Chapter 4

Recurrent Neural Networks

4.1 Model

Definition 4.1. A *simple RNN* (Elman, 1990) with σ activations (where σ is any activation function) is a length-preserving function

$$\begin{aligned} rec\colon (\mathbb{R}^d)^* &\xrightarrow{\text{lp}} (\mathbb{R}^{d'})^* \\ (\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(n-1)}) &\mapsto (\mathbf{h}^{(0)}, \ldots \mathbf{h}^{(n-1)}) \end{aligned} \tag{4.1}$$

where

$$\mathbf{h}^{(-1)} = \mathbf{s} \tag{4.2}$$

$$\mathbf{h}^{(i)} = \sigma\Big(\mathbf{V}\mathbf{h}^{(i-1)} + \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}\Big) \qquad i \in [n] \tag{4.3}$$

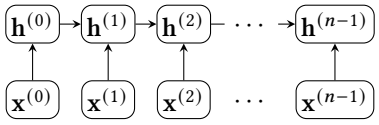


Figure 4.1: A simple RNN.

with parameters

$$\begin{aligned}\mathbf{s} &\in \mathbb{R}^{d'}\\ \mathbf{V} &\in \mathbb{R}^{d'\times d'}\\ \mathbf{W} &\in \mathbb{R}^{d'\times d}\\ \mathbf{b} &\in \mathbb{R}^{d'}\end{aligned}$$

and attributes:

- d is the input size
- d' is the output size.

To use a simple RNN as a function on Σ^*, we represent each symbol by a one-hot vector. To use it to output an accept/reject decision, we use a linear output layer:

$$f = out \circ rec \tag{4.4}$$

where *rec* is a simple RNN and *out* is a linear layer

$$\begin{aligned}out\colon \mathbb{R}^{d'} &\to \mathbb{R}\\ \mathbf{h}^{(i)} &\mapsto \mathbf{W}\mathbf{h}^{(i)} + b\end{aligned} \tag{4.5}$$

with parameters

$$\begin{aligned}\mathbf{W} &\in \mathbb{R}^{1\times d'}\\ b &\in \mathbb{R}.\end{aligned}$$

Then if the output is 0, reject, and if 1, accept. But we will consider other variations below.

4.2 Expressivity

4.2.1 Integer-weight RNNs

The connection between RNNs and finite automata goes all the way back to the beginning; McCulloch and Pitts (1943) called RNNs "nerve nets with circles," and Kleene (1956) reformulated them as finite automata.

But Kleene envisioned an RNN where $\mathbf{h}^{(i)}$ is an arbitrary Boolean function of $\mathbf{h}^{(i-1)}$ and $\mathbf{x}^{(i)}$. In a simple RNN it is not, which raises a question about its expressivity:

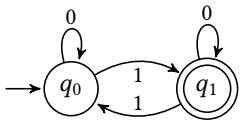


Figure 4.2: A DFA for PARITY

Example 4.2 (Goudreau et al., 1994)**.** Define the language

$$\text{PARITY} = \{\mathbf{w} \in \{0, 1\}^* \mid \mathbf{w} \text{ has an odd number of 1's}\}. \tag{4.6}$$

At each time step t, the vector $\mathbf{h}^{(i)}$ must be able to distinguish between whether the string so far is in PARITY or not. This means that the transition function has to compute $\mathbf{h}^{(i)}$ as the XOR of $\mathbf{h}^{(i-1)}$ and w_t. Since the transition function of a simple RNN is a single-layer FFNN, and a single-layer FFNN cannot compute the XOR function (Theorem 2.4), can a simple RNN recognize PARITY?

Fortunately, the answer is yes, and indeed a simple RNN can simulate any finite automaton, as long as it is coupled with an output layer.

Definition 4.3. A *deterministic finite automaton (DFA)* is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states
- Σ is a finite alphabet
- $\delta \colon Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the initial state
- F is a set of final states.

On input $\mathbf{w} \in \Sigma^*$, the DFA M is intialized in state q_0. If the current state is q, and the next symbol of $\mathbf{w}$ is a, then the machine transitions to state $\delta(q, a)$. After reading the entire string, the DFA accepts if the current state is in F. Otherwise, the DFA rejects $\mathbf{w}$. We say a DFA M recognizes a language $L \subseteq \Sigma^*$ iff it accepts all strings in L and rejects all strings not in L. We say that a language L is regular iff it is recognized by a DFA.

For example, a DFA that recognizes PARITY is shown in Fig. 4.2.

Theorem 4.4 (Minsky, 1967)**.** *Any regular language can be recognized by a network $f = out \circ rec$, where rec is a simple RNN with integer weights and* ReLU *activations, and out is a linear layer.*

Tentative proof. Let L be a regular language over an alphabet Σ, and let L be recognized by a DFA M with states $Q = \{q_0, \ldots, q_{|Q|-1}\}$, start state q_0, transition function $\delta\colon Q \times \Sigma \to Q$, and accept states F. Number the symbols of Σ as $a_0, \ldots, a_{|\Sigma|-1}$. To reduce clutter, we index vectors by states instead of state-numbers, that is, $[q_i]$ instead of $[i]$, and by symbols instead of symbol-numbers, that is, $[a_i]$ instead of $[i]$.

As an initial attempt, define

$$f = out \circ rec \tag{4.7}$$

$$rec\colon (\mathbb{R}^{|\Sigma|})^* \to (\mathbb{R}^{|Q|})^*$$

$$(\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(n-1)}) \mapsto (\mathbf{h}^{(0)}, \ldots, \mathbf{h}^{(n-1)}) \tag{4.8}$$

$$\mathbf{h}^{(-1)}[q] = \mathbb{I}[q = q_0] \qquad q \in Q \tag{4.9}$$

$$\mathbf{g}^{(i)}[q, a] = \mathbb{I}\left[\mathbf{h}^{(i-1)}[q] \wedge \mathbf{x}^{(i)}[a]\right] \qquad i \in [n], q \in Q, a \in \Sigma \tag{4.10}$$

$$\mathbf{h}^{(i)}[r] = \sum_{\substack{q \in Q, a \in \Sigma \\ \delta(q,a)=r}} \mathbf{g}^{(i)}[q, a] \qquad i \in [n], r \in Q \tag{4.11}$$

$$out\colon \mathbb{R}^{|Q|} \to \mathbb{R}$$

$$\mathbf{h}^{(i)} \mapsto \sum_{q \in F} \mathbf{h}^{(i)}[q] \qquad i \in [n]. \tag{4.12}$$

Added 9-18 and 9-19

The notation for *rec* is dense. Intuitively, $\mathbf{h}^{(i)}$ encodes the current state of the DFA after reading the i-th symbol of the input. Equation (4.9) initializes $\mathbf{h}^{(i-1)}$ to the one-hot vector for the initial state q_0. Equation (4.10) prepares for Eq. (4.11) by computing a "one-hot matrix" $\mathbf{g}^{(i)}$ storing the previous state and the current input symbol. Equation (4.11) sets $\mathbf{h}^{(i)}$ to the one-hot vector for the current state.

We had some difficulty with these equations in class, so here are a couple of attempts to elucidate.

As pseudocode Perhaps it is easier to read as pseudocode (assume everything is initialized to zero):

for $q \in Q$ **do**
  if $q = q_0$ **then**
    $\mathbf{h}^{(-1)}[q] \leftarrow 1$ ▷ Eq. (4.9)
for $i \in [n]$ **do**
  for $q \in Q$ **do**
    for $a \in \Sigma$ **do**
      if $\mathbf{h}^{(i-1)}[q] \wedge \mathbf{x}^{(i)}[a]$ **then**
        $\mathbf{g}^{(i)}[q, a] \leftarrow 1$ ▷ Eq. (4.10)
  for $q \in Q$ **do**

for $a \in \Sigma$ **do**
  for $r \in Q$ **do**
    if $\delta(q, a) = r$ **then**
      $\mathbf{h}^{(i)}[r] \leftarrow \mathbf{h}^{(i)}[r] + 1$ ▷ Eq. (4.11)

Example Let's run the DFA in Fig. 4.2 on input `110`. We represent $\mathbf{g}^{(i)}$ as a matrix with rows corresponding to states and columns corresponding to symbols

$$\mathbf{h}^{(i)} = \begin{matrix} q_0 \\ q_1 \end{matrix} \begin{bmatrix} - \\ - \end{bmatrix} \qquad \mathbf{g}^{(i)} = \begin{matrix} \\ q_0 \\ q_1 \end{matrix} \begin{matrix} 0 & 1 \\ \end{matrix} \begin{bmatrix} - & - \\ - & - \end{bmatrix}.$$

Then across the run of the DFA on `110`, we have

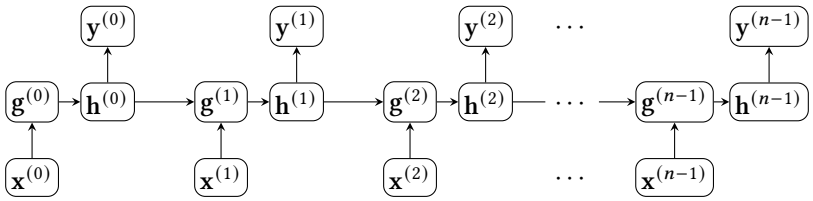
$$\mathbf{h}^{(-1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \mathbf{h}^{(0)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \mathbf{h}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \mathbf{h}^{(2)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
$$\mathbf{g}^{(0)} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad \mathbf{g}^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{g}^{(2)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

Our initial attempt creates a kind of RNN that is equivalent to a DFA, but it's not a simple RNN, because each step of $f.rec$ has two layers (Eqs. (4.10) and (4.11)). The trick is to cut apart the two layers and move the second layer (which is just a linear transformation) to the following time step, where it can be merged into a single layer (because a linear transformation composed with a linear transformation is a linear transformation). See Fig. 4.3. To do this, I think it's convenient to pause this proof and first prove a more general lemma, which will also be useful later. □

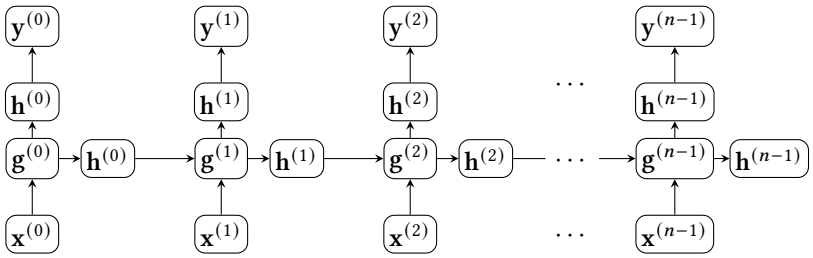
Lemma 4.5. *Let f be a length-preserving function*

$$f \colon (\mathbb{R}^{d_{\text{in}}})^* \xrightarrow{\text{lp}} (\mathbb{R}^{d_{\text{out}}})^*$$
$$(\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(n-1)}) \mapsto (\mathbf{y}^{(0)}, \ldots, \mathbf{y}^{(n-1)}) \tag{4.13}$$
$$\mathbf{h}^{(-1)} = \mathbf{s} \tag{4.14}$$
$$\mathbf{h}^{(i)} = \mathit{step}(\mathbf{h}^{(i-1)}, \mathbf{x}^{(i)}) \qquad i \in [n] \tag{4.15}$$
$$\mathbf{y}^{(i)} = \mathit{out}(\mathbf{h}^{(i)}) \tag{4.16}$$

where $\mathbf{s} \in \mathbb{R}^d$, $\mathit{step} \colon \mathbb{R}^d \times \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^d$ is a two-layer ReLU FFNN, and $\mathit{out} \colon \mathbb{R}^d \to \mathbb{R}^{d_{\text{out}}}$ is an affine transformation. Then there is a network $f' = \mathit{out} \circ \mathit{rec}$, where $f'.rec \colon (\mathbb{R}^{d_{\text{in}}})^* \to (\mathbb{R}^{d'})^*$ is a simple ReLU RNN and $f'.out \colon \mathbb{R}^{d'} \to \mathbb{R}^{d_{\text{out}}}$ is a linear layer, that is equal to f.



(a) Initial attempt (not a simple RNN)



(b) Correct version (simple RNN)

Figure 4.3: Proofs of Theorem 4.4 and Lemma 4.5. (a) Our initial attempt does not fit the definition of a simple RNN, because the $\mathbf{h}^{(i)} \mapsto \mathbf{h}^{(i+1)}$ mapping has two layers. (b) Making the $\mathbf{g}^{(i)}$ the hidden states does fit the definition of a simple RNN, because the $\mathbf{g}^{(i)} \mapsto \mathbf{h}^{(i)}$ mapping is linear, and we can fuse it into both the $\mathbf{h}^{(i)} \mapsto \mathbf{y}^{(i)}$ and $\mathbf{h}^{(i)} \mapsto \mathbf{h}^{(i+1)}$ mappings.

Proof. Let's write out the equations for $f.step$:

$$f.step\colon \mathbb{R}^d \times \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^d$$
$$(\mathbf{h}^{(i-1)}, \mathbf{x}^{(i)}) \mapsto \mathbf{h}^{(i)} \tag{4.17}$$
$$\mathbf{g}^{(i)} = \text{ReLU}(\mathbf{V}\mathbf{h}^{(i-1)} + \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}) \tag{4.18}$$
$$\mathbf{h}^{(i)} = \mathbf{U}\mathbf{g}^{(i)}. \tag{4.19}$$

Without loss of generality, we can assume that the second layer (Eq. (4.18)) doesn't have a bias term. This doesn't have the form that we need for $f'.rec$, because of the second layer. But (as mentioned above) the trick is to cut apart the two layers and move the second layer (which is just a linear transformation) to the following time step, where it can be merged into a single layer (because a linear transformation composed with a linear transformation is a linear transformation). We just have to be careful with the first time step, because there was no previous time step to inherit a $\mathbf{U}$ from.

$$f'.rec\colon (\mathbb{R}^{d_{\text{in}}})^* \xrightarrow{\text{lp}} (\mathbb{R}^{d'})^*$$
$$(\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(n-1)}) \mapsto (\mathbf{g}^{(0)}, \dots, \mathbf{g}^{(n-1)}) \tag{4.20}$$
$$\mathbf{g}^{(-1)} = \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix} \tag{4.21}$$
$$\mathbf{h}^{(i)} = \begin{bmatrix} 0 & \mathbf{0} \\ f.\mathbf{s} & \mathbf{U} \end{bmatrix} \mathbf{g}^{(i-1)} \tag{4.22}$$
$$\mathbf{g}^{(i)} = \text{ReLU}\left(\begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{V} \end{bmatrix} \mathbf{h}^{(i-1)} + \begin{bmatrix} \mathbf{0} \\ \mathbf{W} \end{bmatrix} \mathbf{x}^{(i)} + \begin{bmatrix} 0 \\ \mathbf{b} \end{bmatrix} \right) \quad i \in [n] \tag{4.23}$$
$$f'.out\colon \mathbb{R}^{d'} \to \mathbb{R}^{d_{\text{out}}}$$
$$\mathbf{g}^{(i)} \mapsto f.out\left(\begin{bmatrix} \mathbf{0} & \mathbf{U} \end{bmatrix} \mathbf{g}^{(i)} \right).$$

□

Proof of Theorem 4.4. In our initial attempt, the step function (Eqs. (4.10) and (4.11)) is a two-layer ReLU FFNN. So we can use Lemma 4.5 to convert the network into a simple RNN. □

Do simple RNNs recognize *only* regular languages? It depends. If we restrict the weights and/or activation functions appropriately, then yes.

Theorem 4.6. *Any network $f = out \circ rec$, where rec is a simple RNN with integer weights and* SLU *activations, and out is a linear layer, recognizes a regular language.*

Proof. Because all weights are integers, all activation values (that is, all the components of all the $\mathbf{h}^{(i)}$) will also be integers. Moreover, because each $\mathbf{h}^{(i)}$ is the result of a SLU, it must be in $\{0, 1\}$.

Now we construct a DFA $(Q, \Sigma, \delta, q_0, F)$, where

$$Q = \{q_0\} \cup \{0,1\}^d.$$

That is, there are $2^d + 1$ states: there is the initial state q_0, and there is a state for every possible $\mathbf{h}$ vector. The transition function δ simulates one step of the RNN:

$$\delta(q_0, a) = \text{SLU}(rec.\mathbf{V}(\mathbf{s}) + rec.\mathbf{W}(\mathbf{e}_a) + rec.\mathbf{b}) \qquad a \in \Sigma \tag{4.24}$$

$$\delta(\mathbf{h}, a) = \text{SLU}(rec.\mathbf{V}(\mathbf{h}) + rec.\mathbf{W}(\mathbf{e}_a) + rec.\mathbf{b}) \qquad a \in \Sigma, \mathbf{h} \in \{0,1\}^d. \tag{4.25}$$

And the set of final states F simulates the output layer:

$$F = \{\mathbf{h} \mid out.\mathbf{W}(\mathbf{h}) + out.b = 1\}. \tag{4.26}$$

□

Example Let's convert the RNN below into a DFA.

$$\Sigma = \{\mathsf{a}, \mathsf{b}\}$$

$$rec.\mathbf{s} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$rec.\mathbf{V} = \begin{bmatrix} 1 & -2 \\ -3 & 4 \end{bmatrix} \qquad rec.\mathbf{W} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad rec.\mathbf{b} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

$$out.\mathbf{W} = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad out.b = 0.$$

$$Q = \left\{q_0, \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]\right\}$$

$$\delta(q_0, \mathsf{a}) = \text{SLU}\left(\left[\begin{smallmatrix} 2 \\ -3 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] \qquad \delta(q_0, \mathsf{b}) = \text{SLU}\left(\left[\begin{smallmatrix} 1 \\ -2 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$$

$$\delta\left(\left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right], \mathsf{a}\right) = \text{SLU}\left(\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] \qquad \delta\left(\left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right], \mathsf{b}\right) = \text{SLU}\left(\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$$

$$\delta\left(\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right], \mathsf{a}\right) = \text{SLU}\left(\left[\begin{smallmatrix} -1 \\ 4 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] \qquad \delta\left(\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right], \mathsf{b}\right) = \text{SLU}\left(\left[\begin{smallmatrix} -2 \\ 5 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$$

$$\delta\left(\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right], \mathsf{a}\right) = \text{SLU}\left(\left[\begin{smallmatrix} 2 \\ -3 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] \qquad \delta\left(\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right], \mathsf{b}\right) = \text{SLU}\left(\left[\begin{smallmatrix} 1 \\ -2 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$$

$$\delta\left(\left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right], \mathsf{a}\right) = \text{SLU}\left(\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] \qquad \delta\left(\left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right], \mathsf{b}\right) = \text{SLU}\left(\left[\begin{smallmatrix} -1 \\ 2 \end{smallmatrix}\right]\right) = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$$

$$F = \left\{\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]\right\}.$$

Theorems 4.4 and 4.6 are not exactly converses of each other. But Theorem 4.4 used only integer weights and could have been done using SLU activations. So simple RNNs with integer weights and SLU activations recognize *exactly* the regular languages.

Number of states One possible objection to this equivalence is that Theorem 4.6 converts an RNN with d-dimensional hidden vectors to a DFA with $O(2^d)$ states. Does that mean that even if RNNs and DFAs define the same languages, DFAs do so less efficiently? If you are familiar with nondeterministic finite automata (NFAs), you may be reminded of the fact that an NFA with d states converts to a DFA with 2^d states. One might hope that an NFA can simulate an RNN with d states, but no, it too needs $O(2^d)$ states. But there is an even fancier version of finite automata, *alternating* finite automata (Chandra et al., 1981), that can simulate an RNN using only d states.

Infinite states What happens if we tinker with the assumptions of Theorem 4.6?

Exercise 4.7. Switching from SLU to ReLU activations breaks Theorem 4.6. Construct a network of the form $out \circ rec$, where rec is a simple RNN with integer weights and ReLU activations and out is a linear layer, recognizing the nonregular language

$$\text{MAJORITY} = \{w \in \{0,1\}^* \mid w \text{ has strictly more 1's than 0's}\}.$$

You may change the acceptance criterion to something else (like a positive output for accept and a negative output for reject).

Exercise 4.8. Allowing rational weights also breaks Theorem 4.6. Construct a network of the form $out \circ rec$, where rec is a simple RNN with rational weights and SLU activations and out is a linear layer, recognizing MAJORITY. Again, you may change the acceptance criterion.

To my knowledge, the exact expressivity of these variations of RNNs has not been well explored. The only result I'm aware of relates RNNs to real-time Turing machines (Chen et al., 2017, Sec. 7).

4.2.2 Rational weight RNNs with intermediate steps

If we allow an arbitrary-precision numeric representation, then the power of RNNs increases. In fact, if we allow the RNN to run for a number of *intermediate steps* after the end of the input string but before delivering an accept/reject decision, it can simulate a Turing machine.

We assume that you are familiar with Turing machines. We use the definition of Turing machine in the textbook by Sipser (2013), with one small modification. Just to make sure we're on the same page, we give the barest of definitions here.

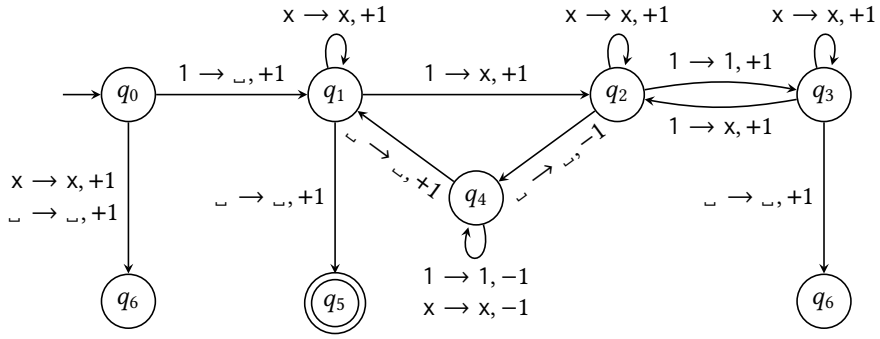
Definition 4.9. A Turing machine is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of states

- Σ is a finite input alphabet, where $\sqcup \notin \Sigma$
- Γ is a finite tape alphabet, where $\Sigma \cup \{\sqcup\} \subseteq \Gamma$
- $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ is the transition function.

The tape has a left end and extends infinitely to the right. On input $\mathbf{w} \in \Sigma^*$, the tape is initialized to $\mathbf{w}\sqcup\sqcup\cdots$. If the current state is q, the current tape symbol is a, and $\delta(q, a) = (r, b, m)$, then the machine enters state r, writes a b, and moves left if $m = -1$, right if $m = +1$. If the machine enters state q_{accept}, it halts and accepts $\mathbf{w}$; if it enters state q_{reject}, it halts and rejects $\mathbf{w}$.

Example 4.10. Here's an example Turing machine (Sipser, 2013), with $q_{\text{start}} = q_0$, $q_{\text{accept}} = q_5$ (marked with a double circle), $q_{\text{reject}} = q_6$. It decides the language $\{1^{2^m} \mid m \geq 0\}$.



The reject state q_6 appears twice to reduce clutter.

The (not very exciting) run of this machine on string 11 is:

$$
\begin{array}{ll}
q_0 & \hat{1}1\sqcup\cdots \\
q_1 & \sqcup\hat{1}\sqcup\cdots \\
q_2 & \sqcup\hat{\mathsf{x}}\sqcup\cdots \\
q_4 & \sqcup\hat{\mathsf{x}}\sqcup\cdots \\
q_4 & \hat{\sqcup}\mathsf{x}\sqcup\cdots \\
q_1 & \sqcup\hat{\mathsf{x}}\sqcup\cdots \\
q_5 & \text{accept}
\end{array}
$$

Theorem 4.11 (Siegelmann and Sontag, 1995)**.** *For any Turing machine M with input alphabet Σ, there is a network $f = out \circ rec$, where rec is a simple RNN with rational weights and* ReLU *activation functions, and out is a linear layer, that is equivalent to M in the following sense: for any string $\mathbf{w} \in \Sigma^*$,*

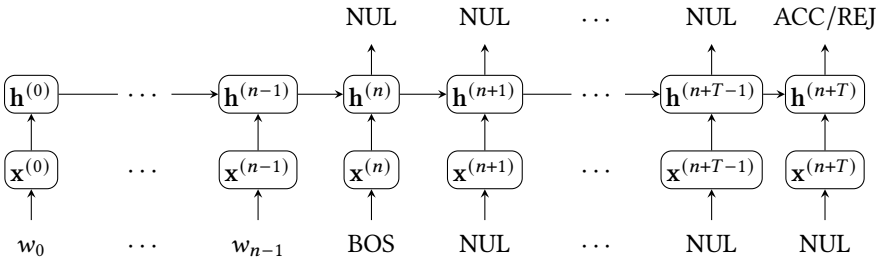


Figure 4.4: The RNN of Theorem 4.11 accepts/rejects after a certain number of intermediate steps.

- *If M halts and accepts on input $\mathbf{w}$, then there is a T such that for all $t \in [T]$, $f(\mathbf{w} \cdot \text{BOS} \cdot \text{NUL}^t) = \mathbf{e}_{\text{NUL}}$ and $f(\mathbf{w} \cdot \text{BOS} \cdot \text{NUL}^T) = \mathbf{e}_{\text{ACC}}$.*
- *If M halts and rejects on input $\mathbf{w}$, then there is a T such that for all $t \in [T]$, $f(\mathbf{w} \cdot \text{BOS} \cdot \text{NUL}^t) = \mathbf{e}_{\text{NUL}}$ and $f(\mathbf{w} \cdot \text{BOS} \cdot \text{NUL}^T) = \mathbf{e}_{\text{REJ}}$.*
- *If M does not halt on input $\mathbf{w}$, then for all $t \geq 0$, $f(\mathbf{w} \cdot \text{BOS} \cdot \text{NUL}^t) = \mathbf{e}_{\text{NUL}}$.*

Proof. We adapt Siegelmann and Sontag's proof to use the definition of Turing machine by Sipser (2013) and our definition of RNN above.

If Γ_M is M's tape alphabet, let $\Gamma = \Gamma_M \cup \{\$\}$, where \$ is a symbol not already in Γ_M, which we will use as a bottom-of-stack marker. Number the symbols of Γ as $a_0, a_1, \ldots, a_{|\Gamma|-1}$. To represent M's tape (which has a leftmost cell but extends infinitely to the right), we use two stacks, $\mathbf{l} = l_0 \cdots l_{|l|-1} \in \Gamma^*$ and $\mathbf{r} = r_0 \cdots r_{|r|-1} \in \Gamma^*$, and a symbol $c \in \Gamma$. These represent the tape

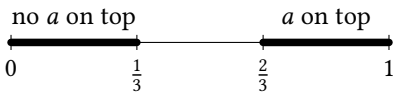
$$l_{|l|-1} l_{|l|-2} \cdots l_1 l_0 c r_0 r_1 \cdots r_{|r|-2} r_{|r|-1} \sqcup \sqcup \cdots$$

with the head is over symbol c.

Then we encode a stack as a vector of $|\Gamma|$ rational numbers using the following mapping:

$$\begin{aligned} \text{stack}\colon \Gamma^* &\to \mathbb{Q}^{|\Gamma|} \\ \text{stack}(\epsilon) &= \mathbf{0} && (4.27) \\ \text{stack}(a_j \cdot \mathbf{z}) &= \tfrac{2}{3}\mathbf{e}_j + \tfrac{1}{3}\text{stack}(\mathbf{z}). && (4.28) \end{aligned}$$

For each $a \in \Gamma$, this encoding puts a "margin" between stacks without an a on top and stacks with an a on top, so that a SLU network can distinguish them:



(The set of possible values is known as the *Cantor set*.)

Then the basic stack operations can be implemented as follows:

$$\text{push}(\mathbf{z}, a_j) = \tfrac{2}{3}\mathbf{e}_j + \tfrac{1}{3}\mathbf{z} \tag{4.29}$$

$$\text{top}(\mathbf{z}) = \text{SLU}(3\mathbf{z} - 1) \tag{4.30}$$

$$\text{pop}(\mathbf{z}) = 3\mathbf{z} - 2\,\text{top}(\mathbf{z}). \tag{4.31}$$

Let Q_M be the states of M. Let Q contain two new states q_0 and q_1, which aren't used by M itself, but by a preprocessing step. Then, number the rest of the states starting with the start state q_2, then $q_3, q_4, \ldots, q_{|Q|+1}$.

The hidden vectors of the RNN are

$$\mathbf{h}^{(i)} = \begin{bmatrix} \mathbf{q}^{(i)} \\ \mathbf{l}^{(i)} \\ \mathbf{c}^{(i)} \\ \mathbf{r}^{(i)} \end{bmatrix} \tag{4.32}$$

where $\mathbf{q}^{(i)}$ is the one-hot vector of the current state, $\mathbf{l}^{(i)}$ and $\mathbf{r}^{(i)}$ are the left and right stacks, respectively, and $\mathbf{c}$ is the one-hot vector of the current symbol. For clarity (I hope), let's write q_i in place of $\mathbf{e}_i$.

The initial vector is

$$\mathbf{h}^{(-1)} = \begin{bmatrix} q_0 \\ \text{stack}(\$) \\ \text{␣} \\ \text{stack}(\$) \end{bmatrix}. \tag{4.33}$$

We initialized both stacks with a \$ on the bottom and a blank as the current symbol.

We've shown previously (Example 3.3) how to define Boolean operators using ReLUs; we also need

$$\text{if}(b, t) = \text{ReLU}(t + b - 1) \tag{4.34}$$

which means, assuming $b \in \{0, 1\}$ and $t \in [0, 1]$, "if $b = 1$, then t, else 0."

The recurrent step is a big piecewise linear function. We break it up into four pieces:

$$\mathit{step}\left(\begin{bmatrix} \mathbf{q} \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r} \end{bmatrix}, \mathbf{x}\right) = \mathit{load}\left(\begin{bmatrix} \mathbf{q} \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r} \end{bmatrix}, \mathbf{x}\right) + \mathit{rewind}\left(\begin{bmatrix} \mathbf{q} \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r} \end{bmatrix}\right) + \mathit{left}\left(\begin{bmatrix} \mathbf{q} \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r} \end{bmatrix}\right) + \mathit{right}\left(\begin{bmatrix} \mathbf{q} \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r} \end{bmatrix}\right). \tag{4.35}$$

The first term initially loads the input string onto the tape, from left to right:

$$load\left(\begin{bmatrix}\mathbf{q}\\ \mathbf{l}\\ \mathbf{c}\\ \mathbf{r}\end{bmatrix}, \mathbf{x}\right) = \text{if}\left(\mathbf{q} = q_0 \wedge \mathbf{x} \neq \text{EOS}, \begin{bmatrix} q_0 \\ \text{push}(\mathbf{l}, \mathbf{x}) \\ \sqcup \\ \mathbf{r}\end{bmatrix}\right) + \text{if}\left(\mathbf{q} = q_0 \wedge \mathbf{x} = \text{EOS}, \begin{bmatrix} q_1 \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r}\end{bmatrix}\right). \tag{4.36}$$

The second term just rewinds the head back to the left end of the tape:

$$rewind\left(\begin{bmatrix}\mathbf{q}\\ \mathbf{l}\\ \mathbf{c}\\ \mathbf{r}\end{bmatrix}\right) = \text{if}\left(\mathbf{q} = q_1 \wedge \text{top}(\mathbf{l}) \neq \$, \begin{bmatrix} q_1 \\ \text{pop}(\mathbf{l}) \\ \text{top}(\mathbf{l}) \\ \text{push}(\mathbf{r}, \mathbf{c})\end{bmatrix}\right) + \text{if}\left(\mathbf{q} = q_1 \wedge \text{top}(\mathbf{l}) = \$, \begin{bmatrix} q_2 \\ \mathbf{l} \\ \mathbf{c} \\ \mathbf{r}\end{bmatrix}\right). \tag{4.37}$$

The third term handles all the left-moving transitions:

$$left\left(\begin{bmatrix}\mathbf{q}\\ \mathbf{l}\\ \mathbf{c}\\ \mathbf{r}\end{bmatrix}\right) = \sum_{\substack{q,q' \in Q \\ a,a' \in \Gamma \\ \delta(q,a)=(q',a',-1)}} \text{if}\left(\mathbf{q} = q \wedge \mathbf{c} = a \wedge \text{top}(\mathbf{l}) \neq \$, \begin{bmatrix} q' \\ \text{pop}(\mathbf{l}) \\ \text{top}(\mathbf{l}) \\ \text{push}(\mathbf{r}, a')\end{bmatrix}\right) + \sum_{\substack{q,q' \in Q \\ a,a' \in \Gamma \\ \delta(q,a)=(q',a',-1)}} \text{if}\left(\mathbf{q} = q \wedge \mathbf{c} = a \wedge \text{top}(\mathbf{l}) = \$, \begin{bmatrix} q' \\ \mathbf{l} \\ a' \\ \mathbf{r}\end{bmatrix}\right). \tag{4.38}$$

The last term handles all the right-moving transitions:

$$right\left(\begin{bmatrix}\mathbf{q}\\ \mathbf{l}\\ \mathbf{c}\\ \mathbf{r}\end{bmatrix}\right) = \sum_{\substack{q,q'\in Q\\ a,a'\in\Gamma\\ \delta(q,a)=(q',a',+1)}} \text{if}\left(\mathbf{q}=q\wedge\mathbf{c}=a\wedge\text{top}(\mathbf{r})\neq\$, \begin{bmatrix}q'\\ \text{push}(\mathbf{l},a')\\ \text{top}(\mathbf{r})\\ \text{pop}(\mathbf{r})\end{bmatrix}\right)$$
$$+ \sum_{\substack{q,q'\in Q\\ a,a'\in\Gamma\\ \delta(q,a)=(q',a',+1)}} \text{if}\left(\mathbf{q}=q\wedge\mathbf{c}=a\wedge\text{top}(\mathbf{r})=\$, \begin{bmatrix}q'\\ \text{push}(\mathbf{l},a')\\ \\ \mathbf{r}\end{bmatrix}\right). \tag{4.39}$$

Finally, we define an output layer such that

$$out\left(\begin{bmatrix}\mathbf{q}\\ \mathbf{l}\\ \mathbf{c}\\ \mathbf{r}\end{bmatrix}\right) = \begin{cases}\mathbf{e}_{\text{ACC}} & \text{if } \mathbf{q}=q_{\text{accept}}\\ \mathbf{e}_{\text{REJ}} & \text{if } \mathbf{q}=q_{\text{reject}}\\ \mathbf{e}_{\text{NUL}} & \text{otherwise.}\end{cases}$$

If we write the function *step* as a FFNN in the most straightforward way, it has four layers, so f is not a simple RNN. So the last step is to get rid of extra ReLU's until the FFNN has just two layers. Then, by Lemma 4.5, f will be a simple RNN. These are all the places where ReLU's are used:

1. The conditional function, $\text{if}(b,t)$.
2. In the conditions of the conditionals, top.
3. In the conditions of the conditionals, $\wedge$.
4. In the "then" part of the conditionals, top and pop.

Our goal is to eliminate all the uses of ReLU other than the first.

(2) Note that if $\text{top}(\mathbf{l})=\$$, then $\mathbf{l}$ is exactly \$ (because \$ only occurs at the bottom of the stack), and therefore $\mathbf{l}[\$]=\frac{2}{3}$ exactly. Otherwise, $\mathbf{l}[\$]\leq\frac{1}{3}$. Furthermore, note that $\text{if}(b,t)$ works even when $b<0$ (standing for false, just like $b=0$). So

$$\text{if}(\text{top}(\mathbf{l})=\$,t)=\text{if}(3\,\mathbf{l}[\$]-1,t).$$

For $\text{top}(\mathbf{l})\neq\$$, we use the fact that

$$\text{if}(\neg b,t)=t-\text{if}(b,t).$$

(3) Conjunction $(x \wedge y)$ is defined as $\mathrm{ReLU}(x+y-1)$, but because $\mathrm{if}(b, t)$ works even when $b < 0$, we have

$$\mathrm{if}(b_1 \wedge b_2, t) = \mathrm{if}(b_1 + b_2 - 1, t).$$

(4) In an expression $\mathrm{if}(b, \mathrm{top}(\mathbf{l}))$ or $\mathrm{if}(b, \mathrm{top}(\mathbf{l}))$, we can merge the two nonlinearities into one:

$$\begin{aligned}
\mathrm{if}(b, \mathrm{top}(\mathbf{l})) &= \mathrm{if}(b, \mathrm{SLU}(3\mathbf{l} - 1)) \\
&= \mathrm{SLU}(3\mathbf{l} - 1 + 2b - 2) \\
\mathrm{if}(b, \mathrm{pop}(\mathbf{l})) &= \mathrm{if}(b, 3\mathbf{l} - 2\,\mathrm{top}(\mathbf{l})) \\
&= 3\,\mathrm{if}(b, \mathbf{l}) - 2\,\mathrm{if}(b, \mathrm{top}(\mathbf{l})).
\end{aligned}$$

□

If the number of intermediate steps is bounded by $T(n)$, then the number of simulated steps of the Turing machine is also bounded by $T(n)$. So we can obtain many other equivalences. For example, RNNs that run for a finite number of steps recognize exactly the decidable languages, and RNNs that run for a polynomial number of steps recognize exactly the languages in P.

4.2.3 Real-weight RNNs with intermediate steps

The last result we look at is purely theoretical: If we allow an RNN to have real weights, how powerful is it?

Theorem 4.12 (Siegelmann and Sontag 1994)**.** *For any language L over Σ, there is a network $f = out \circ rec$, where rec is a simple RNN with real weights and* ReLU *activation functions, and out is a linear layer, that decides L in the following sense: for any string $\mathbf{w} \in \Sigma^*$,*

- *If $\mathbf{w} \in L$, then there is a T such that for all $t \in [T]$, $f(\mathbf{w} \cdot \mathrm{BOS} \cdot \mathrm{NUL}^t) = \mathbf{e}_{\mathrm{NUL}}$ and $f(\mathbf{w} \cdot \mathrm{BOS} \cdot \mathrm{NUL}^T) = \mathbf{e}_{\mathrm{ACC}}$.*
- *If $\mathbf{w} \notin L$, then there is a T such that for all $t \in [T]$, $f(\mathbf{w} \cdot \mathrm{BOS} \cdot \mathrm{NUL}^t) = \mathbf{e}_{\mathrm{NUL}}$ and $f(\mathbf{w} \cdot \mathrm{BOS} \cdot \mathrm{NUL}^T) = \mathbf{e}_{\mathrm{REJ}}$.*

Siegelmann and Sontag (1994) prove a more precise result relating complexity classes of real-weight RNNs with complexity classes of *circuits*, which we will encounter in Section 6.2.2. But their paper is most often cited simply for the claim we have stated above, and we give a much simpler proof here.

Proof. We have already seen how to encode a string over Σ as a vector of $|\Sigma|$ rational numbers. Under the same encoding, we can encode an *infinite* string as

a vector of real numbers. Let's think of an infinite string over Σ as a mapping $w\colon \mathbb{N}_{>0} \to \Sigma$, that is, $w(i)$ is the symbol at position i. Then

$$\operatorname{stack}(w) = \sum_{i=1}^{\infty} \frac{2}{3^i} \mathbf{e}_{w(i)}. \tag{4.40}$$

Given a language L, we can enumerate the (finite) strings of L in order of increasing length, as $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \ldots$. Then we can concatenate them into a single infinite string, $\langle L \rangle = \mathbf{w}^{(0)}\#\mathbf{w}^{(1)}\#\cdots$. For example, if $L = \{\mathbf{ww} \mid \mathbf{w} \in \{\mathsf{a}, \mathsf{b}\}^*\}$, then

$$\langle L \rangle = \#\mathsf{aa}\#\mathsf{bb}\#\mathsf{aaaa}\#\mathsf{abab}\#\mathsf{baba}\#\mathsf{bbbb}\#\mathsf{aaaaaa}\#\cdots.$$

The stack operations are defined exactly as before.

The construction in the proof of Theorem 4.11 can be modified to simulate a Turing machine with two tapes (or a tape and a stack is enough): the first tape is as before, while the second tape is initialized with $\langle L \rangle$. Then construct an RNN that simulates the Turing machine $M =$ "On input $\mathbf{w}$:

1. Compare the string on the first tape ($\mathbf{w}$) with the string on the second tape, starting from the current position up to (but not including) #.
2. If they are equal, *accept*.
3. If $\mathbf{w}$ is shorter than the other string, *reject*.
4. Otherwise, move the first head back to the left end, move the second head immediately to the right of the #, and goto 1. □

Bibliography

Chandra, Ashok K., Dexter Kozen, and Larry J. Stockmeyer (1981). Alternation. In: *J. ACM* 28.1, pp. 114–133.

Chen, Yining, Sorcha Gilroy, Jonathan May, and Kevin Knight (2017). Recurrent neural networks as weighted language recognizers. In: arXiv:1711.05408v1 (earlier version of a paper published at NAACL HLT 2018).

Elman, Jeffrey L. (1990). Finding structure in time. In: *Cognitive Science* 14, pp. 179–211.

Goudreau, Mark W., C. Lee Giles, Srimat T. Chakradhar, and D. Chen (1994). First-order versus second-order single-layer recurrent neural networks. In: *IEEE Transactions on Neural Networks* 5.3, pp. 511–513.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In: *Automata Studies*. Ed. by C. E. Shannon and J. McCarthy. Vol. 34. Annals of Mathematics Studies. Princeton University Press, pp. 3–42.

McCulloch, Warren and Walter Pitts (1943). A logical calculus of ideas immanent in nervous activity. In: *Bulletin of Mathematical Biophysics* 5, pp. 127–147.

Minsky, Marvin L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall.

Siegelmann, Hava T. and Eduardo D. Sontag (1994). Analog computation via neural networks. In: *Theoretical Computer Science* 131.2, pp. 331–360.

— (1995). On the computational power of neural nets. In: *Journal of Computer and System Sciences* 50.1, pp. 132–150.

Sipser, Michael (2013). *Introduction to the Theory of Computation*. 3rd. Cengage Learning.