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

$$\begin{aligned} rec\colon (\mathbb{R}^d)^* &\xrightarrow{\text{lp}} (\mathbb{R}^{d'})^* \\ (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}) &\mapsto (\mathbf{h}^{(1)}, \dots \mathbf{h}^{(T)}) \end{aligned} \tag{4.1}$$

where

$$\mathbf{h}^{(0)} = \mathbf{s} \tag{4.2}$$

$$\mathbf{h}^{(t)} = \sigma\Big(\mathbf{V}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)} + \mathbf{b}\Big) \qquad t = 1, \dots, T \tag{4.3}$$

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

$$\begin{aligned} out \colon \mathbb{R}^{d'} &\to \mathbb{R} \\ \mathbf{h}^{(t)} &\mapsto \mathbf{w} \cdot \mathbf{h}^{(t)} + b \end{aligned} \tag{4.5}$$

with parameters

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

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

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

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

At each time step t, the vector $\mathbf{h}^{(t)}$ 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 $\mathbf{h}^{(t)}$ as the XOR of $\mathbf{h}^{(t-1)}$ and w_t. Since the transition function of a simple RNN is a single-layer FFNN, and single-layer FFNN cannot compute the XOR function (Theorem 2.3), 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.

Theorem 4.3 (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 $Q = \{q_1, \ldots, q_{|Q|}\}$, start state q_1, transition function $\delta \colon Q \times \Sigma \to Q$, and accept states F. Number the symbols of Σ as $a_1, \ldots, a_{|\Sigma|}$.

As an initial attempt, define

$$\begin{aligned}
f &= out \circ rec && && (4.7)\\
rec &\colon (\mathbb{R}^{|\Sigma|})^* \to (\mathbb{R}^{|Q|})^* \\
(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}) &\mapsto (\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(T)}) && && (4.8)\\
\mathbf{h}^{(0)}[i] &= \mathbb{I}[i = 1] && i \in [|Q|] && (4.9)\\
\mathbf{g}^{(t)}[i,j] &= \mathbf{h}^{(t-1)}[i] \wedge \mathbf{x}^{(t)}[j] && t \in [T], i \in [|Q|], j \in [|\Sigma|] && (4.10)\\
\mathbf{h}^{(t)}[k] &= \sum_{\substack{i \in [|Q|], j \in [|\Sigma|] \\ \delta(q_i, a_j) = q_k}} \mathbf{g}^{(t)}[i,j] && t \in [T], k \in [|Q|] && (4.11)\\
out &\colon \mathbb{R}^{|Q|} \to \mathbb{R} \\
\mathbf{h}^{(t)} &\mapsto \sum_{\substack{i \in [|Q|] \\ q_i \in F}} \mathbf{h}^{(t)}[i] && t \in [T]. && (4.12)
\end{aligned}$$

But this doesn't work, 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). 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 4.4. *Let f be a length-preserving function*

$$\begin{aligned}
f &\colon (\mathbb{R}^{d_\text{in}})^* \xrightarrow{\text{lp}} (\mathbb{R}^{d_\text{out}})^* \\
(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}) &\mapsto (\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)}) && && (4.13)\\
\mathbf{h}^{(0)} &= \mathbf{s} && && (4.14)\\
\mathbf{h}^{(t)} &= step(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) && t \in [T] && (4.15)\\
\mathbf{y}^{(t)} &= out(\mathbf{h}^{(t)}) && && (4.16)
\end{aligned}$$

where $\mathbf{s} \in \mathbb{R}^d$, $step \colon \mathbb{R}^d \times \mathbb{R}^{d_\text{in}} \to \mathbb{R}^d$ is any continuous piecewise linear function with a finite number of pieces, and $out \colon \mathbb{R}^d \to \mathbb{R}^{d_\text{out}}$ is an affine transformation.

Then there is a network $g = out \circ rec$, where $g.rec \colon (\mathbb{R}^{d_\text{in}})^* \to (\mathbb{R}^{d'})^*$ is a simple ReLU RNN and $g.out \colon \mathbb{R}^{d'} \to \mathbb{R}^{d_\text{out}}$ is a linear layer, that is equal to f.

Proof. By Theorem 3.6, $f.step$ can be computed by a FFNN

$$\begin{aligned} f.step \colon \mathbb{R}^d \times \mathbb{R}^{d_\text{in}} &\to \mathbb{R}^d \\ (\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) &\mapsto \mathbf{h}^{(t)} && (4.17) \\ \mathbf{h}^{(t)} &= \mathbf{U}\,\text{ReLU}(\mathbf{V}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)} + \mathbf{b}). && (4.18) \end{aligned}$$

Without loss of generality, we can assume that the second layer ($\mathbf{U}$) doesn't have a bias term. This doesn't have the form that we need for $g.rec$, 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.

$$\begin{aligned} g.rec \colon (\mathbb{R}^{d_\text{in}})^* &\xrightarrow{\text{lp}} (\mathbb{R}^{d'})^* \\ (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}) &\mapsto (\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(T)}) && (4.19) \\ \mathbf{h}^{(0)} &= \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix} && (4.20) \\ \mathbf{h}^{(t)} &= \text{ReLU}\left(\begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{V}(f.\mathbf{s}) & \mathbf{VU} \end{bmatrix} \mathbf{h}^{(t-1)} + \begin{bmatrix} \mathbf{0} \\ \mathbf{W} \end{bmatrix} \mathbf{x}^{(t)} + \begin{bmatrix} 0 \\ \mathbf{b} \end{bmatrix} \right) \quad t \in [T] && (4.21) \\ g.out \colon \mathbb{R}^{d'} &\to \mathbb{R}^{d_\text{out}} \\ \mathbf{h}^{(t)} &\mapsto f.out(\begin{bmatrix} \mathbf{0} & \mathbf{U} \end{bmatrix} \mathbf{h}^{(t)}). \end{aligned}$$

□

Proof of Theorem 4.3. In our initial attempt, the step function (Eqs. (4.10) and (4.11)) is continuous piecewise linear with a finite number of pieces; indeed, it's already a two-layer FFNN. So we can use Lemma 4.4 to convert it 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 4.5. *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 will also be integers, and the $\mathbf{h}^{(t)}$ for $t > 0$ will have components all in $\{0, 1\}$. Therefore there is a DFA with 2^d states that simulates f. □

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.

Theorem 4.6 (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 an affine transformation, that is equivalent to M in the following sense: for any string $w \in \Sigma^*$,*

- *If M halts and accepts on input w, then there is a T such that for all $0 < t < T$, $f(w \cdot \mathrm{EOS}^t) = 0$ and $f(w \cdot \mathrm{EOS}^T) = 1$.*
- *If M halts and rejects on input w, then there is a T such that for all $0 < t < T$, $f(w \cdot \mathrm{EOS}^t) = 0$ and $f(w \cdot \mathrm{EOS}^T) = -1$.*
- *If M does not halt on input w, then for all $t > 0$, $f(w \cdot \mathrm{EOS}^t) = 0$.*

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 $a_1, a_2, \ldots, a_{|\Gamma|}$. To represent M's tape (which has a leftmost cell but extends infinitely to the right), we use two stacks $\ell, r \in \Gamma^*$. The stacks $\ell = \ell_1 \cdots \ell_{|\ell|}$ and $r = r_1 \cdots r_{|r|}$ represent the tape

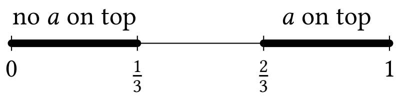
$$\ell_{|\ell|}\ell_{|\ell|-1} \cdots \ell_2\ell_1 r_1 r_2 \cdots r_{|r|-1} r_{|r|} \sqcup \sqcup \cdots$$

and the head is over symbol r_1.

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

$$\begin{aligned} \mathrm{stack} \colon \Gamma^* &\to \mathbb{Q}^{|\Gamma|} \\ \mathrm{stack}(\epsilon) &= \mathbf{0} && (4.22) \\ \mathrm{stack}(a_j \cdot z) &= \tfrac{2}{3}\mathbf{e}_j + \tfrac{1}{3}\mathrm{stack}(z). && (4.23) \end{aligned}$$

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:

$$\begin{aligned} \text{push}(\mathbf{z}, a_j) &= \tfrac{2}{3}\mathbf{e}_j + \tfrac{1}{3}\mathbf{z} && (4.24) \\ \text{top}(\mathbf{z}) &= \text{SLU}(3\mathbf{z} - 1) && (4.25) \\ \text{pop}(z) &= 3z - \text{SLU}(3\mathbf{z} - 1). && (4.26) \end{aligned}$$

Let Q_M be the states of M. Let Q contain two new states q_1 and q_2, 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_3, then $q_4, q_5, \ldots, q_{|Q|+2}$.

The hidden vectors of the RNN are

$$\mathbf{h}^{(t)} = \begin{bmatrix} \mathbf{q}^{(t)} \\ \mathbf{l}^{(t)} \\ \mathbf{r}^{(t)} \end{bmatrix} \tag{4.27}$$

where $\mathbf{q}^{(t)}$ is the one-hot vector of the current state, and $\mathbf{l}^{(t)}$ and $\mathbf{r}^{(t)}$ are the left and right stacks, respectively.

The initial vector is

$$\mathbf{h}^{(0)} = \begin{bmatrix} \mathbf{e}_1 \\ \text{stack}(\$) \\ \text{stack}(\text{␣}\$) \end{bmatrix}. \tag{4.28}$$

We initialized both stacks with a `$` on the bottom, and put an extra blank in the right stack, because it will be convenient later to assume that we never have two empty stacks.

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

$$\text{if}(c, t) = \text{SLU}(t - 1 + c) \qquad \text{if } c \text{ then } t \text{ else } 0 \tag{4.29}$$

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

$$\mathit{step}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}, \mathbf{x}\right) = \mathit{load}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}, \mathbf{x}\right) + \mathit{rewind}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right) + \mathit{left}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right) + \mathit{right}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right). \tag{4.30}$$

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

$$\begin{aligned} \mathit{load}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}, \mathbf{x}\right) = \text{if}&\left(\mathbf{q} = q_1 \wedge \mathbf{x} \neq \text{EOS}, \begin{bmatrix} q_1 \\ \text{push}(\mathbf{l}, \mathbf{x}) \\ \mathbf{r} \end{bmatrix}\right) \\ + \text{if}&\left(\mathbf{q} = q_1 \wedge \mathbf{x} = \text{EOS}, \begin{bmatrix} q_2 \\ \mathbf{l} \\ \mathbf{r} \end{bmatrix}\right). \end{aligned} \tag{4.31}$$

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

$$\mathit{rewind}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right) = \text{if}\left(\mathbf{q} = q_2 \wedge \text{top}(\mathbf{l}) \neq \$, \begin{bmatrix}q_2\\\text{pop}(\mathbf{l})\\\text{push}(\mathbf{r}, \text{top}(\mathbf{l}))\end{bmatrix}\right) + \text{if}\left(\mathbf{q} = q_2 \wedge \text{top}(\mathbf{l}) = \$, \begin{bmatrix}q_3\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right). \tag{4.32}$$

The third term handles all the left-moving transitions:

$$\mathit{left}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right) = \sum_{\substack{(q,a\to q',a',\text{L})\in\delta\\c\in\Gamma\setminus\{\$\}}} \text{if}\left(\mathbf{q} = q \wedge \text{top}(\mathbf{r}) = a \wedge \text{top}(\mathbf{l}) = c, \begin{bmatrix}q'\\\text{pop}(\mathbf{l})\\\text{push}(\mathbf{r}, c)\end{bmatrix}\right) + \sum_{(q,a\to q',a',\text{L})\in\delta} \text{if}\left(\mathbf{q} = q \wedge \text{top}(\mathbf{r}) = a \wedge \text{top}(\mathbf{l}) = \$, \begin{bmatrix}q'\\\text{push}(\text{pop}(\mathbf{l}), c)\\\mathbf{r}\end{bmatrix}\right). \tag{4.33}$$

The last term handles all the right-moving transitions:

$$\mathit{right}\left(\begin{bmatrix}\mathbf{q}\\\mathbf{l}\\\mathbf{r}\end{bmatrix}\right) = \sum_{(q,a\to q',a',\text{R})\in\delta} \text{if}\left(\mathbf{q} = q \wedge \text{top}(\mathbf{r}) = a, \begin{bmatrix}q'\\\text{push}(\mathbf{l}, a')\\\text{pop}(\mathbf{r})\end{bmatrix}\right) + \sum_{(q,\text{␣}\to q',a',\text{R})\in\delta} \text{if}\left(\mathbf{q} = q \wedge \text{top}(\mathbf{r}) = \$, \begin{bmatrix}q'\\\text{push}(\mathbf{l}, a')\\\mathbf{r}\end{bmatrix}\right). \tag{4.34}$$

Siegelmann and Sontag (1994) give essentially the above *step*, which turns out to have four layers, and then construct an RNN that takes four steps to simulate one step of M. They then give a more complicated direct construction of an RNN that only takes one step to simulate each step of M. But here, we use Lemma 4.4 to construct a simple RNN with a recurrent step equivalent to *step*. □

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 4.7 (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 rational weights and* ReLU *activation functions, and out is a linear layer, such that decides L in the following sense: for any string $w \in \Sigma^*$,*

- *If $w \in L$, then there is a T such that for all $0 < t < T$, $f(w \cdot \mathrm{EOS}^t) = 0$ and $f(w \cdot \mathrm{EOS}^T) = 1$.*
- *If $w \notin L$, then there is a T such that for all $0 < t < T$, $f(w \cdot \mathrm{EOS}^t) = 0$ and $f(w \cdot \mathrm{EOS}^T) = -1$.*

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 5.2. 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, $w(t)$ is the symbol at position t. Then

$$\mathrm{stack}(w) = \sum_{t=1}^{\infty} \frac{2}{3^t} \mathbf{e}_{w(t)}. \tag{4.35}$$

Given a language L, we can enumerate the strings of L in order of increasing length, as $w^{(1)}, w^{(2)}, \ldots$. Then we can concatenate them into a single infinite string, $\langle L \rangle = w^{(1)}\#w^{(2)}\# \cdots$. For example, if $L = \{ww \mid w \in \{\mathsf{a}, \mathsf{b}\}^*\}$, 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 4.6 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 w:

1. Compare the string on the first tape (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 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

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.