Neural sequence models as automata computations

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

December 2018
Acknowledgements

Thanks to my advisors Bob and Dana 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. Thank you to everyone in Computational Linguistics at Yale (CLAY) for providing a space to think about NLP from a more theoretical angle. The foundations laid in CLAY paved the way for a lot of the developments in my thesis. I’d also like to thank all the members of the linguistics senior seminar and the advanced NLP seminar for helping me to consolidate my ideas. Any remaining errors are my own.
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, ConvNets, and other architecture by relating them to automata. In particular, I prove that the LSTMs have equivalent generative capacity to counter machines, and the convolutional networks sit somewhere in 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 us understand neural network computation, as well as the relationships between neural networks and natural language grammar.
# Table of contents

List of Theorems ix

1 Introduction 1

2 Recurrent neural networks 3
   2.1 SRNs ....................................................... 5
   2.2 LSTMs ..................................................... 6
      2.2.1 Upper bound on LSTM computation ................. 7
      2.2.2 Lower bound on LSTM computation ................. 8
      2.2.3 Experimental results ................................ 12
   2.3 GRUs ....................................................... 13

3 Other neural sequence models 15
   3.1 Convolutional networks ................................. 15
      3.1.1 Relation to subregular languages .................. 16
   3.2 Transformers ............................................. 18
      3.2.1 Automata-theoretic characterization ............... 19
   3.3 Stack recurrent networks ................................ 20

4 Rational recurrences 21
   4.1 WFSAs ...................................................... 21
   4.2 Simplified counter machines as rational recurrences .. 22
   4.3 General counter machines ................................ 23

5 Implications for natural language 25
   5.1 Semilinearity of counter languages ..................... 25
   5.2 Counter machines and context-free grammars ........... 27

References 31
## Table of contents

### Appendix A  Counter machines  
A.1 Definitions .......................... 34
A.1.1 The general counter machine .... 34
A.1.2 Counter machine variants ......... 35
A.2 Relationships between counter classes .............. 36
A.3 Closure properties of counter classes ................. 40

### Appendix B  Linearly separable expressions  
B.1 Common linearly separable forms ................ 41
List of Theorems

2.0.1 Definition (Neural sequence acceptor) ........................................ 3
2.0.2 Definition (Asymptotic computation) .......................................... 4
2.0.3 Definition (Asymptotic acceptance) ............................................. 4
2.0.4 Definition (Asymptotic configuration) ......................................... 4
2.0.5 Definition (State complexity) ....................................................... 4
2.0.1 Theorem (Upper bound on state complexity) ................................ 5
2.1.1 Theorem (SRN state complexity) ................................................ 5
2.2.1 Definition (LSTM) ................................................................. 6
2.2.1 Theorem (LSTM state complexity) ............................................. 7
2.2.2 Theorem (LSTM upper bound) ................................................... 8
2.2.3 Theorem (LSTM lower bound) ................................................... 8
2.2.2 Definition ................................................................................. 9
2.2.3 Definition ................................................................................. 9
2.2.3 Conjecture .............................................................................. 11
2.3.1 Definition (GRU) ................................................................. 13
2.3.1 Theorem (GRU state complexity) ............................................. 13
2.3.2 Theorem (GRU characterization) ............................................. 14
3.1.1 Definition (Convolutional language acceptor) ............................. 16
3.1.1 Theorem ................................................................................. 16
3.1.2 Definition (Strictly k-local grammar) ....................................... 17
3.1.3 Definition (Strictly local acceptance) ....................................... 17
3.1.4 Definition (\(SL_k\)) ............................................................... 17
3.1.2 Theorem ................................................................................. 17
3.2.1 Definition (Transformer) ........................................................... 19
3.2.1 Theorem ................................................................................. 19
3.3.1 Theorem (Neural stack state complexity) .................................. 20
<table>
<thead>
<tr>
<th>Theorem/Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1 Definition (Path score)</td>
<td>21</td>
</tr>
<tr>
<td>4.1.2 Definition (String score)</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1 Theorem</td>
<td>22</td>
</tr>
<tr>
<td>4.3.1 Conjecture</td>
<td>23</td>
</tr>
<tr>
<td>5.1.1 Definition (Semilinear set)</td>
<td>25</td>
</tr>
<tr>
<td>5.1.2 Definition (Parikh mapping)</td>
<td>25</td>
</tr>
<tr>
<td>5.1.3 Definition (Semi-linear language)</td>
<td>26</td>
</tr>
<tr>
<td>5.1.4 Definition (Stateless simplified counter languages)</td>
<td>26</td>
</tr>
<tr>
<td>5.1.1 Theorem</td>
<td>26</td>
</tr>
<tr>
<td>5.1.1 Conjecture</td>
<td>27</td>
</tr>
<tr>
<td>5.1.2 Conjecture</td>
<td>27</td>
</tr>
<tr>
<td>5.2.1 Definition ($L_n$)</td>
<td>28</td>
</tr>
<tr>
<td>5.2.1 Theorem</td>
<td>28</td>
</tr>
<tr>
<td>A.1.1 Definition (General counter machine)</td>
<td>34</td>
</tr>
<tr>
<td>A.1.2 Definition (Zero-check function)</td>
<td>34</td>
</tr>
<tr>
<td>A.1.3 Definition (Counter machine computation)</td>
<td>34</td>
</tr>
<tr>
<td>A.1.4 Definition (Real-time string acceptance)</td>
<td>35</td>
</tr>
<tr>
<td>A.1.5 Definition (Real-time language acceptance)</td>
<td>35</td>
</tr>
<tr>
<td>A.1.6 Definition (Counter languages)</td>
<td>35</td>
</tr>
<tr>
<td>A.1.7 Definition (Simplified counter machine)</td>
<td>35</td>
</tr>
<tr>
<td>A.1.8 Definition (Simplified counter languages)</td>
<td>35</td>
</tr>
<tr>
<td>A.1.9 Definition (Incremental counter machine)</td>
<td>36</td>
</tr>
<tr>
<td>A.1.10 Definition (Incremental counter languages)</td>
<td>36</td>
</tr>
<tr>
<td>A.1.11 Definition (Stateless counter machine)</td>
<td>36</td>
</tr>
<tr>
<td>A.1.12 Definition (Stateless counter languages)</td>
<td>36</td>
</tr>
<tr>
<td>A.2.1 Theorem</td>
<td>36</td>
</tr>
<tr>
<td>A.2.2 Theorem</td>
<td>37</td>
</tr>
<tr>
<td>A.2.3 Theorem</td>
<td>39</td>
</tr>
<tr>
<td>A.3.1 Theorem (General set operation closure)</td>
<td>40</td>
</tr>
<tr>
<td>B.0.1 Definition (Linearly separable expression)</td>
<td>41</td>
</tr>
<tr>
<td>B.1.1 Theorem (Conjunction)</td>
<td>41</td>
</tr>
<tr>
<td>B.1.2 Theorem (Negation)</td>
<td>42</td>
</tr>
<tr>
<td>B.1.3 Theorem (Disjunction)</td>
<td>42</td>
</tr>
<tr>
<td>B.1.4 Theorem</td>
<td>42</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years, neural networks have achieved tremendous success in 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. [16] 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 neural sequence models to forms of computation that are more well-understood. In theoretical computer science, the computational capacities of many different kinds of automata formalisms are well understood. 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. [23] propose a connection between LSTM computation 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. [17], 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 transformers and convolutional networks (chapter 3). Overall, this leads to a fairly complete automata-theoretic characterization of neural sequence models.

In chapter 4, I introduce the rational recurrences developed by Peng et al. [17], and I show that the simplified counter machines of Weiss et al. [23] are in fact rationally recurrent. This unifies two separate automata-theoretic analyses of recurrent neural networks that exist in the literature.

Finally, in chapter 5, I further discuss the implications of my results for the connection between neural networks systems and natural language. In particular, this chapter focuses on the relationship between counter machines and context-free grammars, as well as whether counter languages are semi-linear. Two appendices provide additional technical results about the counter machines (A) and linearly separable expressions (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.
Chapter 2

Recurrent neural networks

In this chapter, I consider the relationship between automata and various kinds of recurrent neural networks (RNNs). The most basic RNN is the simple recurrent network (SRN) [5]. Gated recurrent networks like the LSTM and GRU are a refinement of the simple recurrent architecture.

Whereas general RNNs are often described as Turing-complete [21], it turns out that this classical reduction relies on two very strong assumptions about RNN computation [23]. First, the number of recurrent computations must be unbounded in the length of the input, whereas, in practice, recurrent neural networks 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 restricts the class of formal languages that an RNN can accept. Thus, it is not an unreasonable question to ask whether a real-time, bounded precision LSTM can 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 my to derive upper and lower bounds on the types of languages they can accept. To do this, we need to introduce a notion of what it means for a neural network to accept a language. We define a neural sequence acceptor as follows:

**Definition 2.0.1** (Neural sequence acceptor). A neural sequence acceptor \( f(x_1, \ldots, x_n) \) is a computation graph unrolled in \( 1, \ldots, n \). Each node \( f \) can be evaluated at time \( t \), and its value \( f_t(x_1, \ldots, x_t) \) is either:

1. A one-hot input \( x_t \) drawn from a finite alphabet
2. A function of other nodes in the graph
A special node $\hat{f}^*$ with range $(0, 1)$ is taken as the output of the network.¹

Languages are discrete sets, whereas neural networks are vector-valued function approximators. I relate the two by considering the asymptotic case of neural network computation, in which the continuous output of a neural network approaches a discrete characteristic function for a language. I will formalize this idea as asymptotic acceptance (2.0.3) after building up a series of definitions.

**Definition 2.0.2 (Asymptotic computation).** A function $\hat{f}_\theta : X \to Y$ parameterized by $\theta$ asymptotically computes $f : X \to Y$ if

$$\lim_{N \to \infty} \hat{f}_N \theta = f.$$  

By writing the limit of $\hat{f}$, I mean the function which $\hat{f}$ converges to pointwise.² We now apply this idea to each layer of a neural network:

**Definition 2.0.3 (Asymptotic acceptance).** A neural network $\hat{f}$ asymptotically accepts a language $L$ if each node $\hat{f}$ asymptotically computes some $f$ with finite range, and the output node $\hat{f}^*$ asymptotically computes $1_L$.

By formulating acceptance in this way, we restrict the amount of precision that each logit in a 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.

This definition also gives us a notion of the memory capacity of a neural architecture of we measure the number of values that some node in the graph can achieve in the limit. Formally, we can write:

**Definition 2.0.4 (Asymptotic configuration).** Let $\hat{f}(x_1, \ldots, x_n)$ be a a neural sequence acceptor for $L$. An asymptotic configuration of a node $\hat{f}_n$ is a possible value of

$$\lim_{N \to \infty} \hat{f}_N(x_1, \ldots, x_n).$$

**Definition 2.0.5 (State complexity).** For a node $\hat{f} \in \hat{r}$, the state complexity $\mathcal{M}(\hat{f}_n)$ is the number of asymptotic configurations of $\hat{f}_n$.

In general, this complexity measure is at most exponential in the sequence length $n$. We will also see that specific architectures like the SRN (2.1.1) and LSTM (2.2.1) have more restricted bounds. I now prove the general bound:

¹Runic font provided by Werner [24].
Theorem 2.0.1 (Upper bound on state complexity). Let \( \hat{f}(x_1, \ldots, x_n) \) be a neural sequence acceptor for \( L \). For each node \( \hat{f}_n \),

\[
\mathbb{M}(\hat{f}_n) = 2^{O(n)}.
\]

Proof. The number of configurations of \( \hat{f}_n \) cannot be more than the number of distinct inputs. By construction, each \( x \) is a one-hot vector over a finite alphabet. Thus, the number of configurations is

\[
O(|\Sigma|^n) = 2^{O(n)}.
\]

Armed with the concept of asymptotic acceptance, we can go on to consider what kinds of languages different kinds of recurrent neural networks are capable of accepting. I will also show how we can derive the state complexity of different architectures and use it to determine what computation they are capable of.

2.1 SRNs

SRNs are the simplest variant of recurrent neural networks. We make the hidden layer recurrent by simply including the output at the previous time step in a standard affine transformation [5]. This can be written as

\[
h_t = \tanh(W x_t + U h_{t-1} + b).
\]

A well-known problem with SRNs is that they struggle with long-term dependencies. One explanation of this is the vanishing gradient problem, which motivated the development of more sophisticated architectures like LSTMs [11]. 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 is unbounded [23]. 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). The SRN cell state \( h_n \in \mathbb{R}^k \) has state complexity

\[
\mathbb{M}(h_n) \leq 2^k = O(1).
\]
Corollary 2.1.1. Let \( \text{RL} \) denote the regular languages. Then,

\[
L(\text{SRN}) \subseteq \text{RL}.
\]

Proof. For every \( t \), each unit of \( h_t \) 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 \(2^k\).

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 \[11\]. Due to strong empirical performance on many language tasks, LSTMs have become the canonical model for NLP.

Interestingly, Weiss et al. \[23\] suggest that another way to understand the success of the LSTM architecture is that they are expressive enough to accept the 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 in LSTMs. In particular, I will derive upper bounds on what kinds of computation LSTMs can perform. Building on Weiss et al. \[23\], I will also provide a constructive reduction from the simplified counter machines to the LSTMs. Together, these two results suggest that the generative capacity of LSTMs is essentially equivalent to that of some class of counter machines.

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

Definition 2.2.1 (LSTM).

\[
i_t = \sigma(W^i x_t + U^i h_{t-1} + b^i) \tag{2.2}
\]

\[
f_t = \sigma(W^f x_t + U^f h_{t-1} + b^f) \tag{2.3}
\]

\[
o_t = \sigma(W^o x_t + U^o h_{t-1} + b^o) \tag{2.4}
\]

\[
\tilde{c}_t = \tanh(W^c x_t + U^c h_{t-1} + b^c) \tag{2.5}
\]
2.2 LSTMs

\[ c_t = i_t \odot c_{t-1} + f_t \odot \tilde{c}_t \]  
\[ h_t = o_t \odot f(c_t) \]

(2.6) \hspace{1cm} (2.7)

In the last equation, we can let \( f \) be either the identity or \( \tanh \). For simplicity, I will only consider the case where \( f = \tanh \). We will refer to \( c_t \) as the network’s hidden cell state, and \( h_t \) is the output that gets passed up to the next layer at the same time step. Both of these vectors are also copied and fed into the hidden unit computation at the next time step.

2.2.1 Upper bound on LSTM computation

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

\[ \mathbb{M}(c_n) = O(n^k). \]

**Proof.** At each time step \( t \), we know that \( i_t, f_t, o_t \in \{0,1\}^k \) and \( \tilde{c}_t \in \{-1,1\}^k \). This allows us to rewrite the elementwise recurrent update as

\[ [c_t]_i = i_t[c_{t-1}]_i + f_t[\tilde{c}_t]_i \]
\[ \implies [c_t]_i = a[c_{t-1}]_i + b \]

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

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

\[ (\arg \min_{s \in S_{t-1}} s) - 1 \]  
\[ (\arg \max_{s \in S_{t-1}} s) + 1. \]

(2.10) \hspace{1cm} (2.11)

It follows that

\[ |S_t| = 2 + |S_{t-1}| \]
\[ \implies |S_n| = O(n) \]
\[ \therefore |S_n|^k = O(n^k). \]  

(2.12) \hspace{1cm} (2.13) \hspace{1cm} (2.14)
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,

$$L(LSTM) \subseteq CL.$$  

*Proof.* The machine that we construct in theorem 2.2.1 takes the form of a general counter machine whose update function is constrained to be linearly separable. This implies that any LSTM-acceptable language is acceptable by a general counter machine. 

Whereas Weiss et al. [23] argued for a lower bound on LSTM computation, theorem 2.2.2 constitutes the first strict upper bound on LSTM computation in the literature. It implies that LSTMs are not powerful enough to deal with even simple context-free languages like $w\#w^R$.

### 2.2.2 Lower bound on LSTM computation

I will prove that the simplified counter languages are a lower bound on the LSTM-acceptable languages. Weiss et al. [23] sketch an argument for how an LSTM can simulate a simplified counter machine. In our terms, they argue that the LSTM cell states can asymptotically compute the counter values of a counter machine.

They also implicitly assert that the state can be encoded in recurrent cell state of the LSTM alongside the counter values. While this is true, I found a construction of this far from trivial due to the fact that the conditions for updating the state must be conditioned by three variables while being computed in a very constrained gating mechanism. Its form prevents a straightforward construction where state is encoded and updated as a one-hot vector.

I provide an alternative construction below. This construction has memory overhead dependent on the value of the transition function $\delta$. In the worst case, the memory overhead is exponential in $k$. It seems like the representation might also allow for the simulation of a complex counter update function.

These proofs follows Weiss et al. [23] in assuming that $h_t = o_t \odot c_t$. I conjecture that the same construction still works when we introduce a tanh nonlinearity, but the analysis is made more complicated. For more technical background on counter machines, refer to appendix A.

**Theorem 2.2.3** (LSTM lower bound). Let $SCL$ be the simplified counter languages, and $2LSTM$ be an LSTM architecture with two feedforward layers after the LSTM layer. Then,

$$SCL \subseteq L(2LSTM).$$
I will demonstrate this construction through a series of three interconnected lemmas.

**Lemma 2.2.3.1.** We can augment the LSTM with cells whose exposed states asymptotically compute \( \neg z(c_t) \).

**Proof.** Let \( c'_t \) and \( h'_t \) represent the augmented cell states and exposed states respectively. The cell state update is given by

\[
c'_t = i_t \odot c'_{t-1} + f_t \odot \tilde{c}'_t.
\]

We will set \( i_t = \vec{0} \) and \( \tilde{c}'_t = \vec{1} \), which reduces our equation to

\[
c'_t = f_t = \sigma(Wx_t + U(c_{t-1}|c'_{t-1}) + b).
\]

Now, we zero out the connections to \( c'_{t-1} \). We also decompose \( W \) into \( UW' \) to get

\[
c'_t = \sigma(UW'x_t + Uc_{t-1} + b).
\]

We now set \( W' \) such that \( W'x_t = u(x_t) \). This leaves

\[
c'_t = \sigma(U(u(x_t) + c_{t-1}) + b) = \sigma(Uc_t + b).
\]

We will now parameterize the weights of this affine transformation in \( N \) such that its limit is the following step function:

\[
\lim_{N \to \infty} c'_t = 1_{x<1}(c_t).
\]

Now, we apply the same construction to \( o_t \) that we did to \( f_t \), except that we choose a sigmoidal curve that approaches \( 1_{x>-1}(c_t) \). Once we have these two limits, we can find the value of the exposed state:

\[
\lim_{N \to \infty} h'_t = \lim_{N \to \infty} o_t \odot f_t = 1_{x<1}(c_t) \odot 1_{x>-1}(c_t) = \neg z(c_t).
\]

Next, we will show state can be encoded in the machine. To do this, we will first introduce some logical predicates applying to a counter machine that let us read off what state the machine is in.

**Definition 2.2.2.** Let \( q_t(i) \) be true of a counter machine if it is in state \( i \) at time \( t \).

**Definition 2.2.3.** Define \( \bar{q}_t(i, \alpha, \tilde{b}) \) such that
\( \bar{\vartheta}_t(i, \alpha, \vec{b}) \iff x_t = \alpha \land q_{t-1}(i) \land z(c_{t-1}) = \vec{b}. \)

Intuitively, \( \bar{\vartheta}_t(i, \alpha, \vec{b}) \) represents whether the machine could have been in state \( i \) and then encountered a certain input and memory configuration. This is inherently connected to the current state, which gives us, by construction, the following fact:

\[
q_t(j) \iff \bigvee_{(i, \alpha, \vec{b}) \in \delta^{-1}(j)} \bar{\vartheta}_t(i, \alpha, \vec{b}). \tag{2.15}
\]

Therefore, knowing the value of \( \bar{\vartheta} \) at any time steps lets us recover the recurrent state. Thus, if we can compute \( \bar{\vartheta}_t \) in the recurrent layer of our LSTM, the following layers will be able to recover the state of the machine. I will now demonstrate a construction that allows us to do this.

**Lemma 2.2.3.2.** We can augment an LSTM with a vector of cells that asymptotically compute \( \bar{\vartheta}_t \).

**Proof.** Let \( d_t \) be the hidden state of our augmented vector. We ignore all weights in the gated computation of \( d_t \) besides the corresponding \( o_t \) and \( f_t \) vectors, which reduces the equation for \( d_t \) to

\[
d_t = o_t \odot f_t.
\]

Now we split the conditions for \( \bar{\vartheta}_t \) into the components

\[
\bar{\vartheta}_t^1(i) \iff q_{t-1}(i) \tag{2.16}
\]

and

\[
\bar{\vartheta}_t^2(\alpha, \vec{b}) \iff x_t = \alpha \land z(c_{t-1}) = \vec{b}. \tag{2.17}
\]

Observe that, if \( f_t \) asymptotically computes \( 1_{\bar{\vartheta}_t^1} \) and \( o_t \) asymptotically computes \( 1_{\bar{\vartheta}_t^2} \), we are left with

\[
\lim_{N \to \infty} d_t = \lim_{N \to \infty} o_t \odot f_t = 1_{\bar{\vartheta}_t^1} \odot 1_{\bar{\vartheta}_t^2} = 1_{\bar{\vartheta}_t^1} \land 1_{\bar{\vartheta}_t^2} = 1_{\bar{\vartheta}_t}.
\]

Thus, we just need to provide constructions for \( f_t \) and \( o_t \). We start with \( f_t \). In the base case, we want

\[
\bar{\vartheta}_t^1(i) \iff i = 0. \tag{2.19}
\]
On the other hand, in the recurrent case, we can rewrite the form of \( \hat{\delta}_t^1 \) by unrolling \( q_{t-1} \) to get

\[
\hat{\delta}_t^1(i) \iff \bigvee_{\langle j, \alpha, \vec{b} \rangle \in \delta^{-1}(i)} \hat{\delta}_{t-1}(j, \alpha, \vec{b}). \tag{2.20}
\]

Each term in this disjunction \( [\hat{\delta}_{t-1}]_i \) is, by inductive assumption, already computed in \( [d_{t-1}]_i \). Therefore \( \hat{\delta}_t^1 \) is linearly separable in \( \hat{\delta}_{t-1} \) (B.1.3), and we can pick \( f_t \) to asymptotically compute it. In order to incorporate the base case, we can add an additional counter \( s \) to the machine with starts out at zero and then remains at 1 for the rest of the computation. We then add \( \neg s_t \) to the disjunction for \( \hat{\delta}_t^1(0) \). In the base case, all the previous \( \hat{\delta} \) terms will be zero, so the expression reduces to equation 2.19. In the recurrent case, we know that \( \neg s_t = 0 \), so we get equation 2.20.

Now, we will move on to computing \( \hat{\delta}_t^2 \) with \( o_t \). First, we can expand

\[
z(c_{t-1}) = \vec{b} \iff \left( \bigwedge_{\{i : b_i = 0\}} [h'_{t-1}]_i \right) \land \left( \bigwedge_{\{i : b_i = 1\}} \neg [h'_{t-1}]_i \right) \tag{2.21}
\]

where \( h'_{t-1} \) is the zero-check counter from the previous lemma. Note that this expression represents \( \hat{\delta}_t^2 \) as a conjunction of positive and negative terms which are already computable. Thus, \( \hat{\delta}_t^2 \) is linearly separable (B.1.1), so we can pick \( o_t \) to compute it.

**Lemma 2.2.3.3.** We can asymptotically compute an acceptance decision from \( \hat{\delta}_t \) in two feedforward layers.

**Proof.** By definition, we accept at time step \( t \) if we are in an accepting configuration. We can write this logically as

\[
a_t \iff \bigvee_{(j, \vec{b}) \in F} (q_t(j) \land z(c_t) = \vec{b}). \tag{2.22}
\]

This expression can be computed from the exposed LSTM state \( h_t \) in two feedforward layers.

These three lemmas suffice to prove theorem 2.2.3. I conjecture that this reduction would also would for an LSTM with only one final feedforward layer, although the reduction would have to be more complicated. This gives the following conjecture:

**Conjecture 2.2.1.**

\[
\text{SCL} \subseteq L(\text{LSTM}).
\]

Either way, the construction in 2.2.3 shows that an LSTM is capable of encoding all the relevant state for simulating a simplified counter machine.
2.2.3 Experimental results

Empirical evidence confirms the theoretical result that counter languages beyond the simplified ones are LSTM-acceptable. Weiss et al. [23] 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 $a^n b 2a$ in a counter-like fashion. In figure 2.1, we see that the cyan cell is incremented by $2\delta$ for an $a$ and decremented by $\delta$ for a $b$. This is not be achievable in a simplified counter machine where we can only add or subtract 1. In figure 2.2, 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 conditioned being conditioned by state.

Other empirical results serve as a sanity check for the theoretical notion of state complexity. Reversing a string losslessly requires a machine to have an exponential number of possible state configurations. On the other hand, LSTMs only have polynomial many states. In figure 2.3, we see that an LSTM fails as expected on the simple task of reversing long binary strings.
2.3 GRUs

GRUs are a popular gated recurrent neural network architecture that are in many ways similar to LSTMs [3]. Rather than having both an include and forget gate, the GRU has a single gate which, along with its complement, modules both the ability to include and forget:

**Definition 2.3.1 (GRU).**

\[
\begin{align*}
    z_t & = \sigma(W^z x_t + U^z h_{t-1} + b^z) \\
    r_t & = \sigma(W^r x_t + U^r h_{t-1} + b^r) \\
    u_t & = \tanh(W^u x_t + U^u (r_t \odot h_{t-1}) + b^u) \\
    h_t & = z_t \odot h_{t-1} + (1 - z_t) \odot u_t.
\end{align*}
\]  

Weiss et al. [23] show empirically and analytically how this architectural difference prevents a GRU from simulating a counter machine like an LSTM can. Similarly, it turns out 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

\[ \mathcal{M}(h_n) = O(1). \]

**Proof.** As with the LSTM, the range of \( z_t \) converges to \( \{0, 1\} \). Thus, we have two possibilities for each value of \( [h_t]_i \): either \( [h_{t-1}]_i \) or \( [u_t]_i \). Let \( S_t \) be the set of values that \( [h_{t-1}]_i \) can attain. We can write

\[ S_t = S_{t-1} \cup \{-1, 1\}. \]
This implies that there are only three possible values for each logit: $-1$, $0$, or $1$. Thus, the number of state configurations of $h_n$ is

$$\mathbf{M}(h_n) \leq 3^k = O(1). \quad (2.28)$$

Using this theorem, we can also 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 $\bar{\sigma}$ construction similar to the one in 2.2.3. For more detail, refer to that proof. I start by defining

$$\bar{\sigma}_t(i, \alpha) \iff q_{t-1}(i) \land x_t = \alpha. \quad (2.29)$$

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

$$\bar{\sigma}_t(i, \alpha) \iff x_t = \alpha \land \bigvee_{(j, \beta) \in \delta^{-1}(i)} \bar{\sigma}_{t-1}(j, \beta). \quad (2.30)$$

This formula is linearly separable in $x_t | \bar{\sigma}_{t-1}$ (B.1.4). Therefore, we can store $\bar{\sigma}_t$ in our hidden state $h_t$ and recurrently compute its update. Then, in a final feedforward layer, we can compute whether we are in an accepting state from the value of $\bar{\sigma}_t$:

$$a_t \iff \bigvee_{i \in F} \bigvee_{(j, \beta) \in \delta^{-1}(i)} \bar{\sigma}_t(j, \beta). \quad (2.31)$$

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

\[\square\]
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 [22]. These models tend to be used in specialized contexts, such as modeling subword information in the case of convolutional networks [13], or machine translation, in the case of transformers [22]. Using the notion of acceptance developed in the last chapter (2.0.3), we can also reason about what kinds of computation these models are capable of from the point of view of formal language theory.

3.1 Convolutional networks

While convolutional networks were originally developed with other purposes in mind [14], they can be use to process variable-length sequences. One popular application of this is to build character-level representations of words [13]. Another example of this is the capsule network architecture of Zhao et al. [25], 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, since there are no recurrent connections, it would ignore all computation besides the last time step. Therefore, we should redefine our acceptance criterion in terms of the whole vector of outputs at each time step.

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. For simplicity, I will consider only the case where we pick the logical function AND. This gives us the following \( k \)-convolutional architecture for a language acceptor:

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

\[
    h_t = W^h(x_{t-k} \cdots x_{t+k}) \tag{3.1}
\]

\[
    y_t = \sigma(W^y h_t + b^y) \tag{3.2}
\]

\[
    a = \prod_{t=1}^{n} y_t \tag{3.3}
\]

In this model, the initial \( k \)-convolutional layer (3.1) produces a vector-valued sequence of outputs. Then, the feedforward layer (3.2) converts this to a sequence of scalars. Finally, this is reduced through multiplication to an acceptance decision (3.3).

### 3.1.1 Relation to subregular languages

One of the most striking things about convolutional language acceptors defined this way is that they are substantially computationally weaker than LSTMs. Right away, we can see that \( L(\text{CNN}) \subseteq \text{RL} \). This is because both state vectors \( h_t \) and \( y_t \) in a convolutional network are restricted to having 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.** \( b^* ab^* \) is not asymptotically acceptable by a convolutional language acceptor.

**Corollary 3.1.1.1** (Convolutional network upper bound).

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

**Proof.** By contradiction. Assume we can write a network that accepts any string with exactly one \( a \) and reject any other string. Then, \( y_t = \vec{1} \) for any string with exactly one \( a \). Consider a string with two \( a \)s at indices \( i \) and \( j \) where \( j - i > 2k + 1 \). Then, there are no columns in the network which receive both \( x_i \) and \( x_j \) as input. But because the network does not accept the string, there must be some index \( k \) such that \( y_k = 0 \). When we replace one \( a \) with a \( b \), then we get a string with exactly one \( a \), but still \( y_k = 0 \).

Thus, to arrive at a characterization of what convolutional sequence acceptors can do, we should move to sub-regular classes of languages. In particular, we will consider the strictly local languages [19], which can be defined as follows:
Definition 3.1.2 (Strictly k-local grammar). A strictly k-local grammar over an alphabet \( \Sigma \) is a set of constraints \( S \) where each \( s \in S \) takes the form

\[
s \in (\Sigma \cup \{\#\})^k
\]

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 \( \sigma \) if, at each index \( i \),

\[
\sigma_i \sigma_{i+1} \ldots \sigma_{i+k-1} \in S.
\]

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. A k-convolutional network can asymptotically accept any strictly \( 2k + 1 \)-local language.

Corollary 3.1.2.1 (Convolutional network lower bound).

\[
\text{SL} \subseteq L(\text{CNN}).
\]

Proof. We can think about the convolutional column at time step \( t \) as verifying whether the \( k \)-gram at position \( t \) is valid.

In the convolutional layer (3.1), each filter will identify whether a particular \( k \)-gram is matched. First, each convolutional filter assigns a positive weight of \( N \) to each valid \([x_i]_i\), and \(-(n + 1)N\) to each invalid \([x_i]_i\). This ensures that the value of each filter will be slightly positive if the \( k \)-gram is matched, and very negative otherwise.

Next, the feedforward layer (3.2) sums over all the filters. This sum will come out to be positive if all of the filters are on. Otherwise, the one filter that is off will dominate, and the sum will come out to be negative. Thus, \( y_t \) is a boolean indicator for local validity.

Finally, the \( \land \)-reduction in the final layer ensures that each boolean \( y_t \) is true. Thus, a \( k \)-convolutional network can simulate the computation of a strictly \( 2k + 1 \)-local grammar.

Interestingly, the tier-based strictly local languages have been proposed as a computational model for natural language phonological grammar [10]. Tier-based strictly local languages are very similar to strictly local languages, except that the local patterns can apply to all
characters in a specific tier of the vocabulary (e.g., vowels) instead of over the full string. In
the field of NLP, convolutional networks have been used to model character-level information
within words [13]. Theorem 3.1.2 provides a theoretical explanation about why this works:
convolutional networks pick up on strictly local dependencies that are similar to those
employed by natural-language phonology.

3.2 Transformers

An attention function can be defined as a mapping from a query vector and paired key-value
vectors to a weighted combination of the values [22]. This output is meant to incorporate
the values that are relevant to the query. For example, we can define a standard dot-product
attention function as

$$\text{attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V.$$  \hspace{1cm} (3.4)

This function gives us a new vector for each query in the matrix $Q$. The vector that is
outputted is a sum of the values in $V$ weighted by the similarity of the corresponding key to
the query.

Attention mechanisms were originally used in sequence-to-sequence 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. Another advantage of adding an attention
mechanism to a recurrent network is that introduces unbounded 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
an attention mechanism is integral for sequence transductions tasks like machine translation
(2.2.1).

Another way of using attention is self-attention, where a sequence attends over itself.
Self-attention has been used in tasks like abstractive summarization [20].

The transformer architecture developed by Vaswani et al. [22] is motivated by the claim
that "attention is all you need". In other words, they replace the recurrent connections in
an encoder network with self attention over the layer. Additionally, they perform multiple
separate instances of these attention heads in parallel, and then concatenate the outputted
vectors. More formally, we can write a mult-head attention layer with $d$ heads as

$$h_i = \text{attention}(W_i^Q Q, W_i^K K, W_i^V V)$$  \hspace{1cm} (3.5)
multihead(Q, K, V) = W^o(h_1∥..∥h_d). \quad (3.6)

The network proposed by Vaswani et al. [22] uses an encoder with self-attention and a decoder that attends over the output of the encoder. Further work has also developed a simplified architecture with a single self-attention module [15, 18]. Radford et al. [18] 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 adapt the single-layer version of this "OpenAI Transformer" as the variant which I consider in my analysis:

**Definition 3.2.1** (Transformer).

\[
\langle k_t, v_t \rangle = W^x x_t \quad (3.7)
\]

\[
h_t = \sigma(\text{multihead}(v_t, K, V)) \quad (3.8)
\]

\[
y_t = \sigma(W^y h_t + b^y) \quad (3.9)
\]

\[
a_t = \sigma(W^a y_t + b^a). \quad (3.10)
\]

One key difference between this model and the model of Radford et al. [18] is that the multihead attention here is not masked. This is because unmasked attention trivially solves the language modeling task, whereas it does not do so for language acceptance. Therefore, I do not make this additional restriction.

### 3.2.1 Automata-theoretic characterization

It has been documented that transformers have difficulty learning positional dependencies without the augmentation of special positional encodings [22]. This motivates the following result:

**Theorem 3.2.1.** A transformer cannot asymptotically accept the language \( ab^n \).

**Proof.** Assume we can accept a string \( ab^n \) for some \( n \). Then, we can swap \( a \) with some random \( b \), and the state of \( h_t \) will be unchanged. Thus, we will accept some string which is not in the language, which means we reach a contradiction. \( \square \)

**Corollary 3.2.1.1.** A transformer cannot accept all regular languages.

Interestingly, this means that there is a language which is acceptable by an LSTM but not by a transformer.
3.3 Stack recurrent networks

One way to make an RNN closer to the context-free grammar is to construct a differentiable pushdown automaton [8, 9]. 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.

Hao et al. [9] show how a stack RNN can solve a variety of formal language tasks. Additionally, the structured memory mechanism allows for interpretability of the algorithm that a stack RNN is learning. More work should be done applying stack neural networks to natural language data to determine whether this type of architecture is practically viable. I have an implementation of the stack neural network architecture that is publically available.\(^1\)

I now demonstrate that a neural stack connected to a controller does not suffer from the same memory constraints as an LSTM. In fact, an asymptotic Stack-NN has exponentially many configurations. Refer to Hao et al. [9] for architectural details.

**Theorem 3.3.1** (Neural stack state complexity). *Let \( S \) be a neural stack with a feedforward controller. Then,*

\[
\mathcal{M}(S_n) = \Omega(2^n).
\]

*Proof.* The stack at time step \( n \) is a matrix \( S \in \mathbb{R}^{nk} \) where each row corresponds to a vector that has been pushed on at each time step. Since the vector that is pushed onto the stack at time \( t \) is a function of \( x_t \) 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 \( \Omega(2^n) \). \( \square \)

---

\(^1\)https://github.com/viking-sudo-rm/StackNN.
Chapter 4

Rational recurrences

Peng et al. [17] introduce the term "rational recurrence" to describe an RNN recurrent update that can be computed elementwise by series of weighted finite-state automata (WFSAs). In recurrent neural networks, the gate update function is expressed as a recurrence

\[ c_t = f(x_t, c_{t-1}). \] (4.1)

For example, in a simple RNN [5], the gate update takes the form

\[ c_t = \tanh(W x_t + U c_{t-1} + b). \] (4.2)

It is worth pointing out that, if we unroll the computation graph of the network, this recurrence becomes a function of the variable-length input prefix \( x_1, \ldots, x_t \). Thus, we will consider recurrent update function to be a vector-valued function of the form \( c : \Sigma^* \to \mathbb{K}^d \). This kind of function, which takes a variable-length sequence as input, is exactly the type of object that can be computed by a WFSA.

4.1 WFSAs

Formally, a WFSA is a non-deterministic automaton where each transition receives a weight [17]. 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 4.1.1** (Path score). The score of a path \( \pi_1, \ldots, \pi_t \) is given by

\[ A[\pi] = \lambda(q_1) \otimes \left( \bigotimes_{i=1}^{t} \tau(\pi_i) \right) \otimes \rho(q_{t+1}). \]
Intuitively, $\tau$ is a function which gives us the score of each transition. Similarly, $\lambda$ gives us the score of starting in each state, and $\rho$ 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 4.1.2** (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.

### 4.2 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. Interestingly, the gating mechanism which dictates how the counters are updated turns out to be a rational recurrence.

**Theorem 4.2.1.** A simplified counter machine is rationally recurrent.

**Proof.** Let $c_t$ be the value of the counters at time $t$. We will now parameterize the counter operations as

$$c_t = r(x_t)c_{t-1} + u(x_t). \quad (4.3)$$

This parameterization allows us to express all of the valid update operations. For $-1$, $+0$, and $+1$, we set $r(x_t) = 1$, and $u(x_t)$ to $-1$, $0$, and $1$ respectively. For $\times 0$, we set $u(x_t) = r(x_t) = 0$.

Next, we can unroll this recurrence in time to get

$$c_t = \sum_{i=1}^{t} \left( \prod_{j=i+1}^{t} r(x_j) \right) u(x_i). \quad (4.4)$$

Each element of this vector is computed by a WFSA of the form:

```
q0 \rightarrow q1
```

- $q_0$ on $x_t : 1$
- $q_1$ on $x_t : u(x_t)$
- Transition $x_t : r(x_t)$
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 $\rho(q_0) = 0$.

\section*{4.3 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:

\begin{equation}
    c_t = r(x_t, z(c_{t-1}))c_{t-1} + u(x_t, z(c_{t-1}))
\end{equation}

\begin{equation}
    \Rightarrow c_t = \sum_{i=1}^{t} \left( \prod_{j=i+1}^{t} r(x_j, z(c_{j-1})) \right) u(x_i, z(c_{i-1})).
\end{equation}

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

Peng et al. [17] notes an analogous problem in reducing the LSTM gate update to a rational recurrence. In their case, the fact that $\tilde{c}$ depends on $h_{t-1}$ 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. [17] to conjecture that the LSTM is not rationally recurrent, I conjecture that the general counter machine is not rationally recurrent:

\textbf{Conjecture 4.3.1.} A general counter machine is not rationally recurrent.

An interesting implication of Peng et al. [17]'s conjecture that LSTMs are not rationally recurrent is that the simplified counter machines are strictly weaker than LSTM computation. This is consistent with empirical examples of languages that an LSTM can model which a simplified counter machine cannot (see section 2.2.3). In this case, we ought to look for a more complex class of counter machines, such as the general counter machines, to model LSTM computation.
Chapter 5

Implications for natural language

Now that we have defined and studied the counter machine formalism, we can ask how counter machines relate to formalisms 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. I will also touch on mildly context-sensitive grammar formalisms.

5.1 Semilinearity of counter languages

Semilinearity is a condition that has been proposed as a desired property for any formalism of natural language syntax [12]. 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. I will now define this concepts one at a time to build up the definition of a semilinear language.

**Definition 5.1.1 (Semilinear set).** A set $S \subseteq \mathbb{Z}^n$ is semilinear if it can be written as the finite union of sets of the form

$$\{wx + b = 0 \mid x \in \mathbb{Z}^n\}.$$

**Definition 5.1.2 (Parikh mapping).** The Parikh mapping of a language $L$ is the set

$$\Psi(L) = \{(\#(\sigma_1, x), \ldots, \#(\sigma_n, x)) \mid x \in L\}.$$

In machine learning terms, we might describe the Parikh mapping of a language as its bag-of-words representation. By translating languages into vector spaces, the Parikh mapping allows us to define the semilinear languages.
Definition 5.1.3 (Semi-linear 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 [12]. 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 can will present a proof for the stateless simplified counter languages.

Definition 5.1.4 (Stateless simplified counter languages). Let $\tilde{\text{QSCL}}$ be the class of languages acceptable by a simplified counter machine with only one state.

Theorem 5.1.1. $L \in \tilde{\text{QSCL}}$ is semilinear.

Proof. We can start by expressing $L$ as

$$L = \bigcup_{b \in F} \{ x \mid c_n(x) = b \}. \quad (5.1)$$

Since semilinear languages are closed under finite union, $L$ is semilinear if each of the following sets is semilinear:

$$L_b = \{ x \mid c_n(x) = b \}. \quad (5.2)$$

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:

$$L_b(i) = \{ x \mid [c_n]_i(x) = b_i \}. \quad (5.3)$$

Now, consider each counter in a machine accepting $L_b(i)$. I claim that, if counter $i$ can be zeroed, then $L_b(i)$ is semilinear. To show this, consider the suffixes of every string in the language after the last occurrence of a character which resets counter $i$. There are two cases:

1. There exists such a suffix $\omega$. Then, we can prepend any string to $\omega$ and the resulting string will be in $L_b(i)$. This means that

$$\Psi(L_b(i)) = \mathbb{Z}^{[\Sigma]}, \quad (5.4)$$

which is a semilinear set.

2. There does not exist such a suffix $\omega$. Then there is no string in the language, so

$$\Psi(L_b(i)) = \emptyset, \quad (5.5)$$
which is also a semilinear set.

All that remains to be proven is that, if counter \( i \) cannot be zeroed, \( L_{b}(i) \) is still semilinear. Since counter \( i \) cannot be reset, we can write

\[
b_{i} = [c_{n}](x) = \sum_{t=1}^{n} u_{i}(x_{t}) = \sum_{\sigma \in \Sigma} \#(\sigma, x) u_{i}(\sigma).
\]

Note that the right half of this equation parameterizes \( \mathbb{Z}^{\Sigma} \). When \( b = 0 \), we target a hyperplane subset of \( \mathbb{Z}^{\Sigma} \), 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 semilinear sets. Therefore, \( L_{b}(i) \) is always 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 extend to SCL, or possibly also to CL.

**Conjecture 5.1.1.** \( L \in \text{SCL} \) is semilinear.

**Conjecture 5.1.2.** \( L \in \text{CL} \) is semilinear.

### 5.2 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 [23].

A surprising classical result is that the language of well-formed preorder expressions is real-time acceptable [7] 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 5.2.1** \((L_n)\). Let \(L_n\) be the language generated by the grammar:

\[
\begin{align*}
  \langle \text{exp} \rangle & \rightarrow \langle \text{VALUE} \rangle \\
  \langle \text{exp} \rangle & \rightarrow \langle \text{UNARY\_OP} \rangle \ \langle \text{exp} \rangle \\
  \langle \text{exp} \rangle & \rightarrow \langle \text{BINARY\_OP} \rangle \ \langle \text{exp} \rangle \ \langle \text{exp} \rangle \\
  \langle \text{exp} \rangle & \rightarrow \langle \text{n\text{-ARY\_OP}} \rangle \ \langle \text{exp} \rangle \ldots \ \langle \text{exp} \rangle 
\end{align*}
\]

Fischer et al. [7]'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 5.2.1.** A real-time \(k\)-counter transducer cannot evaluate preorder 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 \(|Q|^nk\) configurations. I will show by induction that an \(n\)-length prefix of operators can encode \(2^n\) boolean functions. Since \(|Q|^nk < 2^n\) 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 \(x_1, \ldots, x_{n+1}\) followed by symbols \(x_{n+2}, \ldots, x_l\). 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

\[
\llbracket x_1, \ldots, x_{n+1} \rrbracket = \llbracket x_1 \rrbracket (\llbracket x_2, \ldots, x_l-1 \rrbracket, \llbracket x_l \rrbracket).
\] (5.7)

Observe that \(x_2, \ldots, x_{l-1}\) is a prefix of length \(n\). Thus, by inductive hypothesis, \(\llbracket x_2, \ldots, x_{n+1} \rrbracket\) could be one of \(2^n\) possible functions. The compositional relationship in (5.7) introduces a new variable into all of these possible functions, so we get two new functions in \(\llbracket x_1, \ldots, x_{n+1} \rrbracket\) by fixing \(x_1\):
5.2 Counter machines and context-free grammars

\[ f_\wedge = \left[ \land \right] \left( \left[ x_2, \ldots, x_{l-1} \right], \left[ x_l \right] \right) = \left[ x_2, \ldots, x_{l-1} \right] \land \left[ x_l \right] \]  
(5.8)

and

\[ f_\lor = \left[ \lor \right] \left( \left[ x_2, \ldots, x_{l-1} \right], \left[ x_l \right] \right) = \left[ x_2, \ldots, x_{l-1} \right] \lor \left[ x_l \right]. \]  
(5.9)

We can verify that \( f_\wedge \) and \( f_\lor \) 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 \( f_\lor \) it will end in a 0. Therefore, we are introducing exactly two new functions for each \( f \), which means an \( n + 1 \)-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 unbounded memory, since its automata-theoretic counterpart is a pushdown automaton.
References


Appendix A

Counter machines

Informally, counter machines are a class of automata that can use a finite number of integer variables as memory. This is similar in some ways to the abacus machine [2].

Early results in theoretical computer science established that a 2-counter machine with unbounded computation time is Turing-complete [6]. 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. Interestingly, we will see that this is also true of the long short-term memory network (LSTM), which is one reason for thinking there is a connection between the automaton and the neural network architecture.

While the classical literature on counter machines focused more on the unbounded variant, Weiss et al. [23] discuss the real-time variant 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. [23], and some slightly less restricted forms of counter machines. It turns out that some of the restrictions imposed by Weiss et al. [23] on the original counter model severely restrict the computational capacity of the model, whereas others do not change what it can compute.
A.1 Definitions

A.1.1 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.

**Definition A.1.1** (General counter machine). A counter machine is a tuple \((\Sigma, Q, q_0, k, u, \delta, F)\) containing

1. A finite alphabet \(\Sigma\)
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 \(u : \Sigma \times Q \times \{0, 1\}^k \rightarrow \left(\{\lambda x. x + n : n \in \mathbb{Z}\} \cup \{\lambda x. 0\}\right)^k\)
6. A state transition function \(\delta : \Sigma \times Q \times \{0, 1\}^k \rightarrow Q\)
7. An acceptance mask \(F : Q \times \{0, 1\}^k \rightarrow \{0, 1\}\)

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 \(\langle q, \vec{b} \rangle\).

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 \(\langle q, \vec{c} \rangle \in Q \times \mathbb{Z}^k\) be a configuration of machine \(M\). Upon reading input \(x \in \Sigma\), \(M\) transitions into the new configuration \(\langle q', \vec{c}' \rangle\) where

\[
\vec{c}' = u(x, q, z(\vec{c})). \quad (A.1.3.1)
\]

\[
q' = \delta(x, q, z(\vec{c})). \quad (A.1.3.2)
\]
We write this relation as
\[ \langle q, \vec{c} \rangle \rightarrow_x \langle q', \vec{c}' \rangle. \] (A.1.3.3)

**Definition A.1.4** (Real-time string acceptance). A counter machine accepts a string \( x_1, \ldots, x_n \) if
\[ \langle q_0, \vec{0} \rangle \rightarrow_{x_1} \langle q_1, \vec{c}_1 \rangle \rightarrow_{x_2} \ldots \rightarrow_{x_n} \langle q_n, \vec{c}_n \rangle \] (A.1.1)

and
\[ F(q_n, z(\vec{c}_n)). \] (A.1.2)

**Definition A.1.5** (Real-time language acceptance). A counter machine 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.1.2 Counter machine variants

Now, we 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. [23], which they call "SKCM".

**Definition A.1.7** (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.1.8** (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.1.9** (Incremental counter machine). An incremental counter machine is a counter machine where $u$ takes the restricted form

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

**Definition A.1.10** (Incremental counter languages). Let $\text{ICL}$ 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.1.11** (Stateless counter machine). A stateless counter machine is a counter machine with only one state $q_0$.

**Definition A.1.12** (Stateless counter languages). Let $\tilde{\text{QCL}}$ be the set of languages that are acceptable in real time by a stateless counter machines.

### A.2 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.2.1.**

$$a^n b^{2n} \notin \text{SCL}.$$ 

**Corollary A.2.1.1.**

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

**Proof.** Consider the language $a^n b^{2n}$. This is trivially acceptable by CL 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 SCL. 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 $b$s, the additional update to the counters must be $u_b \in \{-2n, 0, 2n\}$.

We need to determine whether the number of $a$s equals the twice number of $b$s 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 $\sigma \in a^* b^*$,
\[ u_a + u_b = 0 \implies z(u_a + u_b) = 0. \tag{A.2.1} \]

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

\[ u_a + u_b \neq 0 \implies z(u_a + u_b) = 1. \tag{A.2.2} \]

So, for any pair of update operations we pick, the counters cannot distinguish whether the number of \( b \)s is twice the number of \( a \)s.

Note that this counter example 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 \( b \)s, 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.2.2.**

\[ \text{CL} = \text{ICL}. \]

**Proof.** Clearly, \( \text{ICL} \subseteq \text{CL} \). The goal is to show that \( \text{CL} \subseteq \text{ICL} \). 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 register \( c \) in the general machine. We will define a vector of registers \( \hat{c}_1, \ldots, \hat{c}_k \) 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 \( \hat{c} \), as well as \( \text{ADD}-\delta \) and \( \text{SUB}-\delta \) operations.

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

\[ c = \sum_{n=1}^{k} n\hat{c}_n. \tag{A.2.3} \]
\[ \hat{c}_1, \ldots, \hat{c}_{k-1} \text{ is one-hot or } \vec{0}. \tag{A.2.4} \]

A natural way of computing the zero mask of the simulated counters follows from these invariants:
\[ z(c) \iff \bigvee_{n=1}^{k} z(\hat{c}_n). \quad (A.2.5) \]

We can simulate adding or subtracting \( \delta \) to \( c \) according to the following update rules:

- **ADD-\( \delta \)**:
  \[
u_i = -1, u_{\min(i+\delta,k)} = +1, u_{i+\delta-k} = +1. \quad (A.2.6)\]

- **SUB-\( \delta \)**:
  \[
u_i = -1, u_{i-\delta} = +1 \quad \text{if } i \geq \delta \\
u_i = -1, u_k = -1, u_{k+i-\delta} = +1 \quad \text{otherwise.} \quad (A.2.7)\]

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 \( \hat{c}_1, \ldots, \hat{c}_{k-1} \) 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

\[(z(\hat{c}), n) \mapsto u_n.\]

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

\[
\sum_{n=1}^{k} n(\hat{c}_n + u_n) = \sum_{n=1}^{k} n\hat{c}_n + \sum_{n=1}^{k} nu_n \\
= c + iu_i + \min(i+\delta,k)u_{\min(i+\delta,k)} + \mathbb{1}_{i+\delta > k}(i+\delta-k)u_{i+\delta-k}. \quad (A.2.8)\]

When \( i + \delta > k \), we get

\[c - i + k + i + \delta - k = c + \delta. \quad (A.2.10)\]

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

\[c - i + i + \delta = c + \delta. \quad (A.2.11)\]

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 **SUB-\( \delta \)** update. When \( i \geq \delta \), the new counter state is given by

\[c - i + i - \delta = c - \delta. \quad (A.2.12)\]
In the complementary case where \( i < \delta \), we get
\[
    c - i - k + k + i - \delta = c - \delta. \tag{A.2.13}
\]

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. \( \square \)

**Theorem A.2.3.**

\[
    CL = \tilde{Q}CL.
\]

**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
\[
    q(M) = q_i \iff \hat{q} = \vec{\omega}(i) \tag{A.2.14}
\]
where \( \vec{\omega}(i) \) 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
\[
    F'(\vec{b}||\vec{\omega}(i)) = F(q_i, \vec{b}). \tag{A.2.15}
\]

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:
\[
    \langle \sigma, \vec{b}, \vec{\omega}(i), \vec{v} \rangle \in u'_1 \iff \langle \sigma, q_i, \vec{b}, \vec{v} \rangle \in u \tag{A.2.16}
\]
and
\[
    \langle \sigma, \vec{b}, -\vec{\omega}(i) + \vec{\omega}(j) \rangle \in u'_2 \iff \langle \sigma, q_i, \vec{b}, q_j \rangle \in \delta. \tag{A.2.17}
\]
Then, we can define \( u' \) in terms of \( u'_1 \) and \( u'_2 \) according to
\[
    u'(\sigma, \vec{b}||\vec{\omega}(i)) = u'_1(\sigma, \vec{b}||\vec{\omega}(i)) \parallel u'_2(\sigma, \vec{b}||\vec{\omega}(i)) \tag{A.2.18}
\]
Note that the state vector updated by \( u'_2 \) is a one-hot encoding of \( q_j \) because
\[
    \vec{\omega}(i) + (-\vec{\omega}(i) + \vec{\omega}(j)) = \vec{\omega}(j), \tag{A.2.19}
\]
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.3 Closure properties of counter classes

**Theorem A.3.1** (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

\[
1_{L'}(\alpha) = p(1_{L_1}(\alpha), \ldots, 1_{L_n}(\alpha)).
\]

**Corollary A.3.1.1** (Complement closure). CL is closed under complement.

**Corollary A.3.1.2** (Intersection closure). CL is closed under intersection.

**Corollary A.3.1.3** (Union closure). CL is closed under union.

**Corollary A.3.1.4** (Set difference closure). CL is closed under set difference.

**Corollary A.3.1.5** (Symmetric difference closure). CL is closed under symmetric difference.

**Proof.** Given finitely many counter machines \( M_1, \ldots, M_n \), 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 \( q' \in Q_1 \times \ldots \times Q_n \) and \( c' \in \mathbb{Z}^{k_1 \times \ldots \times k_n} \). Let \( q' = \langle q_1, \ldots, q_n \rangle \) and analogously for \( b' \) and \( c' \). We can write the update functions for the new machine as

\[
\delta'(x, q', b') = \langle \delta_1(x, q_1, b_1), \ldots, \delta_n(x, q_n, b_n) \rangle \quad \text{(A.3.1)}
\]

and

\[
u'(x, q', b') = \lambda c' \cdot u_1(x, q_1, b_1) || \ldots || u_n(x, q_n, b_n).
\] (A.3.2)

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

\[
F'(q', b') \iff p(F_1(q_1, b_1), \ldots, F_n(q_n, b_n)). \quad \text{(A.3.3)}
\]

Interestingly, all of these closure properties also apply to the simplified counter languages. This is because theorem A.3.1 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 \( \phi : X \rightarrow \{0, 1\} \) is linearly separable in \( x \) if and only if there exists a parameterized affine transformation such that

\[
\lim_{N \to \infty} \sigma \left( N(Wx(N) + b) \right) = \mathbb{1}_\phi(x).
\]

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

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. While not directly related to the topic of counter machines, these will be referenced throughout the main results that I present.

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

\[
\bigwedge_{i=1}^{n} x_i \land \bigwedge_{j=1}^{m} \neg y_j.
\]

**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
Linearly separable expressions

\[ \lim_{N \to \infty} \sigma \left( \sum_{i=1}^{n} N x_i - \sum_{j=1}^{m} N y_j - \frac{(n - 1)}{2} N \right), \]

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 \( \phi(x) \) be a linearly separable form in \( x \). Then the following form is linearly separable in \( x \):

\[ \neg \phi(x). \]

*Proof.* Take an affine transformation for \( \phi \), and then take its additive inverse.

**Theorem B.1.3** (Disjunction). The following formula is linearly separable in \( x \mid y \):

\[ \bigvee_{i=1}^{n} x_i \lor \bigvee_{j=1}^{m} \neg y_j. \]

*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.** The following formula is linearly separable in \( x \mid y \mid z \):

\[ \bigvee_{i=1}^{n} x_i \land \bigwedge_{j=1}^{m} y_j \land \bigwedge_{k=1}^{l} \neg z_k. \]

*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 \( ((n + 1)m + \frac{1}{2})N \). Then, the form of the transformation is

\[ \lim_{N \to \infty} \sigma \left( \sum_{i=1}^{n} N x_i + \sum_{j=1}^{m} (n + 1)N y_j - \sum_{k=1}^{l} (n + 1)N z_k - \left( (n + 1)m + \frac{1}{2} \right) N \right). \]

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, we will get 0.