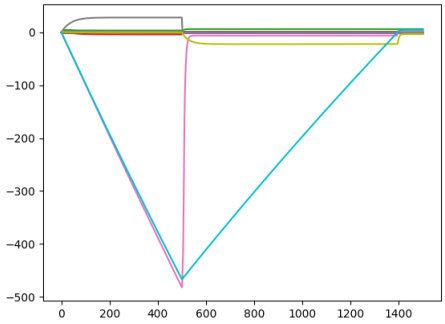


Fig. 4.4 Hidden units of another LSTM language model on $a^{500}b^{1000}$.

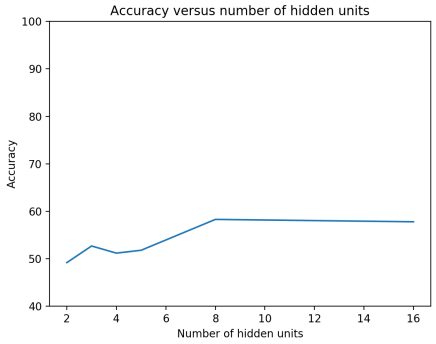


Fig. 4.5 Accuracy of an LSTM trained to reverse long binary strings.

4.3 LSTMs as counter machines

Empirical evidence confirms the theoretical result that counter languages beyond the simplified ones are LSTM-acceptable. Weiss et al. [27] note that, empirically, an LSTM is able to learn to model $a^n b a^n$, which is one example of a non-simplified counter language. Similarly, I found that an LSTM is able to learn to model $a^n b^{2n}$ in a counter-like fashion. In Figure 4.3, we see that the decrement for a b is different depending on whether it is the in first or second half of the b sequence. This suggests that the counter update is being conditioned by state. In Figure 4.4, we see that the cyan cell is incremented by 2δ for an a and decremented by δ for a b. This is not achievable by a simplified counter machine, which can only add or subtract 1. It appears that such a strategy would require utilizing an intermediate values in the LSTM gates. Thus, the machine may be acting in a way that would not be allowed by the asymptotic definition I have assumed.

4.4 Discussion

Overall, this empirical evidence suggests that analyzing the asymptotic case of neural network computation is a good way to understand what neural networks can express in practice. While networks may sometimes learn to utilize intermediate values of squashing functions, state complexity still accurately predicts the effective memory capacity of a variety of neural networks architectures in all cases that I considered. One explanation for this might be that the network can only really utilize a small number of states on the intermediate section of the squashing function, rather than being able to utilize it continuously. Thus, treating the logits discretely would remain sensible.

Chapter 5

Rational recurrences

Peng et al. [21] introduce the term "rational recurrence" to describe an RNN recurrent update that can be computed elementwise by series of weighted finite-state automata (WFSAs). Recall that, in RNNs, the gate update function is expressed as a recurrence

$$\mathbf{c}_t = f(\mathbf{x}_t, \mathbf{c}_{t-1}). \tag{5.1}$$

For example, in an SRN [6], the gate update takes the form

$$\mathbf{c}_t = \tanh(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{c}_{t-1} + \mathbf{b}). \tag{5.2}$$

If we unroll the computation graph of the network, this recurrence becomes a function of the variable-length input prefix $\mathbf{x}_1, .., \mathbf{x}_t$. Thus, we will consider recurrent update function to be a vector-valued function of the form $\mathbf{c} : \Sigma^* \to \mathbb{K}^k$. This kind of function, which takes a variable-length sequence as input, is exactly the type of object that can be computed by a series of WFSAs.

5.1 WFSAs

Formally, a WFSA is a non-deterministic automaton where each transition receives a weight [21]. This allows us to define a numerical score for any input string. The automaton assumes a particular semi-ring $\mathbb{K}$ with operations $\otimes$ and $\oplus$. This allows us to define a score for all paths through the automaton:

Definition 5.1.1 (Path score)**.** The score of a path $\pi_1, .., \pi_t$ is given by

$$A[\pi] = \lambda(q_1) \otimes \big(\bigotimes_{i=1}^{t} \tau(\pi_i)\big) \otimes \rho(q_{t+1}).$$

Semantically, τ is a function which gives us the score of each transition. Similarly, λ gives us the score of starting in each state, and ρ gives the score for ending in any state. These functions generalize the concepts of initial and accepting states. Next, we define the score of for an input string as the sum of the scores over all possible paths:

Definition 5.1.2 (String score)**.** The score of a string x is given by

$$A[x] = \bigoplus_{\pi \in \Pi(x)} A[\pi].$$

We consider the output of a WFSA on a particular string to be the score assigned to the string by the WFSA. Thus, a sequence of k WFSAs will compute a vector $\mathbf{c}_t \in \mathbb{K}^k$.

5.2 Simplified counter machines as rational recurrences

Just like we can write a recurrence relating the hidden states in a recurrent neural network, we can also write a recurrence relating the update to the counter state in a simplified counter machine (A.2.2). Interestingly, the gating mechanism which dictates how the counters are updated turns out to be a rational recurrence.

Theorem 5.2.1 (Rational recurrence of simplified counter machines)**.** *A simplified counter machine is rationally recurrent.*

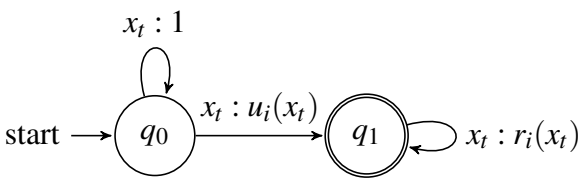
Proof. Let $\mathbf{c}_t$ be the value of the counters at time t. We will now parameterize the counter operations as

$$\mathbf{c}_t = \mathbf{r}(x_t)\mathbf{c}_{t-1} + \mathbf{u}(x_t). \tag{5.3}$$

This parameterization allows us to express all of the valid update operations. For -1, $+0$, and $+1$, we set $\mathbf{r}(x_t) = 1$, and $\mathbf{u}(x_t)$ to -1, 0, and 1 respectively. For $\times 0$, we set $\mathbf{u}(x_t) = \mathbf{r}(x_t) = 0$. Next, we can unroll this recurrence in time to get

$$\mathbf{c}_t = \sum_{i=1}^{t} \Big(\prod_{j=i+1}^{t} \mathbf{r}(x_j) \Big) \mathbf{u}(x_i). \tag{5.4}$$

Element i of this vector is computed by a WFSA of the form:



Assigning q_0 to be the start state means that $\lambda(q_0) = 1$ and $\lambda(q_1) = 0$. Similarly, when I say that q_1 is an accepting state, I mean that $\rho(q_1) = 1$ and $\rho(q_0) = 0$.

□

5.3 General counter machines

Extending this reduction to general counter machines does not seem to work. This is because the update operation is conditioned by the previous counter state in addition to the input symbol:

$$c_t = r(x_t, z(c_{t-1}))c_{t-1} + u(x_t, z(c_{t-1})) \tag{5.5}$$

$$\implies c_t = \sum_{i=1}^{t} \Big(\prod_{j=i+1}^{t} r(x_j, z(c_{j-1}))\Big) u(x_i, z(c_{i-1})). \tag{5.6}$$

This modification means that the values of $\mathbf{r}$ and $\mathbf{u}$ are conditioned by more-than-finite state. Thus, we can no longer use the same scheme to write a WFSA where $\mathbf{r}$ and $\mathbf{u}$ are introduced by a finite number of state transitions.

Peng et al. [21] notes an analogous problem in reducing the LSTM gate update to a rational recurrence. In their case, the fact that $\tilde{\mathbf{c}}_t$ depends on $\mathbf{h}_{t-1}$ prevented the derivation of a rationally recurrent form. Thus, there is a striking similarity between LSTM computation and general counter machine computation. Just as these observations led Peng et al. [21] to conjecture that the LSTM is not rationally recurrent, I conjecture that the general counter machine is not rationally recurrent:

Conjecture 5.3.1 (The general case)**.** *A general counter machine is not rationally recurrent.*

An interesting implication of Peng et al. [21]'s conjecture that LSTMs are not rationally recurrent is that the simplified counter machines are strictly weaker than LSTM computation. This is consistent with the theoretical and empirical arguments that I present in Chapter 4. Therefore, it seems likely that the LSTM languages, although probably not as powerful as the general counter languages, are substantially more powerful than Weiss et al. [27]'s simplified counter languages.

Chapter 6

Implications for natural language

Given that LSTMs seem to act as counter machines, we should ask how counter machines relate to formal models of natural language grammar. This gives us some insight about how similar an LSTM's representation of syntax is to that which exists in the mind. In particular, I consider the linguistic property of semilinearity, as well as the relationship between counter languages and context-free languages.

Finally, I will also discuss the significance of the state complexity measure from the point of view of syntactic structure. We will see that the state complexity of a grammar is inherently related to the types of embedding and agreement that it can be sensitive to.

6.1 Semilinearity of counter languages

Semilinearity is a condition that has been proposed as a desired property for any formalism of natural language syntax [13]. Intuitively, semilinearity ensures that the set of string lengths in a language will not be unnaturally sparse. More formally, we can define a language L to be semi-linear if its Parikh mapping is a semilinear set.

Definition 6.1.1 (Parikh vector)**.** The Parikh vector for a string $x \in L$ is

$$\Psi(x) = \langle \#(\sigma_1, x), .., \#(\sigma_n, x) \rangle .$$

Definition 6.1.2 (Parikh mapping)**.** The Parikh mapping of a language L is

$$\Psi(L) = \{\Psi(x) \mid x \in L\} .$$

In machine learning terms, we might describe the Parikh mapping of a string as its bag-of-characters representation. By translating languages into vector spaces, the Parikh mapping allows us to define the semilinear languages:

Definition 6.1.3 (Semilinear set)**.** A set $S \subseteq \mathbb{N}^n$ is semilinear if it can be written as the finite union of the form

$$\bigcup_{i=1}^{m} \{\mathbf{W}_i\mathbf{x} + \mathbf{b}_i = 0 \mid \mathbf{x} \in \mathbb{N}^n\}.$$

Definition 6.1.4 (Semilinear language)**.** A language L is semilinear if $\Psi(L)$ is semilinear.

Regular languages, context-free grammars, and a variety of mildly context-sensitive grammar formalisms are known to be semilinear [13]. Since counter machines exhibit a lot of the same properties as context-free grammars, it seems reasonable that the counter languages might also be semilinear. While I do not prove this in full generality, I present a proof that the stateless simplified counter languages are semilinear. This line of reasoning might in the future be extended to the general counter languages.

Definition 6.1.5 (Stateless simplified counter languages)**.** Let $\tilde{\text{Q}}$SCL be the class of languages acceptable by a simplified counter machine with only one state.

Theorem 6.1.1 (Semilinearity of $\tilde{\text{Q}}$SCL)**.** $L \in \tilde{\text{Q}}$SCL *is semilinear.*

Proof. We can start by expressing L as

$$L = \bigcup_{b \in F} \{x \mid c_n(x) = b\}. \tag{6.1}$$

Since semilinear languages are closed under finite union, L is semilinear if each of the following sets, which correspond to specific accepting configurations, is semilinear:

$$L_b = \{x \mid \mathbf{c}_n(x) = \mathbf{b}\}. \tag{6.2}$$

Furthermore, this set can be rewritten as the intersection of sets with elementwise constraints. Since semilinear languages are closed under finite intersection, the problem reduces to showing that each $L_b(i)$ is semilinear:

$$L_b(i) = \{x \mid [\mathbf{c}_n]_i(x) = b_i\}. \tag{6.3}$$

I claim that $L_b(i)$ is semilinear. We first consider the simple case where counter i cannot be zeroed. Since counter i cannot be reset, we can write

$$b_i = [c_n]_i(x) = \sum_{t=1}^{n} u_i(x_t) = \sum_{\sigma \in \Sigma} \#(\sigma, x) u_i(\sigma). \tag{6.4}$$

Note that the right half of this equation parameterizes $\mathbb{N}^{|\Sigma|}$. When $b = 0$, we target a subspace of $\mathbb{N}^{|\Sigma|}$, which means $L_b(i)$ is semilinear. When $b = 1$, we target the complement of this hyperplane, which can be expressed as the union of two linear sets. Therefore, $L_b(i)$ is always semilinear.

Now, I claim that $L_b(i)$ is also semilinear when counter i can be zeroed by the specification of the counter machine. To analyze this case, I define $L'_b(i)$ as the language where all the characters that zero out counter i are removed from the alphabet. Clearly, this language is semilinear by the same argument as the simple case we just considered (6.4). I also define the language

$$R_b(i) = L_b(i) - L'_b(i). \tag{6.5}$$

$L_b(i)$ is semilinear if $R_b(i)$ is semilinear since $L_b(i) = L'_b(i) \cup R_b(i)$. Thus, we just need to show that $R_b(i)$ is semilinear to complete the proof. To do this, consider a string $x \in R_b(i)$. There is some index which is the last occurrence of a character that resets counter i. We will define the suffix starting after this index to be the string ω. Observe the set of all ω is $L_b(i)$. Also, we can represent any $x \in R_b(i)$ as

$$\alpha\rho\omega, \tag{6.6}$$

where α is any string over Σ, and ρ is the last symbol which resets counter i. Since all three of these substrings come from semilinear languages and semilinearity is closed under concatenation, $R_b(i)$ is semilinear.

□

While this proof only applies to the stateless simplified counter languages (which are quite a restricted class), I conjecture that a similar argument can be extended to SCL, or possibly even to CL.

Conjecture 6.1.1 (Semilinearity of SCL)**.** $L \in$ SCL *is semilinear.*

Conjecture 6.1.2 (Semilinearity of CL)**.** $L \in$ CL *is semilinear.*

6.2 Counter machines and context-free grammars

Context-free languages do a decent but imperfect job of modeling the hierarchical structure that occurs in natural language [4]. On the other hand, counter machines seem to be a good model for LSTM computation. Thus, comparing the generative capacities of these two

automata architectures is, in some sense, comparing the types of languages that LSTMs can effectively model to natural language.

Context-free grammars and counter machines are both strictly more powerful than regular expressions. This is because, if we ignore the memory mechanism of each machine, we are left with a simple finite-state machine. We know, however, that neither class is a subset of the other. The language $a^n b^n c^n d^n$ is an example of a counter-acceptable language that is not context-free. On the other hand, the reverse language $w\#w^R$ is context-free, but not counter-acceptable [27].

A surprising classical result is that the language of well-formed preorder expressions is real-time acceptable [8] by a 1-counter machine. I say that this is surprising because pre-order boolean expressions have a rich hierarchical structure resembling the syntactic trees of natural language. We can formalize this language L_n as follows:

Definition 6.2.1 (L_n)**.** Let L_n be the language generated by the grammar:

```
<exp> -> <VALUE>
<exp> -> <UNARY_OP> <exp>
<exp> -> <BINARY_OP> <exp> <exp>
<exp> -> <n-ARY_OP> <exp> .. <exp>
```

Fischer et al. [8]'s proof of this theorem essentially uses a counter to keep track of the depth at any given index. If the depth counter returns to its initial value at the end of the string, the machine has verified that the string is well-formed. This algorithm is in some sense agnostic to the actual structure of the string in that it cannot recover the dependencies between tokens. This means that it could not be used to evaluate one of these expressions, for example. This observation motivates the next theorem, which shows that a counter machine is unable to evaluate even a very simple language of expressions:

Theorem 6.2.1 (Weak evaluation)**.** *A real-time k-counter transducer cannot evaluate pre-order boolean expressions.*

Proof. Assume not. Consider the case where the input contains a prefix of n operators. For the machine to evaluate the string correctly, the configuration after character n must encode which boolean function is determined by the prefix.

However, a real-time k-register machine only has $|Q|n^k$ configurations. I will show by induction that an n-length prefix of operators can encode 2^n boolean functions. Since $|Q|n^k < 2^n$ for large enough n, we reach a contradiction.

In the base case, we have a prefix of length zero which is followed by one value. If this value is 0, the expression will evaluate to 0, and if this value is 1, the expression will evaluate to 1. Therefore, we can represent exactly one function, which is the identity.

Consider the inductive case. The expression has a prefix of operators $x_1, .., x_{n+1}$ followed by symbols $x_{n+2}, .., x_l$. First, observe that x_l must be atomic to make the expression syntactically allowable. The value x_l must be the second argument of x_1, which forces everything else to be x_1's first argument. Thus, the semantics of the full expression can be represented as

$$[\![x_1, .., x_{n+1}]\!] = [\![x_1]\!]([\![x_2, .., x_{l-1}]\!], [\![x_l]\!]). \tag{6.7}$$

Observe that $x_2, .., x_{l-1}$ is a prefix of length n. Thus, by inductive hypothesis, $[\![x_2, .., x_{n+1}]\!]$ could be one of 2^n possible functions. The compositional relationship in (6.7) introduces a new variable into all of these possible functions, so we get two new functions in $[\![x_1, .., x_{n+1}]\!]$ by fixing x_1:

$$f_\wedge = [\![\wedge]\!]([\![x_2, .., x_{l-1}]\!], [\![x_l]\!]) = [\![x_2, .., x_{l-1}]\!] \wedge [\![x_l]\!] \tag{6.8}$$

and

$$f_\vee = [\![\vee]\!]([\![x_2, .., x_{l-1}]\!], [\![x_l]\!]) = [\![x_2, .., x_{l-1}]\!] \vee [\![x_l]\!]. \tag{6.9}$$

We can verify that $f_\wedge$ and $f_\vee$ are different functions by considering the first sequence of bits that will satisfy them according to a right-to-left ordering. We see that this sequence for $f_\wedge$ will necessarily end in a 1, whereas for $f_\vee$ it will end in a 0. Therefore, we are introducing exactly two new functions for each f, which means a $n+1$-length sequence can encode $2 \cdot 2^n = 2^{n+1}$ many $n+1$-ary functions.

□

This result relies on the crucial fact that the number of configurations of a general counter machine is bounded by $|Q|n^k$. A context-free grammar, on the other hand, has exponentially many memory configurations.

6.3 State complexity of sentence embedding

Embedding in natural language is the process of placing one sentence within another one. This kind of recursive procedure is one of the things that gives natural language grammars their infinite capacity. Different kinds of embedding have different kinds of processing demands. Interestingly, we can apply the same notion of state complexity that we used to analyze neural network architectures and counter machines to these grammars. What we find is that different levels of state complexity enable different types of embedding.

6.3.1 Right embedding

Different kinds of embedding exist in natural language. *Right embedding* consists of concatenating sentences in a sequence. For example:

(1) a. Gudrun sees Mary.

b. John knows Gudrun sees Mary.

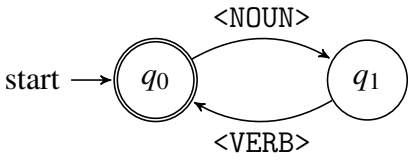
c. I believe John knows Gudrun sees Mary.

Formally speaking, we can construct the following toy grammar to simulate the dependency structures of right embedding:

Definition 6.3.1 (Right embedding grammar)**.** Define the grammar:[1]

```
<sentence> -> #
<sentence> -> <NOUN> <VERB> <sentence>
```

This simple form of embedding can be parsed fairly simply. In fact, doing so only requires a finite number of states. To demonstrate this, we could accept the language generated by Definition 6.3.1 using the following finite-state machine:



6.3.2 Center embedding

Another way to recursively generate sentences is to place the inner sentence within the outer sentence constituent. This is called *center embedding*, and it occurs in constructions like relative clauses:

(2) a. The cat slept.

b. The cat the dog chased slept.

c. ? The cat the dog the boy fed chased slept.

Already with these examples, we see that processing center embedding is fairly memory-intensive. Whereas sentences (2a) and (2b) are clearly grammatical, it takes some time to verify that (2c) checks out. We can build a toy model of center embedding as follows:

[1] I use # here to represent the null string.

Definition 6.3.2 (Center embedding grammar)**.** Define the grammar:

```
<sentence> -> #
<sentence> -> <NOUN> <sentence> <VERB>
```

We can verify whether a sentence is in this language with a linear number of states using a counter machine. This is true because the center embedding language is isomorphic to $a^n b^n$. However, the task becomes more complex if we want to evaluate the semantics of a construction like this, or, similarly, enforce agreement between corresponding nouns and verbs.

6.3.3 Matched center embedding

Matched center embedding is a variant of this center embedding grammar that enforces agreement between the noun and the verb at each level of the embedding:

Definition 6.3.3 (Matched center embedding grammar)**.** Define the grammar:

```
<sentence> -> #
<sentence> -> <NOUN[SG]> <sentence> <VERB[SG]>
<sentence> -> <NOUN[PL]> <sentence> <VERB[PL]>
```

This kind of feature agreement is common in natural language. For example, in English, there is number agreement between a verb and its subject:

(3) a. The **cat** the dog sees **runs**.

b. * The **cat** the dog sees **run**.

Because the grammar needs to keep track of the grammatical number at each depth, the number of states we need to verify that a sentence is in the language becomes exponential. To see this, observe that this grammar is isomorphic to a subset of a Dijk language with two different kinds of parentheses. If we allow sequences of balanced parentheses at each level, we get a grammar that mixes right and center embedding. The state complexity of this grammar remains exponential.

6.3.4 The Linzen agreement task

A toy grammar that mixes center and right embedding while enforcing agreement between nouns and verbs is a good formal model of the Linzen agreement task [16]. This task consists

of reading a sequence of words and then predicting the number of the following verb. The Linzen task has been used in the literature as a method of assessing the syntactic capabilities of different kinds of neural networks. We can view the exponential state complexity of the formal grammar model (Subsection 6.3.3) as a theoretical argument that solving this task requires structure sensitivity.

Interestingly, LSTMs can perform very well on the Linzen task, despite the fact that I have shown that they only have polynomial state complexity (2.2.1). One explanation for this might be that the embedding depth that actually occurs in natural-language data is bounded. In practice, we do not need exponential state complexity to keep track of structure. This hypothesis agrees with the the follow-up analysis done by Linzen et al. [16]. While LSTMs perform remarkably well on the general agreement task, they perform dramatically worse when evaluation targets syntactically complex cases.

6.3.5 Chomsky dependencies

State complexity is inherently related to the type of dependencies between words that a grammar can be sensitive to. By dependencies between words, I mean that the form of one word is linked to the form of a word earlier in the sentence. An example of this from Subsection 6.3.3 is English number agreement. In his foundational work on generative syntax, Chomsky [4] formalizes this notion of a syntactic dependency. I recast this definition as follows:

Definition 6.3.4 (Chomsky dependency)**.** Consider a sentence $x \in L$. There is a dependency between indices i and j with $i < j$ if there exist character y_i, y_j such that

$$x_{1:i-1} y_i x_{i+1:n} \notin L \tag{6.10}$$

and

$$x_{1:i-1} y_i x_{i+1:j-1} y_j x_{j+1:n} \in L. \tag{6.11}$$

Using this definition, we can formalize how many dependencies a sentence contains:

Definition 6.3.5 (Dependency set)**.** The dependency set of a sentence $x \in L$ is the set of tuples $\langle i_k, j_k \rangle$ such that

1. there is a dependency between i_k and j_k in x for each k;
2. $i_k < j_l$ for each k, l;

3. $i_k \neq i_l$ and $j_k \neq j_l$ for each k, l.

The size of this set corresponds to the number of nested dependencies in the sentence. Chomsky [4] observes that, if a sentence $x \in L$ has m dependencies, a finite-state machine which accepts L must have at least 2^m states. A consequence of this fact is that regular languages can only have finitely many dependencies. We can extend this relationship from finite-state machines to machines with bounded state, i.e. where the number of states at any time step is bounded by a function of n.

Remark 6.3.1 (Dependencies and state complexity)**.** *If $x \in L$ has $T(n)$ dependencies, then any machine which accepts L has at least $2^{T(n)}$ states.*

This fact makes interesting predictions about the types of dependencies or agreement that different computational models can be sensitive to. For example, it explains the explosion in state complexity that incorporating agreement caused in Subsection 6.3.3. From the point of view of syntax, the state complexity of a grammar is fundamentally related to its ability to represent agreement and embedding.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge university press, 2002.

[3] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[4] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[5] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in neural information processing systems*, pages 577–585, 2015.

[6] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[7] Patrick C Fischer. Turing machines with restricted memory access. *Information and Control*, 9(4):364–379, 1966.

[8] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Counter machines and counter languages. *Mathematical systems theory*, 2(3):265–283, Sep 1968. ISSN 1433-0490. doi: 10.1007/BF01694011. URL `https://doi.org/10.1007/BF01694011`.

[9] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

[10] Yiding Hao, William Merrill, Dana Angluin, Robert Frank, Noah Amsel, Andrew Benz, and Simon Mendelsohn. Context-free transductions with neural stacks. *arXiv preprint arXiv:1809.02836*, 2018.

[11] Jeffrey Heinz, Chetan Rawal, and Herbert G Tanner. Tier-based strictly local constraints for phonology. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 58–64. Association for Computational Linguistics, 2011.

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

[13] Aravind K Joshi, K Vijay Shanker, and David Weir. The convergence of mildly context-sensitive grammar formalisms. *Technical Reports (CIS)*, page 539, 1990.

[14] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[16] Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of lstms to learn syntax-sensitive dependencies. *arXiv preprint arXiv:1611.01368*, 2016.

[17] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018.

[18] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[21] Hao Peng, Roy Schwartz, Sam Thomson, and Noah A Smith. Rational recurrences. *arXiv preprint arXiv:1808.09357*, 2018.

[22] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/language-unsupervised/language_ understanding_paper. pdf*, 2018.

[23] James Rogers and Geoffrey K Pullum. Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information*, 20(3):329–342, 2011.

[24] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[25] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 440–449, New York, NY, USA, 1992. ACM. ISBN 0-89791-497-X. doi: 10.1145/130385.130432. URL `http://doi.acm.org/10.1145/130385.130432`.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[27] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision RNNs for language recognition. *CoRR*, abs/1805.04908, 2018. URL `http://arxiv.org/abs/1805.04908`.

[28] Carl-Gustav Werner. The allrunes font and package. 2004.

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

[30] Wei Zhao, Jianbo Ye, Min Yang, Zeyang Lei, Suofei Zhang, and Zhou Zhao. Investigating capsule networks with dynamic routing for text classification. *arXiv preprint arXiv:1804.00538*, 2018.

Appendix A

Counter machines

Informally, counter machines are a class of automata that can use a finite number of integer variables as memory. From this point of view, they are similar to the computational model known as the abacus machine [2].

Early results in theoretical computer science established that a 2-counter machine with unbounded computation time is Turing-complete [7]. It turns out, however, that when we restrict the computation to be real-time (i.e. one iteration of computation per input), the computational capacity of counter machines is severely limited. As we have seen, this is a striking similarity between counter machines and LSTMs.

While the classical literature on counter machines focused more on the unbounded variant, Weiss et al. [27] discuss the real-time machine because of its potential relationship to LSTM computation. In particular, they argue that a simplified variant of the counter machines can be simulated by LSTMs, and they provide empirical evidence to justify that LSTM languages models can learn to manipulate their memory cells as counters. They also note that, while their theoretical arguments only hold for a restricted class of counter machines, LSTMs seem to be powerful enough to handle some general counter languages.

In this work, I will focus only on the real-time counter machines as language acceptors. I will attempt to paint a comprehensive picture of counter computation by comparing the sets of languages that different variants of counter machines can accept. In particular, we will look at the general real-time counter machines, the simplified machines from Weiss et al. [27], and some slightly less restricted forms of counter machines. It turns out that some of the restrictions imposed by Weiss et al. [27] on the original counter model severely restrict the computational capacity of the model, whereas others do not change what it can compute.

A.1 The general counter machine

We first define a fully general counter machine, as well as the class of languages that are acceptable by such a machine in real time [7, 8].

Definition A.1.1 (General counter machine)**.** We define a counter machine as a tuple of the form $\langle \Sigma, Q, q_0, k, u, \delta, F \rangle$ containing

1. A finite alphabet Σ
2. A finite set of states Q
3. An initial state q_0
4. A number of counters $k \in \mathbb{N}$
5. A counter update function

$$u : \Sigma \times Q \times \{0,1\}^k \to \left(\{\lambda x.x + n : n \in \mathbb{Z}\} \cup \{\lambda x.0\}\right)^k$$

6. A state transition function

$$\delta : \Sigma \times Q \times \{0,1\}^k \to Q$$

7. An acceptance mask

$$F : Q \times \{0,1\}^k \to \{0,1\}$$

Note that I will generally represent F as a masking function, but at times it will be more convenient to treat it as a set of accepting configurations $\langle q, b \rangle$.

Next, we can define a computational configuration for such a machine, as well as what it means for the machine to accept a string. To do this, we will need a notion of a zero-check function z.

Definition A.1.2 (Zero-check function)**.**

$$z(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{otherwise.} \end{cases}$$

Definition A.1.3 (Counter machine computation)**.** Let $\langle q, \mathbf{c} \rangle \in Q \times \mathbb{Z}^k$ be a configuration of machine M. Upon reading input $x \in \Sigma$, M transitions into the new configuration $\langle q', \mathbf{c}' \rangle$ where

$$c' = \mathbf{u}(x, q, z(\mathbf{c})) \tag{A.1.3.1}$$

and

$$q' = \delta(x, q, z(\mathbf{c})). \tag{A.1.3.2}$$

We write this relation as

$$\langle q, \mathbf{c} \rangle \rightarrow_x \langle q', \mathbf{c}' \rangle. \tag{A.1.3.3}$$

Definition A.1.4 (Real-time string acceptance)**.** A counter machine accepts a string $x_1, .., x_n$ if

$$\langle q_0, \mathbf{0} \rangle \rightarrow_{x_1} \langle q_1, \mathbf{c}_1 \rangle \rightarrow_{x_2} .. \rightarrow_{x_n} \langle q_n, \mathbf{c}_n \rangle \tag{A.1.1}$$

and

$$F\big(q_n, z(\mathbf{c}_n)\big). \tag{A.1.2}$$

Definition A.1.5 (Real-time language acceptance)**.** A counter machines accepts a language L if its accepts each $\alpha \in L$ and rejects each $\beta \notin L$.

Definition A.1.6 (Counter languages)**.** Let CL be the set of languages that are acceptable in real time by a general counter machine.

A.2 Counter machine variants

Now, we can can consider various restrictions of this machine, and the corresponding classes of languages acceptable by such automata. First, we redefine the simplified counter machine discussed by Weiss et al. [27], which they call an "SKCM".

Definition A.2.1 (Simplified counter machine)**.** A simplified counter machine is a counter machine where u takes the restricted form

$$u : \Sigma \rightarrow \{-1,\ +0,\ +1,\ \times 0\}^k.$$

Definition A.2.2 (Simplified counter languages)**.** Let SCL be the set of languages that are acceptable in real time by a simplified counter machine.

We can view the counter update function in the simplified counter machine as having two important restrictions compared to the general machine. First, it can only be conditioned by the input symbol at each time step. Second, its update operation must be a 0 or 1 instead of any arbitrary constant.

Another variant which we consider is the incremental counter machine, which is affected only by the second of these restrictions.

Definition A.2.3 (Incremental counter machine)**.** An incremental counter machine is a counter machine where u takes the restricted form

$$u : \Sigma \times Q \times \{0,1\}^k \to \{-1,\ +0,\ +1,\ \times 0\}^k.$$

Definition A.2.4 (Incremental counter languages)**.** Let ICL be the set of languages that are acceptable in real time by an incremental counter machine.

I will also define a variant of counter machines that operate without state. For simplicity, we will say that the counter machine has exactly one state q_0, but note that this is equivalent to reformulating the counter machine specification with all references to state removed.

Definition A.2.5 (Stateless counter machine)**.** A stateless counter machine is a counter machine with only one state q_0.

Definition A.2.6 (Stateless counter languages)**.** Let $\tilde{\text{Q}}$CL be the set of languages that are acceptable in real time by a stateless counter machines.

A.3 Relationships between counter classes

It turns out that the simplified counter languages are a strict subset of the general counter languages. Their weakness comes from the fact that the counter update function can only be conditioned by the input symbol. A language that illustrates this difference is $a^n b^{2n}$:

Theorem A.3.1 (Weakness of SCL)**.**

$$\text{SCL} \subset \text{CL}.$$

Proof. Consider the language $a^n b^{2n}$. This is trivially acceptable by a 1-counter machine that adds 2 for a and subtracts 1 for b. On the other hand, I claim it cannot be accepted by any simplified machine. We will think about the subproblem of distinguishing between strings in $a^* b^*$ and focus on the value of a single counter. After scanning the a sequence, we know that

its value must be $u_a \in \{-n, 0, n\}$. Then, when we read the bs, the additional update to the counters must be $u_b \in \{-2n, 0, 2n\}$.

We need to determine whether the number of as equals the twice number of bs based on the value of $z(c) = z(u_a + u_b)$. But this cannot be done: if we pick the 0 update for both a and b, then for any $\sigma \in a^*b^*$,

$$u_a + u_b = 0 \implies z(u_a + u_b) = 0. \tag{A.3.1}$$

On the other hand, if we pick any other pair of u_a and u_b, then for any $\sigma \in a^*b^*$,

$$u_a + u_b \neq 0 \implies z(u_a + u_b) = 1. \tag{A.3.2}$$

So, for any pair of update operations we pick, the counters cannot distinguish whether the number of bs is twice the number of as. □

Note that this counterexample breaks down if we allow the counter update to depend on the state. In that case, we can build a machine which has two counters and three states: one which adds 1 to the first counter while it reads bs, another which decrements the first counter and increments the second counter, and a third which decrements the second counter until the end of the string. This motivates the next theorem.

Whereas the simplified counter model is weaker than the general counter machine, just restricting the counter updates to be incremental does not limit the machine's computational power. Similarly, restricting the machine to be stateless does not weaken it. I demonstrate this in the next two theorems.

Theorem A.3.2 (Generality of ICL)**.**

$$\text{CL} = \text{ICL}.$$

Proof. By construction, ICL $\subseteq$ CL. The goal is to show that CL $\subseteq$ ICL. We do this by simulating a single register in the general counter machine with a constant number of registers on the incremental machine.

Consider a counter c in the general machine. We will define a vector of registers $\hat{c}_1, .., \hat{c}_k$ to correspond to c, where k is the maximum value by which c is ever incremented. We will define a way to to read off the value of c from $\hat{\mathbf{c}}$, as well as $\textsc{Add}(\delta)$, $\textsc{Sub}(\delta)$, and $\textsc{Set}(0)$ update operations.

I will define the following invariants over the counter values, and later show that they are preserved by the update operations:

$$c = \sum_{n=1}^{k} n\hat{c}_n \tag{A.3.3}$$

$$\hat{\mathbf{c}} \text{ is one-hot or } \mathbf{0}. \tag{A.3.4}$$

A natural way of computing the zero mask of the simulated counters follows from these invariants:

$$z(c) \iff \bigvee_{n=1}^{k} z(\hat{c}_n). \tag{A.3.5}$$

We can simulate counter updates according to the following operations:

- SET(0):

$$\forall j \quad u_j = \times 0. \tag{A.3.6}$$

- ADD(δ):

$$u_i = -1,\ u_{\min(i+\delta,k)} = +1,\ u_{i+\delta-k} = +1. \tag{A.3.7}$$

- SUB(δ):

$$\begin{cases} u_i = -1,\ u_{i-\delta} = +1 & \text{if } i \geq \delta \\ u_i = -1,\ u_k = -1,\ u_{k+i-\delta} = +1 & \text{otherwise.} \end{cases} \tag{A.3.8}$$

By u_n, we denote the update operation for counter n. If $n \leq 0$ upon evaluation in the expressions below, then we do not apply u_n to any counter. Let i be the nonzero index in $\hat{\mathbf{c}}$ or 0 if this is undefined. Also note that each of these update functions is representable on a counter machine because each can be written as a finite function of the form

$$(z(\hat{\mathbf{c}}), n) \mapsto u_n.$$

Consider the ADD(δ) update. In general, the form of the new value of the counter vector will be given by

$$\sum_{n=1}^{k} n(\hat{c}_n + u_n) = \sum_{n=1}^{k} n\hat{c}_n + \sum_{n=1}^{k} nu_n \tag{A.3.9}$$

$$= c + iu_i + \min(i+\delta,k)u_{\min(i+\delta,k)} + \mathbb{1}_{i+\delta>k}(i+\delta-k)u_{i+\delta-k}. \tag{A.3.10}$$

When $i+\delta > k$, we get

$$c - i + k + i + \delta - k = c + \delta. \tag{A.3.11}$$

In the other case, $i + \delta \leq k$. Then we get

$$c - i + i + \delta = c + \delta. \tag{A.3.12}$$

Either way, the non-leading counters remain a one-hot or zero-hot vector. This is true because the one-hot index is zeroed out, and at most one non-leading index is set to one.

Now, consider the $\textsc{Sub}(\delta)$ update. When $i \geq \delta$, the new counter state is given by

$$c - i + i - \delta = c - \delta. \tag{A.3.13}$$

In the complementary case where $i < \delta$, we get

$$c - i - k + k + i - \delta = c - \delta. \tag{A.3.14}$$

Again, we know that the non-leading counters remain a one-hot or zero-hot vector because index i is always zeroed out, and at most one other non-leading position is set to 1. □

Theorem A.3.3 (Generality of $\tilde{\text{Q}}$CL)**.**

$$\text{CL} = \tilde{\text{Q}}\text{CL}.$$

Proof. Consider a counter machine $M = \langle \Sigma, Q, q_0, k, \delta, u, F \rangle$. We define a new stateless machine M' whose counters are augmented by a vector $\hat{q}$ with length $|Q|$. We initialize $\hat{q}_0 = 1$ and set all other indices to 0. Furthermore, we define as an invariant that

$$q(M) = q_i \iff \hat{q} = \omega(i) \tag{A.3.15}$$

where $\omega(i)$ is a one-hot vector encoding i. This invariant gives us a natural way to check acceptance in the new machine. We can translate the old acceptance function into a stateless version according to

$$F'(\mathbf{b} \| \omega(i)) = F(q_i, \mathbf{b}). \tag{A.3.16}$$

The counter update function in the new machine is slightly more complicated because it needs to deal with both counter and state updates, but we can use a similar trick. First, we define two functions u'_1 and u'_2 which respectively update the inherited counters and state counters:

$$u'_1(x, \mathbf{b} \| \omega(i)) = \mathbf{v} \iff u(x, q_i, \mathbf{b}) = v \tag{A.3.17}$$

and

$$u_2'(x,b\|\omega(i)) = -\omega(i)+\omega(j) \iff \delta(x,q_i,b) = q_j. \tag{A.3.18}$$

Then, we can define u' in terms of u_1' and u_2' according to

$$u'(\sigma,b\|\omega(i)) = u_1'(\sigma,b\|\omega(i)) \parallel u_2'(\sigma,b\|\omega(i)). \tag{A.3.19}$$

Note that the state vector updated by u_2' is a one-hot encoding of q_j because

$$\omega(i)+\big(-\omega(i)+\omega(j)\big) = \omega(j), \tag{A.3.20}$$

which implies that the invariant is preserved. Now, we have a stateless counter machine $M' = \langle \Sigma, k+|Q|, u', F' \rangle$ which simulates M.

□

A.4 Closure properties of counter classes

Theorem A.4.1 (General set operation closure)**.** CL *is closed under any n-ary set-theoretic operation whose result's characteristic function can be written as an n-ary boolean function*

$$\mathbb{1}_{L'}(\alpha) = p\big(\mathbb{1}_{L_1}(\alpha),..,\mathbb{1}_{L_n}(\alpha)\big).$$

Corollary A.4.1.1 (Complement closure)**.** CL *is closed under complement.*

Corollary A.4.1.2 (Intersection closure)**.** CL *is closed under intersection.*

Corollary A.4.1.3 (Union closure)**.** CL *is closed under union.*

Corollary A.4.1.4 (Set difference closure)**.** CL *is closed under set difference.*

Corollary A.4.1.5 (Symmetric difference closure)**.** CL *is closed under symmetric difference.*

Proof. Given finitely many counter machines $M_1,..,M_n$, I will construct M' which runs all the machines in parallel, and then accepts if p holds of the results. We can formalize this by saying that $q' \in Q_1 \times .. \times Q_n$ and $\mathbf{c}' \in \mathbb{Z}^{k_1 \times .. \times k_n}$. Let $q' = \langle q_1,..,q_n \rangle$ and analogously for $\mathbf{b}', \mathbf{c}', \mathbf{u}'$. We can write the update functions for the new machine as

$$\delta'(x,q',\mathbf{b}') = \langle \delta_1(x,q_1,\mathbf{b}_1),..,\delta_n(x,q_n,\mathbf{b}_n) \rangle \tag{A.4.1}$$

and

$$u'(x,q',\mathbf{b}') = \lambda \mathbf{c}'.\mathbf{u}_1(x,q_1,\mathbf{b}_1)\|..\|\mathbf{u}_n(x,q_n,\mathbf{b}_n). \tag{A.4.2}$$

Finally, we can write our acceptance mask in terms of p as

$$F'(q', \mathbf{b}') \iff p\big(F_1(q_1, \mathbf{b}_1), .., F_n(q_n, \mathbf{b}_n)\big). \tag{A.4.3}$$

□

Interestingly, all of these closure properties also apply to the simplified counter languages. This is because Theorem A.4.1 only relies on the structure of F. In other words, we can reformulate a construction in which u is only conditioned on x.

Appendix B

Linearly separable expressions

A linearly separable boolean expression is one where a hyperplane can be used to separate the true settings of variables from the false settings of variables. Since the focus of this work is on neural networks, we will give an equivalent definition in terms of a sigmoidal affine transformation:

Definition B.0.1 (Linearly separable expression)**.** An expression $\phi : \mathbf{X} \to \{0,1\}$ is linearly separable in $\mathbf{x}$ if and only if there exists $\mathbf{W}$ and $\mathbf{b}$ such that

$$\lim_{N\to\infty} \sigma\Big(N\big(\mathbf{W}\mathbf{x}+\mathbf{b}\big)\Big) = \mathbb{1}_{\phi}(\mathbf{x}).$$

It immediately follows from this definition that, if an expression is linearly separable in $\mathbf{x}$, then it is asymptotically computable by a single neural network layer whose input is $\mathbf{x}$.

B.1 Common linearly separable forms

Knowing whether an expression is linearly separable is useful for determining whether it can be computed in one neural network layer. Therefore, I will compile a list here of some forms that are known to be linearly separable. These facts are frequently referenced throughout my main results.

Theorem B.1.1 (Conjunction)**.** *The following formula is linearly separable in* $\mathbf{x}\|\mathbf{y}$*:*

$$\bigwedge_{i=1}^{n} x_i \wedge \bigwedge_{j=1}^{m} \neg y_j.$$

Proof. We pick a weight of N for each x_i, a weight of $-N$ for each y_i, and a bias of $-(n-\frac{1}{2})N$. Then, the form of the transformation is

$$\lim_{N\to\infty} \sigma\big(\sum_{i=1}^{n} Nx_i - \sum_{j=1}^{m} Ny_j - (n-\frac{1}{2})N\big),$$

which will be 1 only when all the x_i are 1 and none of the y_i are 1, and 0 otherwise. □

Theorem B.1.2 (Negation)**.** *Let $\phi(\mathbf{x})$ be a linearly separable form in* $\mathbf{x}$*. Then, the following form is linearly separable in* $\mathbf{x}$*:*

$$\neg\phi(\mathbf{x}).$$

Proof. Take an affine transformation for ϕ, and then take its additive inverse. □

Theorem B.1.3 (Disjunction)**.** *The following formula is linearly separable in* $\mathbf{x}\|\mathbf{y}$*:*

$$\bigvee_{i=1}^{n} x_i \vee \bigvee_{j=1}^{m} \neg y_j.$$

Proof. This form is linearly separable if its negation is linearly separable (B.1.2). Furthermore, since its negation is a conjunction of terms, we know that it is in fact linearly separable (B.1.1). □

Theorem B.1.4 (Disjunction and conjunction)**.** *The following formula is linearly separable in* $\mathbf{x}\|\mathbf{y}\|\mathbf{z}$*:*

$$\bigvee_{i=1}^{n} x_i \wedge \bigwedge_{j=1}^{m} y_j \wedge \bigwedge_{k=1}^{l} \neg z_k.$$

Proof. We pick a weight of N for each x_i, $(n+1)N$ for each y_i, $-(n+1)N$ for each z_i, and a bias of $\big((n+1)m+\frac{1}{2}\big)N$. Then, the form of the transformation is

$$\lim_{N\to\infty} \sigma\bigg(\sum_{i=1}^{n} Nx_i + \sum_{j=1}^{m} (n+1)Ny_j - \sum_{k=1}^{l} (n+1)Nz_l - \big((n+1)m+\frac{1}{2}\big)N\bigg).$$

To make this quantity equal to 1, we require all the y_j to be 1 and all the z_l to be zero, because, if not, all the positive mass from the x_i cannot exceed $nN < (n+1)N$. In addition, we require at least one of the x_i to be on to overcome the additional $\frac{N}{2}$ of the bias term. Otherwise, the sigmoid will come out to 0. □