Verification of Neural Networks

Lecture Notes (in progress)

Benedikt Bollig

Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF
Gif-sur-Yvette, France

March 1, 2024

Contents

Preface

These are the notes of a lecture series given in the academic year 2023/2024 as part of the MPRI 2.8 course "Advanced Techniques of Verification".

The neural-network drawings presented in Sections 3.1 and 3.2 are adopted from Izaak Neutelings [38].

Any comments, suggestions, errata, etc. are very welcome. Please send them by email to `**ll**@lmf.cnrs.fr`.

This work is licensed under a Creative Commons Attribution 4.0 International License.

Chapter 1

Introduction

AI-based systems, particularly neural networks, are playing an important role in our daily lives. Neural networks are used for image and speech recognition, in autonomous cars, medical diagnostics, anomaly detection, financial and weather forecasting, etc. As they are increasingly used in safety-critical applications (e.g., in autonomous cars or medical diagnostics), there are of course high safety requirements for AI-based components. However, neural networks are black-box models with a rather intransparent structure where small changes can have significant effects. Another reason to ask: Can we give formal guarantees for a given neural network? Or, in other work, can we *verify* a neural network?

Formal methods are well-suited to provide answers here. They include techniques that first mathematically model systems and requirements specifications and then algorithmically determine the compatibility of system and specification. Formal methods are often associated with verifying systems written in some programming language. Now, neural networks are not a program in this sense (they are written or trained by a machine). However, like "programs," they follow a precise sequence of instructions and are, in principle, amenable to formal verification.

While the specification of programs is often natural (think of terms like termination or deadlock freedom), it may appear unclear what correctness means for neural networks. For example, when is an image classifier that is supposed to classify animals correct? Probably, when it recognizes a dog as such, a cat as a cat, and so on. But to apply formal methods, we must formalize correctness in a precise mathematical sense. Now, to write down when a picture represents a dog and when a cat, is naturally incredibly challenging. And if we could, we probably would not need a neural network anymore. But there are many other desirable properties that we *can* formalize. For example, when an image classifier is robust, that is, when small changes in a given image do not entirely change the classification. Or when a neural network is fair, that is, when it does not take sensitive features into account in order to assign credits or jobs.

Neural networks often serve as building blocks of larger systems and may act as a controler, i.e., choose an action to be performed in a given state. We are then in the setting of *reactive systems*. Verifying them is particularly challenging, as specifications usually combine temporal properties with arithmetic expressions such a "whenever a system variable x reaches a critical threshold γ, i.e., $x \geq \gamma$, then there is a time point in the near future when it falls back below the threshold, i.e., $x < \gamma$.

Applying formal methods to neural networks is an exciting new field with many interesting developments. We refer here to various survey papers and lecture notes [3, 10, 31, 54, 57] that we recommend for further reading. We do not aim to provide optimal algorithms. The goal of this lecture is to give a sense of what verification of a neural network means and to explore its theoretical possibilities and limitations. However, one should remember that scalability is an essential criterion for verification methods for neural networks, which can have millions of parameters.

Chapter 2

Preliminaries

In this chapter, we recall some standard concepts from linear algebra and automata theory. Linear algebra allows one to describe neural networks in a compact, elegant manner. We will use automata-based techniques (among others) to address their verification.

2.1 Sets, Functions, Vectors, Matrices

Sets and Functions. By $\mathbb{N} = \{0, 1, 2, \ldots\}$, we denote the set of natural numbers, and by $\mathbb{N}_+ = \{1, 2, \ldots, \}$ the set of positive natural numbers. The set of real numbers is denoted $\mathbb{R}$, and the set of rational numbers by $\mathbb{Q}$. As part of an input to a decision problem or of an object like a matrix, we implicitly assume that a rational number is effectively given in terms of binary encodings of its numerator and denominator. For $x, y \in \mathbb{R}$, we let $[x, y] = \{z \in \mathbb{R} \mid x \leq z \leq y\}$, $(x, y] = \{z \in \mathbb{R} \mid x < z \leq y\}$, and so forth.

For functions $f : A \to B$ and $g : B \to C$, we denote by $g \circ f : A \to C$ the composition of f and g, defined by $(g \circ f)(a) = g(f(a))$ for all $a \in A$. Moreover, given $A' \subseteq A$, we let $f\big|_{A'} : A' \to B$ denote the *restriction* of f to the domain A'.

For a finite set A, the number of elements in A is denoted by $|A|$.

Vectors and Matrices. Let $m, n \in \mathbb{N}_+$. For a vector $\mathbf{x} \in \mathbb{R}^n$ and $i \in \{1, \ldots, n\}$, we let x_i refer to the i-th component of $\mathbf{x}$, i.e., $\mathbf{x} = (x_1, \ldots, x_n)^\top$. If $n = 1$, we may simply write x for $\mathbf{x}$. Similarly, $\mathbf{y} = (y_1, \ldots, y_n)^\top$, $\mathbf{x}' = (x'_1, \ldots, x'_n)^\top$, and so on. Moreover, for a matrix $\mathbf{A} \in \mathbb{R}^{n\times m}$, $i \in \{1, \ldots, n\}$, and $j \in \{1, \ldots, m\}$, we let $a_{i,j}$ refer to the element of $\mathbf{A}$ in the i-th row and j-th column, i.e.,

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \in \mathbb{R}^{m\times n}\,.$$

Similarly, $\mathbf{A}' = \left(a'_{i,j}\right)_{i,j}$ and so forth.

For $f = (f_1, \ldots, f_m)$ with $f_1, \ldots, f_m : \mathbb{R} \to \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^m$, we define

$$f(\mathbf{x}) = (f_1(x_1), \ldots, f_m(x_m))^\top \in \mathbb{R}^m\,.$$

The *vertical concatenation* of matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$ is defined by $\mathbf{A} \boxminus \mathbf{B} = \mathbf{C} \in \mathbb{R}^{(m+k)\times n}$ where $c_{i,j} = a_{i,j}$ for all $i \in \{1,\dots,m\}$ and $j \in \{1,\dots,n\}$, and $c_{i,j} = b_{i,j}$ for all $i \in \{m+1,\dots,m+k\}$ and $j \in \{1,\dots,n\}$. The *horizontal concatenation* of $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{m\times k}$ is defined accordingly by $\mathbf{A} \boxbar \mathbf{B} = \mathbf{C} \in \mathbb{R}^{m\times(n+k)}$ where $c_{i,j} = a_{i,j}$ for all $i \in \{1,\dots,m\}$ and $j \in \{1,\dots,n\}$, and $c_{i,j} = b_{i,j}$ for all $i \in \{1,\dots,m\}$ and $j \in \{n+1,\dots,n+k\}$. The special case of vectors is defined analogously. In particular, the vertical concatenation of $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$ is $\mathbf{x} \boxminus \mathbf{y} = (x_1,\dots,x_m,y_1,\dots,y_n)^\top \in \mathbb{R}^{m+n}$.

For a given $m \in \mathbb{N}_+$ (which we suppose to be clear from the context), we let

$$\mathrm{argmax} : \begin{cases} \mathbb{R}^m \to 2^{\{1,\dots,m\}} \\ \mathbf{x} \mapsto \{i \in \{1,\dots,m\} \mid x_i = \max(\mathbf{x})\}\,. \end{cases}$$

For $m \in \mathbb{N}_+$, the set of permutations $\pi : \{1,\dots,m\} \to \{1,\dots,m\}$ is denoted S_m. We extend π to $\pi : \mathbb{R}^m \to \mathbb{R}^m$ letting $\pi(\mathbf{x}) = (x_{\pi(1)},\dots,x_{\pi(m)})$.

Below we define two properties that play an important rule in the realm of neural networks:

Definition 2.1: Permutation Equivariance and Invariance

Let $m \in \mathbb{N}_+$ and A be a set.

- A function $f : \mathbb{R}^m \to A$ is called *permutation invariant* if, for all $\mathbf{x} \in \mathbb{R}^m$ and $\pi \in S_m$, we have $f(\mathbf{x}) = f(\pi(\mathbf{x}))$.
- A function $f : \mathbb{R}^m \to \mathbb{R}^m$ is called *permutation equivariant* if, for all $\mathbf{x} \in \mathbb{R}^m$ and $\pi \in S_m$, we have $f(\pi(\mathbf{x})) = \pi(f(\mathbf{x}))$.

2.2 Languages and Büchi Automata

Automata are a useful tool in verification and for deciding arithmetic theories such as Presburger arithmetic and linear real arithmetic [20]. In this course, we will need to decide linear real arithmetic and, to do so, rely on Büchi automata, which are devices that run over infinite words (or strings). Later on, these infinite words will represent real numbers.

An *alphabet* is a nonempty set (possibly infinite). Let Σ be an alphabet. A *finite word* over Σ of length $n \in \mathbb{N}$ is a finite sequence $w = u_1 \dots u_n$ with $u_1,\dots,u_n \in \Sigma$. We denote the length n of w by $|w|$. If $n = 0$, then w is the empty word, denoted by ε. An infinite word over Σ is a countably infinite sequence $w = u_1u_2u_3\dots$ with $u_1,u_2,\dots \in \Sigma$. The set of finite words over Σ is denoted by Σ^*, the set of nonempty finite words over Σ by Σ^+ (i.e., $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$), and the set of infinite words by Σ^ω.

We will often deal with mappings of the form $\delta : Q \times \Sigma \to Q$ where Q is a (possibly infinite) set. This mapping can be inductively extended to $\hat{\delta} : Q \times \Sigma^* \to Q$ letting $\hat{\delta}(q,\varepsilon) = q$ and, for $w \in \Sigma^*$ and $u \in \Sigma$, $\hat{\delta}(q, w\cdot u) = \delta(\hat{\delta}(q,w),u)$. Abusing notation, we usually still write δ instead of $\hat{\delta}$.

In the remainder of this section, we consider finite alphabets.

Definition 2.2: Büchi Automaton

Let Σ be a finite alphabet. A *Büchi automaton* over Σ is a tuple $\mathcal{A} = (Q, \Delta, \iota, F)$ where

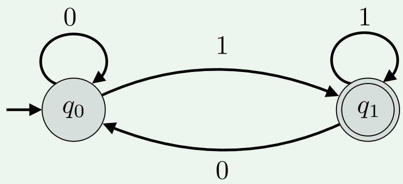
- Q is a finite set of *states*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is the set of *transitions*
- $\iota \in Q$ is the *initial state*, and
- $F \subseteq Q$ is the set of *final states*.

Büchi automaton $\mathcal{A}$ recognizes a language $L(\mathcal{A}) \subseteq \Sigma^\omega$ as follows. A *run* of $\mathcal{A}$ is an infinite sequence $\rho = q_0 u_1 q_1 u_2 q_2 \ldots \in Q(\Sigma Q)^\omega$ such that $q_0 = \iota$ and, for all $i \in \mathbb{N}_+$, $(q_{i-1}, u_i, q_i) \in \Delta$. The *label* of ρ is defined as $label(\rho) = u_1 u_2 u_3 \ldots \in \Sigma^\omega$. Run ρ is called *accepting* if it sees some final state infinitely often, i.e., the set $\{i \in \mathbb{N} \mid q_i \in F\}$ is infinite. Finally, the *language recognized by* $\mathcal{A}$ is defined by

$$L(\mathcal{A}) = \{label(\rho) \mid \rho \text{ is an accepting run of } \mathcal{A}\} \subseteq \Sigma^\omega .$$

Example 2.1:

The figure below depicts a Büchi automaton $\mathcal{A} = (Q, \Delta, \iota, F)$ over the alphabet $\Sigma = \{0, 1\}$. We have $Q = \{q_0, q_1\}$, $\iota = q_0$, and $F = \{q_1\}$. The set of transitions Δ includes $(q_0, 0, q_0)$, $(q_0, 1, q_1)$, etc. The language $L(\mathcal{A})$ is the set of words from Σ^ω in which letter 1 occurs infinitely often.



Büchi automata enjoy several useful closure and decidability properties:

Theorem 2.1: Closure Properties of Büchi Automata

Let Σ be a finite alphabet and let $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2$ be Büchi automata over Σ. We can effectively construct a Büchi automaton

(a) $\mathcal{A}_1 \cup \mathcal{A}_2$ over Σ such that $L(\mathcal{A}_1 \cup \mathcal{A}_2) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$;

(b) $\mathcal{A}_1 \cap \mathcal{A}_2$ over Σ such that $L(\mathcal{A}_1 \cap \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$;

(c) $\overline{\mathcal{A}}$ over Σ such that $L(\overline{\mathcal{A}}) = \Sigma^\omega \setminus L(\mathcal{A})$.

Theorem 2.2: Büchi Automata Emptiness

The following problem is decidable:

Input: A finite alphabet Σ and a Büchi automaton over Σ.

Question: Is $L(\mathcal{A})$ nonempty?

The complexity is linear in the number of states and transitions.

For more background on languages and automata, we refer the reader to [52].

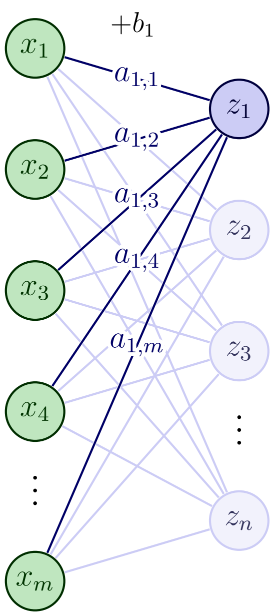
Chapter 3

Feed-Forward Neural Networks

In this chapter, we define feed-forward neural networks and their verification problem. We also present a specification language for neural networks. It is based on linear real arithmetic, which encompasses, as we will see, many pertinent properties of neural networks.

3.1 Definition of Neural Networks

A neural network is a stack of layers. Every layer transforms an input vector into an output vector. The more layers we have, the deeper is the network. Technically, *deep learning* starts when there are at least two layers. A layer and its computation are depicted in Figure 3.1. It consists of n input nodes and m output nodes, and it maps an input vector $\mathbf{x} = (x_1, \dots, x_m)^\top \in \mathbb{R}^m$ to an output vector $\mathbf{y} = (y_1, \dots, y_n)^\top \in \mathbb{R}^n$. Here, intermediate values z_i are computed as a linear combination $b_i + \sum_j a_{i,j} \cdot x_j$. Thus, the transformation induced by a layer can be written in terms of matrix multiplications.



$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$\mathbf{z} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = f(\mathbf{z})$$

Figure 3.1: A neural network layer with activation function $f : \mathbb{R}^n \to \mathbb{R}^n$

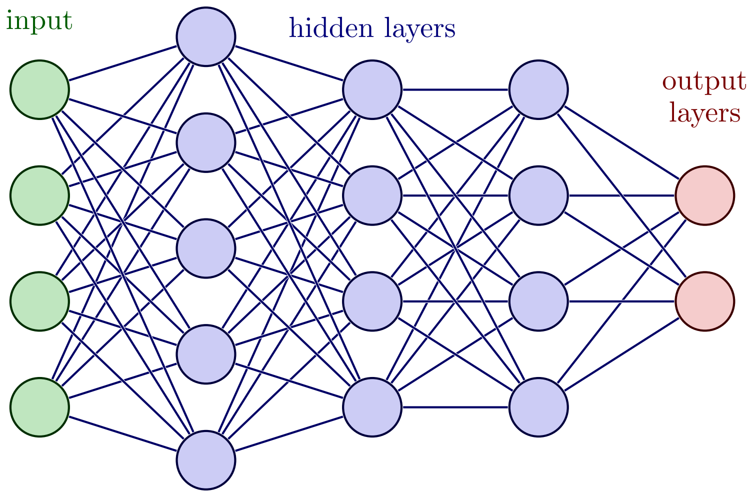


Figure 3.2: Structure of a neural network with 4 layers (+ 1 input layer)

Definition 3.1: Layer

Let $m, n \in \mathbb{N}_+$. A *feed-forward layer* with input dimension m and output dimension n is a triple $\mathscr{L} = (\mathbf{A}, \mathbf{b}, f)$ where $\mathbf{A} \in \mathbb{Q}^{n \times m}$ is the *weight matrix*, $\mathbf{b} \in \mathbb{Q}^n$ is the *bias vector*, and $f : \mathbb{R}^n \to \mathbb{R}^n$ is the *activation function*.

We let $in(\mathscr{L}) = m$ denote the *input dimension* of $\mathscr{L}$. Moreover, $out(\mathscr{L}) = n$ is the *output dimension*, which is usually referred to as the *number of neurons* of $\mathscr{L}$. In other words, the weights of the i-th layer are given by the i-th row of $\mathbf{A}$. We also let $dim(\mathscr{L}) = (m, n)$ (note the difference from the dimension of the matrix $\mathbf{A}$. Layer $\mathscr{L}$ defines the function

$$[\![\mathscr{L}]\!] : \begin{cases} \mathbb{R}^{in(\mathscr{L})} \to \mathbb{R}^{out(\mathscr{L})} \\ \mathbf{x} \mapsto f(\mathbf{A} \cdot \mathbf{x} + \mathbf{b}) \, . \end{cases}$$

Let us turn to neural networks. A *(feed-forward) neural network* is a sequence of feed-forward layers such that neighboring layers have compatible input/output dimensions. The idea is that the output of one layer is the input to the next layer. Thus, the function defined by a neural network is the composition of the functions defined by its layers. The neural network illustrated in Figure 3.2 consists of 4 layers (we omit weights and activation functions).

Definition 3.2: Feed-forward Neural Network

A *feed-forward neural network* is a sequence $\mathcal{N} = (\mathscr{L}^{(1)}, \ldots, \mathscr{L}^{(\ell)})$ of layers $\mathscr{L}^{(k)} = (\mathbf{A}^{(k)}, \mathbf{b}^{(k)}, f^{(k)})$ such that $out(\mathscr{L}^{(k)}) = in(\mathscr{L}^{(k+1)})$ for all $k \in \{1, \ldots, \ell - 1\}$.

As, in this, chapter, we only talk about feed-forward neural networks (rather than recurrent neural networks or graph neural networks), we just say *neural network*.

Note that, in the literature, the number of layers usually considers the collection of input values as a separate layer, This indeed makes sense when looking at the graph representation of a neural network (cf. Example 3.1). Thus, while $\mathcal{N} = (\mathscr{L}^{(1)}, \ldots, \mathscr{L}^{(\ell)})$ has ℓ layers, the “graph representation” exhibits $\ell + 1$ layers.

Similarly to a layer, $in(\mathcal{N}) = in(\mathscr{L}^{(1)})$ is the *input dimension*, and $out(\mathcal{N}) = out(\mathscr{L}^{(\ell)})$ the output dimension of $\mathcal{N}$. In addition, we let $dim(\mathcal{N}) = (in(\mathcal{N}), out(\mathcal{N}))$. The neural network computes $[\![\mathcal{N}]\!] : \mathbb{R}^{in(\mathcal{N})} \to \mathbb{R}^{out(\mathcal{N})}$ defined as the function composition

$$[\![\mathcal{N}]\!] = [\![\mathscr{L}^{(\ell)}]\!] \circ \ldots \circ [\![\mathscr{L}^{(1)}]\!] .$$

We will switch between a graph representation and matrix view at discretion. Moreover, we will consider $\mathbb{R}^m$ and $\mathbb{R}^n$ as sets of vectors or sets of tuples whatever is more convenient. In particular, we may simply write $[\![\mathcal{N}]\!](x_1, \ldots, x_m) = (y_1, \ldots, y_n)$ instead of $[\![\mathcal{N}]\!]((x_1, \ldots, x_m)^\top) = ((y_1, \ldots, y_n)^\top)$.

Two neural networks $\mathcal{N}_1 = (\mathscr{L}_1^{(1)}, \ldots, \mathscr{L}_1^{(\ell_1)})$ and $\mathcal{N}_2 = (\mathscr{L}_2^{(1)}, \ldots, \mathscr{L}_2^{(\ell_2)})$ such that $out(\mathcal{N}_1) = in(\mathcal{N}_2)$ can be concatenated, and we let

$$\mathcal{N}_1 \cdot \mathcal{N}_2 = (\mathscr{L}_1^{(1)}, \ldots, \mathscr{L}_1^{(\ell_1)}, \mathscr{L}_2^{(1)}, \ldots, \mathscr{L}_2^{(\ell_2)}) .$$

Note that $dim(\mathcal{N}_1 \cdot \mathcal{N}_2) = (in(\mathcal{N}_1), out(\mathcal{N}_2))$.

Activation Functions. There are different types of activation functions used in practice. They usually depend on the concrete task at hand (e.g., classification vs. regression) and the type of the layer (internal vs. output layer), but also on training-related issues such as the choice of the loss function (a precise discussion is, however, not in the scope of this course). Often, a layer $\mathscr{L} = (\mathbf{A}, \mathbf{b}, f)$ has a *local activation function* f in the following sense: There is $g : \mathbb{R} \to \mathbb{R}$ such that, for all $\mathbf{x} \in \mathbb{R}^n$ and $i \in \{1, \ldots, n\}$, we have $f(\mathbf{x}) = (g(x_1), \ldots, g(x_n))^\top$. The definitions and graphs of some common local activation functions are given in Figure 3.3 (the activation function NLReLU was recently studied in [33]). Their extensions to $\mathbb{R}^n \to \mathbb{R}^n$ are denoted in the same way (σ, tanh, and ReLU), where we always assume that n is clear from the context. Another example of a local activation function is the identity function, which we denote by $\mathrm{id} : \mathbb{R}^n \to \mathbb{R}^n$ (again assuming that n is understood). An instance of a *global activation function* is

$$\text{softmax} : \begin{cases} \mathbb{R}^n \to \mathbb{R}^n \\ \mathbf{x} \mapsto (y_1, \ldots, y_n) \text{ where } y_i = \dfrac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} \end{cases}$$

It “squeezes” or normalizes a vector so that it represents a probability distribution. It is, therefore, frequently used in classification tasks.

An important class of neural network employs ReLU activation functions (or the identity function, mostly in output layers):

Definition 3.3: Standard and ReLU Neural Network

A neural network $\mathcal{N} = (\mathscr{L}^{(1)}, \ldots, \mathscr{L}^{(\ell)})$ with layers $\mathscr{L}^{(k)} = (\mathbf{A}^{(k)}, \mathbf{b}^{(k)}, f^{(k)})$ is called a *ReLU neural network* if, for all $k \in \{1, \ldots, \ell\}$, we have $f^{(k)} \in \{\mathrm{id}, \mathrm{ReLU}\}$.

Classification vs. Regression. As far as feed-forward neural networks are concerned, two predominant classes of tasks are *classification* and *regression*. In a regression problem, the goal is to predict a continuous, numerical value or quantity. In other words, one is

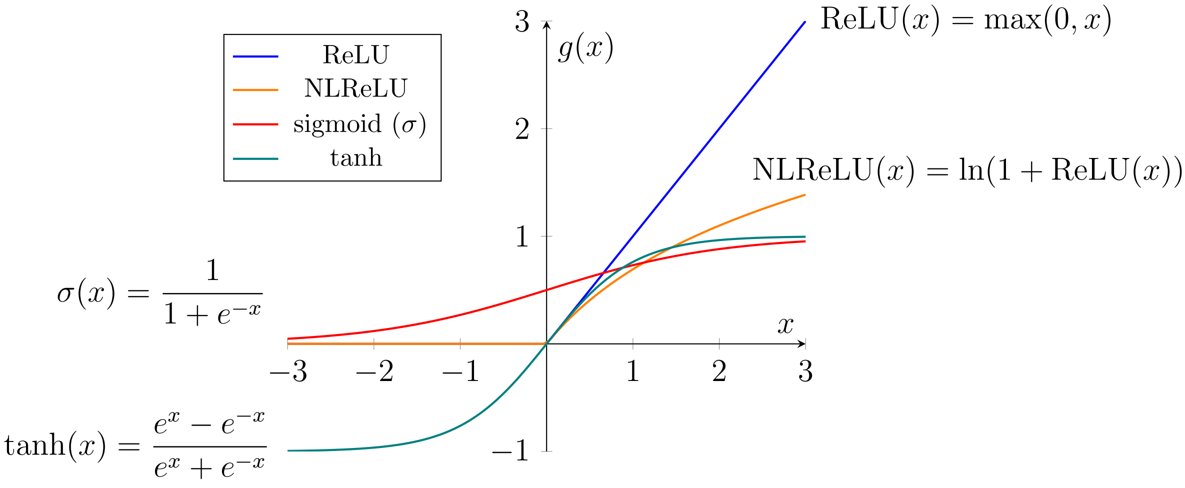
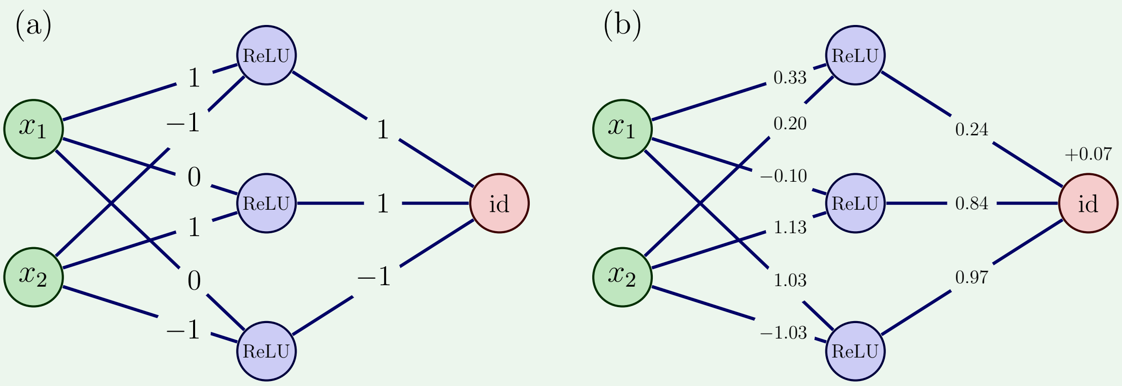


Figure 3.3: (Local) activation functions given by $g : \mathbb{R} \to \mathbb{R}$

trying to find a relationship between input features and the output, which is a real-valued number. Corresponding tasks can be price prediction, weather forecast, or estimating health indicators. In a classification problem, objects are assigned a category or label among m labels. In that case, frequently, the last layer has output dimension n and uses a softmax activation function, which works well with the categorical cross-entropy loss function during training. When $\mathcal{N}$ acts as a classifier with $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}^n$, element $\mathbf{x} \in \mathbb{R}^n$ is assigned category $\min(\operatorname{argmax}([\![\mathcal{N}]\!](\mathbf{x}))) \in \{1, \ldots, n\}$. In the case of binary classification (i.e., in presence of two classes to choose from), it is also common to have $n = 1$ and σ as activation function in the last layer, and to choose one class or the other depending on whether $[\![\mathcal{N}]\!](\mathbf{x}) \in [0, 1]$ exceeds a given threshold $\gamma \in [0, 1]$. In the case of binary classification, possible applications are object detection, fraud detection, medical diagnostics, etc.

Example 3.1: Neural Networks

Below are two simple neural networks:



(a) We have $\mathcal{N} = (\mathscr{L}^{(1)}, \mathscr{L}^{(2)})$ where $\mathscr{L}^{(k)} = (\mathbf{A}^{(k)}, \mathbf{b}^{(k)}, f^{(k)})$. Function $f^{(1)}$ is the ReLU activation function and $f^{(2)}$ is the identity. Moreover, we have

$\mathbf{b}^{(1)} = (0,0,0)^\top$, $\mathbf{b}^{(2)} = (0)$,

$$\mathbf{A}^{(1)} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \in \mathbb{R}^{3\times 2}\,, \text{ and } \mathbf{A}^{(2)} = \begin{pmatrix} 1 & 1 & -1 \end{pmatrix} \in \mathbb{R}^{1\times 3}\,.$$

Note that $\mathcal{N}$ computes the maximum function, i.e., $[\![\mathcal{N}]\!](x_1, x_2) = \max(x_1, x_2) = \max(x_1 - x_2, 0) + x_2$ for all $(x_1, x_2) \in \mathbb{R}^2$.

(b) This neural network has the same structure and activation functions as in (a), but different weights: We have $\mathbf{b}^{(1)} = (0,0,0)^\top$, $\mathbf{b}^{(2)} = (0.07)$,

$$\mathbf{A}^{(1)} = \begin{pmatrix} 0.33 & 0.2 \\ -0.1 & 1.13 \\ 1.03 & -1.03 \end{pmatrix} \in \mathbb{R}^{3\times 2}\,, \text{ and } \mathbf{A}^{(2)} = \begin{pmatrix} 0.24 & 0.84 & 0.97 \end{pmatrix} \in \mathbb{R}^{1\times 3}\,.$$

It is, however, less clear what $\mathcal{N}$ computes. For example, we have $[\![\mathcal{N}]\!](3, 2) = 3.049$, $[\![\mathcal{N}]\!](4, 9) = 9.026$, and $[\![\mathcal{N}]\!](4, 93) = 92.79$.

The first neural network is taken from [21], where the authors study the question how many layers are needed to compute certain functions in a ReLU neural network

Convolutional Neural Networks. Note that the neural networks that we defined above are fully connected: every neuron is connected to all neurons in the previous layer. In particular, all the weights/parameters in a weight matrix are, in principle, trainable and can be adjusted by the learning algorithm. There are other types of neural networks, such as convolutional neural networks (CNNs), that relax this condition. A CNN is illustrated in Figure 3.4. Certain neurons in CNNs share weights and they are sparse in the sense that some of the connections to preceding neurons are missing. A CNN can be captured in terms of our definition by setting the corresponding weights to 0. However, one should have in mind that, when considering training algorithms, one has to make a distinction between trainable parameters and those that cannot be modified. When defining neural networks as graphs, this is simple, as one would just omit the corresponding edges.

3.2 A Specification Language for Neural Networks

In this section, we will discuss what it means for a neural network to be *correct*. Here, we mean correctness in a strict mathematical sense. This, in turn, requires a formal specification to be given. Below are some examples of specifications. After discussing them, we will see a formal specification language that encompasses all of them.

(a) $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}$ computes the maximum function.

(b) $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}^m$ implements a sorting algorithm.

(c) $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}^n$ is permutation invariant.

(d) $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}^m$ is permutation equivariant.

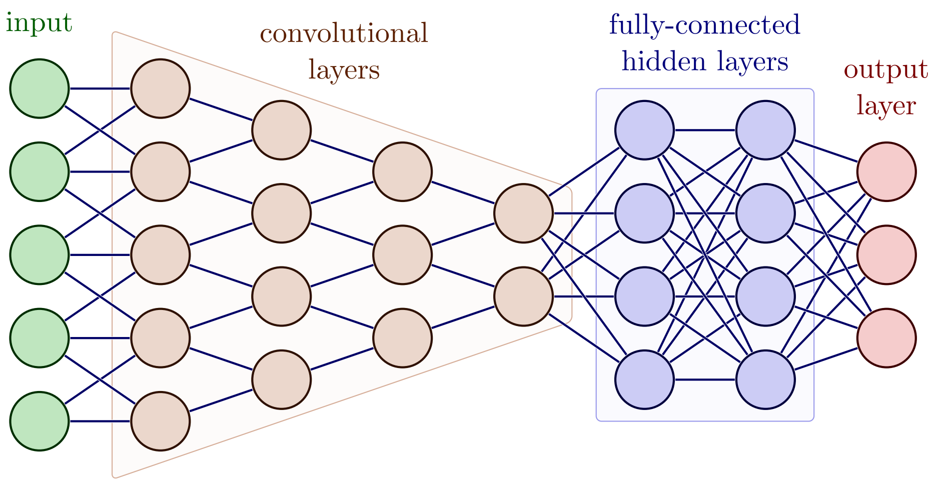


Figure 3.4: A convolutional neural network (CNN)

(e) $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}^n$ such that, for given sets $R \subseteq \mathbb{R}^m$ and $K \subseteq \{1, \ldots, m\}$, the following holds: for all $\mathbf{x}, \mathbf{x}' \in R$ satisfying $x_i = x'_i$ for all $i \in K$, we have $\text{argmax}([\![\mathcal{N}]\!](\mathbf{x})) = \text{argmax}([\![\mathcal{N}]\!](\mathbf{x}'))$.

(f) $[\![\mathcal{N}]\!] : \mathbb{R}^m \to \mathbb{R}^n$ such that, for a given set $R \subseteq \mathbb{R}^m$ and $\varepsilon > 0$, the following holds: for all $\mathbf{x} \in R$ and $\mathbf{x}' \in \mathbb{R}^m$ such that $\|\mathbf{x}, \mathbf{x}'\|_{\mathsf{Manhattan}} \leq \varepsilon$, we have $\text{argmax}([\![\mathcal{N}]\!](\mathbf{x})) = \text{argmax}([\![\mathcal{N}]\!](\mathbf{x}'))$.[1]

(g) $[\![\mathcal{N}_1]\!], [\![\mathcal{N}_2]\!] : \mathbb{R}^m \to \mathbb{R}^n$ such that, for a given set $R \subseteq \mathbb{R}^m$, the following holds: for all $\mathbf{x} \in R$, we have $\text{argmax}([\![\mathcal{N}_1]\!](\mathbf{x})) = \text{argmax}([\![\mathcal{N}_2]\!](\mathbf{x}))$.

Exercise 3.1:

For all of the above properties, discuss whether they are typically relevant for machine-learning tasks and provide corresponding examples. Can you come up with other relevant specifications? Are there specifications that imply others?

Solution:

(a) Computing the maximum function is not a typical machine-learning task: it does not require learning from data to discover unknown or hidden patterns (though, in principle, the task can be addressed by learning a machine-learning model).

(b) The same discussion as for (a) applies to the sorting problem.

(c) Here, we aggregate multiple inputs into one or several values. Suppose every input value is the age of a particular person in a group of m people, and that the neural networks makes a prediction on the number of votes that a particular candidate may receive in the upcoming election. The prediction should not depend on the order of the input values. In other words, permuting the input values should result in the same output value.

[1] For $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$, the *Manhattan distance* of $\mathbf{x}$ and $\mathbf{x}'$ is defined as $\|\mathbf{x}, \mathbf{x}'\|_{\mathsf{Manhattan}} = \sum_{i=1}^m |x_i - x'_i|$.

(d) Similarly to the previous case, suppose that each input represents the probability of a medical staff member being infected, and the output determines, for every medical staff member, whether they should be tested or quarantined. Again, the decision per agent should not depend on the order of the input values. That is, permuting the input values should result in the same output values permuted.

(e) Suppose the neural network takes features of a person as input, such as age, gender, salary, etc., and the neural network's task is to decide whether a loan is granted. This decision should be independent of sensitive features like gender or ethnicity. Such a property is referred to as a fairness property. The property thus considers that the features in $\{1, \ldots, m\} \setminus K$ are sensitive.

(f) We can think of $\mathcal{N}$ as an image classifier. If R represents a set of images, e.g., the set the classifier was trained on, we would like $\mathcal{N}$ to be robust in the sense that small perturbations on the input images do not change the predictions. This would be particularly important when $\mathcal{N}$ is used in an autonomous car to detect traffic signs.

(g) The formula states that neural networks $\mathcal{N}_1$ and $\mathcal{N}_2$ are equivalent on an input set R. That is, for every input in R, they yield the same index/class from $\{1, \ldots, n\}$. Suppose that $\mathcal{N}_2$ is much smaller than $\mathcal{N}_1$. If the property is satisfied, we could safely replace $\mathcal{N}_1$ with the more efficient neural network $\mathcal{N}_2$.

Exercise 3.2:

Let $n = 3$. For each of the specifications (a) and (b), provide a neural network satisfying it.

Exercise 3.3:

Consider the neural networks from Example 3.1 and the properties (a)–(f) above. For every combination of a neural network $\mathcal{N}$ and a property φ, verify whether φ is a suitable specification for $\mathcal{N}$ (syntactically) and, if so, whether $\mathcal{N}$ satisfies φ. Can the two neural networks from the example be considered equivalent? And what would a corresponding specification look like?

The formal specification language for neural networks will be based on *linear real arithmetic*, a decidable logic that allows one to combine logical connectives with linear expressions. It is defined over an infinite countable set of variables $\mathcal{X} = \{x, y, x_1, x_2, \ldots\}$ that range over the real numbers.[2]

[2] Note that we use x etc. to denote both real numbers and variables. Variables will be *interpreted* as real numbers so that denoting them in the same way makes sense. However, it is important to keep in mind that variables and real numbers are different objects.

Definition 3.4: Linear Real Arithmetic

Formulas from LRA (linear real arithmetic) are given by the following grammar:

$$\begin{array}{llll} \text{terms} & t & ::= & a \cdot x \mid b \mid t + t \\ \text{formulas} & \varphi & ::= & t \leq t \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \end{array}$$

where $x \in \mathcal{X}$ and $a, b \in \mathbb{Q}$.

An occurrence of a variable is *free* in an LRA formula φ if it is not in the scope of a quantifier $\exists/\forall$. A variable is called free in φ if it has some free occurrence in φ. Given a tuple of variables $\mathbf{x} = (x_1, \dots, x_n)$, we may write $\varphi(\mathbf{x})$ or $\varphi(x_1, \dots, x_n)$ for a formula φ whose free variables are among $x_1, \dots, x_n$ (though not all of them need to have a free occurrence). A *sentence* is a formula without free variables.

We define some common abbreviations such as x for $1 \cdot x$, $t_1 = t_2$ for $t_1 \leq t_2 \wedge t_2 \leq t_1$, $t_1 < t_2$ for $t_1 \leq t_2 \wedge \neg(t_1 = t_2)$, $\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$ $\varphi \Rightarrow \psi$ for $\neg\varphi \vee \psi$, $\varphi \Leftrightarrow \psi$ for $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$, $\forall x.\varphi$ for $\neg\exists x.\neg\varphi$, etc. We will also write $\exists x_1, \dots, x_n.\varphi$ instead of $\exists x_1.\exists x_2.\dots\exists x_n.\varphi$ and so forth.

Moreover, for (tuples of) variables x, $\mathbf{y} = (y_1, \dots, y_n)$, $\mathbf{y}' = (y'_1, \dots, y'_n)$, and a set $K \subseteq \{1, \dots, m\}$, we define the following abbreviations:

$$\begin{aligned} x \in \mathbf{y} \;&\equiv\; \bigvee_{i=1}^{n} x = y_i \\ x = \max(\mathbf{y}) \;&\equiv\; x \in \mathbf{y} \;\wedge\; \bigwedge_{i=1}^{n} y_i \leq x \\ \operatorname{argmax}(\mathbf{y}) = K \;&\equiv\; \bigwedge_{i \in K} y_i = \max(\mathbf{y}) \;\wedge\; \bigwedge_{i \in \{1,\dots,m\} \setminus K} \neg(y_i = \max(\mathbf{y})) \\ \operatorname{argmax}(\mathbf{y}) = \operatorname{argmax}(\mathbf{y}') \;&\equiv\; \bigvee_{K \subseteq \{1,\dots,n\}} (\operatorname{argmax}(\mathbf{y}) = K \;\wedge\; \operatorname{argmax}(\mathbf{y}') = K) \end{aligned}$$

The latter definition takes into account that argmax returns the set of indices carrying the maximal value in a tuple/vector.

Example 3.2:

Consider the formulas

$$\varphi_1(x, y) = x < y \;\Rightarrow\; \exists z.(x < z \;\wedge\; z < y)$$

$$\varphi_2(x, y) = x < y \;\Rightarrow\; ((x < 0.5 \cdot x + 0.5 \cdot y) \;\wedge\; (0.5 \cdot x + 0.5 \cdot y < y))$$

Both, φ_1 and φ_2 have free variables x and y. The formulas $\forall x.\forall y.\varphi_1(x, y)$ and $\forall x.\forall y.\varphi_2(x, y)$ are sentences, as they do not have any free variables.

The semantics of LRA formulas is defined inductively. To evaluate formulas with free variables, such as $\varphi(x, y) = (x \leq 0.2 \cdot y)$, we need to assign values to x and y. This is done by an *interpretation function* $I : \mathcal{X} \to \mathbb{R}$. The above formula φ is evaluated to true iff $I(x) \leq 0.2 \cdot I(y)$. Towards the semantics, we will first assign to each term t a real number $I(t) \in \mathbb{R}$ inductively as follows:

- $I(a \cdot x) = a \cdot I(x)$
- $I(b) = b$
- $I(t_1 + t_2) = I(t_1) + I(t_2)$

Now, models of formulas φ are interpretation functions:

- $I \models t_1 \leq t_2$ if $I(t_1) \leq I(t_2)$
- $I \models \neg\varphi$ if $I \not\models \varphi$
- $I \models \varphi \vee \psi$ if $I \models \varphi$ or $I \models \psi$
- $I \models \exists x.\varphi$ if there is $r \in \mathbb{R}$ such that $I[x \mapsto r] \models \varphi$

Here, $I[x \mapsto r]$ is the interpretation function that coincides with I on all variables apart from x, while x is mapped to r.

We say that formula φ is *satisfiable* if there is an interpretation function I such that $I \models \varphi$. Note that, to evaluate a formula (a term), it is enough to know the interpretation of the free variables (variables, respectively) that occur in it. Therefore, given a formula $\varphi(x_1, \ldots, x_n)$ and $r_1, \ldots, r_n \in \mathbb{R}$, we write $\models \varphi(r_1, \ldots, r_n)$ if $I \models \varphi(x_1, \ldots, x_n)$ for some interpretation I such that maps $I(x_i) = r_i$ for all $i \in \{1, \ldots, n\}$. In the particular case where φ is a sentence, satisfiability is independent of an interpretation function. That is, we either have one of the following:

- $I \models \varphi$ for all interpretation functions I
- $I \not\models \varphi$ for all interpretation functions I

In the former case, we write $\models \varphi$, and we say that φ is *true*. In the latter case, φ is *false*.

Example 3.3:

We continue Example 3.2. The sentences $\forall x.\forall y.\varphi_1(x, y)$ and $\forall x.\forall y.\varphi_2(x, y)$ are both true. On the other hand, $\forall x.\forall y.\exists z.(x < z \ \wedge \ z < y)$ is false.

Definition 3.5: Satisfiability Problem

For a class of formulas $\mathcal{F}$, the decision problem $\mathsf{SAT}(\mathcal{F})$ is defined as follows:

Input: A formula $\varphi \in \mathcal{F}$.

Question: Is φ satisfiable?

Note that free variables in the given formula φ are implicitly interpreted as existential variables, as the question is whether *there is* a suitable interpretation function. Moreover, if φ is a sentence, then the problem amounts to asking if φ is true.

The definition of $\mathsf{SAT}(\mathcal{F})$ applies to all classes of formulas $\mathcal{F}$ that we consider in this lecture, as they will all be based on interpretation functions of the form $I : \mathcal{X} \to \mathbb{R}$.

Theorem 3.1:

The problem $\mathsf{SAT}(\mathrm{LRA})$ is decidable.

Before we prove this theorem, we give our specification language for neural networks. It is basically LRA, but with an additional predicate that allows us to talk about the input-output relation induced by a neural network.

Definition 3.6: Neural Network Logic

Formulas from NNL (neural network logic) are given by the following grammar:

$$\begin{aligned} t &::= a \cdot x \mid b \mid t + t \\ \varphi &::= t \leq t \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \mathcal{N}(x_1, \dots, x_m) = (y_1, \dots, y_n) \end{aligned}$$

where $\mathcal{N}$ is a neural network with input dimension $m \in \mathbb{N}_+$ and output dimension $n \in \mathbb{N}_+$, $x, x_1 \dots, x_m, y_1, \dots, y_n \in \mathcal{X}$, and $a, b \in \mathbb{Q}$.

Note that, in the formula $\mathcal{N}(x_1, \dots, x_m) = (y_1, \dots, y_n)$, the variables $x_1, \dots, x_m$ and $y_1, \dots, y_n$ are free. As NNL is an extension of LRA, it only remains to define the semantics of $\mathcal{N}(x_1, \dots, x_m) = (y_1, \dots, y_n)$:

$$I \models \mathcal{N}(x_1, \dots, x_m) = (y_1, \dots, y_n) \;\text{ if }\; [\![\mathcal{N}]\!](I(x_1), \dots, I(x_m)) = (I(y_1), \dots, I(y_n))$$

Accordingly, given an NNL sentence φ, we write $\models \varphi$ (and say that φ is true) if $I \models \varphi$ for some/all I. Given an NNL formula φ containing the neural networks $\mathcal{N}_1, \dots, \mathcal{N}_k$, we may write $\varphi[\mathcal{N}_1, \dots, \mathcal{N}_k]$ instead of just φ to highlight that φ talks about $\mathcal{N}_1, \dots, \mathcal{N}_k$.[3]

For a set of NNL formulas $\mathcal{F}$ and a set of activation functions $\mathfrak{A}$, we denote by $\mathcal{F}[\mathfrak{A}]$ the set of formulas $\varphi \in \mathcal{F}$ such that every neural network occurring in φ uses only activation functions from $\{\mathrm{id}\} \cup \mathfrak{A}$. To simplify notation further, we may just write a list of functions instead of a set. For example, NNL[ReLU] is the set of NNL formulas whose neural networks use the identity function or ReLU in their layers. Similarly $\mathrm{NNL}[\mathrm{ReLU}, \sigma, \tanh]$ allows for activation functions from $\{\mathrm{id}\} \cup \{\mathrm{ReLU}, \sigma, \tanh\}$, while $\mathrm{NNL}[\emptyset]$ admits only the identity function.

Now, let us define some concrete NNL specifications:

Exercise 3.4:

Let $\mathcal{N}$ be a neural network with $in(\mathcal{N}) = out(\mathcal{N}) = 2$. Write NNL sentences $\varphi_1[\mathcal{N}]$ and $\varphi_2[\mathcal{N}]$ such that the following hold:

- $[\![\mathcal{N}]\!]$ is surjective iff $\models \varphi_1[\mathcal{N}]$

[3] We could have defined NNL formulas using "neural network variables" so that models interpret these variables as neural networks, but this would cause some notational overhead.

– $[\![\mathcal{N}]\!]$ is injective iff $\models \varphi_2[\mathcal{N}]$

Solution:

– $\varphi_1[\mathcal{N}] = \forall y_1, y_2. \exists x_1, x_2. \mathcal{N}(x_1, x_2) = (y_1, y_2)$

– $\varphi_2[\mathcal{N}] = \forall x_1, x_2, x_1', x_2', y_1, y_2. \left(\begin{array}{c} \mathcal{N}(x_1, x_2) = (y_1, y_2) \ \wedge \ \mathcal{N}(x_1', x_2') = (y_1, y_2) \\ \Rightarrow \\ x_1 = x_1' \ \wedge \ x_2 = x_2' \end{array} \right)$

Exercise 3.5:

Write an NNL sentence for the XOR function: A given neural network $\mathcal{N}$ should compute a function $\mathbb{R}^2 \to \mathbb{R}$ that, for every input $(x_1, x_2) \in \{0, 1\}^2$, outputs the truth value $x_1 \oplus x_2$.

Exercise 3.6:

Write NNL sentences for the properties (a)–(g) given at the beginning of Section 3.2: For each property P among (a)–(f), define an NNL sentence $\varphi[\mathcal{N}]$ such that $\models \varphi[\mathcal{N}]$ iff $\mathcal{N}$ satisfies P. For property (g), write an NNL sentence $\varphi[\mathcal{N}_1, \mathcal{N}_2]$ such that $\models \varphi[\mathcal{N}_1, \mathcal{N}_2]$ iff $\mathcal{N}_1, \mathcal{N}_2$ satisfy (g).

Solution:

For a (definable) set $R \subseteq \mathbb{R}^m$, let $\varphi_R(x_1, \ldots, x_m)$ be an LRA formula such that, for all $r_1, \ldots, r_m \in \mathbb{R}$, we have $(r_1, \ldots, r_m) \in R$ iff $\models \varphi_R(r_1, \ldots, r_m)$.

(a) $\forall x_1, \ldots, x_m, y. \left(\mathcal{N}(x_1, \ldots, x_m) = y \quad \Rightarrow \quad y = \max(x_1, \ldots, x_m)\right)$

(b) $\forall x_1, \ldots, x_m, y_1, \ldots, y_m. \left(\begin{array}{c} \mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_m) \\ \Rightarrow \\ \displaystyle\bigwedge_{1 \le i < j \le m} y_i \le y_j \ \wedge \ \bigvee_{\pi \in S_m} \bigwedge_{i=1}^{m} x_i = y_{\pi(i)} \end{array} \right)$

(c) $\forall x_1, \ldots, x_m. \exists y_1, \ldots, y_n. \displaystyle\bigwedge_{\pi \in S_m} \mathcal{N}(x_{\pi(1)}, \ldots, x_{\pi(m)}) = (y_1, \ldots, y_n)$

(d) $\forall x_1, \ldots, x_m, y_1, \ldots, y_m. \left(\begin{array}{c} \mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_m) \\ \Rightarrow \\ \displaystyle\bigwedge_{\pi \in S_m} \mathcal{N}(x_{\pi(1)}, \ldots, x_{\pi(m)}) = (y_{\pi(1)}, \ldots, y_{\pi(m)}) \end{array} \right)$

(e) $\forall \mathbf{x}, \mathbf{x}', \mathbf{y}, \mathbf{y}'. \left(\begin{array}{c} \mathcal{N}(\mathbf{x}) = \mathbf{y} \wedge \mathcal{N}(\mathbf{x}') = \mathbf{y}' \wedge \varphi_R(\mathbf{x}) \wedge \varphi_R(\mathbf{x}') \wedge \bigwedge_{i \in K} x_i = x'_i \\ \Rightarrow \\ \mathrm{argmax}(\mathbf{y}) = \mathrm{argmax}(\mathbf{y}') \end{array} \right)$

(f) $\forall \mathbf{x}, \mathbf{x}', \mathbf{y}, \mathbf{y}', \mathbf{z}. \left(\begin{array}{c} \left(\begin{array}{ll} & \mathcal{N}(\mathbf{x}) = \mathbf{y} \wedge \mathcal{N}(\mathbf{x}') = \mathbf{y}' \wedge \varphi_R(\mathbf{x}) \\ \wedge & \bigwedge_{i=1}^{m} \left(\begin{array}{ll} & x_i \leq x'_i \Rightarrow z_i = x'_i - x_i \\ \wedge & x'_i < x_i \Rightarrow z_i = x_i - x'_i \end{array} \right) \\ \wedge & z_1 + \ldots + z_m \leq \varepsilon \end{array} \right) \\ \Rightarrow \\ \mathrm{argmax}(\mathbf{y}) = \mathrm{argmax}(\mathbf{y}') \end{array} \right)$

(g) $\forall \mathbf{x}, \mathbf{y}, \mathbf{y}'. \left(\begin{array}{c} \mathcal{N}_1(\mathbf{x}) = \mathbf{y} \wedge \mathcal{N}_2(\mathbf{x}) = \mathbf{y}' \wedge \varphi_R(\mathbf{x}) \\ \Rightarrow \\ \mathrm{argmax}(\mathbf{y}) = \mathrm{argmax}(\mathbf{y}') \end{array} \right)$

An NNL formula $\varphi[\mathcal{N}_1, \ldots, \mathcal{N}_k]$ is considered to represent a neural network specification. Verifying $\mathcal{N}_1, \ldots, \mathcal{N}_k$ amounts to deciding whether $\varphi[\mathcal{N}_1, \ldots, \mathcal{N}_k]$ is satisfiable. We will assume that $\mathcal{N}_1, \ldots, \mathcal{N}_k$ are all ReLU neural networks, i.e., $\varphi \in$ NNL[ReLU]. To establish decidability of verification, due to Theorem 3.1, it is then enough to show that φ can be translated into an equivalent LRA sentence.

Definition 3.7:

Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be classes of formulas (whose semantics depends on interpretation functions $I : \mathcal{X} \to \mathbb{R}$). We write $\mathcal{F}_1 \leq \mathcal{F}_2$ if there is an algorithm that translates every $\varphi(x_1, \ldots, x_n) \in \mathcal{F}_1$ into $\tilde{\varphi}(x_1, \ldots, x_n) \in \mathcal{F}_2$ such that, for all interpretation functions I, we have $I \models \varphi$ iff $I \models \tilde{\varphi}$.

If the translation can be done in polynomial time, then we write $\mathcal{F}_1 \leq_{\mathrm{poly}} \mathcal{F}_2$.

Proposition 3.1:

We have NNL[ReLU] $\leq_{\mathrm{poly}}$ LRA.

Proof. We only need to consider formulas involving neural networks. To do so, we translate every neural network $\mathcal{N}$, say with input dimension m and output dimension n, into an LRA formula $\varphi_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n)$ such that

$$\begin{array}{l} \text{for all } r_1, \ldots, r_m, s_1, \ldots, s_n \in \mathbb{R}, \text{ we have} \\ [\![\mathcal{N}]\!](r_1, \ldots, r_m) = (s_1, \ldots, s_n) \text{ iff } \models \varphi_{\mathcal{N}}(r_1, \ldots, r_m, s_1, \ldots, s_n) \end{array} \tag{3.1}$$

We proceed by induction and first suppose that $\mathcal{N} = (\mathscr{L})$ has one layer $\mathscr{L} = (\mathbf{A}, \mathbf{b}, f)$.

If $f = \mathrm{id}$, we set

$$\varphi_{\mathcal{N}}(x_1, \dots, x_m, y_1, \dots, y_n) \;=\; \bigwedge_{i=1}^{n} y_i = b_i + \sum_{j=1}^{m} a_{i,j} \cdot x_j \,.$$

If $f = \mathrm{ReLU}$, we set

$$\varphi_{\mathcal{N}}(x_1, \dots, x_m, y_1, \dots, y_n) \;=\; \bigwedge_{i=1}^{n} \exists z. \left(\begin{array}{cc} & z = b_i + \displaystyle\sum_{j=1}^{m} a_{i,j} \cdot x_j \\ \wedge & \left(\begin{array}{cc} & (z \le 0 \;\wedge\; y_i = 0) \\ \vee & (z > 0 \;\wedge\; y_i = z) \end{array} \right) \end{array} \right) .$$

Then, statement (3.1) holds by the very definitions.

Now assume $\mathcal{N} = (\mathscr{L}^{(1)}, \dots, \mathscr{L}^{(\ell)}, \mathscr{L}^{(\ell+1)})$ with $\ell \ge 1$. Let $\mathcal{N}_1 = (\mathscr{L}^{(1)}, \dots, \mathscr{L}^{(\ell)})$ and $\mathcal{N}_2 = (\mathscr{L}^{(k+1)})$ and assume we already have formulas $\varphi_{\mathcal{N}_1}(x_1, \dots, x_m, z_1, \dots, z_k)$ and $\varphi_{\mathcal{N}_2}(z_1, \dots, z_k, y_1, \dots, y_n)$ as required, where $k = \mathit{out}(\mathcal{N}_1) = \mathit{in}(\mathcal{N}_2)$. Then, we set

$$\varphi_{\mathcal{N}}(x_1, \dots, x_m, y_1, \dots, y_n) = \exists z_1, \dots, z_k. \left(\begin{array}{cc} & \varphi_{\mathcal{N}_1}(x_1, \dots, x_m, z_1, \dots, z_k) \\ \wedge & \varphi_{\mathcal{N}_2}(z_1, \dots, z_k, y_1, \dots, y_n) \end{array} \right) .$$

Indeed, for all $r_1, \dots, r_m, s_1, \dots, s_n \in \mathbb{R}$ we have

$$\begin{array}{ll} & [\![\mathcal{N}]\!](r_1, \dots, r_m) = (s_1, \dots, s_n) \\ \text{iff} & ([\![\mathcal{N}_2]\!] \circ [\![\mathcal{N}_1]\!])(r_1, \dots, r_m) = (s_1, \dots, s_n) \\ \text{iff} & \text{there are } p_1, \dots, p_k \in \mathbb{R}\text{:} \left(\begin{array}{rl} & [\![\mathcal{N}_1]\!](r_1, \dots, r_m) = (p_1, \dots, p_k) \\ \text{and} & [\![\mathcal{N}_2]\!](p_1, \dots, p_k) = (s_1, \dots, s_n) \end{array} \right) \\ \text{iff} & \text{there are } p_1, \dots, p_k \in \mathbb{R}\text{:} \left(\begin{array}{rl} & \models \varphi_{\mathcal{N}_1}(r_1, \dots, r_m, p_1, \dots, p_k) \\ \text{and} & \models \varphi_{\mathcal{N}_2}(p_1, \dots, p_k, s_1, \dots, s_n) \end{array} \right) \\ \text{iff} & \models \varphi_{\mathcal{N}}(r_1, \dots, r_m, s_1, \dots, s_n) \,. \end{array}$$

Now, to translate a given NNL formula φ into the LRA formula $\tilde{\varphi}$ as required, we replace, in φ, every occurrence of an atomic subformula $\mathcal{N}(x_1, \dots, x_m) = (y_1, \dots, y_n)$ by $\varphi_{\mathcal{N}}(x_1, \dots, x_m, y_1, \dots, y_n)$. □

Exercise 3.7:

For the NNL sentence φ for the maximum function, and the neural network $\mathcal{N}$ from Example 3.1(a), determine $\tilde{\varphi}$ according to Proposition 3.1.

As a corollary from Proposition 3.1 and Theorem 3.1 (decidability of SAT(LRA)), we obtain the following result:

Theorem 3.2:

The problem SAT(NNL[ReLU]) is decidable.

We still need to show Theorem 3.1, i.e., decidability of LRA, which deserves its own section.

3.3 Proof of Decidability of LRA

We use automata-theoretic techniques, which are versatile tools for deciding arithmetic theories and can often be easily extended to cover even richer theories [20]. The automata-theoretic approach to deciding arithmetic theories goes back to Büchi [11]. Given an LRA formula φ, the idea is to construct a Büchi automaton $\mathcal{A}_\varphi$ such that φ is satisfiable iff the language of $\mathcal{A}_\varphi$ is nonempty.

In the description below, we adopt several constructions and some notation from the paper [43], where Sälzer et al. provide translations of neural networks into Büchi automata.

Encoding Real Numbers as Words. The main idea is to encode a real number as a word over the alphabet $\Sigma = \{0, 1, \bullet, +, -\}$, and a k-tuple of real numbers as a word over Σ^k. We call a word $w \in \Sigma^\omega$ *well-formed* if it is of the form

$$w = su_{n-1} \dots u_0 \bullet v_1 u_2 \dots \in \{+, -\}\{0, 1\}^*\{\bullet\}\{0, 1\}^\omega$$

with $n \geq 0$, $s \in \{+, -\}$, and $u_{n-1}, \dots, u_0, v_1, u_2, \dots \in \{0, 1\}$. Then, w encodes the real number in decimal system

$$dec(w) = (-1)^{sign(s)} \cdot \left(\sum_{i=0}^{n-1} u_i \cdot 2^i + \sum_{i=1}^{\infty} v_i \cdot \frac{1}{2^i} \right)$$

where $sign(+) = 0$ and $sign(-) = 1$. Note that w determines a unique value $dec(w)$. On the other hand, given $r \in \mathbb{R}$, there may be several well-formed words $w \in \Sigma^\omega$ such that $dec(w) = r$. For example, we have $dec(+0 \bullet 000 \dots) = dec(-000 \bullet 000 \dots) = 0$ and $dec(+0110 \bullet 1000 \dots) = dec(+110 \bullet 0111 \dots) = 6.5$.

Let $k \geq 0$. A word

$$w = \begin{bmatrix} u_{1,1} \\ \vdots \\ u_{k,1} \end{bmatrix} \begin{bmatrix} u_{1,2} \\ \vdots \\ u_{k,2} \end{bmatrix} \begin{bmatrix} u_{1,3} \\ \vdots \\ u_{k,3} \end{bmatrix} \dots \in (\Sigma^k)^\omega$$

may be seen as the k-tuple $(u_{1,1}u_{1,2}u_{1,3} \dots, \ \dots \ , u_{k,1}u_{k,2}u_{k,3} \dots)$ of words over Σ. Thus, a language $L \subseteq (\Sigma^k)^\omega$ can, equivalently, be considered as a relation $L \subseteq (\Sigma^\omega)^k$ and the latter is the view that we mostly adopt in the following.

Now, a tuple $(w_1, \dots, w_k) \in (\Sigma^k)^\omega$ of well-formed words encodes the tuple of real numbers $(dec(w_1), \dots, dec(w_k)) \in \mathbb{R}^k$. However, the algorithmic manipulations we perform on automata to simulate arithmetic operations require that the comma $\bullet$ in the binary representations be aligned. Therefore, we introduce the set of *(k-ary) well-formed words*, denoted WF^k. A word from $(\Sigma^k)^\omega$ is *well-formed* if it is of the form

$$\begin{bmatrix} s_1 \\ \vdots \\ s_k \end{bmatrix} \begin{bmatrix} u_{1,n-1} \\ \vdots \\ u_{k,n-1} \end{bmatrix} \dots \begin{bmatrix} u_{1,0} \\ \vdots \\ u_{k,0} \end{bmatrix} \begin{bmatrix} \bullet \\ \vdots \\ \bullet \end{bmatrix} \begin{bmatrix} v_{1,1} \\ \vdots \\ v_{k,1} \end{bmatrix} \begin{bmatrix} v_{1,2} \\ \vdots \\ v_{k,2} \end{bmatrix} \dots$$

such that $s_i \in \{+, -\}$, $u_{i,j} \in \{0, 1\}$, and $v_{i,j} \in \{0, 1\}$ for all i and j. In particular, all $\bullet$ are aligned in the same column. If $k = 0$, we have a unique infinite word over a singleton alphabet, which we define to be well-formed. Recall that we may consider $\mathrm{WF}^k \subseteq (\Sigma^\omega)^k$. For $\mathbf{w} \in (\Sigma^\omega)^k$, we let w_i refer to the i-th component of $\mathbf{w}$, i.e., $\mathbf{w} = (w_1, \dots, w_k)$.

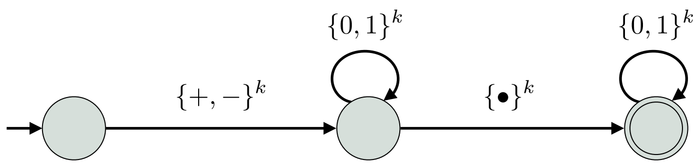


Figure 3.5: The Büchi automaton $\mathcal{A}^k_{\mathrm{wf}}$

Proposition 3.2:

For $k \geq 0$, we can effectively construct a Büchi automaton $\mathcal{A}^k_{\mathrm{wf}}$ over Σ^k such that $L(\mathcal{A}^k_{\mathrm{wf}}) = \mathrm{WF}^k$.

Proof. The automaton $\mathcal{A}^k_{\mathrm{wf}}$ is given in Figure 3.5. It checks that each "row" is contained in $\{+,-\}\{0,1\}^*\{\bullet\}\{0,1\}^\omega$ and that the $\bullet$-symbols are aligned in the same column. □

We present two further useful automata constructions.

Proposition 3.3: Büchi Automata Projection

Let $k \geq 1$ and $i \in \{0, 1, \ldots, k\}$. Let $\mathcal{A}$ be a Büchi automaton over Σ^k. We can effectively construct a Büchi automaton $\mathit{proj}_{\leq i}(\mathcal{A})$ over Σ^i such that

$$L(\mathit{proj}_{\leq i}(\mathcal{A})) = \{(w_1, \ldots, w_i) \mid (w_1, \ldots, w_k) \in L(\mathcal{A})\}.$$

Proof. Let $\mathcal{A} = (Q, \Delta, \iota, F)$ be a Büchi automaton over Σ^k. The Büchi automaton $\mathit{proj}_{\leq i}(\mathcal{A})$ has the same structure (i.e., the same states, initial states, transitions, final states). The only thing that changes are the transition *labels*, which instead of k symbols, $[u_1, \ldots, u_k]$, only contain the first i symbols $[u_1, \ldots, u_i]$. That is, we set $\mathit{proj}_{\leq i}(\mathcal{A}) = (Q, \Delta', \iota, F)$ with

$$\Delta' = \{(p, [u_1, \ldots, u_i], q) \mid (p, [u_1, \ldots, u_k], q) \in \Delta\}.$$

Note that, when $i = 0$, $\mathit{proj}_{\leq i}(\mathcal{A})$ is a Büchi automaton over a single-letter alphabet. The correctness proof is now straightforward. □

Applying projection to well-formed words does not necessarily preserve closure under removing leading zeros. Therefore, we will make use of another useful closure property.

Proposition 3.4:

Let $k \geq 1$ and let $\mathcal{A}$ be a Büchi automaton over Σ^k such that $L(\mathcal{A}) \subseteq \mathrm{WF}^k$. We can construct a Büchi automaton $\mathit{cl}(\mathcal{A})$ over Σ^k such that $L(\mathit{cl}(\mathcal{A}))$ is the least set satisfying the following:

- $L(\mathcal{A}) \subseteq L(\mathit{cl}(\mathcal{A}))$ and
- for all words $\mathbf{w} \in L(\mathit{cl}(\mathcal{A}))$ of the form $\mathbf{w} = (s_1, \ldots, s_k)(0, 0, \ldots, 0)\mathbf{w}'$ (i.e., $s_1, \ldots, s_k \in \{+,-\}$), we have $(s_1, \ldots, s_k)\mathbf{w}' \in L(\mathit{cl}(\mathcal{A}))$.

Exercise 3.8:

Prove Proposition 3.4.

From LRA to Büchi automata. We are now ready to describe how Büchi automata can be used to decide whether a given LRA sentence is true. We start with some constructions for k-ary relations of real numbers that will serve as building blocks in the translation. They are due to [43].

Proposition 3.5: Büchi Automata Constructions

Let $k \geq 1$, $i, j, i_1, i_2 \in \{1, \dots, k\}$, and $a, b \in \mathbb{Q}$. We can effectively construct automata $\mathcal{A}^k_{i=j}$, $\mathcal{A}^k_{i \leq j}$, $\mathcal{A}^k_{i=\mathsf{add}(i_1,i_2)}$, $\mathcal{A}^k_{i=\mathsf{mult}(a,j)}$, and $\mathcal{A}^k_{i=\mathsf{const}(b)}$ over Σ^k such that

$$L(\mathcal{A}^k_{i=j}) = \{\mathbf{w} \in \mathrm{WF}^k \mid dec(w_i) = dec(w_j)\}$$

$$L(\mathcal{A}^k_{i\leq j}) = \{\mathbf{w} \in \mathrm{WF}^k \mid dec(w_i) \leq dec(w_j)\}$$

$$L(\mathcal{A}^k_{i=\mathsf{add}(i_1,i_2)}) = \{\mathbf{w} \in \mathrm{WF}^k \mid dec(w_i) = dec(w_{i_1}) + dec(w_{i_1})\}$$

$$L(\mathcal{A}^k_{i=\mathsf{mult}(a,j)}) = \{\mathbf{w} \in \mathrm{WF}^k \mid dec(w_i) = a \cdot dec(w_j)\}$$

$$L(\mathcal{A}^k_{i=\mathsf{const}(b)}) = \{\mathbf{w} \in \mathrm{WF}^k \mid dec(w_i) = b\}$$

Proof. We start with equality and addition. The other automata will build on them. We only consider constructions with pairwise distinct indices. The remaining cases follow similar patterns.

Equality. The equality Büchi automaton $\mathcal{A}^2_{1=2}$ is depicted in Figure 3.6. Note that two binary representations of real numbers can still be equal when they have different signs. The right-hand side of the automaton also takes care of real numbers that have a suffix of the form 1^ω. The general case $\mathcal{A}^k_{i=j}$ allows for arbitrary values in the components different from i and j (while checking that the word be well-formed).

Addition. Consider first the Büchi automaton $\mathcal{B}^k_{i=\mathsf{add}(i_1,i_2)}$ as illustrated in Figure 3.7 for the special case $\mathcal{B}^3_{3=\mathsf{add}(1,2)}$ (the cases with different signs are similar, since $x_3 = x_1 - x_2$ iff $x_1 = x_2 + x_3$ and so on). It performs bitwise addition of the first two numbers, while checking, each time, whether a carry bit has to be produced. Whereas this procedure yields, for all possible input strings, at least one valid result, $L(\mathcal{B}^k_{i=\mathsf{add}(i_1,i_2)})$ does not contain *all* $\mathbf{w} \in \mathrm{WF}^k$ such that $dec(w_i) = dec(w_{i_1}) + dec(w_{i_1})$. For example, $L(\mathcal{B}^3_{3=\mathsf{add}(1,2)})$ does not contain the word

$$\begin{bmatrix}+\\+\\+\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}\bullet\\\bullet\\\bullet\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\dots$$

but instead

$$\begin{bmatrix}+\\+\\+\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}\bullet\\\bullet\\\bullet\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\dots$$

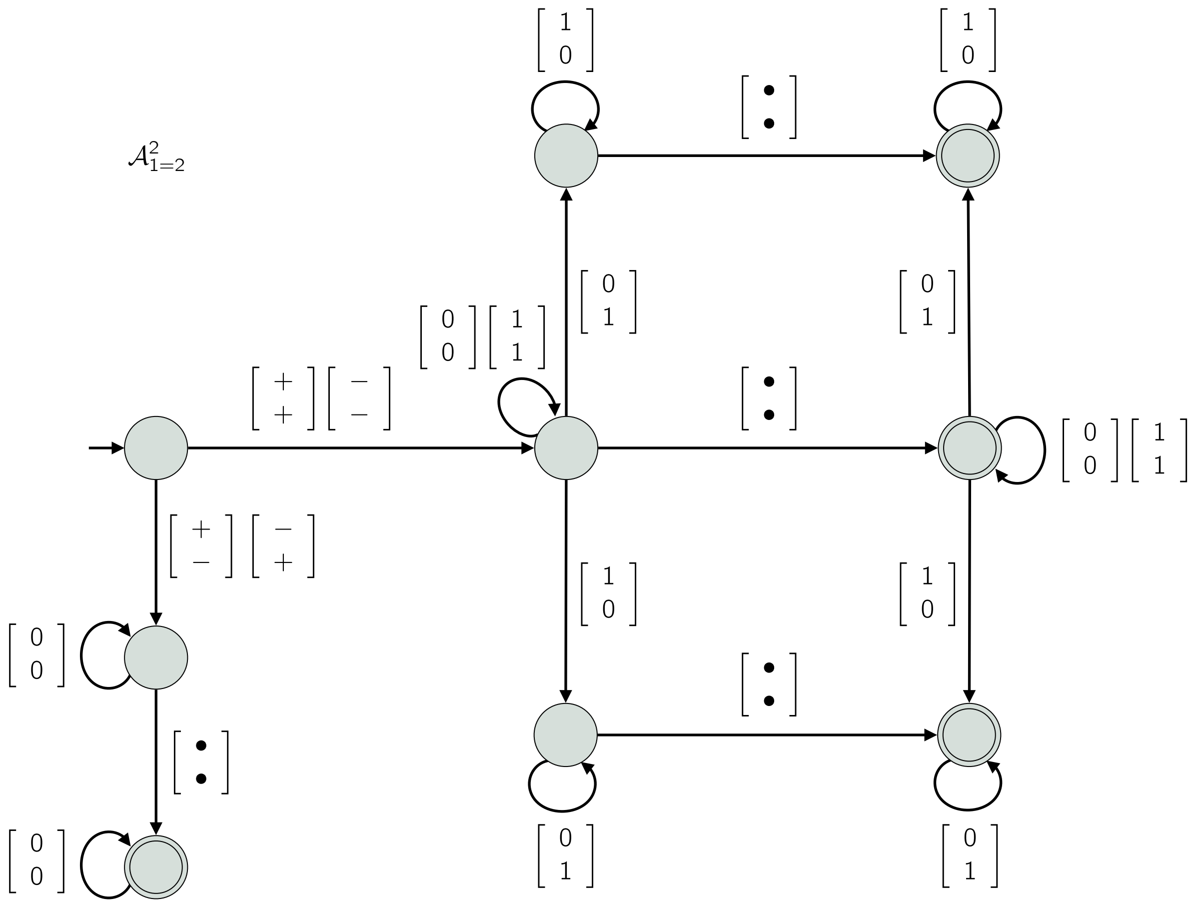


Figure 3.6: The Büchi automaton $\mathcal{A}^2_{1=2}$

We include all admissible triples using an additional tape and the equality automaton defining

$$\mathcal{A}^k_{i=\mathsf{add}(i_1,i_2)} = cl(proj_{\leq k}(\mathcal{B}^{k+1}_{k+1=\mathsf{add}(i_1,i_2)} \cap \mathcal{A}^{k+1}_{i=k+1}))\,.$$

Multiplication. The cases $a \in \{0,1\}$ are easy. Suppose $a = 2$. We create an additional tape, $k+1$, to "copy" the value of tape j using the Büchi automaton for equality. Tape i should then contain the sum of tapes j and $k+1$. We finally erase tape $k+1$. All this is realized by

$$\mathcal{A}^k_{i=\mathsf{mult}(2,j)} = cl(proj_{\leq k}(\mathcal{A}^{k+1}_{k+1=j} \cap \mathcal{A}^{k+1}_{i=\mathsf{add}(j,k+1)}))\,.$$

Now suppose a is an integer such that $a \geq 3$ (negative constants are handled similarly). Let $u_{n-1}\ldots u_0 \in \{0,1\}^*$ be the binary representation of a of minimal length, i.e., $a = dec(+u_{n-1}\ldots u_0 \bullet 000\ldots)$. Furthermore, let $K = \{i_1,\ldots,i_d\} \subseteq \{0,\ldots,n-1\}$ be the set of indices ℓ such that $u_\ell = 1$. In particular, $n-1 \in K$. Note that, for all $x \in \mathbb{R}$, we have

$$a \cdot x = \sum_{\ell=0}^{n-1} u_\ell \cdot 2^\ell \cdot x = \sum_{\ell \in K} u_\ell \cdot 2^\ell \cdot x\,.$$

Thus, we can set

$$\mathcal{A}^k_{i=\mathsf{mult}(a,j)} = cl(proj_{\leq k}(\mathcal{A}^{k+n}_{k+1=j} \cap \bigcap_{\ell=1}^{n-1} \mathcal{A}^{k+n}_{k+1+\ell=\mathsf{mult}(2,k+\ell)} \cap \mathcal{A}^{k+n}_{i=\mathsf{add}(k+1+i_1,\ldots,k+1+i_d)}))\,.$$

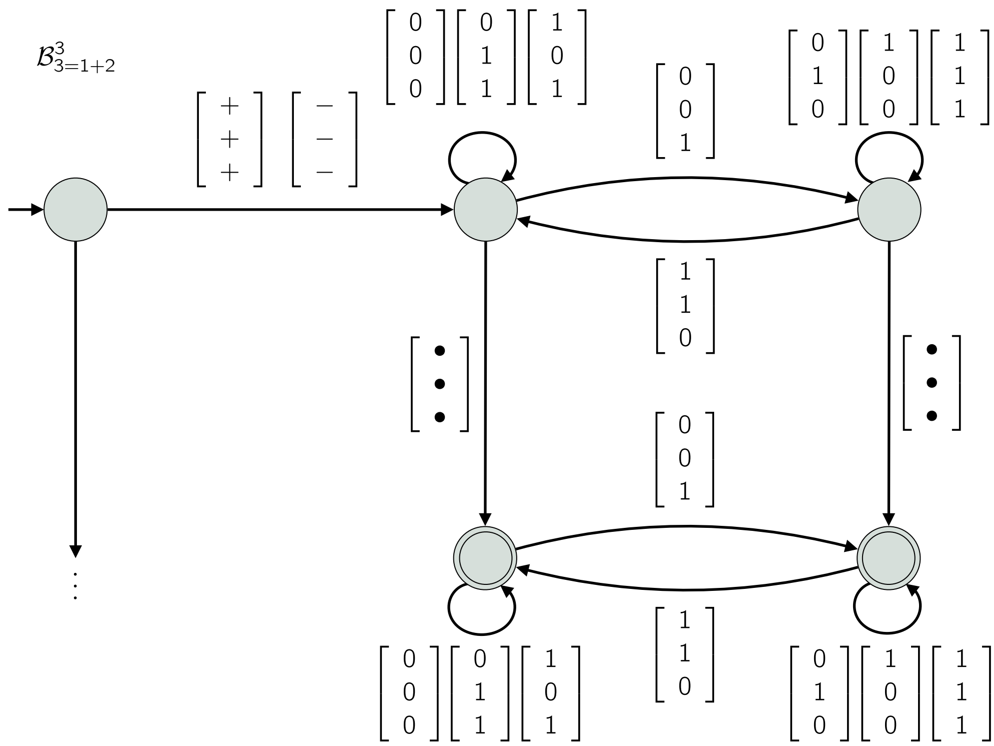


Figure 3.7: The intermediate Büchi automaton $\mathcal{B}^3_{3=\mathsf{add}(1,2)}$

Note that we use a Büchi automaton performing an addition of possibly more than two elements. We obtain the corresponding Büchi automaton by induction, letting, for $n \geq 3$,

$$\mathcal{A}^k_{i=\mathsf{add}(i_1,\dots,i_n)} = cl(proj_{\leq k}(\mathcal{A}^{k+1}_{k+1=\mathsf{add}(i_1,\dots,i_{n-1})} \cap \mathcal{A}^k_{i=\mathsf{add}(i_n,k+1)}))$$

Finally, suppose that $a = \frac{m}{n}$ for integers $m, n \in \mathbb{N}_+$. For any real numbers $x, y \in \mathbb{R}$, we have $x = \frac{m}{n} \cdot y$ iff $n \cdot x = m \cdot y$. Thus, we can set

$$\mathcal{A}^k_{i=\mathsf{mult}(a,j)} = cl(proj_{\leq k}(\mathcal{A}^{k+2}_{k+1=\mathsf{mult}(n,i)} \cap \mathcal{A}^{k+2}_{k+2=\mathsf{mult}(m,j)} \cap \mathcal{A}^{k+2}_{k+1=k+2}))\,.$$

The remaining constructions are left as an exercise. □

Exercise 3.9:

Determine the Büchi automata $\mathcal{A}^k_{i\leq j}$ and $\mathcal{A}^k_{i=\mathsf{const}(b)}$ (for $b \in \mathbb{Q}$).

From LRA to Büchi Automata. We now have all the ingredients to transform a given formula into a Büchi automaton that we can then test for nonemptiness. For simplicity, we suppose we are given a sentence, without free variables. Given an LRA sentence, we first transform it into a logically equivalent formula in Prenex normal form, i.e., into a sentence of the form

$$\Psi = \theta_1 x_1 \dots \theta_m x_m.\Phi(x_1, \dots, x_m)$$

where $\theta_1, \dots, \theta_m \in \{\exists, \neg\exists\}$, the $x_1, \dots, x_m$ are pairwise distinct, and Φ is quantifier-free. For background on the Prenex normal form, we refer the reader to [14]. Let $t_1, \dots, t_n$ be all *distinct occurrences* of terms in Φ, including[4] $x_1, \ldots, x_m$. That is, $m \leq n$ and $t_i = x_i$ for all $i \in \{1, \ldots, m\}$.

With $\mathbf{w} = (w_1, \ldots, w_m, w_{m+1}, \ldots, w_n) \in \mathrm{WF}^n$, we associate an arbitrary interpretation function $I_\mathbf{w}$ such that $I_\mathbf{w}(x_i) = \mathit{dec}(w_i)$ for all $i \in \{1, \ldots, m\}$.

Lemma 3.1:

We can construct a Büchi automaton $\mathcal{A}_{\mathsf{term}}$ over Σ^n such that

$$L(\mathcal{A}_{\mathsf{term}}) = \{\mathbf{w} \in \mathrm{WF}^n \mid I_\mathbf{w}(t_i) = \mathit{dec}(w_i) \text{ for all } i \in \{m+1, \ldots, n\}\}\,.$$

Proof. We set

$$\mathcal{A}_{\mathsf{term}} = \bigcap_{i=m+1}^{n} \mathcal{B}_i^n$$

where

$$\mathcal{B}_i^n = \begin{cases} \mathcal{A}^n_{i=\mathsf{add}(j,k)} & \text{if } t_i = t_j + t_k \\ \mathcal{A}^n_{i=\mathsf{mult}(a,j)} & \text{if } t_i = a \cdot x_j \\ \mathcal{A}^n_{i=\mathsf{const}(b)} & \text{if } t_i = b \end{cases}$$

with $a, b \in \mathbb{Q}$. Correctness follows from Proposition 3.5 by induction. □

Example 3.4:

Consider the sentence

$$\Psi = \neg\exists x_1.\exists x_2.\underbrace{(x_1 \leq x_2 \wedge \neg((x_1 \leq 0.5 \cdot x_1 + 0.5 \cdot x_2)\ \wedge\ (0.5 \cdot x_1 + 0.5 \cdot x_2 \leq x_2)))}_{\Phi(x_1, x_2)}$$

in Prenex normal form. The terms occurring in Φ are $x_1, x_2, 0.5 \cdot x_1, 0.5 \cdot x_2, 0.5 \cdot x_1 + 0.5 \cdot x_2$, and we have

$$\begin{array}{r} x_1 \\ x_2 \\ 0.5 \cdot x_1 \\ 0.5 \cdot x_2 \\ 0.5 \cdot x_1 + 0.5 \cdot x_2 \end{array} \begin{bmatrix} + \\ + \\ + \\ + \\ + \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \ldots \in L(\mathcal{A}_{\mathsf{term}})\,.$$

Using $\mathcal{A}_{\mathsf{term}}$, we now proceed by induction to transform any subformula $\varphi(x_1, \ldots, x_m)$ of Φ (including Φ) into a Büchi automaton $\mathcal{B}_\varphi$ over Σ^n such that, for all $\mathbf{w} \in L(\mathcal{A}_{\mathsf{term}})$,

$$\mathbf{w} \in L(\mathcal{B}_\varphi) \text{ iff } I_\mathbf{w} \models \varphi(x_1, \ldots, x_m)\,. \tag{3.2}$$

Term Comparison. Suppose $\varphi(x_1, \ldots, x_m) = (t_i \leq t_j)$. We set

$$\mathcal{B}_{t_i \leq t_j} = \mathcal{A}^n_{i \leq j}\,.$$

[4]Strictly speaking, a variable is not a term.

Let $\mathbf{w} = (w_1, \ldots, w_n) \in L(\mathcal{A}_{\mathsf{term}})$. We will show (3.2).

Suppose $\mathbf{w} \in L(\mathcal{B}_\varphi)$. Since $\mathbf{w} \in L(\mathcal{A}_{\mathsf{term}})$, we have $I_\mathbf{w}(t_i) = \mathit{dec}(w_i)$ and $I_\mathbf{w}(t_j) = \mathit{dec}(w_j)$. By $\mathbf{w} \in L(\mathcal{A}^n_{i \le j})$ and Proposition 3.5, we have $\mathit{dec}(w_i) \le \mathit{dec}(w_j)$. Thus, $I_\mathbf{w}(t_i) \le I_\mathbf{w}(t_j)$, which implies $I_\mathbf{w} \models t_i \le t_j$.

Conversely, suppose $I_\mathbf{w} \models t_i \le t_j$. Then, $I_\mathbf{w}(t_i) \le I_\mathbf{w}(t_j)$. Since $\mathbf{w} \in L(\mathcal{A}_{\mathsf{term}})$, we also have $\mathit{dec}(w_i) \le \mathit{dec}(w_j)$, which implies $\mathbf{w} \in L(\mathcal{A}^n_{i \le j})$. We conclude $\mathbf{w} \in L(\mathcal{B}_{t_i \le t_j})$.

Disjunction. Suppose $\varphi(x_1, \ldots, x_m) = \varphi_1 \vee \varphi_2$. We set

$$\mathcal{B}_{\varphi_1 \vee \varphi_2} = \mathcal{B}_{\varphi_1} \cup \mathcal{B}_{\varphi_2} \,.$$

To show (3.2), let $\mathbf{w} \in L(\mathcal{A}_{\mathsf{term}})$. We have $\mathbf{w} \in L(\mathcal{B}_{\varphi_1 \vee \varphi_2})$ iff $\mathbf{w} \in L(\mathcal{B}_{\varphi_1}) \cup L(\mathcal{B}_{\varphi_2})$ iff (by induction hypothesis) $I_\mathbf{w} \models \varphi_1 \vee \varphi_2$.

Negation. Suppose $\varphi(x_1, \ldots, x_m) = \neg\psi$. We set

$$\mathcal{B}_{\neg\psi} = \overline{\mathcal{B}_\psi} \,.$$

Let us show (3.2). For $\mathbf{w} \in L(\mathcal{A}_{\mathsf{term}})$, we have $\mathbf{w} \in L(\mathcal{B}_{\neg\psi})$ iff $\mathbf{w} \notin L(\mathcal{B}_\psi)$ iff (by induction hypothesis) $I_\mathbf{w} \not\models \psi$ iff $I_\mathbf{w} \models \neg\psi$.

We thus hold a Büchi automaton $\mathcal{B}_\Phi$ over Σ^n such that, for all $\mathbf{w} \in L(\mathcal{A}_{\mathsf{term}})$,

$$\mathbf{w} \in L(\mathcal{B}_\Phi) \text{ iff } I_\mathbf{w} \models \Phi(x_1, \ldots, x_m) \,.$$

Consider the Büchi automaton $\mathcal{A}_\Phi = \mathit{cl}(\mathit{proj}_{\le m}(\mathcal{B}_\Phi \cap \mathcal{A}_{\mathsf{term}}))$.

Lemma 3.2:

We have

$$L(\mathcal{A}_\Phi) = \{\mathbf{w} \in \mathrm{WF}^m \mid I_\mathbf{w} \models \Phi(x_1, \ldots, x_m)\} \,.$$

Proof. Let $\mathbf{w} \in L(\mathcal{A}_\Phi)$. There is $\mathbf{w}' = (w'_1, \ldots, w'_n) \in L(\mathcal{B}_\Phi \cap \mathcal{A}_{\mathsf{term}})$ such that, for all $i \in \{1, \ldots, m\}$, the word w'_i equals w_i modulo (possibly) extra leading zeros. In particular, $\mathit{dec}(w'_i) = \mathit{dec}(w_i)$ for all $i \in \{1, \ldots, m\}$. Due to $\mathbf{w}' \in L(\mathcal{B}_\Phi)$, we have $I_{\mathbf{w}'} \models \Phi$ and, therefore, $I_\mathbf{w} \models \Phi$.

Conversely, assume $\mathbf{w} = (w_1, \ldots, w_m) \in \mathrm{WF}^m$ such that $I_\mathbf{w} \models \Phi(x_1, \ldots, x_m)$. Let $\mathbf{w}' = (w'_1, \ldots, w'_n) \in \mathrm{WF}^n$ such that,

- for all $i \in \{1, \ldots, m\}$, w'_i equals w_i but possibly has extra leading zeros,
- for all $i \in \{m+1, \ldots, n\}$, $\mathit{dec}(w'_i) = I_\mathbf{w}(t_i)$.

Then, we have $\mathbf{w}' \in L(\mathcal{A}_{\mathsf{term}})$ and, since $I_{\mathbf{w}'} \models \Phi$, $\mathbf{w}' \in L(\mathcal{B}_\Phi)$. Altogether, we have $(w'_1, \ldots, w'_m) \in \mathit{proj}_{\le m}(\mathcal{B}_\Phi \cap \mathcal{A}_{\mathsf{term}})$ and, finally, $\mathbf{w} = (w_1, \ldots, w_m) \in L(\mathcal{A}_\Phi)$. $\square$

Towards a Büchi automaton for $\Psi = \theta_1 x_1 \dots \theta_m.x_m \Phi(x_1, \dots, x_m)$ where $\theta_i \in \{\exists, \neg\exists\}$, we add quantifier (and so eliminate free variables) by induction: For $i = m, \dots, 0$ (the number of free variables yet to be eliminated), letting

$$\Psi_i = \theta_{i+1} x_{i+1} \dots \theta_m x_m.\Phi(x_1, \dots, x_i)\,,$$

we build a Büchi automaton $\mathcal{A}_{\Psi_i}$ over Σ^i such that

$$L(\mathcal{A}_{\Psi_i}) = \{\mathbf{w} \in \mathrm{WF}^i \mid I_{\mathbf{w}} \models \Psi_i(x_1, \dots, x_i)\}\,. \tag{3.3}$$

Note that $\mathcal{A}_{\Psi_m} = \mathcal{A}_\Phi$. Now, let $i = m-1, \dots, 0$. Towards $\mathcal{A}_{\Psi_i}$, we will have to eliminate free variable x_{i+1}.

Existential Quantification. If $\theta_{i+1} = \exists$, then we set

$$\mathcal{A}_{\Psi_i} = cl(proj_{\leq i}(\mathcal{A}_{\Psi_{i+1}}))\,.$$

We show (3.3). We certainly have $\mathcal{A}_{\Psi_i} \subseteq \mathrm{WF}^i$. Let $\mathbf{w} = (w_1, \dots, w_i) \in \mathrm{WF}^i$.

Suppose $\mathbf{w} \in L(\mathcal{A}_{\Psi_i})$. Then, there is $\mathbf{w}' \in L(\mathcal{A}_{\Psi_{i+1}}) \subseteq \mathrm{WF}^{i+1}$ such that, for all $i \in \{1, \dots, m\}$, w'_i equals w_i modulo extra leading zeros. By induction hypothesis, we have that $I_{\mathbf{w}'} \models \Psi_{i+1}(x_1, \dots, x_{i+1})$. But this implies $I_{\mathbf{w}} \models \exists x_{i+1}.\Psi_{i+1}(x_1, \dots, x_{i+1})$.

Conversely, suppose $I_{\mathbf{w}} \models \Psi_i = \exists x_{i+1}.\Psi_{i+1}(x_1, \dots, x_{i+1})$. There is a real number $r \in \mathbb{R}$ such that $I_{\mathbf{w}}[x_{i+1} \mapsto r] \models \Psi_{i+1}(x_1, \dots, x_{i+1})$. Let $\mathbf{w}' = (w'_1, \dots, w'_{i+1}) \in \mathrm{WF}^{i+1}$ such that $dec(w'_{i+1}) = r$ and, for all $i \in \{1, \dots, m\}$, w'_i equals w_i but possibly has extra leading zeros. We have $I_{\mathbf{w}'} \models \Psi_{i+1}(x_1, \dots, x_{i+1})$. By induction hypothesis, $\mathbf{w}' \in L(\mathcal{A}_{\Psi_{i+1}})$. We conclude $\mathbf{w} \in L(\mathcal{A}_{\Psi_i})$.

Negated Existential Quantification. If $\theta_i = \neg\exists$, then we set

$$\mathcal{A}_{\Psi_i} = \overline{cl(proj_{\leq i}(\mathcal{A}_{\Psi_{i+1}}))} \cap \mathcal{A}^i_{\mathrm{wf}}\,.$$

That is, we first apply projection and closure as in the previous case, then complement the automaton, and finally intersect it with $\mathcal{A}^i_{\mathrm{wf}}$. By the preceding discussion, we have

$$\begin{aligned}
& \mathbf{w} \in L(\mathcal{A}_{\Psi_i}) \\
\text{iff}\quad & \mathbf{w} \in \mathrm{WF}^i \text{ and } I_{\mathbf{w}} \not\models \exists x_{i+1}.\Psi_{i+1}(x_1, \dots, x_{i+1}) \\
\text{iff}\quad & \mathbf{w} \in \mathrm{WF}^i \text{ and } I_{\mathbf{w}} \models \underbrace{\neg\exists x_{i+1}.\Psi_{i+1}(x_1, \dots, x_{i+1})}_{=\Psi_i}
\end{aligned}$$

.

To wrap up, we obtain the Büchi automaton $\mathcal{A}_{\Psi_0}$ over Σ^0 (a singleton alphabet) such that $L(\mathcal{A}_{\Psi_0}) \neq \emptyset$ iff $\models \Psi_0$ (recall that $\Psi_0 = \Psi$). Nonemptiness of the language of a Büchi automaton is a decidable problem due to Theorem 2.2.

Historical Notes and Remarks on Complexity. LRA corresponds to the first-order theory of real numbers with addition, denoted by $\mathrm{FO}(\mathbb{R}, +, <)$. Tarski showed that even $\mathrm{FO}(\mathbb{R}, +, \cdot, <)$, i.e., non-linear real arithmetic (including multiplication), is decidable [51]. While his algorithm was nonelementary, $\mathrm{FO}(\mathbb{R}, +, \cdot, <)$ was shown to be solvable in doubly exponential time [12]. The automata-based decidability proof for $\mathrm{FO}(\mathbb{R}, +, <)$ presented in this section can be easily extended to $\mathrm{FO}(\mathbb{R}, +, <, \mathbb{Z})$, which has an additional unary predicate for the integers $\mathbb{Z}$. Another well-known decidable theory is $\mathrm{FO}(\mathbb{N}, +, <)$, also known as Presburger arithmetic. On the other hand, $\mathrm{FO}(\mathbb{N}, +, \cdot, <)$, i.e., Peano arithmetic, is undecidable due to Gödel [19].

The automata-theoretic approach to deciding $\mathrm{FO}(\mathbb{R}, +, <, \mathbb{Z})$ is due to Büchi [11]. Note that projection and negation together may cause an exponential blow-up in the automata size, which a priori does not allow us to infer an elementary bound on the automata size and computation time. Boigelot, Jodogne, and Wolper showed that the models of every $\mathrm{FO}(\mathbb{R}, +, <, \mathbb{Z})$-formula are recognized by a *weak* deterministic Büchi automaton, in which every strongly connected component contains either only final states or only non-final states [9]. Löding showed that minimization of these automata can be reduced, in linear time, to the minimization of DFAs [34]. Moreover, a doubly and triply exponential upper bound on the size of minimal weak deterministic Büchi automata for formulas from $\mathrm{FO}(\mathbb{R}, +, <)$ and $\mathrm{FO}(\mathbb{R}, +, <, \mathbb{Z})$ were shown by Klaedtke [29] and, respectively, Eisinger [15]. For Presburger arithmetic and finite automata, a triply exponential upper bound is due to Klaedtke [28]. These (finite) automata can even be computed in triply exponential time, as was shown by Durand-Gasselin and Habermehl [13]. These results suggest that automata-based decision procedures may still enjoy a good complexity and run well in practice. Various automata-based libraries for deciding arithmetic theories are available [1, 6, 9].

NNL specifications can often do without quantifier alternation, i.e., they belong to the existential fragment, which has a favorable complexity. We will address this in the next section. We will also discuss theories that correspond to deciding NNL sentences with activation functions such as σ and tanh [23].

3.4 An Efficiently Solvable Fragment of NNL

Our goal is now to identify a fragment of NNL[ReLU] that comes with an efficiently solvable satisfiability problem. Observe that many specifications that we have seen previously have a relatively simple structure in the sense that they can do without quantifier alternation. For example, consider Exercise 3.4. While φ_1 has one quantifier alternation, φ_2 is the negation of an existential formula:

$$\varphi_2[\mathcal{N}] = \neg\exists x_1, x_2, x_1', x_2', y_1, y_2. \begin{pmatrix} \mathcal{N}(x_1, x_2) = (y_1, y_2) \;\wedge\; \mathcal{N}(x_1', x_2') = (y_1, y_2) \\ \wedge \\ \neg(x_1 = x_1') \;\vee\; \neg(x_2 = x_2') \end{pmatrix}$$

We will indeed show that satisfiability for existential NNL[ReLU] formulas is NP-complete (and, therefore, satisfiability for universal NNL[ReLU] formulas coNP-complete).

Let us first define the corresponding fragment:

Definition 3.8: Existential NNL (∃NNL)

Formulas from ∃NNL are given by the following grammar:

$$\begin{array}{rcl} t & ::= & a \cdot x \mid b \mid t + t \\ \varphi & ::= & t \leq t \mid t < t \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \mid \\ & & \mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n) \mid \neg\left(\mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n)\right) \end{array}$$

where $\mathcal{N}$ is a neural network with input dimension $m \in \mathbb{N}_+$ and output dimension $n \in \mathbb{N}_+$, $x, x_1 \ldots, x_m, y_1, \ldots, y_n \in \mathcal{X}$, and $a, b \in \mathbb{Q}$.

Note that, as formulas are in negation normal form (negation is pushed inwards), we include atomic formulas of the form $t_1 < t_2$ and $\neg\left(\mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n)\right)$, which are otherwise no longer expressible. We also use the abbreviations $t_1 = t_2 \equiv t_1 \leq t_2 \wedge t_2 \leq t_1$ and $t_1 \neq t_2 \equiv t_1 < t_2 \vee t_2 < t_1$. Note that existential quantifiers are not strictly necessary when considering satisfiability. However, we include them to be able to preserve the number of free variables when translating forth and back between various logics.

We will see that we can translate ∃NNL[ReLU] into existential LRA, which is defined analogously:

Definition 3.9: Existential Linear Real Arithmetic (∃LRA)

Formulas from ∃LRA are generated by the following grammar:

$$\begin{array}{rcl} t & ::= & a \cdot x \mid b \mid t + t \\ \varphi & ::= & t \leq t \mid t < t \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \end{array}$$

where $x \in \mathcal{X}$ and $a, b \in \mathbb{Q}$.

We use the abbreviations $t_1 = t_2$ and $t_1 \neq t_2$ as in ∃NNL.

Theorem 3.3: Complexity of ∃LRA

The problem $\mathsf{SAT}(\exists\text{LRA})$ is NP-complete.

The decision procedure can resort to SMT solvers combining, for example, DPLL(T), Simplex (linear programming) and Tseitin's transformation. See [3] or [30] for a detailed exposition.

Actually, we still need to say what the size of a formula is. It is the length of the formula when constants are written in binary encoding: The length of a rational number m/n (the fraction being simplified), with $m \in \mathbb{Z}$ and $n \in \mathbb{N}_+$, is defined as $1 + \lceil \log(|m| + 1) + 1 \rceil + \lceil \log(n + 1) + 1 \rceil$. Similarly, concerning ∃NNL[ReLU], we define the size of a matrix/vector as the sum of the sizes of all its coefficients.

Our goal is to show the following result:

Theorem 3.4:

The problem $\mathsf{SAT}(\exists\text{NNL}[\text{ReLU}])$ is NP-complete.

The lower bound follows from the obvious fact that $\exists$LRA $\leq_{\text{poly}}$ $\exists$NNL[ReLU]. However, we will show it for a smaller fragment of $\exists$NNL.

The upper bound stated in Theorem 3.4 is due to the following reduction:

Lemma 3.3:

We have $\exists$NNL[ReLU] $\leq_{\text{poly}}$ $\exists$LRA.

Proof. We proceed like in the general case of Lemma 3.1 and translate $\mathcal{N}$, say with input dimension m and output dimension n, into $\exists$LRA formulas $\varphi_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n)$ and $\varphi'_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n)$ such that, for all $r_1, \ldots, r_m, s_1, \ldots, s_n \in \mathbb{R}$ the following hold:

– $[\![\mathcal{N}]\!](r_1, \ldots, r_m) = (s_1, \ldots, s_n)$ iff $\models \varphi_{\mathcal{N}}(r_1, \ldots, r_m, s_1, \ldots, s_n)$

– $[\![\mathcal{N}]\!](r_1, \ldots, r_m) \neq (s_1, \ldots, s_n)$ iff $\models \varphi'_{\mathcal{N}}(r_1, \ldots, r_m, s_1, \ldots, s_n)$

We define $\varphi_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n)$ exactly like in the proof of Lemma 3.1. Note that this gives indeed rise to an $\exists$LRA formula. Moreover, we set

$$\varphi'_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \exists z_1, \ldots, z_n. \left(\begin{array}{ll} & \varphi_{\mathcal{N}}(x_1, \ldots, x_m, z_1, \ldots, z_n) \\ \wedge & \displaystyle\bigvee_{i=1}^{n} y_i \neq z_i \end{array} \right),$$

which is an $\exists$LRA formula, too. The overall translation produces a formula of polynomial size. $\square$

We will now present a stronger lower bound, due to [26, 44, 45], for the *reachability* fragment of $\exists$NNL[ReLU]:

Definition 3.10: Reachability Formulas

A formula from REACH is an $\exists$NNL formula of the form

$$\mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n) \;\wedge\; \varphi_1(x_1, \ldots, x_m) \;\wedge\; \varphi_2(y_1, \ldots, y_n)$$

where φ_1 and φ_2 are quantifier-free formulas generated by the grammar

$$\begin{array}{rcl} t & ::= & a \cdot x \;\mid\; b \;\mid\; t + t \\ \varphi & ::= & t \leq t \;\mid\; \varphi \wedge \varphi \end{array}$$

with $x \in \mathcal{X}$ and $a, b \in \mathbb{Q}$.

In particular, all variables are implicitly existentially quantified.

Theorem 3.5: Reachability is NP-hard [44, 45]

The problem $\mathsf{SAT}(\text{REACH}[\text{ReLU}])$ is NP-hard.

Proof. We follow [44, 45] and provide a polynomial-time reduction from 3SAT (satisfiability of 3CNF formulas). A 3CNF formula is a conjunction of clauses. Every clause is the disjunction of three literals. A literal is a propositional variable X_i or its negation $\neg X_i$. An example formula over the variables X_1, X_2, X_3, X_4 with three clauses is

$$\underbrace{(X_1 \vee X_2 \vee X_3)}_{C_1} \;\wedge\; \underbrace{(\neg X_1 \vee X_2 \vee \neg X_3)}_{C_2} \;\wedge\; \underbrace{(\neg X_2 \vee X_3 \vee X_4)}_{C_3}\,.$$

The satisfiability problem for 3CNF formulas is NP-complete: Given a 3CNF formula $\Phi = C_1 \wedge \ldots \wedge C_k$ over variables $X_1, \ldots, X_m$, is there a valuation $val : \{X_1, \ldots, X_m\} \to \{0,1\}$ such that $val(\Phi) = 1$ (with the canonical extension of val to formulas)?

We will first build a neural network $\mathcal{N}$ such that

$$[\![\mathcal{N}]\!]\big|_{\{0,1\}^m} : \begin{cases} \{0,1\}^m \to \mathbb{R} \\ \mathbf{r} = (r_1, \ldots, r_m) \mapsto val_{\mathbf{r}}(C_1) + \ldots + val_{\mathbf{r}}(C_k) \end{cases}$$

where $val_{\mathbf{r}}(X_i) = r_i$ for all $i \in \{1, \ldots, m\}$. The neural network $\mathcal{N}$ is illustrated in the upper part of Figure 3.8. Correctness follows because, for all $r_1, r_2, r_3 \in \{0,1\}$, we have

$$\begin{aligned} & 1 - \text{ReLU}(1 - r_1 - r_2 - r_3) \\ = \; & 1 - \max(0, 1 - r_1 - r_2 - r_3) \\ = \; & \begin{cases} 0 & \text{if } r_1 = r_2 = r_3 = 0 \\ 1 & \text{otherwise.} \end{cases} \end{aligned}$$

It follows that

$$\Phi \text{ is satisfiable} \quad \text{iff} \quad \left(\begin{array}{ll} & \mathcal{N}(x_1, \ldots, x_m) = y \\ \wedge & \displaystyle\bigwedge_{i=1}^{m} (x_i = 0 \;\vee\; x_i = 1) \\ \wedge & y = k \end{array} \right) \text{ is satisfiable.}$$

However, the latter is not a REACH formula (there cannot be a suitable REACH formula, as $\{0,1\}^m$ is not a hyperreactangle). So we add some more neurons to $\mathcal{N}$ that allow us to enforce $x_i \in \{0,1\}$ in the sentence (cf. lower part of Figure 3.8). We then obtain a neural network $\mathcal{N}'$ with $in(\mathcal{N}') = m$ and $out(\mathcal{N}') = m + 1$.

Note that, for all $r \in \mathbb{R}$, the following holds:

$$\begin{aligned} & bool(r) \\ = \; & \text{ReLU}(r - 0.5) + \text{ReLU}(0.5 - r) - 0.5 \\ = \; & \begin{cases} r - 1 & \text{if } r \geq 0.5 \\ -r & \text{if } r < 0.5 \end{cases} \end{aligned}$$

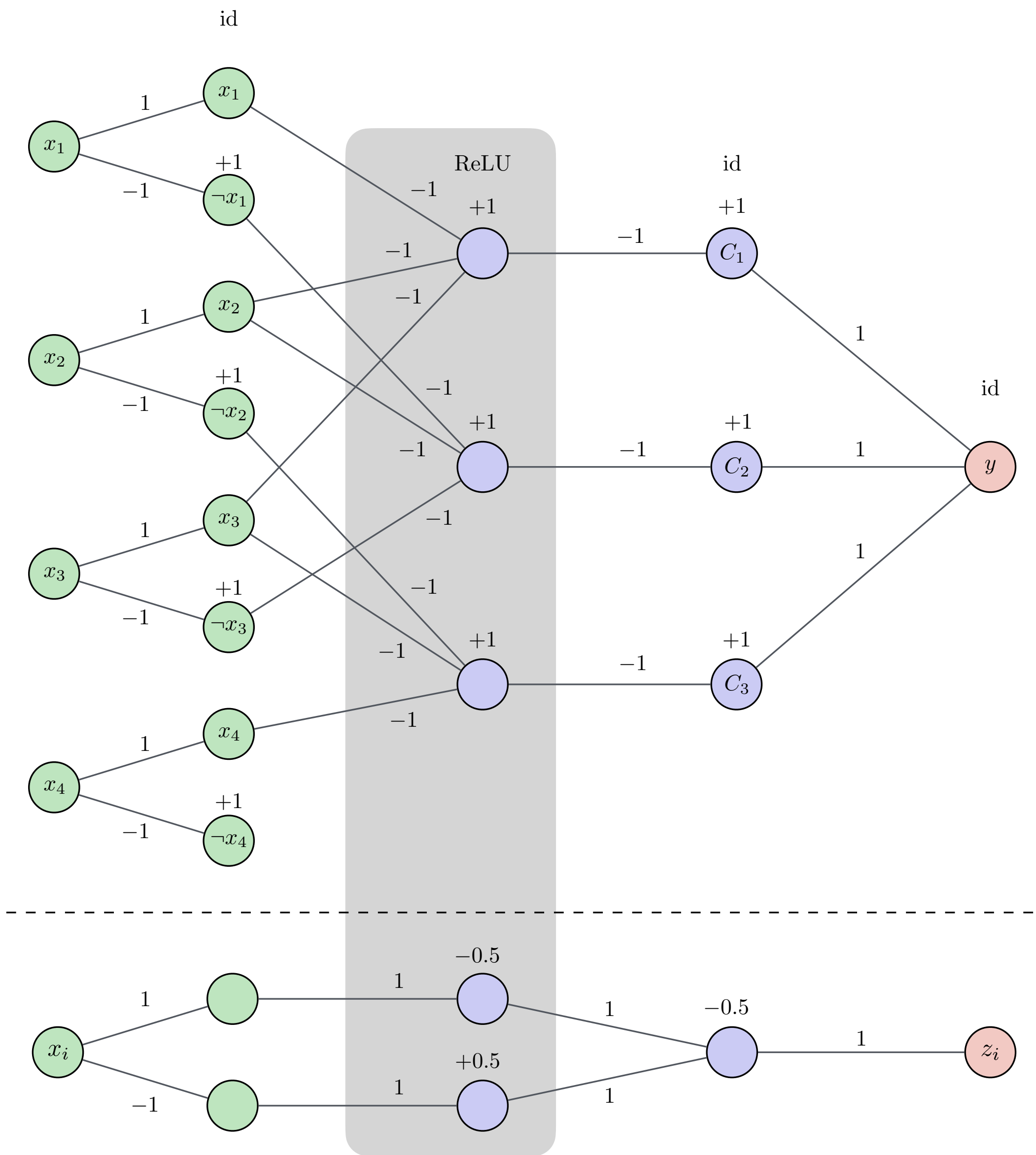


Figure 3.8: Neural network for $(X_1 \vee X_2 \vee X_3) \ \wedge \ (\neg X_1 \vee X_2 \vee \neg X_3) \ \wedge \ (\neg X_2 \vee X_3 \vee X_4)$

In particular, we have $bool(r) = 0$ iff $r \in \{0, 1\}$. Therefore,

$$\Phi \text{ is satisfiable} \quad \text{iff} \quad \begin{pmatrix} & \mathcal{N}'(x_1, \ldots, x_m) = (y, z_1, \ldots, z_m) \\ \wedge & \displaystyle\bigwedge_{i=1}^{m} z_i = 0 \\ \wedge & y = k \end{pmatrix} \quad \text{is satisfiable.}$$

As the latter is a REACH formula whose size (in particular $\mathcal{N}'$) is polynomial, we are done. □

Theorem 3.6: NP-hardness for ReLU layers [44, 45]

NP-hardness from Theorem 3.5 also holds for ReLU neural networks such that all but the last layers are ReLU layers.

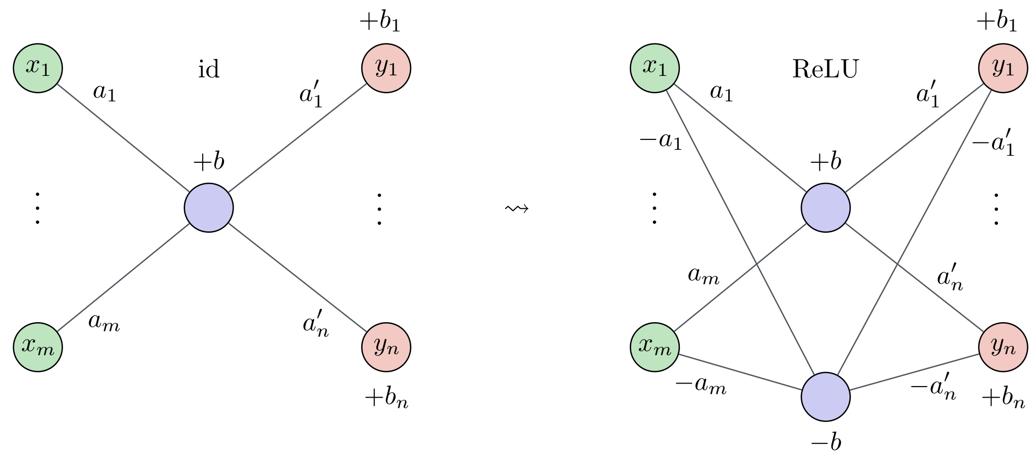


Figure 3.9: Replacing an id neuron with ReLU neurons

Proof. We replace a neuron with activation function id by a gadget employing activation function ReLU. This is illustrated in Figure 3.9. Indeed, we have

$$\begin{aligned}
&\Big(b + \sum_{i=1}^{m} a_i x_i\Big) \cdot a'_j \\
&= \mathrm{ReLU}\Big(b + \sum_{i=1}^{m} a_i x_i\Big) \cdot a'_j + \mathrm{ReLU}\Big(-\big(b + \sum_{i=1}^{m} a_i x_i\big)\Big) \cdot (-a'_j) \\
&= \mathrm{ReLU}\Big(b + \sum_{i=1}^{m} a_i x_i\Big) \cdot a'_j + \mathrm{ReLU}\Big(-b + \sum_{i=1}^{m} -a_i x_i\Big) \cdot (-a'_j)
\end{aligned}$$

This concludes the proof. □

NP-hardness results for even stronger restrictions on the neural networks can be found in [44, 45, 56].

Observe that, when the only activation function allowed is the identity function id, satisfiability of REACH[∅] can be reduced to a linear-programming problem. Therefore, we obtain the following result:

Theorem 3.7:

The problem SAT(REACH[∅]) is solvable in polynomial time.

Recall that many NNL specifications are negations of ∃NNL[ReLU] sentences. As a corollary of Theorem 3.4, we can cover this case, too:

Theorem 3.8:

The following decision problem is coNP-complete.

Input: A sentence $\varphi \in \exists$NNL[ReLU].

Question: Do we have $\models \neg\varphi$?

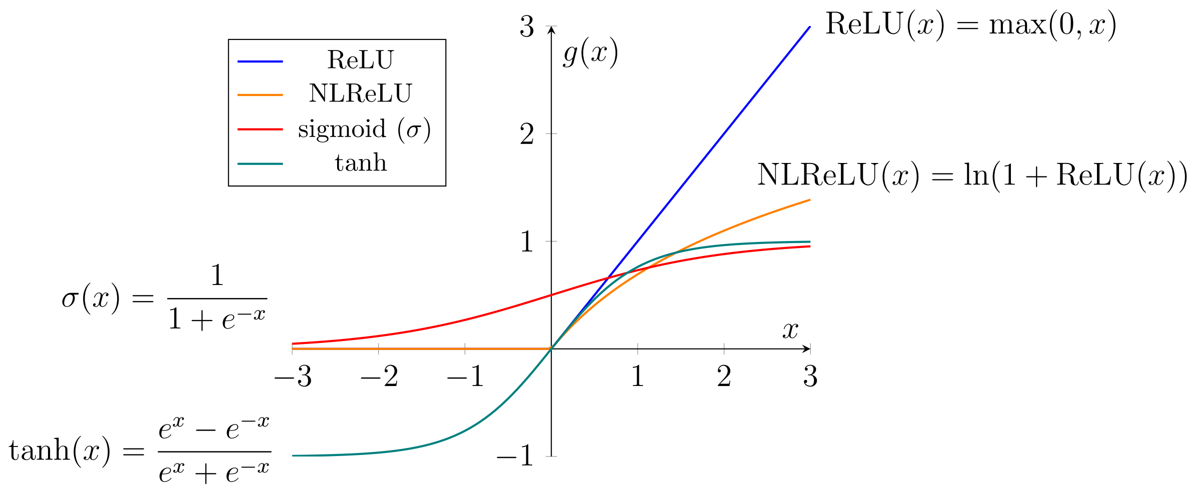


Figure 3.10: (Local) activation functions given by $g : \mathbb{R} \to \mathbb{R}$

Exercise 3.10:

In Theorem 3.8, why do we have to restrict inputs to sentences?

3.5 Beyond ReLU Neural Networks

So far, we mainly considered ReLU neural networks. But how about decidability for general neural networks? In this section, we establish equivalence between verification for a particular set of activation functions and LRA extended by the exponential function [23].

Definition 3.11: First-Order Formulas over the Real Exponential Field

Formulas from REF (first-order formulas over the *real exponential field*) are defined as follows:

$$\begin{array}{rcl} t & ::= & a \mid x \mid t + t \mid t \cdot t \mid e^t \\ \varphi & ::= & t \leq t \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \end{array}$$

where $x \in \mathcal{X}$ and $a \in \mathbb{Q}$.

The semantics of the new terms is given as follows (for an interpretation function I):

- $I(e^t) = e^{I(t)}$
- $I(t_1 \cdot t_2) = I(t_1) \cdot I(t_2)$

Let NNL* be a shorthand for $\mathrm{NNL}[\mathrm{ReLU}, \mathrm{NLReLU}, \sigma, \tanh]$. We recall these activation functions in Figure 3.10. The decidability status of SAT(REF), also called *Tarski's exponential function problem*, is unknown. We show a result, due to [23], stating that the problem is effectively equivalent to SAT(NNL*).

Theorem 3.9: Expressive Equivalence of NNL* and REF [23]

We have $\mathrm{NNL}^* \leq \mathrm{REF}$ and $\mathrm{REF} \leq \mathrm{NNL}^*$.

Proof. The statement follows from the two propositions below. $\square$

Proposition 3.6:

We have $\mathrm{NNL}^* \leq \mathrm{REF}$.

Proof. It remains to consider neural networks $\mathcal{N} = (\mathscr{L})$ with $\mathscr{L} = (\mathbf{A}, \mathbf{b}, f)$ and $f = \sigma$ or $f = \tanh$ or $f = \mathrm{NLReLU}$. Let $m = \mathit{in}(\mathscr{L})$ and $n = \mathit{out}(\mathscr{L})$.

We construct an REF formula $\varphi_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n)$ such that, for all real numbers $r_1, \ldots, r_m, s_1, \ldots, s_n \in \mathbb{R}$,

$$[\![\mathcal{N}]\!](r_1, \ldots, r_m) = (s_1, \ldots, s_n) \text{ iff } \models \varphi_{\mathcal{N}}(r_1, \ldots, r_m, s_1, \ldots, s_n)\,.$$

We set

$$\varphi_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n) \;=\; \bigwedge_{i=1}^{n} \exists z. \left(\begin{array}{ll} & z = b_i + \sum_{j=1}^{m} a_{i,j} \cdot x_j \\ \wedge & \alpha_f(z, y_i) \end{array} \right).$$

where

$$\alpha_f(z, y_i) = \begin{cases} y_i \cdot (1 + e^z) = e^z & \text{if } f = \sigma \\ y_i \cdot (e^{2z} + 1) = (e^{2z} - 1) & \text{if } f = \tanh \\ \left(\begin{array}{ll} & (z \leq 0 \ \wedge \ y_i = 0) \\ \vee & (z > 0 \ \wedge \ e^{y_i} = z + 1) \end{array} \right) & \text{if } f = \mathrm{NLReLU} \end{cases}$$

The rest of the induction follows exactly the lines of the proof of Proposition 3.1. $\square$

Now, we translate REF formulas into NNL^* formulas. In the following, let $\eta = \mathrm{NLReLU}$.

Proposition 3.7:

We have $\mathrm{REF} \leq_{\mathrm{poly}} \mathrm{NNL}^*$.

Towards the proof, we establish a series of intermediary results:

Lemma 3.4:

For every function $f : \mathbb{R} \to \mathbb{R}$ from $\{\mathrm{ReLU}, \eta, \sigma, \tanh, \sigma^{-1}, \tanh^{-1}\}$, there is an NNL^* formula $\varphi_f(x, y) \in$ such that, for all interpretation functions I, we have

$$I \models \varphi_f(x, y) \;\text{ iff }\; f(I(x)) = I(y)\,.$$

Proof. For $f \in \{\mathrm{ReLU}, \eta, \sigma, \tanh\}$, let $\mathcal{N}_f = (\mathscr{L})$ where $\mathscr{L} = ((1), (0), f)$. We have $[\![\mathcal{N}_f]\!] = f$. Therefore, we can set $\varphi_f(x, y) = (\mathcal{N}_f(x) = y)$. Moreover, we let $\varphi_{\sigma^{-1}}(x, y) = (\mathcal{N}_\sigma(y) = x)$ and $\varphi_{\tanh^{-1}}(x, y) = (\mathcal{N}_{\tanh}(y) = x)$. $\square$

Henceforth, we will write $f(x) = y$ as a shorthand for $\varphi_f(x, y)$.

Lemma 3.5:

There is an NNL* formula $\varphi_{\ln}(x, y)$ such that, for all interpretation functions I, we have

$$I \models \varphi_{\ln}(x, y) \;\text{ iff }\; \ln(I(x)) = I(y)\,.$$

Proof. We will express ln in terms of σ^{-1}, $\tanh^{-1}$, and η:

- For all $r \in \mathbb{R}$ such that $r \geq 1$:

$$\eta(r-1) = \ln(r)$$

- For all $r \in \mathbb{R}$ such that $0 < r < 1$:

$$\begin{aligned}
\sigma^{-1}(r) &= \ln(r) - \ln(1-r) \\
\tanh^{-1}(r) &= \frac{1}{2}(\ln(1+r) - \ln(1-r)) \\
\sigma^{-1}(r) - 2\tanh^{-1}(r) + \eta(r) &= \ln(r) - \ln(1-r) - \ln(1+r) + \ln(1-r) + \ln(1+r) \\
&= \ln(r)
\end{aligned}$$

Thus, we can set

$$\varphi_{\ln}(x, y) = \left(\begin{array}{ll}
 & x > 0 \\
\wedge & (x \geq 1 \;\Rightarrow\; \exists z.(z = x - 1 \wedge \eta(z) = y)) \\
\wedge & (x < 1 \;\Rightarrow\; \exists z_1, z_2, z_3. \left(\begin{array}{ll} & y = z_1 - 2 \cdot z_2 + z_3 \\ \wedge & \sigma^{-1}(x) = z_1 \\ \wedge & \tanh^{-1}(x) = z_2 \\ \wedge & \eta(x) = z_3 \end{array}\right)
\end{array}\right).$$

This concludes the proof. ☐

Lemma 3.6:

There is an NNL* formula $\varphi_{\exp}(x, y)$ such that, for all interpretation functions I, we have

$$I \models \varphi_{\exp}(x, y) \;\text{ iff }\; e^{I(x)} = I(y)\,.$$

Proof. We set $\varphi_{\exp}(x, y) = \varphi_{\ln}(y, x)$ (obtained by exchanging x and y in $\varphi_{\ln}(x, y)$). ☐

Henceforth, we will write $e^x = y$ as a shorthand for $\varphi_{\exp}(x, y)$.

Lemma 3.7:

There is an NNL* formula $\varphi_{\mathsf{mult}}(x, y, z)$ such that, for all interpretation functions I, we have

$$I \models \varphi_{\mathsf{mult}}(x, y, z) \;\text{ iff }\; I(x) \cdot I(y) = I(z)\,.$$

Proof. Recall that, for all $r, s \in \mathbb{R}$ with $r, s > 0$, we have $r \cdot s = e^{\ln(r)+\ln(s)}$. We can set

$$\varphi_{\mathsf{mult}}(x,y,z) = \left(\begin{array}{lll} \wedge & (x = 0 \vee y = 0) \;\Rightarrow\; z = 0 & \\ \wedge & (x > 0 \wedge y > 0) \;\Rightarrow\; \exists z_1, z_2, z_3. & \left(\begin{array}{ll} & \ln(x) = z_1 \\ \wedge & \ln(y) = z_2 \\ \wedge & z_1 + z_2 = z_3 \\ \wedge & e^{z_3} = z \end{array}\right) \\ \wedge & (x < 0 \wedge y > 0) \;\Rightarrow\; \exists z_1, z_2, z_3, x', z'. & \left(\begin{array}{ll} & x' = -x \\ & \ln(x') = z_1 \\ \wedge & \ln(y) = z_2 \\ \wedge & z_1 + z_2 = z_3 \\ \wedge & e^{z_3} = z' \\ \wedge & z = -z' \end{array}\right) \\ \wedge & \cdots & \end{array}\right)$$

Completing the formula is left as an exercise. $\square$

Exercise 3.11:

Fill the dots in the definition of $\varphi_{\mathsf{mult}}(x,y,z)$ in the proof of Lemma 3.7.

Henceforth, we will write $x \cdot y = z$ as a shorthand for $\varphi_{\mathsf{mult}}(x,y,z)$.

We are now ready to prove Proposition 3.7 stating that REF $\leq$ NNL*:

Proof of Proposition 3.7. Consider a formula $\varphi = (t_1 \leq t_2)$ and let $\mathfrak{T}$ be the set of subterms occurring in t_1 or t_2. We replace φ with

$$\exists (z_t)_{t \in \mathfrak{T}}.(z_{t_1} \leq z_{t_2} \;\wedge\; \psi_{t_1} \;\wedge\; \psi_{t_2})$$

where ψ_{t_1} and ψ_{t_2} are defined inductively by

$$\begin{array}{lcl} \psi_a & = & (z_a = a) \\ \psi_x & = & (z_x = x) \\ \psi_{t+t'} & = & (z_t + z_{t'} = z_{t+t'} \;\wedge\; \psi_t \;\wedge\; \psi_{t'}) \\ \psi_{t \cdot t'} & = & (z_t \cdot z_{t'} = z_{t \cdot t'} \;\wedge\; \psi_t \;\wedge\; \psi_{t'}) \\ \psi_{e^t} & = & (e^{z_t} = z_{e^t} \;\wedge\; \psi_t) \end{array}$$

Doing this with all inequalities $t_1 \leq t_2$, we obtain the desired NNL* formula. $\square$

Chapter 4

Recurrent Neural Networks

Feed-forward neural networks process one single input vector and produce an output vector. In this chapter, we look at neural networks that process *sequences*. More precisely, they may translate sequences of input vectors into sequences of output vectors, or serve as sequence *classifiers*.

4.1 Definition and Semantics

We consider a simple class of recurrent neural networks, also known as Elman neural networks [16], rather than more advanced architectures such as long short-term memory (LSTM) [22].

Definition 4.1: Recurrent Neural Network

A *recurrent neural network (RNN)* is a triple $\mathcal{R} = (\mathscr{L}^{\mathsf{in}}, \mathscr{L}^{\mathsf{out}}, \mathbf{h}^{(0)})$ where, for some $m, n, o \in \mathbb{N}_+$,

- $\mathscr{L}^{\mathsf{in}}$ is a feed-forward layer with $dim(\mathscr{L}^{\mathsf{in}}) = (n + m, n)$,
- $\mathscr{L}^{\mathsf{out}}$ is a feed-forward layer with $dim(\mathscr{L}^{\mathsf{out}}) = (n, o)$,
- $\mathbf{h}^{(0)} \in \mathbb{Q}^n$ is the *initial (hidden) state vector*.

If $\mathscr{L}^{\mathsf{in}}$ uses activation function f and $\mathscr{L}^{\mathsf{out}}$ uses activation function g, we call $\mathcal{R}$ an (f, g)-RNN. We call

- $state(\mathcal{R}) = n$ the *state dimension* of $\mathcal{R}$ (i.e., the dimension of each hidden state),
- $in(\mathcal{R}) = m$ the *input dimension* of $\mathcal{R}$ (i.e., the dimension of each input symbol),
- $out(\mathcal{R}) = o$ the *output dimension* of $\mathcal{R}$ (i.e., the dimension of each output symbol).

However, states are considered to be hidden, and the type of the input-output relation is solely described by $dim(\mathcal{R}) = (m, o)$. When we see $\mathcal{R}$ as a *sequence-to-sequence* trans-

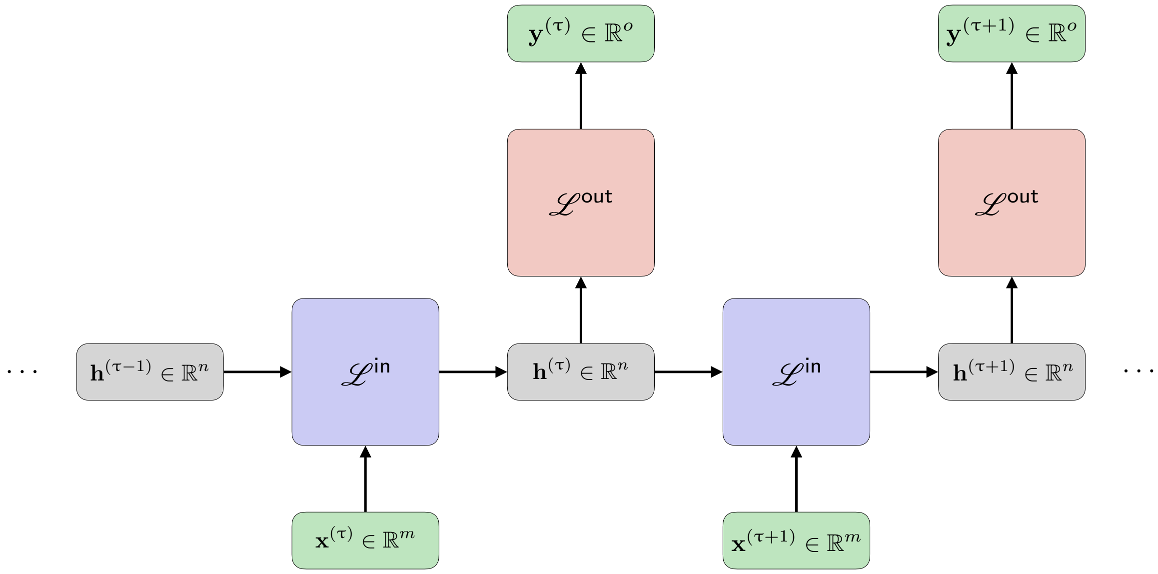


Figure 4.1: Sequence processing by an RNN through time

ducer[1], we consider the mapping

$$[\![\mathcal{R}]\!] : \begin{cases} (\mathbb{R}^m)^+ \to (\mathbb{R}^o)^+ \\ \mathbf{x}^{(1)} \dots \mathbf{x}^{(\ell)} \mapsto \mathbf{y}^{(1)} \dots \mathbf{y}^{(\ell)} \end{cases}$$

where, for all discrete time points $\tau \in \{1, \dots, \ell\}$,

$$\begin{aligned} \mathbf{h}^{(\tau)} &= [\![\mathscr{L}^{\mathsf{in}}]\!](\mathbf{h}^{(\tau-1)} \boxminus \mathbf{x}^{(\tau)}) \\ \mathbf{y}^{(\tau)} &= [\![\mathscr{L}^{\mathsf{out}}]\!](\mathbf{h}^{(\tau)}) \end{aligned}$$

Recall that $\mathbf{h}^{(\tau-1)} \boxminus \mathbf{x}^{(\tau)}$ is the vertical concatenation of $\mathbf{h}^{(\tau-1)}$ and $\mathbf{x}^{(\tau)}$. The computation is illustrated in Figure 4.1. That is, along the way, $\mathcal{R}$ also computes a sequence of *hidden states* $\mathbf{h}^{(1)} \dots \mathbf{h}^{(\ell)} \in (\mathbb{R}^m)^+$. Note that $[\![\mathcal{R}]\!]$ is length preserving, i.e., an input sequence containing ℓ vectors is always mapped to an output sequence containing ℓ vectors.

It can actually be useful to view $\mathcal{R}$ as an (infinite) automaton coming with a state-transition function $\delta_{\mathcal{R}} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ defined by $\delta_{\mathcal{R}}(\mathbf{h}, \mathbf{x}) = [\![\mathscr{L}^{\mathsf{in}}]\!](\mathbf{h} \boxminus \mathbf{x})$, which is canonically extended to sequences. When considering $\mathcal{R}$ as a *sequence-to-vector* transducer, we would be iterested in the mapping

$$\langle\!\langle \mathcal{R} \rangle\!\rangle : \begin{cases} (\mathbb{R}^m)^+ \to \mathbb{R}^o \\ w \mapsto [\![\mathscr{L}^{\mathsf{out}}]\!](\delta_{\mathcal{R}}(\mathbf{h}^{(0)}, w)) \,. \end{cases}$$

RNNs as String Transducers. Though input and output symbols are vectors over real numbers, they allow us to deal with symbols from a finite alphabet as well. This is important in natural language processing, where words are considered as single symbols and are encoded, for example via one-hot encoding, as vectors.

[1]We use the terms *sequence* and *string* interchangeably. However, we rather avoid the term *word* in this context. In NLP, words often refer to what we call letters (or tokens).

Definition 4.2: One-Hot Encoding

Let Σ be a finite alphabet with $|\Sigma| = m$. A *one-hot encoding* enc_Σ of Σ is an injective mapping $\Sigma \to \{0,1\}^m \subseteq \mathbb{R}^m$ such that, for all $\alpha \in \Sigma$, the vector $\mathbf{x} = enc_\Sigma(\alpha)$ satisfies $\sum_{i=1}^m x_i = 1$. For example, for $\Sigma = \{\alpha, \beta\}$, we may set $enc_\Sigma(\alpha) = (1,0)^\top$ and $enc_\Sigma(\beta) = (0,1)^\top$.

The corresponding *one-hot decoding* is the mapping $dec_\Sigma : \mathbb{R}^m \to \Sigma$ defined by $dec_\Sigma(\mathbf{x}) = \alpha$ where α is such that $\operatorname{argmax}(enc_\Sigma(\alpha)) = \min(\operatorname{argmax}(\mathbf{x}))$. For example, we have $dec_\Sigma((0.7, 0.5)^\top) = dec_\Sigma((0.5, 0.5)^\top) = \alpha$, and $dec_\Sigma((0.2, 0.5)^\top) = \beta$.

Remark 4.1: Binary Alphabets

If Σ is a binary alphabet such as $\{\alpha_0, \alpha_1\}$, we may also choose the dimension to be $m = 1$ and define $enc_\Sigma(\alpha_i) = i$ as well as $dec_\Sigma(x) = \alpha_1$ iff $x > \theta$ (or $x \geq \theta$) for some given threshold θ. This is particularly common in the output layer in combination with activation function σ and threshold $\theta = 0.5$.

The mappings enc_Σ and dec_Σ are extended to sequences as expected, i.e., $enc_\Sigma(\alpha_1 \dots \alpha_\ell) = enc_\Sigma(\alpha_1) \dots enc_\Sigma(\alpha_\ell)$ and $dec_\Sigma(\mathbf{x}^{(1)} \dots \mathbf{x}^{(\ell)}) = dec_\Sigma(\mathbf{x}^{(1)}) \dots dec_\Sigma(\mathbf{x}^{(\ell)})$. In the following, we assume that all finite alphabets come with their one-hot encoding/decoding.

Let Σ and Γ be finite alphabets with associated one-hot encodings enc_Σ and enc_Γ, and decodings dec_Σ and dec_Γ. Furthermore, assume $|\Sigma| = in(\mathcal{R})$ and $|\Gamma| = out(\mathcal{R})$. Then, RNN $\mathcal{R}$ defines the mappings

$$[\![\mathcal{R}]\!]_{\Sigma,\Gamma} : \begin{cases} \Sigma^+ \to \Gamma^+ \\ w \mapsto dec_\Gamma([\![\mathcal{R}]\!](enc_\Sigma(w))) \end{cases} \qquad \langle\!\langle \mathcal{R} \rangle\!\rangle_{\Sigma,\Gamma} : \begin{cases} \Sigma^+ \to \Gamma \\ w \mapsto dec_\Gamma(\langle\!\langle \mathcal{R} \rangle\!\rangle(enc_\Sigma(w))) \end{cases}$$

Remark 4.2: Alternative Semantics

If the activation function of $\mathscr{L}^{\mathsf{out}}$ is softmax, we may consider the mapping

$$\langle\!\langle \mathcal{R} \rangle\!\rangle^{\mathsf{prob}}_{\Sigma,\Gamma} : \begin{cases} \Sigma^+ \to \mathit{Distr}(\Gamma) \\ w \mapsto \langle\!\langle \mathcal{R} \rangle\!\rangle(enc_\Sigma(w)) \end{cases}$$

where $\mathit{Distr}(\Gamma)$ is the set of probability distributions over Γ. The idea is that, for the vector $\mathbf{y} = \langle\!\langle \mathcal{R} \rangle\!\rangle^{\mathsf{prob}}_{\Sigma,\Gamma}(w)$, the probability of selecting $\beta \in \Gamma$ as the next letter is $y_{\operatorname{argmax}(enc_\Gamma(\beta))}$. This view has applications in in character-level language modeling or text generation.

In the context of reactive systems, which are supposed to run forever, it is particularly interesting to consider *infinitary* versions of $[\![\mathcal{R}]\!]$ and $\langle\!\langle \mathcal{R} \rangle\!\rangle$, which are of the following form and whose definitions are as expected:

$$[\![\mathcal{R}]\!]^\omega : (\mathbb{R}^m)^\omega \to (\mathbb{R}^o)^\omega$$

$$[\![\mathcal{R}]\!]^\omega_{\Sigma,\Gamma} : \Sigma^\omega \to \Gamma^\omega$$

RNNs as String Classifiers. Let $\mathcal{R}$ be an RNN and $m = in(\mathcal{R})$. If $out(\mathscr{L}^{\mathsf{out}}) = 1$, then $\mathcal{R}$ can be viewed as a classifier of strings over $\mathbb{R}^m$: For $\bowtie \in \{\geq, >, =\}$ and $\theta \in \mathbb{R}$, we define

$$L^{\bowtie\theta}(\mathcal{R}) = \{w \in (\mathbb{R}^m)^+ \mid \langle\!\langle\mathcal{R}\rangle\!\rangle(w) \bowtie \theta\}$$

to be the *language* of $\mathcal{R}$, i.e., the set of strings that have a sufficient "score", or *acceptance probability*, and are therefore classified by $\mathcal{R}$ as positive.[2]

Suppose we are given a finite alphabet Σ. The RNN $\mathcal{R}$ is called an RNN *over* Σ if $m = |\Sigma|$. Like above, $\mathcal{R}$ can then take one-hot encoded letters from Σ as inputs. If, again, $out(\mathscr{L}^{\mathsf{out}}) = 1$, we can interpret $\mathcal{R}$ as a classifier of strings over Σ. For $\bowtie \in \{\geq, >, =\}$ and $\theta \in \mathbb{R}$, we let

$$L_\Sigma^{\bowtie\theta}(\mathcal{R}) = \{w \in \Sigma^+ \mid \langle\!\langle\mathcal{R}\rangle\!\rangle(enc_\Sigma(w)) \bowtie \theta\}$$

be the *language* of $\mathcal{R}$ over Σ.

Example 4.1: Example Applications of RNNs

Among the classical practical applications of RNNs, let us mention price prediction, text translation, or text classification. Several concrete use cases are given in the blog article [25]. Consider, for example, an email spam classifier that processes a text, i.e., a sequence over a finite alphabet Σ (where each letter represents a single word), and outputs a probability of being classified as spam. In other words, we are interested in filtering emails contained in a language of the form $L_\Sigma^{\geq\theta}(\mathcal{R})$. Hereby, it is up to the user to decide on the threshold θ to achieve a reasonable trade-off between sensitivity (true positive rate) and specificity (true negative rate).

4.2 Undecidability of the Emptiness Problem

In this section, we demonstrate that, unfortunately, determining the emptiness of the language of a given RNN, when treated as a string classifier, is undecidable.

Theorem 4.1: Undecidability of RNN Language (Non)emptiness

For all $\bowtie \in \{\geq, >, =\}$, the following decision problem is undecidable:

Input: A finite alphabet Σ and a (ReLU, σ)-RNN $\mathcal{R}$ over Σ with $out(\mathcal{R}) = 1$.

Question: Do we have $L_\Sigma^{\bowtie\frac{1}{2}}(\mathcal{R}) \neq \emptyset$?

As this fundamental decision problem is already undecidable for RNNs, this will also be the case for the majority of non-trivial verification tasks. Thus, it is important to identify suitable abstractions or restrictions. There has been an ongoing and fruitful effort to establish positive verification results specifically designed for RNNs [2, 24, 27, 42, 53].

The rest of this section is dedicated to the proof of Theorem 4.1. We first show that (ReLU, σ)-RNNs can simulate all *probabilistic finite automata (PFAs)*. PFAs were intro-

[2] If the activation function of $\mathscr{L}^{\mathsf{out}}$ is the sigmoid function σ, we can also interpret the score as *acceptance probability*.

duced and studied by Rabin [41]. Their (non)emptiness problem is undecidable and they are strictly more expressive than finite automata.

Definition 4.3: Probabilistic Finite Automaton (PFA)

Let Σ be a finite alphabet. A *probabilistic finite automaton (PFA)* over Σ is a tuple $\mathcal{A} = ((\mathbf{P}^\alpha)_{\alpha\in\Sigma}, \boldsymbol{\lambda}, \boldsymbol{\gamma})$ where, for some $n \in \mathbb{N}_+$,

- $\mathbf{P}^\alpha = (p^\alpha_{j,i}) \in ([0,1] \cap \mathbb{Q})^{n\times n}$ such that, for all $i \in \{1,\dots,n\}$, $\sum_{j=1}^n p^\alpha_{j,i} = 1$,
- $\boldsymbol{\lambda} \in \{0,1\}^n$ is the *initial state vector* (a column vector) such that $\sum_{i=1}^n \lambda_i = 1$,
- $\boldsymbol{\gamma} \in \{0,1\}^n$ is the *final state vector* (a row vector).

Note that $\mathbf{P}^\alpha$ is a *left stochastic matrix*. We call it the *transition-probability matrix* of $\alpha \in \Sigma$. The intuition is that $\mathcal{A}$ has an (implicit) set of states $Q = \{q_1,\dots,q_n\}$, and $p^\alpha_{j,i}$ is the probability that, when reading α in state q_i, we go to state q_j.[3] Thus, $\mathcal{A}$ has an equivalent representation as a *state-transition graph* (Q, ι, Δ, F), which is basically a finite automaton with transition probabilities. The initial state is $\iota = q_i$ for $\lambda_i = 1$, and the set of final states is $F = \{q_i \mid i \in \{1,\dots,n\} \text{ with } \gamma_i = 1\}$. Moreover, we have a set of transitions $\Delta \subseteq Q \times \Sigma \times (0,1] \times Q$ containing, for all $\alpha \in \Sigma$ and all $i,j \in \{1,\dots,n\}$ such that $p^\alpha_{j,i} > 0$, the transition $(q_i, \alpha, p^\alpha_{j,i}, q_j)$. We will switch between these representations at discretion.

PFA $\mathcal{A}$ defines the mapping

$$\delta_{\mathcal{A}} : \begin{cases} \mathbb{R}^n \times \Sigma \to \mathbb{R}^n \\ (\mathbf{x}, \alpha) \mapsto \mathbf{P}^\alpha \cdot \mathbf{x}\,. \end{cases}$$

Suppose that $\delta_{\mathcal{A}}$ gets $(\mathbf{x}, \alpha)$ as arguments and that $\mathbf{x}$ represents a probability distribution over $\{q_1,\dots,q_n\}$, i.e., x_i is the probability of being in state q_i. Then, letting $\mathbf{x}' = \delta_{\mathcal{A}}(\mathbf{x}, \alpha)$, x'_i can naturally be interpreted as the probability of being in state q_i after reading α. Using the extension of $\delta_{\mathcal{A}}$ to strings, we obtain the mapping

$$[\![\mathcal{A}]\!] : \begin{cases} \Sigma^* \to [0,1] \\ w \mapsto \boldsymbol{\gamma} \cdot \delta_{\mathcal{A}}(\boldsymbol{\lambda}, w)\,. \end{cases}$$

Thus, starting from the initial state, $[\![\mathcal{A}]\!](w)$ is the probability of reaching a final state after reading w. Finally, for $\bowtie \in \{\geq, >, =\}$ and $\theta \in [0,1]$, we define the language

$$L^{\bowtie\theta}(\mathcal{A}) = \{w \in \Sigma^+ \mid [\![\mathcal{A}]\!](w) \bowtie \theta\}\,.$$

We restrict to nonempty strings here, aligning with the definition of RNN languages.

[3] We choose $p^\alpha_{j,i}$ rather than the more standard $p^\alpha_{i,j}$, as it reflects the the way computations are represented in the layers of RNNs, where the transition matrix on the left is multiplied with the current state on the right hand side.

Remark 4.3: Reactive vs. Generative Probabilistic Automata

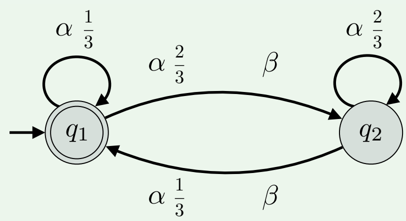
PFAs due to Rabin are also called *reactive*, as they yield a probability distribution when providing an input. On the other hand, *generative* probabilistic automata (also called Segala automata [46]) generate the next letter according to a probability distribution over $\Sigma \times Q$. In formal terms, we may consider that reactive PFAs (as we study here) are equipped with a transition function $\delta : Q \times \Sigma \to \mathit{Distr}(Q)$, whereas generative PFAs have a transition function of type $\delta : Q \to \mathit{Distr}(\Sigma \times Q)$.

Example 4.2: Probabilistic Finite Automaton

Let $\Sigma = \{\alpha, \beta\}$. Consider the PFA $\mathcal{A} = ((\mathbf{P}^\alpha, \mathbf{P}^\beta), \boldsymbol{\lambda}, \boldsymbol{\gamma})$ over Σ given by

$$\boldsymbol{\lambda} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \mathbf{P}^\alpha = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} \end{pmatrix} \qquad \mathbf{P}^\beta = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \boldsymbol{\gamma} = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

Alternatively, $\mathcal{A}$ can be represented by its state-transition diagram:



For $k \in \mathbb{N}$, we have

$$[\![\mathcal{A}]\!](\alpha\beta^k) = \begin{pmatrix} 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}^k \cdot \begin{pmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{cases} \frac{1}{3} & \text{if } k \text{ is even} \\ \frac{2}{3} & \text{if } k \text{ is odd} \end{cases}$$

It is easy to see that every string ending in α leads to the final state with probability $\frac{1}{3}$. Thus, no such strings is in $L^{\geq \frac{1}{2}}(\mathcal{A})$. Altogether, we obtain

$$L^{\geq \frac{1}{2}}(\mathcal{A}) = \{\beta^k \mid k \text{ is odd}\} \cup \{w\alpha\beta^k \mid w \in \{\alpha, \beta\}^* \text{ and } k \text{ is odd}\}\,.$$

Note that the image of $[\![\mathcal{A}]\!]$ is finite since we have $[\![\mathcal{A}]\!](w) \in \{0, \frac{1}{3}, \frac{2}{3}, 1\}$ for all $w \in \Sigma^+$. In general, however, this is not the case.

PFAs enjoy some essential closure properties:

Lemma 4.1: Closure Properties of PFAs

Let Σ be a finite alphabet, $p \in [0, 1]$ be rational, and $\mathcal{A}$ and $\mathcal{B}$ be PFAs over Σ. We can effectively construct PFAs $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ over Σ such that, for all $w \in \Sigma^+$,

- $[\![\mathcal{C}_1]\!](w) = 1 - [\![\mathcal{A}]\!](w)$,
- $[\![\mathcal{C}_2]\!](w) = p \cdot [\![\mathcal{A}]\!](w) + (1 - p) \cdot [\![\mathcal{B}]\!](w)$, and

– $[\![\mathcal{C}_3]\!](w) = [\![\mathcal{A}]\!](w) \cdot [\![\mathcal{B}]\!](w)$.

Exercise 4.1:

Prove Lemma 4.1.

The next theorem states that RNNs with a particular set of activation functions are at least as expressive as PFAs. It constitutes the first step towards showing that emptiness of RNNs is undecidable.

Theorem 4.2: RNNs Recognize All PFA Languages

Let Σ be a finite alphabet and $\mathcal{A}$ be a PFA over Σ. Moreover, let $\bowtie \in \{\geq, >, =\}$ and $\theta \in [0,1]$ be rational. We can effectively construct a (ReLU, σ)-RNN $\mathcal{R}$ over Σ such that $out(\mathcal{R}) = 1$ and

$$L_\Sigma^{\bowtie \frac{1}{2}}(\mathcal{R}) = L^{\bowtie \theta}(\mathcal{A})\,.$$

Before we prove the theorem formally, we illustrate the construction using an example.

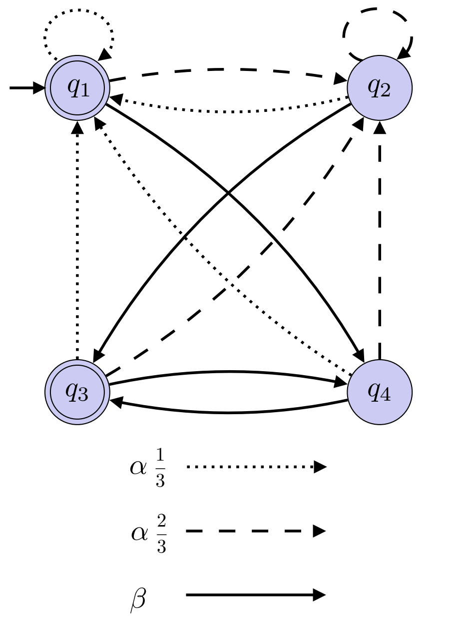
Example 4.3: From PFAs to RNNs

An obvious first idea is to choose $\mathscr{L}^{\mathsf{in}}$ such that $in(\mathscr{L}^{\mathsf{in}}) = |\Sigma|$ (with one-hot encoded letters as inputs) and $state(\mathscr{L}^{\mathsf{in}}) = |Q|$ as state dimension. The corresponding matrix $\mathbf{A}^{\mathsf{in}}$ would then have dimension $|Q| \times (|Q| + |\Sigma|)$. However, in $\mathcal{A}$, we have $|\Sigma| \cdot |Q|^2$ transition probabilities, which thus may not fit into $\mathbf{A}^{\mathsf{in}}$.

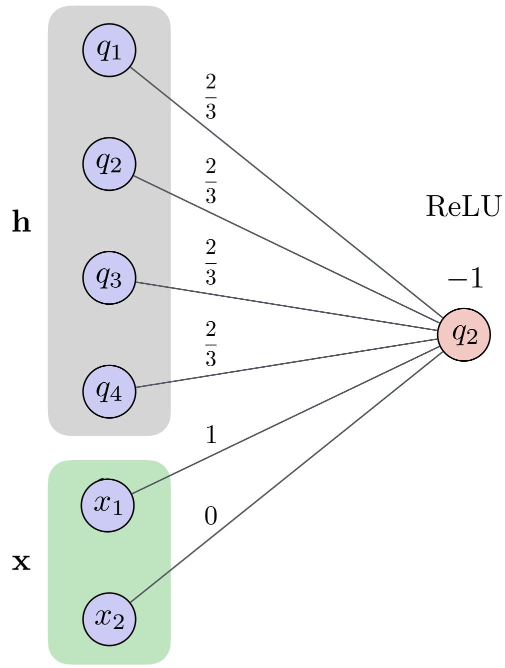
The solution will be to "augment" the set of states so that every state has incoming transitions of a unique letter type. Consider, for example, the PFA $\mathcal{A}$ from Example 4.2. Recall that $\Sigma = \{\alpha, \beta\}$ and suppose $enc_\Sigma(\alpha) = (1,0)^\top$ and $enc_\Sigma(\beta) = (0,1)^\top$. The PFA $\mathcal{A}'$ illustrated in Figure 4.2a, is equivalent to the PFA $\mathcal{A}$ from Example 4.2, i.e., $[\![\mathcal{A}]\!](w) = [\![\mathcal{A}']\!](w)$ for all $w \in \Sigma^*$. However, every state of $\mathcal{A}'$ now has only incoming transitions labeled with a dedicated letter α or β (which we accordingly call an α- or β-state). Note that $\mathcal{A}'$ is given by the following ingredients:

$$\boldsymbol{\lambda} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \mathbf{P}^\alpha = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} & \frac{2}{3} & \frac{2}{3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad \mathbf{P}^\beta = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \boldsymbol{\gamma} = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

Thanks to the transformation, the transition-probability matrices $\mathbf{P}^\alpha$ and $\mathbf{P}^\beta$ no longer "overlap": for all $i, j \in \{1,2,3,4\}$, at most one of $p^\alpha_{j,i}$ and $p^\beta_{j,i}$ is non-zero. That is, the sum $\mathbf{P}^\alpha$ and $\mathbf{P}^\beta$ accommodates all probabilities occurring in $\mathcal{A}'$ in a single matrix of dimension $\mathbb{R}^{4\times4}$. The layer $\mathscr{L}^{\mathsf{in}}$ of the desired RNN is illustrated in Figure 4.2b for a neuron belonging to the α-state q_2. If the input letter is α, i.e., the input vector is $(1,0)^\top$, the output neuron indeed receives the updated probability



(a) PFA $\mathcal{A}'$ where every state has a unique "incoming letter"



(b) A neuron for target α-state q_2

of being in q_2 after processing the input. If, on the other hand, the input is β and, respectively, $(0,1)^\top$, then the result before applying ReLU will be ≤ 0 so that the overall result is 0. As q_2 is an α-state, the probability of being in q_2 after processing β is indeed 0.

The study of the expressive power of RNNs and their extensions and variants in terms of formal languages has a long tradition and is still is an active research field [36]. In this context, a prominent result by Siegelmann and Sontag states that RNNs can simulate all Turing machines [48]. A general technique of transforming finite-state machines into equivalent RNNs goes back to Minsky [37]. It has recently been generalized to *generative* PFAs [50], which define probability distributions over Σ^* (cf. Remark 4.2). Here, we apply this technique to translate (reactive) PFAs into RNNs, aiming to show undecidability of the RNN emptiness problem.

Now, let us formally prove Theorem 4.2.

Proof of Theorem 4.2. Let Σ be a finite alphabet and $m = |\Sigma|$. Let $\mathcal{A} = ((\mathbf{P}^\alpha)_{\alpha\in\Sigma}, \boldsymbol{\lambda}, \boldsymbol{\gamma})$ be a PFA over Σ, with state-transition representation (Q, ι, Δ, F) where $Q = \{q_1, \ldots, q_n\}$. Moreover, fix $\bowtie \in \{\geq, >, =\}$ and a rational number $\theta \in [0,1]$.

Without loss of generality, we assume that, for every $i_1, i_2, j \in \{1, \ldots, n\}$, $\alpha, \beta \in \Sigma$, and $p_1, p_2 \in (0,1]$ such that $(q_{i_1}, \alpha, p_1, q_j) \in \Delta$ and $(q_{i_2}, \beta, p_2, q_j) \in \Delta$, we have $\alpha = \beta$.[4] We call state q_j whose incoming transitions all carry letter α an *α-state*. Note that the transition probability matrices are not "overlapping", i.e., for all $\alpha, \beta \in \Sigma$ and $i, j \in \{1, \ldots, n\}$ such that $p^\alpha_{j,i} > 0$ and $p^\beta_{j,i} > 0$, we have $\alpha = \beta$.

[4] If $\mathcal{A}$ does not satisfy this property, we can convert it choosing $Q \times \Sigma$ as new set of states and introducing, for all $(q_i, \alpha, p, q_j) \in \Delta$, transitions $((q_i, \beta), \alpha, p, (q_j, \alpha))$ for all $\beta \in \Sigma$.

We now formally construct the corresponding (ReLU, σ)-RNN $\mathcal{R} = (\mathscr{L}^{\mathsf{in}}, \mathscr{L}^{\mathsf{out}}, \mathbf{h}^{(0)})$ over Σ such that $\mathit{out}(\mathscr{L}^{\mathsf{out}}) = 1$ and $L_{\Sigma}^{\bowtie\frac{1}{2}}(\mathcal{R}) = L(\mathcal{A})$.

- We let $\mathbf{h}^{(0)} = \boldsymbol{\lambda}$.
- Define $\mathscr{L}^{\mathsf{in}} = (\mathbf{A}^{\mathsf{in}}, \mathbf{b}^{\mathsf{in}}, \mathrm{ReLU})$ as follows: Let $\mathbf{A}^{\mathsf{in}} = \mathbf{P} \mathbin{\Box\!\Box} \mathbf{C}$ be the concatenation of the matrices $\mathbf{P} \in [0,1]^{n \times n}$ and $\mathbf{C} \in \{0,1\}^{n \times m}$ defined by

$$\mathbf{P} = \sum_{\alpha \in \Sigma} \mathbf{P}^{\alpha}$$

  and, for all $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, m\}$,

$$c_{j,k} = \begin{cases} 1 & \text{if } \operatorname{argmax}(\mathit{enc}_{\Sigma}(\alpha)) = k \\ 0 & \text{otherwise.} \end{cases}$$

  Finally, for all $j \in \{1, \ldots, n\}$, we set $b_j^{\mathsf{in}} = -1$.
- We define $\mathscr{L}^{\mathsf{out}} = (\mathbf{A}^{\mathsf{out}}, \mathbf{b}^{\mathsf{out}}, \sigma)$ by $\mathbf{A}^{\mathsf{out}} = \boldsymbol{\gamma}$ and $\mathbf{b}^{\mathsf{out}} = (-\theta)$.

Correctness Proof. Let us show that, for all hidden states $\mathbf{h} \in [0,1]^n$ and $\alpha \in \Sigma$, $\delta_{\mathcal{R}}(\mathbf{h}, \mathit{enc}_{\Sigma}(\alpha)) = \delta_{\mathcal{A}}(\mathbf{h}, \alpha)$. Let $\mathbf{x} = \mathit{enc}_{\Sigma}(\alpha)$. We have

$$\begin{aligned} \delta_{\mathcal{R}}(\mathbf{h}, \mathit{enc}_{\Sigma}(\alpha)) &= \delta_{\mathcal{R}}(\mathbf{h}, \mathbf{x}) = [\![\mathscr{L}^{\mathsf{in}}]\!](\mathbf{h} \boxminus \mathbf{x}) \\ &= \mathrm{ReLU}(\mathbf{A}^{\mathsf{in}} \cdot (\mathbf{h} \boxminus \mathbf{x}) + \mathbf{b}^{\mathsf{in}}) \\ &= \mathrm{ReLU}((\mathbf{P} \mathbin{\Box\!\Box} \mathbf{C}) \cdot (\mathbf{h} \boxminus \mathbf{x}) + \mathbf{b}^{\mathsf{in}}) \\ &= \mathrm{ReLU}(\mathbf{P} \cdot \mathbf{h} + \mathbf{C} \cdot \mathbf{x} + \mathbf{b}^{\mathsf{in}}) \\ &= \mathrm{ReLU}((\textstyle\sum_{\beta \in \Sigma} \mathbf{P}^{\beta}) \cdot \mathbf{h} + \mathbf{C} \cdot \mathbf{x} + \mathbf{b}^{\mathsf{in}}) \end{aligned}$$

Let $\mathbf{y} = (y_1, \ldots, y_n)^{\top} = \mathbf{C} \cdot \mathbf{x} + \mathbf{b}^{\mathsf{in}}$. For all $j \in \{1, \ldots, n\}$, we have

$$y_j = \begin{cases} 0 & \text{if } q_j \text{ is an } \alpha\text{-state} \\ -1 & \text{otherwise.} \end{cases}$$

Thus, letting $\mathbf{z} = (\sum_{\beta \in \Sigma} \mathbf{P}^{\beta}) \cdot \mathbf{h} + \mathbf{y} \in \mathbb{R}^n$, we get, for all $j \in \{1, \ldots, n\}$,

$$\begin{cases} z_j = \mathbf{P}_j^{\alpha} \cdot \mathbf{h} & \text{if } q_j \text{ is an } \alpha\text{-state} \\ z_j \leq 0 & \text{otherwise.} \end{cases}$$

where $\mathbf{P}_j^{\alpha}$ is the j-th row of $\mathbf{P}^{\alpha}$. We conclude

$$\delta_{\mathcal{R}}(\mathbf{h}, \mathit{enc}_{\Sigma}(\alpha)) = \mathrm{ReLU}(\mathbf{z}) = \mathbf{P}^{\alpha} \cdot \mathbf{h} = \delta_{\mathcal{A}}(\mathbf{h}, \alpha)\,.$$

With this, we obtain that, for all $w \in \Sigma^+$,

$$\begin{aligned}
&\langle\!\langle \mathcal{R} \rangle\!\rangle(enc_\Sigma(w)) \bowtie \tfrac{1}{2}\\
\text{iff } &[\![\mathscr{L}^{\mathsf{out}}]\!](\delta_{\mathcal{R}}(\mathbf{h}^{(0)}, enc_\Sigma(w))) \bowtie \tfrac{1}{2}\\
\text{iff } &[\![\mathscr{L}^{\mathsf{out}}]\!](\delta_{\mathcal{A}}(\boldsymbol{\lambda}, w)) \bowtie \tfrac{1}{2}\\
\text{iff } &\sigma(\boldsymbol{\gamma} \cdot \delta_{\mathcal{A}}(\boldsymbol{\lambda}, w) - \theta) \bowtie \tfrac{1}{2}\\
\text{iff } &\boldsymbol{\gamma} \cdot \delta_{\mathcal{A}}(\boldsymbol{\lambda}, w) - \theta \bowtie 0\\
\text{iff } &\boldsymbol{\gamma} \cdot \delta_{\mathcal{A}}(\boldsymbol{\lambda}, w) \bowtie \theta\\
\text{iff } &[\![\mathcal{A}]\!](w) \bowtie \theta
\end{aligned}$$

We have shown $L_\Sigma^{\bowtie \frac{1}{2}}(\mathcal{R}) = L^{\bowtie \theta}(\mathcal{A})$. □

To prove Theorem 4.1, i.e., undecidability of RNN language emptiness, it remains to establish the corresponding facts for PFAs, depending on $\bowtie \in \{\geq, >, =\}$.

The following undecidability results for PFAs are due to [7] and [39]. The proofs were later simplified and strengthened in [18]. For a concise overview of what is decidable and undecidable in PFAs, we refer to [17].

Theorem 4.3: Undecidability of PFA Language Emptiness [7, 39]

The following three decision problems are undecidable:

Input: A finite alphabet Σ and a PFA $\mathcal{A}$ over Σ.

Question 1: Do we have $L^{=\frac{1}{2}}(\mathcal{A}) \neq \emptyset$ (i.e., $[\![\mathcal{A}]\!](w) = \frac{1}{2}$ for some $w \in \Sigma^+$)?

Question 2: Do we have $L^{\geq \frac{1}{4}}(\mathcal{A}) \neq \emptyset$?

Question 3: Do we have $L^{> \frac{1}{8}}(\mathcal{A}) \neq \emptyset$?

Proof. We first consider Question 1. The proof is a reduction from the following modified Post's correspondence problem (modified PCP)[5].

Input: A finite alphabet Σ and morphisms $f_1, f_2 : \Sigma^* \to \{0,1\}^*$ such that $f_i(\alpha) \in 1(0+1)^*$ for all $i \in \{1,2\}$ and $\alpha \in \Sigma$.

Question: Is there $w \in \Sigma^+$ such that $f_1(w) = f_2(w)$?

Given an instance f_1, f_2 of the modified PCP, we will effectively construct a PFA $\mathcal{A}$ over Σ such that, for all $w \in \Sigma^+$, we have $f_1(w) = f_2(w)$ iff $[\![\mathcal{A}]\!](w) = \frac{1}{2}$, which implies the theorem.

[5]The standard undecidable PCP does not have the restriction $f_i(\alpha) \in 1(0+1)^*$. However, when we start with an unrestricted instance of the form $g_1, g_2 : \Sigma \to \{0,1\}^+$, we can translate it into $f_1, f_2 : \Sigma \to \{0,1\}^*$ defined by $f_i = f \circ g_i$ where $f : \{0,1\}^* \to \{0,1\}^*$ is the morphism given by $f(0) = 10$ and $f(0) = 11$. Then, $f_i(\alpha)$ starts with 1 for all $\alpha \in \Sigma$. Moreover, we easily see that $g_1(w) = g_2(w)$ for some $w \in \Sigma^+$ iff $f_1(w) = f_2(w)$ for some $w \in \Sigma^+$

Let $\overline{\varepsilon} = 0$ and, for $u_1 \dots u_k \in \{0,1\}$ with $k \geq 1$,

$$\begin{aligned}\overline{u_1 \dots u_k} &= 0.u_k u_{k-1} \dots u_1 \text{ (in binary)} \\ &= \frac{u_k}{2^1} + \frac{u_{k-1}}{2^2} + \dots + \frac{u_1}{2^k}\,.\end{aligned}$$

Thanks to the modified PCP, we have, for all $w \in \Sigma^*$, $f_1(w) = f_2(w)$ iff $\overline{f_1(w)} = \overline{f_2(w)}$.

Towards the PFA $\mathcal{A}$, we will construct two PFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ over Σ such that, for all $i \in \{1,2\}$ and $w \in \Sigma^*$, we get

$$[\![\mathcal{A}_i]\!](w) = \overline{f_i(w)}\,. \tag{4.1}$$

We then combine $\mathcal{A}_1$ and $\mathcal{A}_2$ using Lemma 4.1 and obtain a PFA $\mathcal{A}$ over Σ such that, for all $w \in \Sigma^*$,

$$[\![\mathcal{A}]\!](w) = \frac{1}{2}([\![\mathcal{A}_1]\!](w) + (1 - [\![\mathcal{A}_2]\!](w)))\,.$$

Then, we are done as $[\![\mathcal{A}]\!](w) = \frac{1}{2}$ iff $\overline{f_1(w)} = \overline{f_2(w)}$ iff $f_1(w) = f_2(w)$.

PFA Evaluating Binary Numbers. The main building block in the construction of both $\mathcal{A}_1$ and $\mathcal{A}_2$ will be a PFA $\mathcal{B} = ((\mathbf{P}^0, \mathbf{P}^1), \boldsymbol{\lambda}, \boldsymbol{\gamma})$ over $\{0,1\}$ such that, for all $\nu \in \{0,1\}^*$, $[\![\mathcal{B}]\!](\nu) = \overline{\nu}$.[6] Thus, $\mathcal{B}$ "evaluates" ν. It is given by

$$\boldsymbol{\lambda} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \mathbf{P}^0 = \begin{pmatrix} 1 & \frac{1}{2} \\ 0 & \frac{1}{2} \end{pmatrix} \qquad \mathbf{P}^1 = \begin{pmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & 1 \end{pmatrix} \qquad \boldsymbol{\gamma} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

We show $[\![\mathcal{B}]\!](\nu) = \overline{\nu}$, by induction on $k = |\nu|$. The claim clearly holds for $k = 0$, i.e., $\nu = \varepsilon$. Moreover, for the case $k = 1$,

$$\begin{aligned} [\![\mathcal{B}]\!](0) &= p^0_{2,1} = 0.0 = \overline{0} \\ \text{and} \quad [\![\mathcal{B}]\!](1) &= p^1_{2,1} = 0.1 = \overline{1} \end{aligned}$$

Now suppose, for $\nu = u_1 \dots u_k$ with $k \geq 1$, that $[\![\mathcal{B}]\!](\nu) = \overline{\nu}$ holds. Moreover, assume

$$\delta_{\mathcal{B}}(\boldsymbol{\lambda}, \nu) = \begin{pmatrix} 1 - p \\ p \end{pmatrix}$$

for suitable $p \in [0,1]$. In particular,

$$p = [\![\mathcal{B}]\!](\nu) = \overline{\nu}\,. \tag{4.2}$$

Let $u \in \{0,1\}$. We have

$$\begin{aligned} [\![\mathcal{B}]\!](u_1 \dots u_k u) &= \begin{pmatrix} 0 & 1 \end{pmatrix} \cdot \mathbf{P}^u \cdot \begin{pmatrix} 1 - p \\ p \end{pmatrix} = \begin{cases} \dfrac{p}{2} & \text{if } u = 0 \\[2ex] \dfrac{1}{2} + \dfrac{p}{2} & \text{if } u = 1 \end{cases} \\ &= \frac{u}{2} + \frac{p}{2} \overset{(4.2)}{=} 0.u \ + \ 0.0u_k \dots u_1 = 0.uu_k \dots u_1 \ = \ \overline{u_1 \dots u_k u} \end{aligned}$$

[6] Note that $\mathbf{P}^0$ and $\mathbf{P}^1$ should not be confused with powers of some matrix $\mathbf{P}$.

Construction of $\mathcal{A}$. Let $i \in \{1,2\}$. Building on the PFA $\mathcal{B} = ((\mathbf{P}^0, \mathbf{P}^1), \boldsymbol{\lambda}, \boldsymbol{\gamma})$, we now construct the PFA $\mathcal{A}_i = ((\mathbf{P}^\alpha)_{\alpha\in\Sigma}, \boldsymbol{\lambda}, \boldsymbol{\gamma})$ over Σ such that, for all strings $w \in \Sigma^*$, we have $[\![\mathcal{A}_i]\!](w) = \overline{f_i(w)}$. For $\nu = u_1 \dots u_k$ with $k \geq 1$, define $\mathbf{P}^\nu = \mathbf{P}^{u_k} \cdot \ldots \cdot \mathbf{P}^{u_1}$. With this, given a letter $\alpha \in \Sigma$, we let $\mathbf{P}^\alpha = \mathbf{P}^{f_i(\alpha)}$. Indeed, for all $w = \alpha_1 \dots \alpha_\ell \in \Sigma^*$, we obtain

$$\begin{aligned}
[\![\mathcal{A}_i]\!](w) &= \boldsymbol{\gamma} \cdot \mathbf{P}^{\alpha_\ell} \cdot \ldots \cdot \mathbf{P}^{\alpha_1} \cdot \boldsymbol{\lambda} \\
&= \boldsymbol{\gamma} \cdot \mathbf{P}^{f_i(\alpha_\ell)} \cdot \ldots \cdot \mathbf{P}^{f_i(\alpha_1)} \cdot \boldsymbol{\lambda} \\
&= \boldsymbol{\gamma} \cdot \mathbf{P}^{f_i(w)} \cdot \boldsymbol{\lambda} = \overline{f_i(w)}\,.
\end{aligned}$$

This concludes the proof of undecidability of the first problem.

Questions 2 and 3. Question 1 can be reduced to Question 2 as follows: From the given PFA $\mathcal{A}$, we construct, using Theorem 4.1, a PFA $\mathcal{B}$ over Σ such that, for all $w \in \Sigma^*$, we have

$$[\![\mathcal{B}]\!](w) = [\![\mathcal{A}]\!](w) \cdot (1 - [\![\mathcal{A}]\!](w))\,.$$

Note that $\frac{1}{4}$ is the global maximum of the function $r \mapsto r \cdot (1-r)$, which is only reached for the argument $r = \frac{1}{2}$. Thus, we have $[\![\mathcal{B}]\!](w) \geq \frac{1}{4}$ iff $[\![\mathcal{A}]\!](w) = \frac{1}{2}$.

Undecidability of Question 3 can be obtained by analyzing the transition probabilities in the automata considered so far. We refer the reader to [18]. $\square$

We now have shown Theorem 4.1, using the effective constructions from Theorems 4.2 and 4.3 to reduce PFA emptiness problems for $\bowtie \in \{\geq, >, =\}$ to RNN emptiness. Specifically, for every PFA $\mathcal{A}$ over Σ, we constructed (ReLU, σ)-RNNs $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3$ over Σ whose output layers have output dimension 1 and such that the following hold:

$$\begin{aligned}
L_\Sigma^{=\frac{1}{2}}(\mathcal{R}_1) &= L^{=\frac{1}{2}}(\mathcal{A}) \\
L_\Sigma^{\geq\frac{1}{2}}(\mathcal{R}_2) &= L^{\geq\frac{1}{4}}(\mathcal{A}) \\
L_\Sigma^{>\frac{1}{2}}(\mathcal{R}_3) &= L^{>\frac{1}{8}}(\mathcal{A})
\end{aligned}$$

Chapter 5

Attention and Transformers

Transformers have been introduced by Vaswani as a powerful alternative to RNNs and their variants [55]. Like RNNs, transformers can be used as *sequence-to-sequence* transducers, as they process sequences of arbitrary length. For language-recognition tasks, we will consider only the encoder (or decoder) part, as is the case in popular language-model architectures like BERT and GPT.

Verification issues for transformers have only been addressed sparingly so far. The main purpose of this chapter is to highlight a few interesting questions for future research.

5.1 Attention

An essential component of a transformer is *attention*, specifically, an *attention head*. There is an analogy between attention heads and channels of convolutional neural networks. Both allow the network to focus on different aspects and parts of an input. Therefore, one usually has several attention heads per layer.

Definition 5.1: Attention Head

An attention head $\mathcal{H} = (\mathbf{Q}, \mathbf{K}, \mathbf{V})$ is given by three matrices (i.e., linear transformations) $\mathbf{Q}, \mathbf{K} \in \mathbb{Q}^{n_{\mathsf{key}} \times n}$ and $\mathbf{V} \in \mathbb{Q}^{n_{\mathsf{value}} \times n}$ for some $n, n_{\mathsf{key}}, n_{\mathsf{value}} \in \mathbb{N}_+$. They are respectively called *query*, *key*, and *value matrix*.

We let $in(\mathcal{H}) = n$, $out(\mathcal{H}) = n_{\mathsf{value}}$, and $dim(\mathcal{H}) = (n, n_{\mathsf{value}})$. Later, we will see that n can be understood as the dimension of the *hidden state* of a transformer. Attention head $\mathcal{H}$ defines a mapping

$$[\![\mathcal{H}]\!] : \begin{cases} (\mathbb{R}^n)^+ \times \mathbb{R}^n \to \mathbb{R}^{n_{\mathsf{value}}} \\ (\mathbf{z}^{(1)} \dots \mathbf{z}^{(\ell)}, \mathbf{x}) \mapsto \mathbf{y} \end{cases}$$

which allows it to situate a vector (word/letter embedding or hidden state) inside a whole sequence. Let $\mathbf{q} = \mathbf{Q} \cdot \mathbf{x}$. Moreover, for $i \in \{1, \dots, \ell\}$, let $\mathbf{k}^{(i)} = \mathbf{K} \cdot \mathbf{z}^{(i)}$ and $\mathbf{v}^{(i)} = \mathbf{V} \cdot \mathbf{z}^{(i)}$. The semantics is then given by

$$\mathbf{y} = \sum_{i=1}^{\ell} p_i \cdot \mathbf{v}^{(i)}$$

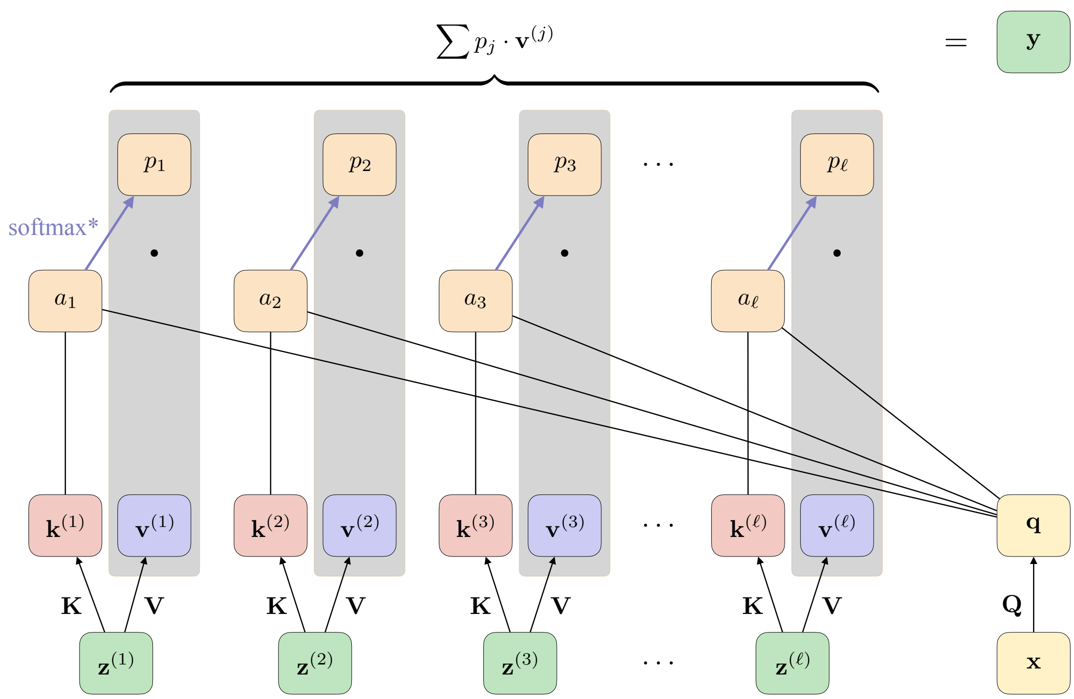


Figure 5.1: Illustration of an attention head $\mathcal{H} = (\mathbf{Q}, \mathbf{K}, \mathbf{V})$

where $(p_1, \ldots, p_\ell) = \mathit{Weights}(a_1, \ldots, a_\ell)$ with

$$a_i = \frac{1}{\sqrt{n_{\mathsf{key}}}} \cdot (\mathbf{q}^\top \cdot \mathbf{k}^{(i)}) \,.$$

The scaling parameter $\frac{1}{\sqrt{n_{\mathsf{key}}}}$ is optional. The function $\mathit{Weights} : \mathbb{R}^+ \to \mathbb{R}^+$ is a length-preserving *weight function*. Usually, one chooses $\mathit{Weights} = \text{softmax}^* \colon \mathbb{R}^+ \to \mathbb{R}^+$, where the latter is softmax adapted for sequences, i.e., for a variable number of input arguments, instead of a fixed number of input values.

For theoretical considerations, one sometimes replaces softmax* by other weight functions, in particular:

– min-argmax$^*\colon \mathbb{R}^+ \to \{0,1\}^+$, also called *leftmost-hard attention*, and

– avg-argmax$^*\colon \mathbb{R}^+ \to \mathbb{R}$, called *average-hard attention*.

Here, min-argmax$^*(x_1, \ldots, x_\ell) = (b_1, \ldots, b_\ell)$ where $b_i = 1$ iff $i = \min(\text{argmax}(x_1, \ldots, x_\ell))$. For example, min-argmax$^*(3,7,4,7) = (0,1,0,0)$. Moreover, avg-argmax* takes the average for all maximal elements, i.e., avg-argmax$^*(x_1, \ldots, x_\ell) = (b_1, \ldots, b_\ell)$ where

$$b_i = \frac{1}{|\text{argmax}(x_1, \ldots, x_\ell)|}$$

for all $i \in \text{argmax}(x_1, \ldots, x_\ell)$, and $b_i = 0$ for all other i. For example, avg-argmax$^*(3,7,4,7) = (0, 0.5, 0, 0.5)$.

The working principle of attention heads is illustrated in Figure 5.1.

In the following, we develop an attention head that will later serve as a building block of transformers (cf. Examples 5.2 and Examples 5.3).

Exercise 5.1: Attention Head Computing the Maximum

Define an attention head $\mathcal{H}_{\mathsf{max}} = (\mathbf{Q}, \mathbf{K}, \mathbf{V})$, using avg-argmax* as weight function and with parameters $n = 3$ and $n_{\mathsf{key}} = n_{\mathsf{value}} = 1$, such that $[\![\mathcal{H}_{\mathsf{max}}]\!] : (\mathbb{R}^3)^+ \times \mathbb{R}^3 \to \mathbb{R}$ where, for all $z_1, \ldots, z_\ell, x \in \mathbb{R}$,

$$[\![\mathcal{H}_{\mathsf{max}}]\!]((z_1, 0, 1)^\top \ldots (z_\ell, 0, 1)^\top, (x, 0, 1)^\top) = \max\{z_1, \ldots, z_\ell\}\,.$$

Solution:

Note that $dim(\mathcal{H}_{\mathsf{max}}) = (3, 1)$. The matrices can be chosen as follows:

$$\mathbf{Q} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \qquad\qquad \mathbf{K} = \mathbf{V} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

This actually also works with min-argmax* instead of avg-argmax*. An illustration of $\mathcal{H}_{\mathsf{max}}$ can be found in Figure 5.5.

Several attention heads can be combined to form layers.

Definition 5.2: Multi-Head Attention Layer

A *(multi-head) attention layer* $\mathscr{A} = (\mathcal{H}^{(1)}, \ldots, \mathcal{H}^{(d)}, \mathbf{W})$ has $d \geq 1$ attention heads $\mathcal{H}^{(i)} = (\mathbf{Q}^{(i)}, \mathbf{K}^{(i)}, \mathbf{V}^{(i)})$ and one additional linear transformation in terms of a matrix $\mathbf{W}$. We require that all attention heads share the same dimensions $n, n_{\mathsf{key}}, n_{\mathsf{value}} \in \mathbb{N}_+$ and that $\mathbf{W} \in \mathbb{Q}^{n \times (d \cdot n_{\mathsf{value}})}$.

We let $in(\mathscr{A}) = out(\mathscr{A}) = n$ and $dim(\mathscr{A}) = (n, n)$. We can assign to $\mathscr{A}$ three different semantics:

– The *(cross-)attention semantics* is given by

$$[\![\mathscr{A}]\!] : \begin{cases} (\mathbb{R}^n)^+ \times (\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+ \\ (w, \mathbf{x}^{(1)} \ldots \mathbf{x}^{(\ell)}) \mapsto \mathbf{y}^{(1)} \ldots \mathbf{y}^{(\ell)} \end{cases}$$

where

$$\mathbf{y}^{(i)} = \mathbf{W} \cdot (\mathcal{H}^{(1)}(w, \mathbf{x}^{(i)}) \boxminus \ldots \boxminus \mathcal{H}^{(d)}(w, \mathbf{x}^{(i)}))\,.$$

– The *self-attention semantics* is defined by

$$[\![\mathscr{A}]\!]_{\mathsf{self}} : \begin{cases} (\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+ \\ w \mapsto [\![\mathscr{A}]\!](w, w)\,. \end{cases}$$

– Finally, the *masked self-attention semantics* is defined by

$$[\![\mathscr{A}]\!]_{\mathsf{masked}} : \begin{cases} (\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+ \\ \mathbf{x}^{(1)} \ldots \mathbf{x}^{(\ell)} \mapsto \mathbf{y}^{(1)} \ldots \mathbf{y}^{(\ell)} \end{cases}$$

where (letting $w_i = \mathbf{x}^{(1)} \ldots \mathbf{x}^{(i)}$)

$$\mathbf{y}^{(i)} = \mathbf{W} \cdot (\mathcal{H}^{(1)}(w_i, \mathbf{x}^{(i)}) \boxminus \ldots \boxminus \mathcal{H}^{(d)}(w_i, \mathbf{x}^{(i)}))\,.$$

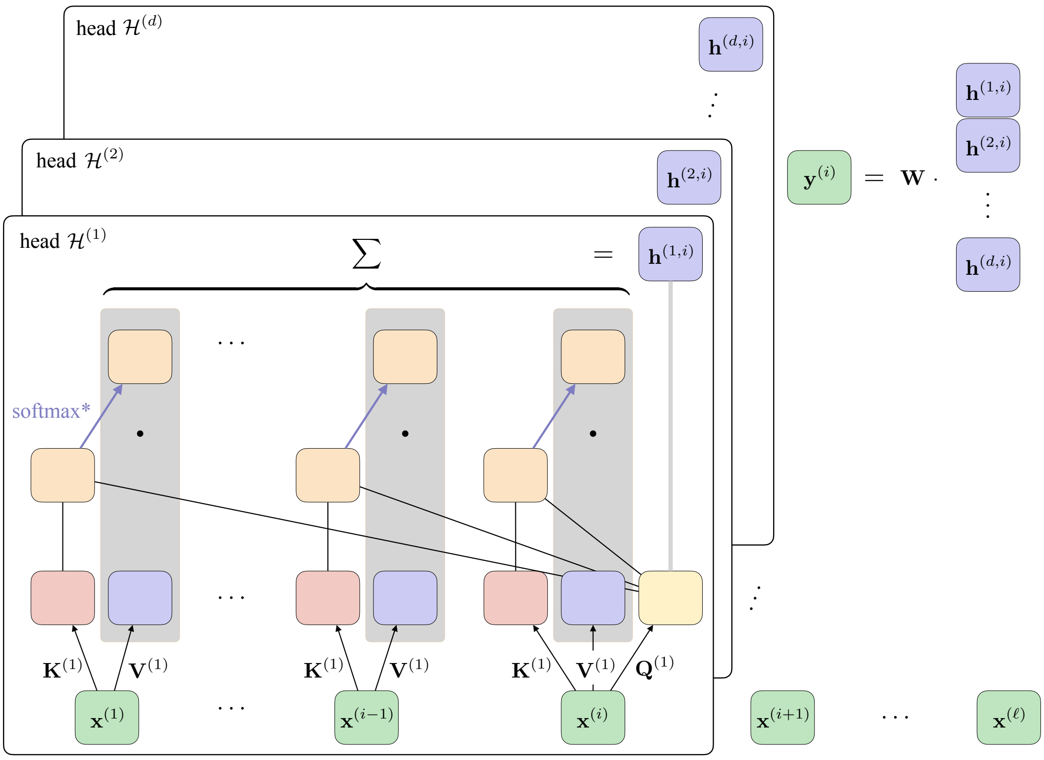


Figure 5.2: A multi-head attention layer with masked self-attention semantics

Note that $[\![\mathscr{A}]\!]_{\mathsf{self}}$ and $[\![\mathscr{A}]\!]_{\mathsf{masked}}$ are length-preserving, and $[\![\mathscr{A}]\!]$ is length-preserving in the second argument. Masked self-attention is illustrated in Figure 5.2.

Example 5.1: Attention Layer

We continue Exercise 5.1. Let $\mathcal{H}_{\mathsf{max}}$ be the attention head developed there. We obtain a (single-head) attention layer $\mathscr{A}_{\mathsf{max}}$ when we add the matrix $\mathbf{W}_{\mathsf{max}} = (0,1,0)^\top \in \mathbb{Q}^{3\times 1}$, which writes the result delivered by $\mathcal{H}_{\mathsf{max}}$ into the second component of the three-dimensional zero-vector. For an illustration, consider Figure 5.7.

5.2 The Transformer Architecture

Encoder Layer. An *encoder layer* is of the form $\mathcal{E} = (\mathscr{A}, \mathcal{N})$. It has two components, a multi-head attention layer $\mathscr{A}$ (with self-attention semantics) and a feed-forward neural network $\mathcal{N}$ with $dim(\mathscr{A}) = dim(\mathcal{N}) = (n,n)$ for some $n \in \mathbb{N}_+$.[1] Abusing notation, $[\![\mathcal{N}]\!]$ can be extended to a mapping $(\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+$ letting $[\![\mathcal{N}]\!](\mathbf{x}^{(1)}\dots\mathbf{x}^{(\ell)}) = \mathcal{N}(\mathbf{x}^{(1)})\dots\mathcal{N}(\mathbf{x}^{(\ell)})$. We can now define $[\![\mathcal{E}]\!] : (\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+$ by

$$[\![\mathcal{E}]\!](w) = Norm(\widehat{w} + [\![\mathcal{N}]\!](\widehat{w}))$$
$$\text{where} \quad \widehat{w} = Norm(w + [\![\mathscr{A}]\!]_{\mathsf{self}}(w))\,.$$

[1] Typically (but not mandatorily), $\mathcal{N}$ is a two-layer neural network with ReLU and id as activation functions, respectively.

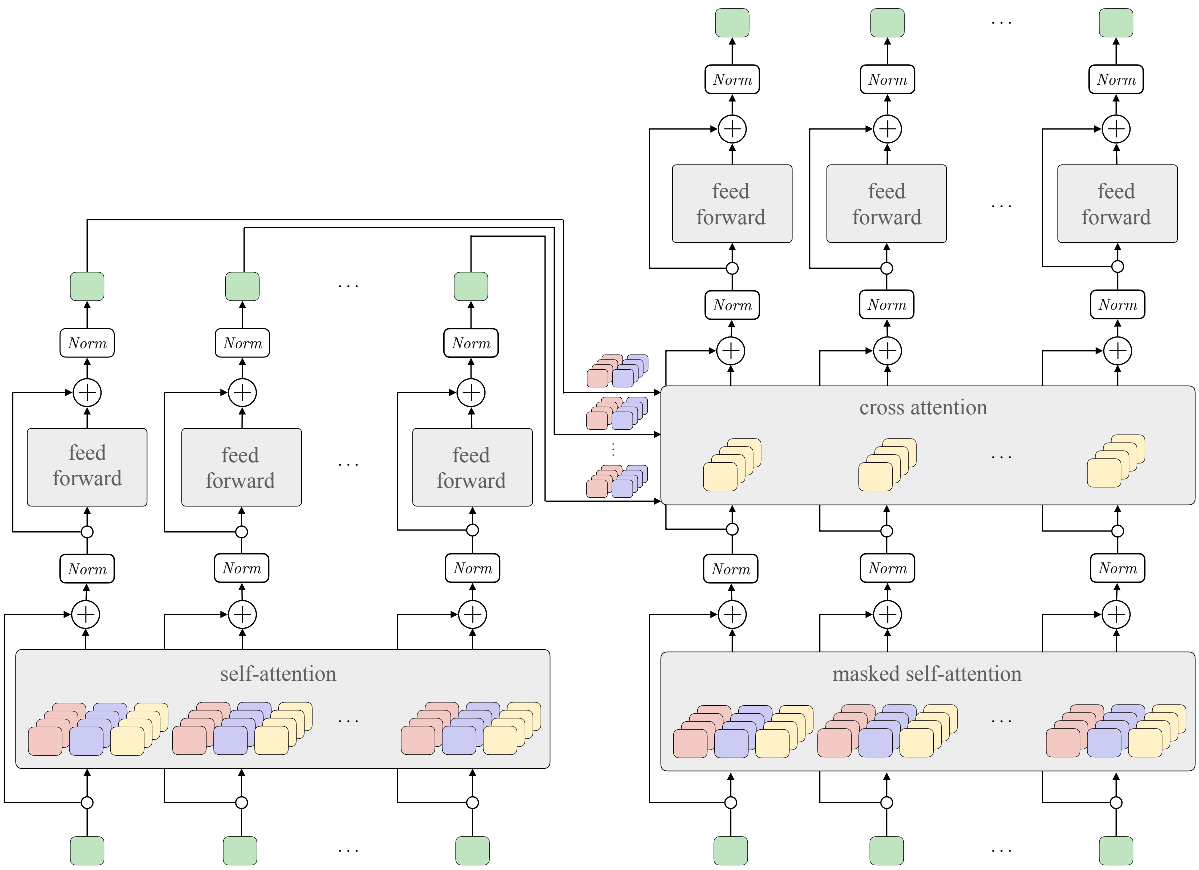


Figure 5.3: The interplay between an encoder layer (left) and a decoder layer (right)

Here, $\mathit{Norm} : (\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+$ is the length-preserving *layer norm*[2] and addition is position-wise (thus length preserving, too). Accordingly, we define $\mathit{dim}(\mathcal{E}) = (n, n)$.

The structure of an encoder layer is illustrated on the left-hand side of Figure 5.3.

Remark 5.1: Normalization and Addition

Optionally, the layer norm(s) may be chosen to be the identity function. Moreover, addition (also called residual connection) may be omitted. However, both greatly help in the training process of transformers. Moreover, in some applications one may consider adopting the masked self-attention semantics. That is, alternative semantics for encoder layer $\mathcal{E}$ can be given as

$$
\begin{aligned}
& [\![\mathcal{E}]\!] = [\![\mathcal{N}]\!] \circ [\![\mathscr{A}]\!]_{\mathsf{self}} \\
\text{or } & [\![\mathcal{E}]\!] = [\![\mathcal{N}]\!] \circ [\![\mathscr{A}]\!]_{\mathsf{masked}} .
\end{aligned}
$$

Decoder Layer. A decoder layer is similar to a encoder layer, but as part of its input comes from an encoder layer, its semantics is described in a different way. A *decoder layer* (of dimension n) is of the form $\mathcal{D} = (\mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \mathcal{N})$. It features two multi-head attention layers $\mathscr{A}^{(1)}$ and $\mathscr{A}^{(2)}$ and, like the encoder layer, a feed-forward neural network

[2] The layer norm usually includes learnable parameters, which we omit in the definition of $\mathcal{E}$ for simplicity.

$\mathcal{N}$ such that $dim(\mathscr{A}^{(1)}) = dim(\mathscr{A}^{(2)}) = dim(\mathcal{N}) = (n,n)$. Its semantics is a mapping $[\![\mathcal{D}]\!] : (\mathbb{R}^n)^+ \times (\mathbb{R}^n)^+ \to (\mathbb{R}^n)^+$ defined by

$$\begin{aligned}[\![\mathcal{D}]\!](w_{\mathsf{enc}}, w) &= Norm(w_2 + [\![\mathcal{N}]\!](w_2)) \\ \text{where}\quad w_2 &= Norm(w_1 + [\![\mathscr{A}^{(2)}]\!](w_{\mathsf{enc}}, w_1)) \\ w_1 &= Norm(w + [\![\mathscr{A}^{(1)}]\!]_{\mathsf{masked}}(w))\,.\end{aligned}$$

Thus, $\mathscr{A}^{(1)}$ is actually a masked multi-head attention layer, and $\mathscr{A}^{(2)}$ is a cross multi-head attention layer. We define $dim(\mathcal{E}) = (n,n)$. Note that $[\![\mathcal{D}]\!]$ is length-preserving in its second argument. The decoder layer is illustrated on the right-hand side of Figure 5.3.

Note that Remark 5.1 applies here as well, i.e., normalization and residual connections are optional. However, in NLP tasks, applying the masked-self attention semantics in decoder layers allows one to feed complete input and output sequences during training while avoiding that the decoder can "look into the future". In fact, its decisions should be based solely on what it has read/produced so far.

Transformer. Transformers were initially introduced for machine translation. For that case, we assume ordered finite alphabets Σ and Γ (of words or tokens). We assume that Γ contains a *start-of-sequence* symbol SOS and an *end-of-sequence* symbol EOS. They indicate when the translation of the output sentence will start and end, respectively. The semantics of a transformer $\mathcal{T}$ over Σ and Γ will define a (partial and not necessarily length-preserving) mapping

$$[\![\mathcal{T}]\!]_{\Sigma,\Gamma} : \Sigma^+ \to \Gamma^*\,.$$

We start with a length-preserving encoding of the input sequence, realized by an *embedding* $\mathsf{emb}_\Sigma : \Sigma^+ \to (\mathbb{Q}^n)^+$. It is obtained from a *word embedding*[3] $\mathsf{WE}_\Sigma : \Sigma \to \mathbb{Q}^n$ and a *positional encoding* $\mathsf{PE} : \mathbb{N}_+ \to \mathbb{Q}^n$ and defined, for $w = \alpha_1 \ldots \alpha_\ell \in \Sigma^+$ by $\mathsf{emb}_\Sigma(w) = \mathbf{x}^{(1)} \ldots \mathbf{x}^{(\ell)}$ where $\mathbf{x}^{(i)} = \mathsf{WE}(\alpha_i) + \mathsf{PE}(i)$. We include the mappings emb_Σ and emb_Γ in the definition of a transformer, as they are in principle learnable.

A transformer consists of a stack of encoder layers and a stack of decoder layers.[4]

Definition 5.3: Transformer

A *(machine-translation) transformer* with hidden-state dimension $n \in \mathbb{N}_+$ over Σ and Γ is a tuple

$$\mathcal{T} = (\mathsf{emb}_\Sigma, \mathsf{emb}_\Gamma, (\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(\kappa)}), (\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(\kappa)}), \mathcal{N}_{\mathsf{out}})$$

such that

- $\mathsf{emb}_\Sigma : \Sigma^+ \to (\mathbb{Q}^n)^+$ and $\mathsf{emb}_\Gamma : \Gamma^+ \to (\mathbb{Q}^n)^+$ are embeddings,
- $\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(\kappa)}$ are encoder layers with $dim(\mathcal{E}^{(i)}) = (n,n)$,
- $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(\kappa)}$ are decoder layers with $dim(\mathcal{D}^{(i)}) = (n,n)$,

[3] The one-hot encoding enc_Σ is a special case of a word embedding.

[4] RNNs and LSTMS can also be presented in that way, especially when we consider text generation. We will present transformers in their full form, but then focus on encoders and their capability as language recognizers.

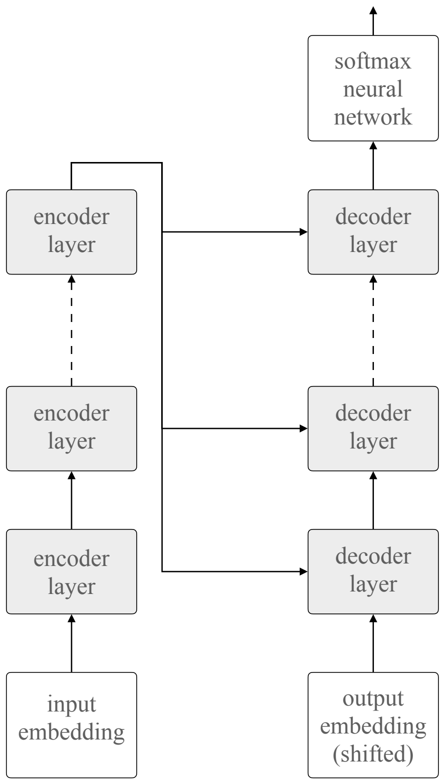


Figure 5.4: The interplay between encoder layers (left) and decoder layers (right)

– $\mathcal{N}_{\mathsf{out}}$ is a feed-forward neural network with a softmax activation function in its last layer and $dim(\mathcal{N}_{\mathsf{out}}) = (n, |\Gamma|)$.

Before defining $[\![\mathcal{T}]\!]_{\Sigma,\Gamma}$, we define an intermediate semantics $[\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma} : \Sigma^+ \times \Gamma^+ \to \Gamma$, which, for a given input sequence $w_{\mathsf{in}} \in \Sigma^+$ and an output sequence $w_{\mathsf{out}} \in \Gamma^+$ generated so far, provides the next output letter $[\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, w_{\mathsf{out}}) \in \Gamma$ to be appended to w_{out}. To determine $[\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, w_{\mathsf{out}})$, we compute

$$
\begin{aligned}
w^{(\kappa)}_{\mathsf{in}} &= [\![\mathcal{E}^{(\kappa)}]\!](w^{(\kappa-1)}_{\mathsf{in}}) & \qquad w^{(\kappa)}_{\mathsf{out}} &= [\![\mathcal{D}^{(\kappa)}]\!](w^{(\kappa)}_{\mathsf{in}}, w^{(\kappa-1)}_{\mathsf{out}}) \\
&\vdots & &\vdots \\
w^{(2)}_{\mathsf{in}} &= [\![\mathcal{E}^{(2)}]\!](w^{(1)}_{\mathsf{in}}) & w^{(2)}_{\mathsf{out}} &= [\![\mathcal{D}^{(2)}]\!](w^{(\kappa)}_{\mathsf{in}}, w^{(1)}_{\mathsf{out}}) \\
w^{(1)}_{\mathsf{in}} &= [\![\mathcal{E}^{(1)}]\!](w^{(0)}_{\mathsf{in}}) & w^{(1)}_{\mathsf{out}} &= [\![\mathcal{D}^{(1)}]\!](w^{(\kappa)}_{\mathsf{in}}, w^{(0)}_{\mathsf{out}}) \\
w^{(0)}_{\mathsf{in}} &= \mathsf{emb}_{\Sigma}(w_{\mathsf{in}}) & w^{(0)}_{\mathsf{out}} &= \mathsf{emb}_{\Gamma}(w_{\mathsf{out}})
\end{aligned}
$$

This interplay between encoder and decoder layers is illustrated in Figures 5.3 and 5.4.

With this, $[\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, w_{\mathsf{out}})$ is the $\min(\operatorname{argmax}(\mathcal{N}_{\mathsf{out}}(\mathbf{x})))$-th letter from Γ where $\mathbf{x}$ is the last vector in $w^{(\kappa)}_{\mathsf{out}}$. Now, SOS is a dummy symbol that allows the decoder to produce a first output. Thus, $[\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, \mathsf{SOS})$ generates the first letter after "reading" the input string w_{in}. Continuing this scheme, we let

$$\beta_1 = [\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, \mathsf{SOS})$$
$$\beta_2 = [\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, \mathsf{SOS}\,\beta_1)$$
$$\beta_3 = [\![\mathcal{T}]\!]^{\mathsf{next}}_{\Sigma,\Gamma}(w_{\mathsf{in}}, \mathsf{SOS}\,\beta_1\beta_2)$$
$$\vdots$$

Consider the smallest index $\ell \geq 1$ such that $\beta_{\ell+1} = \mathsf{EOS}$. If ℓ does not exist, $[\![\mathcal{T}]\!]_{\Sigma,\Gamma}(w_{\mathsf{in}})$ is undefined. Otherwise, $[\![\mathcal{T}]\!]_{\Sigma,\Gamma}(w_{\mathsf{in}}) = \beta_1 \ldots \beta_\ell$.

5.3 Encoder-Only Transformers

Based on the general transformer architecture, we now extract simple architectures, solely based on encoder layers, that define simpler functions or serve as language recognizer.

Henceforth, all encoder layers $\mathcal{E} = (\mathscr{A}, \mathcal{N})$ may or may not use *masked* self-attention. Also recall that residual connections and the normalization are optional in every layer.

Definition 5.4: Encoder-Only Transformer

An *encoder-only transformer* is a tuple

$$\mathcal{T} = (\mathcal{N}_{\mathsf{in}}, \mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(\kappa)}, \mathcal{N}_{\mathsf{out}})$$

where, for some $m, n, o \in \mathbb{N}_+$,

- $\mathcal{N}_{\mathsf{in}}$ and $\mathcal{N}_{\mathsf{out}}$ are feed-forward neural networks with $\mathit{dim}(\mathcal{N}_{\mathsf{in}}) = (m, n)$ and $\mathit{dim}(\mathcal{N}_{\mathsf{out}}) = (n, o)$, and
- $\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(\kappa)}$ are encoder layers with $\mathit{dim}(\mathcal{E}^{(i)}) = (n, n)$.

Similarly to RNNs, we let $\mathit{in}(\mathcal{T}) = m$, $\mathit{out}(\mathcal{T}) = o$, $\mathit{dim}(\mathcal{T}) = (m, o)$, and $\mathit{state}(\mathcal{T}) = n$. Analogously, we can now define a *length-preserving* mapping $[\![\mathcal{T}]\!] : (\mathbb{R}^m)^+ \to (\mathbb{R}^o)^+$ as the function composition

$$[\![\mathcal{T}]\!] = [\![\mathcal{N}_{\mathsf{out}}]\!] \circ [\![\mathcal{E}^{(\kappa)}]\!] \circ \ldots \circ [\![\mathcal{E}^{(1)}]\!] \circ [\![\mathcal{N}_{\mathsf{in}}]\!]$$

where, again, $[\![\mathcal{N}_{\mathsf{in}}]\!]$ and $[\![\mathcal{N}_{\mathsf{out}}]\!]$ are straightforwardly extended to sequences. We also define $\langle\!\langle \mathcal{T} \rangle\!\rangle : (\mathbb{R}^m)^+ \to \mathbb{R}^o$ such that $\langle\!\langle \mathcal{T} \rangle\!\rangle(w)$ returns the last vector in the sequence $[\![\mathcal{T}]\!](w)$.

Just as for RNNs, we can view an encoder-only transformer as a sequence classifier. Suppose $\mathit{dim}(\mathcal{T}) = (m, 1)$. Then, for $\bowtie \in \{\geq, >, =\}$ and $\theta \in \mathbb{R}$, we can define the language

$$L^{\bowtie\theta}(\mathcal{T}) = \{w \in (\mathbb{R}^m)^+ \mid \langle\!\langle \mathcal{T} \rangle\!\rangle(w) \bowtie \theta\}\,.$$

Again, we can adjust this definition to cope with languages over a finite alphabet Σ coming with a one-hot encoding enc_Σ. If $m = |\Sigma|$ and $\mathit{dim}(\mathcal{T}) = (m, 1)$, then we let

$$L^{\bowtie\theta}_{\Sigma}(\mathcal{T}) = \{w \in \Sigma^+ \mid \langle\!\langle \mathcal{T} \rangle\!\rangle(\mathit{enc}_\Sigma(w)) \bowtie \theta\}\,.$$

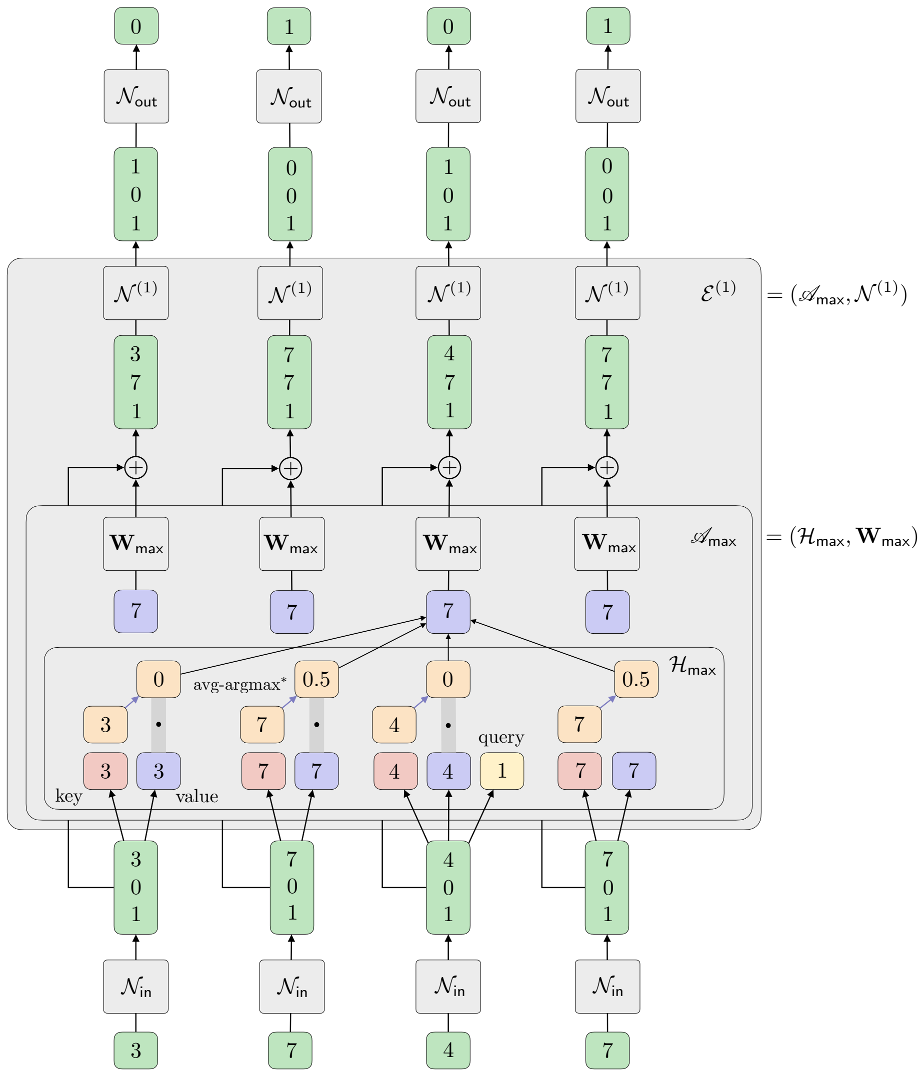


Figure 5.5: Transformer implementing argmax*

Example 5.2: Encoder-Only Transformer For argmax*

We continue Example 5.1. Consider the length-preserving mapping argmax* : $\mathbb{R}^+ \to \{0,1\}^+$ over arbitrarily long sequences of real numbers. We will define an encoder-only transformer $\mathcal{T}$ with $dim(\mathcal{T}) = (1,1)$ and $state(\mathcal{T}) = 3$ such that $[\![\mathcal{T}]\!] = \text{argmax}^*$. It is illustrated in Figure 5.5.

In a preprocessing step, we apply $\mathcal{N}_{\mathsf{in}}$ such that $\mathcal{N}_{\mathsf{in}}(x) = (x,0,1)^\top$. Next, we use self-attention in $\mathscr{A}_{\mathsf{max}}$. Thus, $\mathcal{H}$ outputs, at every position, the maximum number occurring in the sequence. Thanks to the residual connection, the ouput is added to the input vectors. It remains to identify the positions where the first two components are identical (those are the positions originally carrying the maximum number). This is taken care of by the neural network $\mathcal{N}^{(1)}$, which outputs 0 iff the first two components are equal. More precisely, for $\mathbf{x} = (x_1, x_2, x_3)^\top$,

$$[\![\mathcal{N}^{(1)}]\!](\mathbf{x}) = \begin{cases} (0,0,1)^\top & \text{if } x_1 = x_2 \\ (1,0,1)^\top & \text{if } x_1 \neq x_2 \end{cases}$$

Hereby, we leave the third component unchanged, as this will be useful in the subsequent example. The neural network $\mathcal{N}^{(1)}$ makes use of heaviside as (local) activation function, defined by

$$\text{heaviside}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0\,. \end{cases}$$

However, we could also use a combination of ReLU and σ, adjusting the interpretation of the output accordingly. Finally, $\mathcal{N}_{\mathsf{out}}$ inverts the first component. The specification of $\mathcal{N}_{\mathsf{in}}$, $\mathcal{N}^{(1)}$, and $\mathcal{N}_{\mathsf{out}}$ is left to the reader as an exercise.

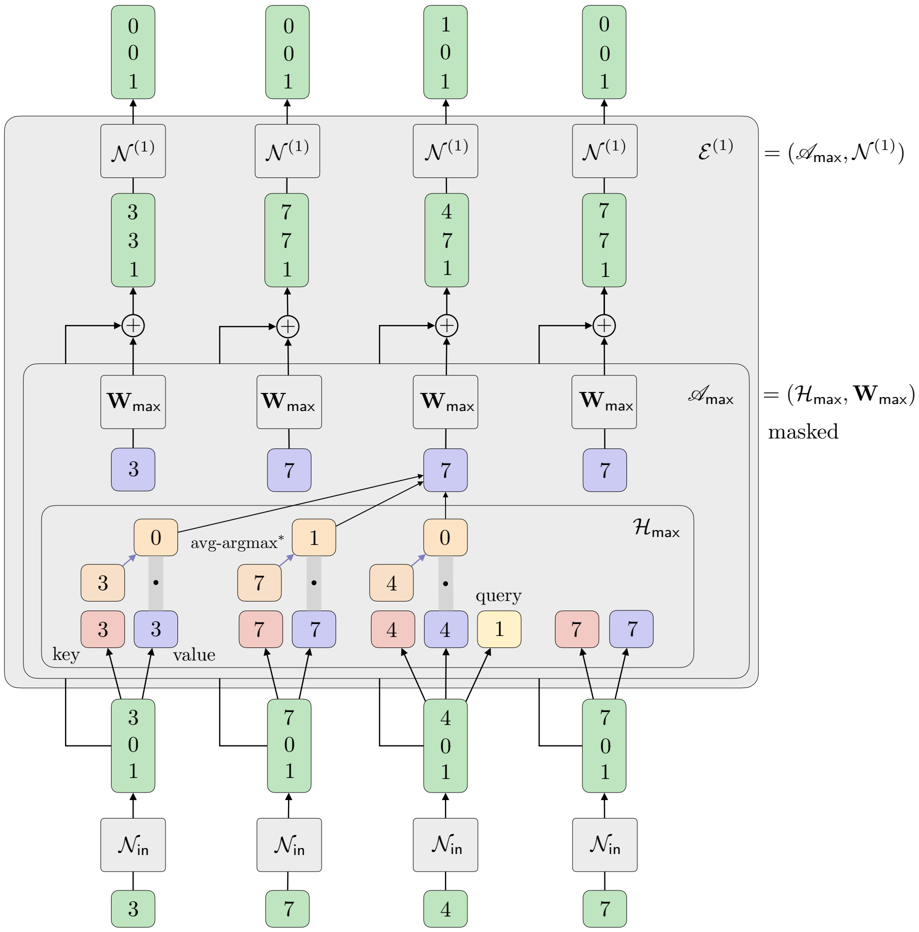


Figure 5.6: Transformer recognizing sorted sequences (lower part)

Example 5.3: Encoder-Only Transformer Recognizing Sorted Sequences

Again, we will build on Example 5.1. Our goal now is to model a function $f : \mathbb{R}^+ \to \{0, 1\}$ that returns 1 if the input sequence is sorted, and 0 otherwise. We will define an encoder-only transformer $\mathcal{T}$ with $dim(\mathcal{T}) = (1, 1)$ and $state(\mathcal{T}) = 3$ such that, for all $w \in \mathbb{R}^+$, the last element in the sequence $[\![\mathcal{T}]\!](w)$ is $f(w)$. In other words, $L^{=1}(\mathcal{T}) = L^{\geq 0.5}(\mathcal{T})$ is the set of sorted sequences over $\mathbb{R}$. The transformer $\mathcal{T} = (\mathcal{N}_{\mathsf{in}}, \mathcal{E}^{(1)}, \mathcal{E}^{(2)}, \mathcal{N}_{\mathsf{out}})$ is illustrated in Figures 5.6 and Figures 5.7.

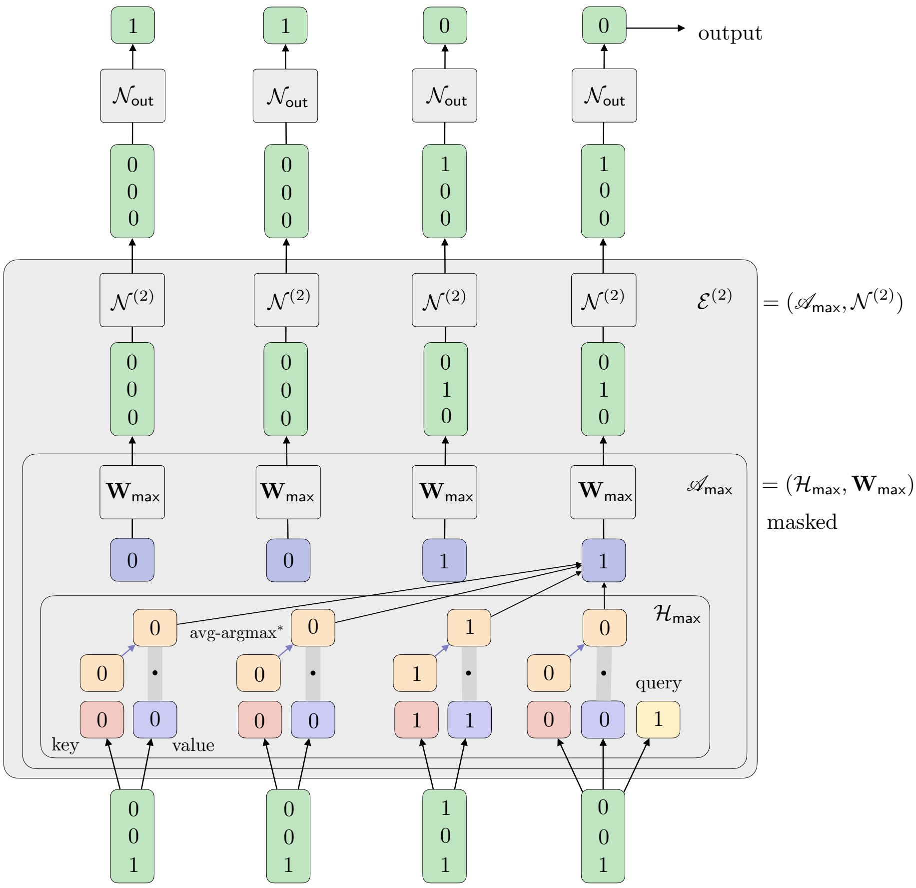


Figure 5.7: Transformer recognizing sorted sequences (upper part)

We use the very same components as in Example 5.2. However, we now adopt the *masked* self-attention semantics. In that case, every position i in the sequence can only "see" the previous positions (including i). Thus, at the last position, we use $\mathcal{H}_{\mathsf{max}}$ in the second encoder layer to detect whether *some* violation of the order has occurred. Note that the neural network $\mathcal{N}^{(2)}$ will just invert the first two components so that we can apply $\mathcal{N}_{\mathsf{out}}$ as in the previous case.

Note that the encoder layer $\mathcal{E}^{(2)}$ could also use the non-masked self-attention semantics, as we are only interested in the output at the very last position.

Example 5.4: Well-Formed Bracket Strings

The last example, which is due to [8], is concerned with the language L over the finite alphabet $\Sigma = \{\langle\,,\rangle\}$ given by the following grammar:

$$A ::= \langle\, A \,\rangle \;\mid\; AA \;\mid\; \varepsilon$$

Thus, L is the set of well-formed bracket strings. Figure 5.8 sketches an encoder-only transformer $\mathcal{T}$ with two encoder layers such that $L^{=0}(\mathcal{T}) = L$. Here, we assume $enc_\Sigma(\,\langle\,) = (1,0)^\top$ and $enc_\Sigma(\,\rangle\,) = (0,1)^\top$.

The idea is that the first component of a global state checks whether, in every prefix, there are at least as many opening as closing brackets. The second component checks whether, in the entire word, there are as many closing as opening brackets. A violation of the former property would result in some value 1 in the first component after the first encoder layer. If not violated, the original string is valid if the second component is zero, too. Zeroness in both cases is checked the the second encoder layer.

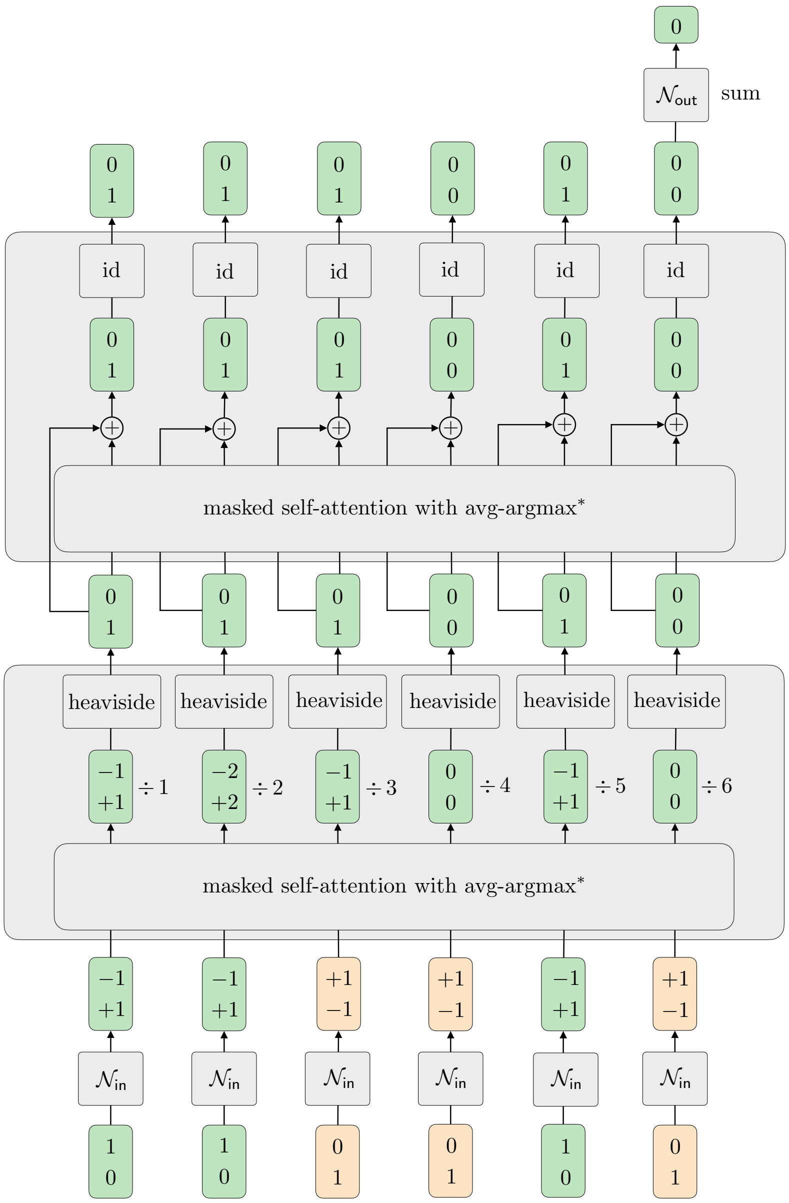


Figure 5.8: Transformer recognizing well-formed bracket strings

A series of recent papers established a rather complete picture of the language classes defined by transformer architectures, both in terms of logic and circuit complexity. Examples include [4, 5, 8, 35, 49]. To the best of our knowledge, only few works address the verification of transformers. Exceptions are [32, 47], which study robustness verification. General positive decidability results for transformers have yet to be explored. Due to

Turing completeness of the general architecture [40], the challenge relies in identifying architectures that allow for deciding interesiting properties. This represents a crucial area for future research, particularly in the context of verification. A related question is what a useful specification could be, for example in the spirit of NNL.

Bibliography

[1] The LASH toolset. `https://people.montefiore.uliege.be/boigelot/research/lash/`.

[2] Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. Verification of rnn-based neural agent-environment systems. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 6006–6013. AAAI Press, 2019.

[3] Aws Albarghouthi. *Introduction to Neural Network Verification.* verifieddeeplearning.com, 2021. `http://verifieddeeplearning.com`.

[4] Dana Angluin, David Chiang, and Andy Yang. Masked hard-attention transformers and boolean RASP recognize exactly the star-free languages. *CoRR*, abs/2310.13897, 2023.

[5] Pablo Barceló, Alexander Kozachinskiy, Anthony Widjaja Lin, and Vladimir V. Podolskii. Logical languages accepted by transformer encoders with hard attention. *CoRR*, abs/2310.03817, 2023.

[6] Bernd Becker, Christian Dax, Jochen Eisinger, and Felix Klaedtke. LIRA: Handling Constraints of Linear Arithmetics over the Integers and the Reals. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 307–310. Springer, 2007.

[7] Alberto Bertoni. The solution of problems relative to probabilistic automata in the frame of the formal languages theory. In Dirk Siefkes, editor, *GI - 4. Jahrestagung, Berlin, 9.-12. Oktober 1974*, volume 26 of *Lecture Notes in Computer Science*, pages 107–112. Springer, 1974.

[8] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 7096–7116. Association for Computational Linguistics, 2020.

[9] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real Variables. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 611–625. Springer, 2001.

[10] Benedikt Bollig, Martin Leucker, and Daniel Neider. A Survey of Model Learning Techniques for Recurrent Neural Networks. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 81–97. Springer, 2022.

[11] J. Richard Büchi. Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Logic, Methodology and Philosophy of Science*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966.

[12] George E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition - Preliminary Report. *SIGSAM Bull.*, 8(3):80–90, 1974.

[13] Antoine Durand-Gasselin and Peter Habermehl. Ehrenfeucht-Fraïssé goes elementarily automatic for structures of bounded degree. In Christoph Dürr and Thomas Wilke, editors, *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, volume 14 of *LIPIcs*, pages 242–253. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[14] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer, 1994.

[15] Jochen Eisinger. Upper Bounds on the Automata Size for Integer and Mixed Real and Integer Linear Arithmetic (Extended Abstract). In Michael Kaminski and Simone Martini, editors, *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2008.

[16] Jeffrey L. Elman. Finding structure in time. *Cogn. Sci.*, 14(2):179–211, 1990.

[17] Nathanaël Fijalkow. Undecidability results for probabilistic automata. *ACM SIGLOG News*, 4(4):10–17, 2017.

[18] Hugo Gimbert and Youssouf Oualhadj. Probabilistic automata on finite words: Decidable and undecidable problems. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 527–538. Springer, 2010.

[19] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[20] Christoph Haase. Approaching Arithmetic Theories with Finite-State Automata. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings*, volume 12038 of *Lecture Notes in Computer Science*, pages 33–43. Springer, 2020.

[21] Christoph Hertrich, Amitabh Basu, Marco Di Summa, and Martin Skutella. Towards Lower Bounds on the Depth of ReLU Neural Networks. *SIAM J. Discret. Math.*, 37(2):997–1029, 2023.

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

[23] Omri Isac, Yoni Zohar, Clark W. Barrett, and Guy Katz. DNN Verification, Reachability, and the Exponential Function Problem. In Guillermo A. Pérez and Jean-François Raskin, editors, *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*, volume 279 of *LIPIcs*, pages 26:1–26:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[24] Yuval Jacoby, Clark W. Barrett, and Guy Katz. Verifying recurrent neural networks using invariant inference. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2020.

[25] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, May 2015. Accessed: December 19, 2023.

[26] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[27] Igor Khmelnitsky, Daniel Neider, Rajarshi Roy, Xuan Xie, Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Martin Leucker, and Lina Ye. Analysis of recurrent neural networks via property-directed verification of surrogate models. *Int. J. Softw. Tools Technol. Transf.*, 25(3):341–354, 2023.

[28] Felix Klaedtke. Bounds on the automata size for Presburger arithmetic. *ACM Trans. Comput. Log.*, 9(2):11:1–11:34, 2008.

[29] Felix Klaedtke. Ehrenfeucht-Fraïssé goes automatic for real addition. *Inf. Comput.*, 208(11):1283–1295, 2010.

[30] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[31] Martin Leucker. Formal Verification of Neural Networks? In Gustavo Carvalho and Volker Stolz, editors, *Formal Methods: Foundations and Applications - 23rd Brazilian Symposium, SBMF 2020, Ouro Preto, Brazil, November 25-27, 2020, Proceedings*, volume 12475 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2020.

[32] Brian Hsuan-Cheng Liao, Chih-Hong Cheng, Hasan Esen, and Alois Knoll. Are transformers more robust? towards exact robustness verification for transformers.

In Jérémie Guiochet, Stefano Tonetta, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security - 42nd International Conference, SAFECOMP 2023, Toulouse, France, September 20-22, 2023, Proceedings*, volume 14181 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2023.

[33] Yang Liu, Jianpeng Zhang, Chao Gao, Jinghua Qu, and Lixin Ji. Natural-logarithm-rectified activation function in convolutional neural networks. *CoRR*, abs/1908.03682, 2019.

[34] Christof Löding. Efficient minimization of deterministic weak ω-automata. *Inf. Process. Lett.*, 79(3):105–109, 2001.

[35] William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth threshold circuits. *Trans. Assoc. Comput. Linguistics*, 10:843–856, 2022.

[36] William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. A formal hierarchy of RNN architectures. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 443–459. Association for Computational Linguistics, 2020.

[37] Marvin Lee Minsky. *Neural Nets and the Brain Model Problem.* PhD thesis, Princeton University, 1954.

[38] Izaak Neutelings. Neural network diagrams with tikz. `https://tikz.net/neural_networks/`, 2021.

[39] Azaria Paz. *Introduction to probabilistic automata.* Academic Press, 1971.

[40] Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *J. Mach. Learn. Res.*, 22:75:1–75:35, 2021.

[41] Michael O. Rabin. Probabilistic automata. *Inf. Control.*, 6(3):230–245, 1963.

[42] Wonryong Ryou, Jiayu Chen, Mislav Balunovic, Gagandeep Singh, Andrei Marian Dan, and Martin T. Vechev. Scalable polyhedral verification of recurrent neural networks. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 225–248. Springer, 2021.

[43] Marco Sälzer, Eric Alsmann, Florian Bruse, and Martin Lange. Verifying And Interpreting Neural Networks using Finite Automata. *CoRR*, abs/2211.01022, 2022.

[44] Marco Sälzer and Martin Lange. Reachability is NP-Complete Even for the Simplest Neural Networks. In Paul C. Bell, Patrick Totzke, and Igor Potapov, editors, *Reachability Problems - 15th International Conference, RP 2021, Liverpool, UK, October 25-27, 2021, Proceedings*, volume 13035 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2021.

[45] Marco Sälzer and Martin Lange. Reachability in simple neural networks. *Fundam. Informaticae*, 189(3-4):241–259, 2022.

[46] Roberto Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[47] Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. Robustness verification for transformers. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[48] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *J. Comput. Syst. Sci.*, 50(1):132–150, 1995.

[49] Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. Transformers as recognizers of formal languages: A survey on expressivity. *CoRR*, abs/2311.00208, 2023.

[50] Anej Svete and Ryan Cotterell. Recurrent neural language models as probabilistic finite-state automata. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 8069–8086. Association for Computational Linguistics, 2023.

[51] Alfred Tarski. A Decision Method for Elementary Algebra and Geometry. *RAND Corporation Paper*, 1948.

[52] Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 389–455. Springer, 1997.

[53] Hoang-Dung Tran, Sung Woo Choi, Xiaodong Yang, Tomoya Yamaguchi, Bardh Hoxha, and Danil V. Prokhorov. Verification of recurrent neural networks with star reachability. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2023, San Antonio, TX, USA, May 9-12, 2023*, pages 6:1–6:13. ACM, 2023.

[54] Caterina Urban and Antoine Miné. A Review of Formal Methods applied to Machine Learning. *CoRR*, abs/2104.02466, 2021.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[56] Adrian Wurm. Complexity of reachability problems in neural networks. In Olivier Bournez, Enrico Formenti, and Igor Potapov, editors, *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, volume 14235 of *Lecture Notes in Computer Science*, pages 15–27. Springer, 2023.

[57] Huan Zhang, Kaidi Xu, Shiqi Wang, and Cho-Jui Hsieh. Formal Verification of Deep Neural Networks: Theory and Practice. `https://neural-network-verification.com`, 2022.