Verification of Neural Networks

Lecture Notes (in progress)

Benedikt Bollig

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

December 20, 2023
# Contents

1. **Introduction**  
   4

2. **Preliminaries**  
   6
   2.1 Sets, Functions, Vectors, Matrices  
   6
   2.2 Languages and Büchi Automata  
   7

3. **Feed-Forward Neural Networks**  
   10
   3.1 Definition of Neural Networks  
   10
   3.2 A Specification Language for Neural Networks  
   14
   3.3 Proof of Decidability of LRA  
   23
   3.4 An Efficiently Solvable Fragment of NNL  
   31
   3.5 Beyond ReLU Neural Networks  
   37

4. **Recurrent Neural Networks**  
   41
   4.1 Definition and Semantics  
   41
   4.2 Undecidability of the Emptiness Problem  
   44
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".

Some of the neural network drawings are adopted from Izaak Neutelings [33].

Any comments, suggestions, errata, etc. are very welcome. Please send them by email to **ll**@lmf.cnrs.fr.
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 controller, 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 as "whenever a system variable $x$ reaches a critical threshold $\gamma$, 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 \[3, 7, 28, 46, 48\] 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 \( f\rvert_{A'} : A' \to B \) 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 \( x \in \mathbb{R}^n \) and \( i \in \{1, \ldots, n\} \), we let \( x_i \) refer to the \( i \)-th component of \( x \), i.e., \( x = (x_1, \ldots, x_n)^\top \). If \( n = 1 \), we may simply write \( x \) for \( x \). Similarly, \( y = (y_1, \ldots, y_n)^\top \), \( x' = (x'_1, \ldots, x'_n)^\top \), and so on. Moreover, for a matrix \( A \in \mathbb{R}^{n \times m} \), \( i \in \{1, \ldots, n\} \), and \( j \in \{1, \ldots, m\} \), we let \( a_{i,j} \) refer to the element of \( A \) in the \( i \)-th row and \( j \)-th column, i.e.,

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

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

For \( f = (f_1, \ldots, f_m) \) with \( f_1, \ldots, f_m : \mathbb{R} \to \mathbb{R} \) and \( x \in \mathbb{R}^m \), we define \( f(x) = (f_1(x_1), \ldots, f_m(x_m))^\top \in \mathbb{R}^m \).
The vertical concatenation of matrices \( A \in \mathbb{R}^{m \times n} \) and \( B \in \mathbb{R}^{k \times n} \) is defined by \( A \uplus B = C \) where \( c_{ij} = a_{ij} \) for all \( i \in \{1, \ldots, m\} \) and \( j \in \{1, \ldots, n\} \), and \( c_{ij} = b_{ij} \) for all \( i \in \{m+1, \ldots, m+k\} \) and \( j \in \{1, \ldots, n\} \). The horizontal concatenation of \( A \in \mathbb{R}^{m \times n} \) and \( B \in \mathbb{R}^{m \times k} \) is defined by \( A \oplus B = C \) where \( c_{ij} = a_{ij} \) for all \( i \in \{1, \ldots, m\} \) and \( j \in \{1, \ldots, n\} \), and \( c_{ij} = b_{ij} \) for all \( i \in \{1, \ldots, m\} \) and \( j \in \{n+1, \ldots, n+k\} \). The special case of vectors is defined analogously. In particular, the vertical concatenation of \( \mathbf{x} \in \mathbb{R}^m \) and \( \mathbf{y} \in \mathbb{R}^m \) is \( \mathbf{x} \uplus \mathbf{y} = (x_1, \ldots, x_m, y_1, \ldots, y_n)^T \in \mathbb{R}^{m+n} \).

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

\[
\argmax : \left\{ \mathbb{R}^m \to 2^{\{1, \ldots, m\}} \right\} : \mathbf{x} \mapsto \left\{ i \in \{1, \ldots, m\} \mid x_i = \max(x) \right\}.
\]

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

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

**Definition 2.1: Permutation Equivariance and Invariance**

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

1. 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 \( f(\mathbf{x}) = f(\pi(\mathbf{x})) \).
2. 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 \cite{17}. 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. Later on, these infinite words will represent real numbers.

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

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 \( \delta \) instead of \( \hat{\delta} \).

In the remainder of this section, we consider finite alphabets.
Definition 2.2: Büchi Automaton

Let $\Sigma$ be a finite alphabet. A Büchi automaton over $\Sigma$ is a tuple $A = (Q, \Delta, \iota, F)$ 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 $A$ recognizes a language $L(A) \subseteq \Sigma^\omega$ as follows. A run of $A$ 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 $\rho$ is defined as $\text{label}(\rho) = u_1 u_2 u_3 \ldots \in \Sigma^\omega$. Run $\rho$ 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 $A$ is defined by

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

Example 2.1:

The figure below depicts a Büchi automaton $A = (Q, \Delta, \iota, F)$ over the alphabet $\Sigma = \{0, 1\}$. We have $Q = \{q_0, q_1\}$, $\iota = q_0$, and $F = \{q_1\}$. The set of transitions $\Delta$ includes $(q_0, 0, q_0)$, $(q_0, 1, q_1)$, etc. The language $L(A)$ is the set of words from $\Sigma^\omega$ 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 $\Sigma$ be a finite alphabet and let $A, A_1, A_2$ be Büchi automata over $\Sigma$. We can effectively construct a Büchi automaton

(a) $A_1 \cup A_2$ over $\Sigma$ such that $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$;

(b) $A_1 \cap A_2$ over $\Sigma$ such that $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$;

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

Theorem 2.2: Büchi Automata Emptiness

The following problem is decidable:

**Input:** A finite alphabet $\Sigma$ and a Büchi automaton over $\Sigma$. 
Question: Is $L(A)$ nonempty?

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

For more background on languages and automata, we refer the reader to [44].
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 $x = (x_1, \ldots, x_m)^T \in \mathbb{R}^m$ to an output vector $y = (y_1, \ldots, y_n)^T \in \mathbb{R}^n$. 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.

Figure 3.1: A neural network layer with activation function $f : \mathbb{R}^n \to \mathbb{R}^n$
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

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 layer \( \mathcal{L} \) with input dimension \( m \) and output dimension \( n \) is a triple \( \mathcal{L} = (A, b, f) \) where \( A \in \mathbb{Q}^{n \times m} \) is the weight matrix, \( b \in \mathbb{Q}^n \) is the bias vector, and \( f : \mathbb{R}^n \to \mathbb{R}^n \) is the activation function.

We let \( \text{in}(\mathcal{L}) := m \) denote the input dimension of \( \mathcal{L} \). Moreover, \( \text{out}(\mathcal{L}) := n \) is the output dimension, which is usually referred to as the number of neurons of \( \mathcal{L} \). In other words, the weights of the \( i \)-th layer are given by the \( i \)-th row of \( A \). Layer \( \mathcal{L} \) defines the function

\[
[\mathcal{L}] : \begin{cases} 
\mathbb{R}^{\text{in}(\mathcal{L})} \to \mathbb{R}^{\text{out}(\mathcal{L})} \\
\mathbf{x} \mapsto f(A \cdot \mathbf{x} + b) 
\end{cases}
\]

Let us turn to neural networks. A (feed-forward) neural network is a sequence of 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 \( \mathcal{N} = (\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(\ell)}) \) of layers \( \mathcal{L}^{(k)} = (A^{(k)}, b^{(k)}, f^{(k)}) \) such that \( \text{out}(\mathcal{L}^{(k)}) = \text{in}(\mathcal{L}^{(k+1)}) \) 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 \( \mathcal{N} = (\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(\ell)}) \) has \( \ell \) layers, the “graph representation” exhibits \( \ell + 1 \) layers.
Similarly to a layer, \( \text{id} \) assuming that \( R \) activation function is the identity function, which we denote by \( \text{id} \):
\[
\text{where we always assume that } R \text{ in } [29]).
\]

Their extensions to functions are given in Figure 3.3 (the activation function NLReLU was recently studied following sense: There is \( L \) of this course). Often, a layer such as the choice of the loss function (a precise discussion is, however, not in the scope 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 \( L = (A, b, f) \) has a local activation function \( f \) in the following sense: There is \( g : R \to R \) such that, for all \( x \in R^n \) and \( i \in \{1, \ldots, n\} \), we have \( f(x) = (g(x_1), \ldots, g(x_n))^\top \). The definitions and graphs of some common local activation functions are given in Figure 3.3 (the activation function NLReLU was recently studied in [29]). Their extensions to \( R^n \to R^n \) are denoted in the same way (\( \sigma, \text{tanh}, \) 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 \( \text{id} : R^n \to R^n \) (again assuming that \( n \) is understood). An instance of a global activation function is
\[
\text{softmax} : \begin{cases} \quad R^n \to R^n \\ \quad x \mapsto (y_1, \ldots, y_n) \text{ where } y_i = \frac{e^{x_i}}{\sum_{j=1} e^{x_j}} \end{cases}
\]
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 \( N = (\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(\ell)}) \) with layers \( \mathcal{L}^{(k)} = (A^{(k)}, b^{(k)}, f^{(k)}) \) is called a ReLU neural network if, for all \( k \in \{1, \ldots, \ell\} \), we have \( f^{(k)} \in \{\text{id}, \text{ReLU}\} \).

**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
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 \( \mathbb{R}^m \to \mathbb{R}^n \), element \( \mathbf{x} \in \mathbb{R}^n \) is assigned category \( \text{argmax}(\mathcal{N}(\mathbf{x})) \in \{1, \ldots, n\} \). In the case of binary classification (i.e., in presence of two classes to choose from), it is also common to choose \( n = 1 \) and \( \sigma \) 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)

\[
\begin{align*}
\sigma(x) &= \frac{1}{1 + e^{-x}} \\
\text{tanh}(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}
\end{align*}
\]

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

(a) We have \( \mathcal{N} = (\mathcal{L}^{(1)}, \mathcal{L}^{(2)}) \) where \( \mathcal{L}^{(k)} = (\mathbf{A}^{(k)}, \mathbf{b}^{(k)}, f^{(k)}) \). Function \( f^{(1)} \) is the ReLU activation function and \( f^{(2)} \) is the identity. Moreover, we have
\[ b^{(1)} = (0, 0, 0)^\top, \ b^{(2)} = (0), \]

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

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 \( b^{(1)} = (0, 0, 0)^\top, b^{(2)} = (0.07), \)

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

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

The first neural network is taken from [18], 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) \( [\mathcal{N}] : \mathbb{R}^m \rightarrow \mathbb{R} \) computes the maximum function.

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

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

(d) \( [\mathcal{N}] : \mathbb{R}^m \rightarrow \mathbb{R}^m \) is permutation equivariant.
Figure 3.4: A convolutional neural network (CNN)

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

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

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

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.

For $x, y \in \mathbb{R}^m$, the Manhattan distance of $x$ and $x'$ is defined as $\|x, x'\|_{\text{Manhattan}} = \sum_{i=1}^{m} |x_i - x'_i|$.
(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 \( \varphi \), verify whether \( \varphi \) is a suitable specification for \( \mathcal{N} \) (syntactically) and, if so, whether \( \mathcal{N} \) satisfies \( \varphi \). 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:

\[
\begin{align*}
t & ::= a \cdot x \mid b \mid t + t \\
\varphi & ::= t \leq t \mid \neg \varphi \mid \varphi \lor \varphi \mid \exists x. \varphi
\end{align*}
\]

where \( x \in X \) and \( a, b \in \mathbb{Q} \).

An occurrence of a variable is free in an LRA formula \( \varphi \) if it is not in the scope of a quantifier \( \exists/\forall \). A variable is called free in \( \varphi \) if it has some free occurrence in \( \varphi \). Given a tuple of variables \( x = (x_1, \ldots, x_n) \), we may write \( \varphi(x) \) or \( \varphi(x_1, \ldots, x_n) \) for a formula \( \varphi \) whose free variables are among \( x_1, \ldots, x_n \) (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 \( t_1 \leq t_2 \land t_2 \leq t_1 \), \( t_1 < t_2 \) for \( t_1 \leq t_2 \land \neg(t_1 = t_2) \), \( \varphi \land \psi \) for \( \neg(\neg \varphi \lor \neg \psi) \Rightarrow \psi \), \( \varphi \lor \psi \) for \( \neg \varphi \land \neg \psi \Rightarrow \varphi \lor \psi \), \( \varphi \land \psi \) for \( \neg \varphi \lor \neg \psi \), etc. We will also write \( \exists x_1, \ldots, x_n. \varphi \) instead of \( \exists x_1. \exists x_2. \ldots \exists x_n. \varphi \) and so forth.

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

\[
\begin{align*}
\forall \ x \in y & \equiv \bigvee_{i=1}^{n} x = y_i \\
x \in y & \equiv \bigwedge_{i=1}^{n} y_i \leq x \\
x = \max(y) & \equiv x \in y \land \bigwedge_{i=1}^{n} y_i \leq x \\
\arg\max(y) = K & \equiv \bigwedge_{i \in K} y_i = \max(y) \land \bigwedge_{i \in \{1, \ldots, m\} \setminus K} \neg(y_i = \max(y)) \\
\arg\max(y) = \arg\max(y') & \equiv \bigvee_{K \subseteq \{1, \ldots, n\}} \left( \arg\max(y) = K \land \arg\max(y') = K \right)
\end{align*}
\]

The latter definition takes into account that \( \arg\max \) returns the set of indices carrying the maximal value in a tuple/vector.

Example 3.2:

Consider the formulas

\[
\begin{align*}
\varphi_1(x, y) &= x < y \Rightarrow \exists z . (x < z \land z < y) \\
\varphi_2(x, y) &= x < y \Rightarrow ((x < 0.5 \cdot x + 0.5 \cdot y) \land (0.5 \cdot x + 0.5 \cdot y < y))
\end{align*}
\]

Both, \( \varphi_1 \) and \( \varphi_2 \) 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 $\varphi$ 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 $\varphi$ are interpretation functions:

- $I |\models t_1 \leq t_2$ if $I(t_1) \leq I(t_2)$
- $I |\models \neg \varphi$ if $I |\not\models \varphi$
- $I |\models \varphi \lor \psi$ 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 $\varphi$ 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 $I |\models \varphi(r_1, \ldots, r_n)$ 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 $\varphi$ 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 $\varphi$ is **true**. In the latter case, $\varphi$ 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, $\forall x.\forall y.\exists z.(x < z \land z < y)$ is false.

**Definition 3.5: Satisfiability Problem**

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

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

**Question:** Is $\varphi$ satisfiable?

Note that free variables in the given formula $\varphi$ are implicitly interpreted as existential variables, as the question is whether there is a suitable interpretation function. Moreover, if $\varphi$ is a sentence, then the problem amounts to asking if $\varphi$ is true.
The definition of \( \text{SAT}(F) \) applies to all classes of formulas \( F \) that we consider in this lecture, as they will all be based on interpretation functions of the form \( I : X \to \mathbb{R} \).

**Theorem 3.1:**

The problem \( \text{SAT}(\text{LRA}) \) 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:

\[
\begin{align*}
  t & ::= a \cdot x \mid b \mid t + t \\
  \varphi & ::= t \leq t \mid \neg \varphi \mid \varphi \lor \varphi \mid \exists x.\varphi \mid \mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n)
\end{align*}
\]

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

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

\[
I \models \mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n) \text{ if } [\mathcal{N}](I(x_1), \ldots, I(x_m)) = (I(y_1), \ldots, I(y_n))
\]

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

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

Now, let us define some concrete NNL specifications:

**Exercise 3.4:**

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

- \( [\mathcal{N}] \) is surjective iff \( I \models \varphi_1[\mathcal{N}] \)

---

\[3\]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.
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

- \([\mathcal{N}]\) is injective iff \(\models \varphi_2[\mathcal{N}]\)

Solution:

- \(\varphi_1[\mathcal{N}] = \forall y_1, y_2. \exists x_1, x_2. \mathcal{N}(x_1, x_2) = (y_1, y_2)\)
- \(\varphi_2[\mathcal{N}] = \forall x_1, x_2, x'_1, x'_2, y_1, y_2. \left(\mathcal{N}(x_1, x_2) = (y_1, y_2) \wedge \mathcal{N}(x'_1, x'_2) = (y_1, y_2)\right) \Rightarrow x_1 = x'_1 \wedge x_2 = x'_2\)

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) \(\forall x_1, \ldots, x_m, y. (\mathcal{N}(x_1, \ldots, x_m) = y \Rightarrow y = \max(x_1, \ldots, x_m))\)

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

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

(d) \(\forall x_1, \ldots, x_m, y_1, \ldots, y_m. \left(\mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_m) \Rightarrow \bigwedge_{\pi \in S_m} \mathcal{N}(x_{\pi(1)}, \ldots, x_{\pi(m)}) = (y_{\pi(1)}, \ldots, y_{\pi(m)})\right)\)
An NNL formula $\varphi[N_1, \ldots, N_k]$ is considered to represent a neural network specification. Verifying $N_1, \ldots, N_k$ amounts to deciding whether $\varphi[N_1, \ldots, N_k]$ is satisfiable. We will assume that $N_1, \ldots, N_k$ are all ReLU neural networks, i.e., $\varphi \in \text{NNL}[\text{ReLU}]$. To establish decidability of verification, due to Theorem 3.1, it is then enough to show that $\varphi$ can be translated into an equivalent LRA sentence.

**Definition 3.7:**

Let $F_1$ and $F_2$ be classes of formulas (whose semantics depends on interpretation functions $I : \mathcal{X} \to \mathbb{R}$). We write $F_1 \leq F_2$ if there is an algorithm that translates every $\varphi(x_1, \ldots, x_n) \in F_1$ into $\tilde{\varphi}(x_1, \ldots, x_n) \in F_2$ such that, for all interpretation functions $I$, we have $I |\varphi$ iff $I |\tilde{\varphi}$.

If the translation can be done in polynomial time, then we write $F_1 \leq_{\text{poly}} F_2$.

**Proposition 3.1:**

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

**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 $\varphi_N(x_1, \ldots, x_m, y_1, \ldots, y_n)$ such that

\[ \text{for all } r_1, \ldots, r_m, s_1, \ldots, s_n \in \mathbb{R}, \text{ we have } \]  
\[ \mathcal{N}(r_1, \ldots, r_m) = (s_1, \ldots, s_n) \text{ iff } \models \varphi_N(r_1, \ldots, r_m, s_1, \ldots, s_n) \]  

(3.1)

We proceed by induction and first suppose that $\mathcal{N} = (\mathcal{L})$ has one layer $\mathcal{L} = (A, b, f)$.
If \( f = \text{id} \), we set
\[
\varphi_N(x_1, \ldots, x_m, y_1, \ldots, y_n) = \bigwedge_{i=1}^n y_i = b_i + \sum_{j=1}^m a_{i,j} \cdot x_j.
\]

If \( f = \text{ReLU} \), we set
\[
\varphi_N(x_1, \ldots, x_m, y_1, \ldots, y_n) = \bigwedge_{i=1}^n \exists z. \begin{cases} 
  z = b_i + \sum_{j=1}^m a_{i,j} \cdot x_j \\
  (z \leq 0 \land y_i = 0) \\
  (z > 0 \land y_i = z)
\end{cases}
\]

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

Now assume \( N = (L^{(1)}, \ldots, L^{(\ell)}, L^{(\ell+1)}) \) with \( \ell \geq 1 \). Let \( N_1 = (L^{(1)}, \ldots, L^{(\ell)}) \) and \( N_2 = (L^{(k+1)}) \) and assume we already have formulas \( \varphi_{N_1}(x_1, \ldots, x_m, z_1, \ldots, z_k) \) and \( \varphi_{N_2}(z_1, \ldots, z_k, y_1, \ldots, y_n) \) as required, where \( k = \text{out}(N_1) = \text{in}(N_2) \). Then, we set
\[
\varphi_N(x_1, \ldots, x_m, y_1, \ldots, y_n) = \exists z_1, \ldots, z_k. \left( \varphi_{N_1}(x_1, \ldots, x_m, z_1, \ldots, z_k) \land \varphi_{N_2}(z_1, \ldots, z_k, y_1, \ldots, y_n) \right)
\]

Indeed, for all \( r_1, \ldots, r_m, s_1, \ldots, s_n \in \mathbb{R} \) we have
\[
\llbracket N \rrbracket (r_1, \ldots, r_m) = (s_1, \ldots, s_n)
\]
iff \( \llbracket N_2 \rrbracket \circ \llbracket N_1 \rrbracket (r_1, \ldots, r_m) = (s_1, \ldots, s_n) \)
iff there are \( p_1, \ldots, p_k \in \mathbb{R} \):
\[
\left( \begin{array}{c}
\llbracket N_1 \rrbracket (r_1, \ldots, r_m) = (p_1, \ldots, p_k) \\
\llbracket N_2 \rrbracket (p_1, \ldots, p_k) = (s_1, \ldots, s_n)
\end{array} \right)
\]
iff there are \( p_1, \ldots, p_k \in \mathbb{R} \):
\[
\left( \begin{array}{c}
\models \varphi_{N_1}(r_1, \ldots, r_m, p_1, \ldots, p_k) \\
\models \varphi_{N_2}(p_1, \ldots, p_k, s_1, \ldots, s_n)
\end{array} \right)
\]
iff \( \models \varphi_N(r_1, \ldots, r_m, s_1, \ldots, s_n) \).

Now, to translate a given NNL formula \( \varphi \) into the LRA formula \( \hat{\varphi} \) as required, we replace, in \( \varphi \), every occurrence of an atomic subformula \( N(x_1, \ldots, x_m) = (y_1, \ldots, y_n) \) by \( \varphi_N(x_1, \ldots, x_m, y_1, \ldots, y_n) \).

**Exercise 3.7:**
For the NNL sentence \( \varphi \) for the maximum function, and the neural network \( N \) from Example 3.1(a), determine \( \hat{\varphi} \) 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 [17]. The automata-theoretic approach to deciding arithmetic theories goes back to Büchi [8]. Given an LRA formula $\varphi$, the idea is to construct a Büchi automaton $A_\varphi$ such that $\varphi$ is satisfiable iff the language of $A_\varphi$ is nonempty.

In the description below, we adopt several constructions and some notation from the paper [37], 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 $\Sigma^k$. We call a word $w \in \Sigma^\omega$ well-formed if it is of the form

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

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

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

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

Let $k \geq 0$. A word

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

may be seen as the $k$-tuple $(u_{1,1}u_{1,2}u_{1,3} \ldots, \ldots, u_{k,1}u_{k,2}u_{k,3} \ldots)$ of words over $\Sigma$. 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 $(w_1, \ldots, w_k) \in (\Sigma^k)^\omega$ of well-formed words encodes the tuple of real numbers $(\text{dec}(w_1), \ldots, \text{dec}(w_k)) \in \mathbb{R}^k$. 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 $\text{WF}^k$. A word from $(\Sigma^k)^\omega$ is well-formed if it is of the form

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

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 $\text{WF}^k \subseteq (\Sigma^\omega)^k$. For $w \in (\Sigma^\omega)^k$, we let $w_i$ refer to the $i$-th component of $w$, i.e., $w = (w_1, \ldots, w_k)$.
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

Figure 3.5: The Büchi automaton $\mathcal{A}_{wf}^k$

**Proposition 3.2:**

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

*Proof.* The automaton $\mathcal{A}_{wf}^k$ 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. \qed

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 $\Sigma^k$. We can effectively construct a Büchi automaton $\text{proj}_{\leq i}(\mathcal{A})$ over $\Sigma^i$ such that $L(\text{proj}_{\leq i}(\mathcal{A})) = \{(w_1,\ldots,w_i) \mid (w_1,\ldots,w_k) \in L(\mathcal{A})\}$.

*Proof.* Let $\mathcal{A} = (Q,\Delta,\iota,F)$ be a Büchi automaton over $\Sigma^k$. The Büchi automaton $\text{proj}_{\leq i}(\mathcal{A})$ 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 $\text{proj}_{\leq i}(\mathcal{A}) = (Q,\Delta',\iota,F)$ with

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

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

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 $\Sigma^k$ such that $L(\mathcal{A}) \subseteq \text{WF}^k$. We can construct a Büchi automaton $\text{cl}(\mathcal{A})$ over $\Sigma^k$ such that $L(\text{cl}(\mathcal{A}))$ is the least set satisfying the following:

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

24
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 [37].

**Proposition 3.5: Büchi Automata Constructions**

Let $k \geq 1$, $i,j,i_1,i_2 \in \{1, \ldots, k\}$, and $a,b \in \mathbb{Q}$. We can effectively construct automata $A_{i=j}^k$, $A_{i\leq j}^k$, $A_{i=\text{add}(i_1,i_2)}^k$, $A_{i=\text{mult}(a,j)}^k$, and $A_{i=\text{const}(b)}^k$ over $\Sigma^k$ such that

- $L(A_{i=j}^k) = \{ w \in WF^k \mid \text{dec}(w_i) = \text{dec}(w_j) \}$
- $L(A_{i\leq j}^k) = \{ w \in WF^k \mid \text{dec}(w_i) \leq \text{dec}(w_j) \}$
- $L(A_{i=\text{add}(i_1,i_2)}^k) = \{ w \in WF^k \mid \text{dec}(w_i) = \text{dec}(w_{i_1}) + \text{dec}(w_{i_2}) \}$
- $L(A_{i=\text{mult}(a,j)}^k) = \{ w \in WF^k \mid \text{dec}(w_i) = a \cdot \text{dec}(w_j) \}$
- $L(A_{i=\text{const}(b)}^k) = \{ w \in WF^k \mid \text{dec}(w_i) = b \}$

**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 $A_{i=2}^2$ 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 $1^\omega$. The general case $A_{i=j}^k$ 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 $B_{i=\text{add}(i_1,i_2)}^k$ as illustrated in Figure 3.7 for the special case $B_{3=\text{add}(1,2)}^3$ (the cases with different signs are similar, since $x_3 = x_1 - x_2 \text{ iff } x_1 = x_2 + x_3$ 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, $L(B_{i=\text{add}(i_1,i_2)}^k)$ does not contain all $w \in WF^k$ such that $\text{dec}(w_i) = \text{dec}(w_{i_1}) + \text{dec}(w_{i_2})$. For example, $L(B_{3=\text{add}(1,2)}^3)$ does not contain the word

\[
\begin{bmatrix}
+ & 0 & 0 & \bullet & 1 & 1 & 1 & 1 \\
+ & 1 & 0 & \bullet & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

but instead

\[
\begin{bmatrix}
+ & 0 & 0 & \bullet & 1 & 1 & 1 & 1 \\
+ & 0 & 1 & \bullet & 1 & 1 & 1 & 1 \\
\end{bmatrix}
\]
We include all admissible triples using an additional tape and the equality automaton defining

\[ \mathcal{A}_k^{\text{add}(i_1,i_2)} := \text{cl}(\text{proj}_{\leq k}(B_{k+1}^{k+1} \cap \mathcal{A}_{k+1}^{k+1} \cap \mathcal{A}_{i_{k+1}}^{k+1})). \]

**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

\[ \mathcal{A}_i^{\text{mult}(2,j)} := \text{cl}(\text{proj}_{\leq k}(A_{k+1}^{k+1} \cap \mathcal{A}_{i_{k+1}}^{k+1})). \]

Now suppose \( a \) is an integer such that \( a \geq 3 \) (negative constants are handled similarly). Let \( u_{n-1} \ldots u_0 \in \{0, 1\}^n \) be the binary representation of \( a \) of minimal length, i.e., \( a = \text{dec}(+u_{n-1} \ldots u_0 \cdot 000 \ldots) \). Furthermore, let \( K = \{i_1, \ldots, i_d\} \subseteq \{0, \ldots, n-1\} \) be the set of indices \( \ell \) 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

\[ \mathcal{A}_i^{\text{mult}(a,j)} := \text{cl}(\text{proj}_{\leq k}(A_{k+1}^{k+n} \cap \bigcap_{\ell=1}^{n-1} A_{k+1+\ell=\text{mult}(2,k+\ell)}^{k+n} \cap \mathcal{A}_{i_{k+1+i_1+\ldots+i_d}}^{k+n})). \]
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 $n \geq 3$,

$$A_{i=\text{add}(i_1, \ldots, i_n)}^k := cl(proj_{\leq k}(A_{k+1=\text{add}(i_1, \ldots, i_{n-1})}^k \cap A_{i=\text{add}(i_n, k+1)}))$$

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

$$A_{i=\text{mult}(a, j)}^k := cl(proj_{\leq k}(A_{k+1=\text{mult}(i, n)}^{k+2} \cap A_{k+2=\text{mult}(m, j)}^k \cap A_{k+1=\text{k+2}})).$$

The remaining constructions are left as an exercise. \qed

**Exercise 3.9:**

Determine the Büchi automata $A_{i\leq j}^k$ and $A_{i=\text{const}(b)}^k$ (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 non-emptiness. 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

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

where $\theta_1, \ldots, \theta_m \in \{\exists, \forall\}$, the $x_1, \ldots, x_m$ are pairwise distinct, and $\Phi$ is quantifier-free. For background on the Prenex normal form, we refer the reader to [11]. Let $t_1, \ldots, t_n$ be
all distinct occurrences of terms in \( \Phi \), including \( x_1, \ldots, x_m \). That is, \( m \leq n \) and \( t_i = x_i \) for all \( i \in \{1, \ldots, m\} \).

With \( w = (w_1, \ldots, w_m, w_{m+1}, \ldots, w_n) \in \text{WF}^n \), we associate an arbitrary interpretation function \( I_w \) such that \( I_w(t_i) = \text{dec}(w_i) \) for all \( i \in \{1, \ldots, m\} \).

**Lemma 3.1:**
We can construct a Büchi automaton \( A_{\text{term}} \) over \( \Sigma^n \) such that
\[
L(A_{\text{term}}) = \{ w \in \text{WF}^n \mid I_w(t_i) = \text{dec}(w_i) \text{ for all } i \in \{m+1, \ldots, n\} \}.
\]

**Proof.** We set
\[
A_{\text{term}} := \bigcap_{i=m+1}^n B_i^n
\]
where
\[
B_i^n = \begin{cases}
A_i^n = \text{add}(j,k) & \text{if } t_i = t_j + t_k \\
A_i^n = \text{mult}(a,j) & \text{if } t_i = a \cdot x_j \\
A_i^n = \text{const}(b) & \text{if } t_i = b
\end{cases}
\]
with \( a, b \in \mathbb{Q} \). Correctness follows from Proposition 3.5 by induction. \( \square \)

**Example 3.4:**
Consider the sentence
\[
\Psi = \neg \exists x_1 \exists x_2. \left( x_1 \leq x_2 \land \neg((x_1 \leq 0.5 \cdot x_1 + 0.5 \cdot x_2) \land (0.5 \cdot x_1 + 0.5 \cdot x_2 \leq x_2)) \right)
\]
\( \Phi(x_1, x_2) \)
in Prenex normal form. The terms occurring in \( \Phi \) 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
\[
\begin{array}{cccccccc}
x_1 & + & 0 & 1 & \bullet & 0 & 0 & 0 \ 
x_2 & + & 0 & 0 & \bullet & 0 & 0 & 0 \ 0.5 \cdot x_1 & + & 0 & 0 & \bullet & 1 & 0 & 0 \ 0.5 \cdot x_2 & + & 0 & 0 & \bullet & 1 & 0 & 0 \ 0.5 \cdot x_1 + 0.5 \cdot x_2 & + & 0 & 0 & \bullet & 1 & 0 & 0 \\
\end{array}
\]
Using \( A_{\text{term}} \), we now proceed by induction to transform any subformula \( \varphi(x_1, \ldots, x_m) \) of \( \Phi \) (including \( \Phi \)) into a Büchi automaton \( B_\varphi \) over \( \Sigma^n \) such that, for all \( w \in L(A_{\text{term}}) \),
\[
w \in L(B_\varphi) \text{ iff } I_w \models \varphi(x_1, \ldots, x_m) . \quad (3.2)
\]

**Term Comparison.** Suppose \( \varphi(x_1, \ldots, x_m) = (t_i \leq t_j) \). We set
\[
B_{t_i \leq t_j} = A_i^{t_j}.
\]

\footnote{Strictly speaking, a variable is not a term.}
Let \( w = (w_1, \ldots, w_n) \in L(\mathcal{A}_{\text{term}}) \). We will show (3.2).

Suppose \( w \in L(B_\varphi) \). Since \( w \in L(\mathcal{A}_{\text{term}}) \), we have \( I_w(t_i) = \text{dec}(w_i) \) and \( I_w(t_j) = \text{dec}(w_j) \).

By \( w \in L(A^n_{\leq j}) \) and Proposition 3.5, we have \( \text{dec}(w_i) \leq \text{dec}(w_j) \). Thus, \( I_w(t_i) \leq I_w(t_j) \), which implies \( I_w \models t_i \leq t_j \).

Conversely, suppose \( I_w \models t_i \leq t_j \). Then, \( I_w(t_i) \leq I_w(t_j) \). Since \( w \in L(\mathcal{A}_{\text{term}}) \), we also have \( \text{dec}(w_i) \leq \text{dec}(w_j) \), which implies \( w \in L(A^n_{\leq j}) \). We conclude \( w \in L(B_{t_i \leq t_j}) \).

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

\[
B_{\varphi_1 \lor \varphi_2} = B_{\varphi_1} \cup B_{\varphi_2}.
\]

To show (3.2), let \( w \in L(\mathcal{A}_{\text{term}}) \). We have \( w \in L(B_{\varphi_1 \lor \varphi_2}) \) iff \( w \in L(B_{\varphi_1}) \cup L(B_{\varphi_2}) \) iff (by induction hypothesis) \( I_w \models \varphi_1 \lor \varphi_2 \).

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

\[
B_{\neg \psi} = \overline{B_\psi}.
\]

Let us show (3.2). For \( w \in L(\mathcal{A}_{\text{term}}) \), we have \( w \in L(B_{\neg \psi}) \) iff \( w \not\in L(B_\psi) \) iff (by induction hypothesis) \( I_w \not\models \psi \) iff \( I_w \models \neg \psi \).

We thus hold a Büchi automaton \( B_\varphi \) over \( \Sigma^n \) such that, for all \( w \in L(\mathcal{A}_{\text{term}}) \),

\[
w \in L(B_\varphi) \text{ iff } I_w \models \Phi(x_1, \ldots, x_m).
\]

Consider the Büchi automaton \( \mathcal{A}_\varphi = \text{cl}(\text{proj}_{\leq m}(B_\varphi \cap \mathcal{A}_{\text{term}})). \)

**Lemma 3.2:**

We have

\[
L(\mathcal{A}_\varphi) = \{ w \in \text{WF}^m \mid I_w \models \Phi(x_1, \ldots, x_m) \}.
\]

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

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

\begin{itemize}
  \item for all \( i \in \{1, \ldots, m\} \), \( w'_i \) equals \( w_i \) but possibly has extra leading zeros,
  \item for all \( i \in \{m+1, \ldots, n\} \), \( \text{dec}(w'_i) = I_w(t_i) \).
\end{itemize}

Then, we have \( w' \in L(\mathcal{A}_{\text{term}}) \) and, since \( I_{w'} \models \Phi \), \( w' \in L(B_\varphi) \). Altogether, we have \( (w'_1, \ldots, w'_m) \in \text{proj}_{\leq m}(B_\varphi \cap \mathcal{A}_{\text{term}}) \) and, finally, \( w = (w_1, \ldots, w_m) \in L(\mathcal{A}_\varphi). \)
Towards a Büchi automaton for $\Psi = \theta_1 x_1 \ldots \theta_m x_m \Phi(x_1, \ldots, x_m)$ where $\theta_i \in \{\exists, \neg \exists\}$, we add quantifier (and so eliminate free variables) by induction: For $i = m, \ldots, 0$ (the number of free variables yet to be eliminated), letting

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

we build a Büchi automaton $A_{\Psi_i}$ over $\Sigma^i$ such that

$$L(A_{\Psi_i}) = \{ w \in \text{WF}^i \mid I_w \models \Psi_i(x_1, \ldots, x_i) \}. \quad (3.3)$$

Note that $A_{\Psi_m} = A_{\Psi}$. Now, let $i = m - 1, \ldots, 0$. Towards $A_{\Psi_i}$, we will have to eliminate free variable $x_{i+1}$.

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

$$A_{\Psi_i} = cl(proj_{\leq i}(A_{\Psi_{i+1}})).$$

We show (3.3). We certainly have $A_{\Psi_i} \subseteq \text{WF}^i$. Let $w = (w_1, \ldots, w_i) \in \text{WF}^i$.

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

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

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

$$A_{\Psi_i} = cl(proj_{\leq i}(A_{\Psi_{i+1}})) \cap A_{\Psi_i}^\text{nf}.\quad$$

That is, we first apply projection and closure as in the previous case, then complement the automaton, and finally intersect it with $A_{\Psi_i}^\text{nf}$. By the preceding discussion, we have

$$w \in L(A_{\Psi_i})$$

iff $w \in \text{WF}^i$ and $I_w \not\models \exists x_{i+1}.\Psi_{i+1}(x_1, \ldots, x_{i+1})$

iff $w \in \text{WF}^i$ and $I_w \models \neg \exists x_{i+1}.\Psi_{i+1}(x_1, \ldots, x_{i+1})$

$$= \Psi_i.$$

To wrap up, we obtain the Büchi automaton $A_{\Psi_0}$ over $\Sigma^0$ (a singleton alphabet) such that $L(A_{\Psi_0}) \neq \emptyset$ 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 $\text{FO}(\mathbb{R}, +, <)$. Tarski showed that even $\text{FO}(\mathbb{R}, +, \cdot, <)$, i.e., non-linear real arithmetic (including multiplication), is decidable [43]. While his algorithm was nonelementary, $\text{FO}(\mathbb{R}, +, \cdot, <)$ was shown to
be solvable in doubly exponential time \cite{9}. The automata-based decidability proof for \( \text{FO}(\mathbb{R}, +, <) \) presented in this section can be easily extended to \( \text{FO}(\mathbb{R}, +, <, \mathbb{Z}) \), which has an additional unary predicate for the integers \( \mathbb{Z} \). Another well-known decidable theory is \( \text{FO}(\mathbb{N}, +, <) \), also known as Presburger arithmetic. On the other hand, \( \text{FO}(\mathbb{N}, +, \cdot, <) \), i.e., Peano arithmetic, is undecidable due to Gödel \cite{16}.

The automata-theoretic approach to deciding \( \text{FO}(\mathbb{R}, +, <, \mathbb{Z}) \) is due to Büchi \cite{8}. 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 \( \text{FO}(\mathbb{R}, +, <, \mathbb{Z}) \)-formula are recognized by a \emph{weak} deterministic Büchi automaton, in which every strongly connected component contains either only final states or only non-final states \cite{6}. Löding showed that minimization of these automata can be reduced, in linear time, to the minimization of DFAs \cite{30}. Moreover, a doubly and triply exponential upper bound on the size of minimal weak deterministic Büchi automata for formulas from \( \text{FO}(\mathbb{R}, +, <) \) and \( \text{FO}(\mathbb{R}, +, <, \mathbb{Z}) \) were shown by Klaedtke \cite{26} and, respectively, Eisinger \cite{12}. For Presburger arithmetic and finite automata, a triply exponential upper bound is due to Klaedtke \cite{25}. These (finite) automata can even be computed in triply exponential time, as was shown by Durand-Gasselin and Habermehl \cite{10}. 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 \cite{1,4,6}.

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 \( \sigma \) and \( \tanh \) \cite{20}.

### 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 \( \varphi_1 \) has one quantifier alternation, \( \varphi_2 \) is the negation of an existential formula:

\[
\varphi_2[\mathcal{N}] = \neg \exists x_1, x_2, x'_1, x'_2, y_1, y_2. \left( \begin{array}{l}
\mathcal{N}(x_1, x_2) = (y_1, y_2) \\
\mathcal{N}(x'_1, x'_2) = (y_1, y_2) \\
\neg(x_1 = x'_1) \vee \neg(x_2 = x'_2)
\end{array} \right)
\]

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:

\[
\begin{align*}
  t &::= a \cdot x \mid b \mid t + t \\
  \varphi &::= t \leq t \mid t < t \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \exists x. \varphi \\
  \mathcal{N}(x_1, \ldots, x_m) &= (y_1, \ldots, y_n) \mid \neg (\mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n))
\end{align*}
\]

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 \( \neg (\mathcal{N}(x_1, \ldots, x_m) = (y_1, \ldots, y_n)) \), which are otherwise no longer expressible. We also use the abbreviations \( t_1 = t_2 \equiv t_1 \leq t_2 \land t_2 \leq t_1 \) and \( t_1 \neq t_2 \equiv t_1 < t_2 \lor t_2 < t_1 \). 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:

\[
\begin{align*}
  t &::= a \cdot x \mid b \mid t + t \\
  \varphi &::= t \leq t \mid t < t \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \exists x. \varphi
\end{align*}
\]

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 \( \text{SAT}(∃LRA) \) 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 [27] 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 $\text{SAT}(\exists\text{NNL}[\text{ReLU}])$ is NP-complete.

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

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

Lemma 3.3:  
We have $\exists\text{NNL}[\text{ReLU}] \leq_{\text{poly}} \exists\text{LRA}$.

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 $\exists\text{LRA}$ formulas $\varphi_N(x_1, \ldots, x_m, y_1, \ldots, y_n)$ and $\varphi'_N(x_1, \ldots, x_m, y_1, \ldots, y_n)$ 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 $\models \varphi_N(r_1, \ldots, r_m, s_1, \ldots, s_n)$
- $[\mathcal{N}](r_1, \ldots, r_m) \neq (s_1, \ldots, s_n)$ iff $\models \varphi'_N(r_1, \ldots, r_m, s_1, \ldots, s_n)$

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

$$\varphi'_N(x_1, \ldots, x_m, y_1, \ldots, y_n) = \exists z_1, \ldots, z_n. \left( \varphi_N(x_1, \ldots, x_m, z_1, \ldots, z_n) \land \bigwedge_{i=1}^n y_i \neq z_i \right),$$

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

We will now present a stronger lower bound, due to [23, 38, 39], for the reachability fragment of $\exists\text{NNL}[\text{ReLU}]$:

Definition 3.10: Reachability Formulas

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

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

where $\varphi_1$ and $\varphi_2$ are quantifier-free formulas generated by the grammar

\[ t ::= a \cdot x \mid b \mid t + t \]
\[ \varphi ::= t \leq t \mid \varphi \land \varphi \]

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 [38, 39]

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

Proof. We follow [38, 39] 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

\[
C_1 = (X_1 \lor X_2 \lor X_3) \land (\neg X_1 \lor X_2 \lor \neg X_3) \land (\neg X_2 \lor X_3 \lor X_4).
\]

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

We will first build a neural network \( N \) such that

\[
[N]_{\{0,1\}^m} : \begin{cases} 
\{0,1\}^m \to \mathbb{R} \\
(r_1, \ldots, r_m) \mapsto \text{val}_r(C_1) + \ldots + \text{val}_r(C_k)
\end{cases}
\]

where \( \text{val}_r(X_i) = r_i \) for all \( i \in \{1, \ldots, m\} \). The neural network \( N \) 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

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

It follows that

\[
\Phi \text{ is satisfiable iff } \begin{cases} 
N(x_1, \ldots, x_m) = y \\
\land \bigwedge_{i=1}^{m} (x_i = 0 \lor x_i = 1) \\
\land y = k
\end{cases}
\]

is satisfiable.

However, the latter is not a REACH formula (there cannot be a suitable REACH formula, as \( \{0,1\}^m \) is not a hyperrectangle). So we add some more neurons to \( N \) 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 \( N' \) with \( \text{in}(N') = m \) and \( \text{out}(N') = m + 1 \).

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

\[
\text{bool}(r) := \text{ReLU}(r - 0.5) + \text{ReLU}(0.5 - r) - 0.5 = \begin{cases} 
r - 1 & \text{if } r \geq 0.5 \\
-r & \text{if } r < 0.5
\end{cases}
\]

34
In particular, we have $\text{bool}(r) = 0$ iff $r \in \{0, 1\}$. Therefore,

\[
\Phi \text{ is satisfiable \ iff } \left( \bigwedge_{i=1}^{m} z_i = 0 \right) \text{ is satisfiable.}
\]

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**  

NP-hardness from Theorem 3.5 also holds for ReLU neural networks such that all but the last layers are ReLU layers.
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

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

\[
\left(b + \sum_{i=1}^{m} a_i x_i\right) \cdot a_j' = \text{ReLU} \left(b + \sum_{i=1}^{m} a_i x_i\right) \cdot a_j' + \text{ReLU} \left(- \left(b + \sum_{i=1}^{m} a_i x_i\right)\right) \cdot (-a_j') = \text{ReLU} \left(b + \sum_{i=1}^{m} a_i x_i\right) \cdot a_j' + \text{ReLU} \left(-b + \sum_{i=1}^{m} -a_i x_i\right) \cdot (-a_j')
\]

This concludes the proof.

NP-hardness results for even stronger restrictions on the neural networks can be found in [38,39,47].

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

Theorem 3.7:
The problem SAT(REACH[\emptyset]) is solvable in polynomial time.

Recall that many NNL specifications are negations of \exists 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?
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

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 [20].

**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:

\[
\begin{align*}
    t &::= a \mid x \mid t + t \mid t \cdot t \mid e^t \\
    \varphi &::= t \leq t \mid \neg \varphi \mid \varphi \lor \varphi \mid \exists x. \varphi
\end{align*}
\]

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 \( \text{NNL}^* \) be a shorthand for \( \text{NNL}[\text{ReLU}, \text{NLReLU}, \sigma, \tanh] \). 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 [20], stating that the problem is effectively equivalent to SAT(\( \text{NNL}^* \)).

**Theorem 3.9: Expressive Equivalence of NNL* and REF [20]**

We have NNL* \( \leq \) REF and REF \( \leq \) NNL*. 

---

Exercise 3.10:

In Theorem 3.8, why do we have to restrict inputs to sentences?
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

Proof. The statement follows from the two propositions below.

Proposition 3.6:

We have \( \text{NNL}^* \leq \text{REF} \).

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

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

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

We set

\[
\varphi_{\mathcal{N}}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \bigwedge_{i=1}^n \exists z. \left( z = b_i + \sum_{j=1}^m a_{i,j} \cdot x_j \right) \land \alpha_f(z, y_i).
\]

where

\[
\alpha_f(z, y_i) = \begin{cases} 
    y_i \cdot (1 + e^z) = e^z & \text{if } f = \sigma \\
    y_i \cdot (e^{2z} + 1) = (e^{2z} - 1) & \text{if } f = \tanh \\
    (z \leq 0 \land y_i = 0) \lor (z > 0 \land e^{y_i} = z + 1) & \text{if } f = \text{NLReLU}
\end{cases}
\]

The rest of the induction follows exactly the lines of the proof of Proposition 3.6.

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

Proposition 3.7:

We have \( \text{REF} \leq_{\text{poly}} \text{NNL}^* \).

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

Lemma 3.4:

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

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

Proof. For \( f \in \{\text{ReLU}, \eta, \sigma, \tanh\} \), let \( \mathcal{N}_f = (\mathcal{L}) \) where \( \mathcal{L} = ((1), (0), f) \). 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 \( \varphi_{\tanh^{-1}}(x, y) = (\mathcal{N}_{\tanh}(y) = x) \).

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
\[
I \models \varphi_{\ln}(x,y) \iff \ln(I(x)) = I(y).
\]

Proof. We will express \( \ln \) in terms of \( \sigma^{-1} \), \( \tanh^{-1} \), and \( \eta \):

- For all \( r \in \mathbb{R} \) such that \( r \geq 1 \):
  \[
  \eta(r - 1) = \ln(r)
  \]

- For all \( r \in \mathbb{R} \) such that \( 0 < r < 1 \):
  \[
  \sigma^{-1}(r) = \ln(r) - \ln(1 - r)
  \]
  \[
  \tanh^{-1}(r) = \frac{1}{2}(\ln(1 + r) - \ln(1 - r))
  \]
  \[
  \sigma^{-1}(r) - 2 \tanh^{-1}(r) + \eta(r) = \ln(r) - \ln(1 - r) - \ln(1 + r) + \ln(1 - r) + \ln(1 + r)
  \]
  \[
  = \ln(r)
  \]

Thus, we can set
\[
\varphi_{\ln}(x,y) = \begin{cases} 
  x > 0 \\
  \land (x \geq 1 \Rightarrow \exists z. (z = x - 1 \land \eta(z) = y)) \\
  \land (x < 1 \Rightarrow \exists z_1, z_2, z_3. (y = z_1 - 2 \cdot z_2 + z_3) \land \sigma^{-1}(x) = z_1 \land \tanh^{-1}(x) = z_2 \land \eta(x) = z_3)
\end{cases}
\]

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
\[
I \models \varphi_{\exp}(x,y) \iff e^{I(x)} = I(y).
\]

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 \( \varphi_{\text{mult}}(x,y,z) \) such that, for all interpretation functions \( I \), we have
\[
I \models \varphi_{\text{mult}}(x,y,z) \iff I(x) \cdot I(y) = I(z).
\]
CHAPTER 3. FEED-FORWARD NEURAL NETWORKS

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

\[
\varphi_{\text{mult}}(x, y, z) = \left\{ \begin{array}{l}
\land (x = 0 \lor y = 0) \Rightarrow z = 0 \\
\land (x > 0 \land y > 0) \Rightarrow \exists z_1, z_2, z_3. \\
\land x' = -x \\
\land \ln(x') = z_1 \\
\land \ln(y) = z_2 \\
\land z_1 + z_2 = z_3 \\
\land e^{z_3} = z \\
\land \ldots \\
\end{array} \right.
\]

Completing the formula is left as an exercise. \( \square \)

Exercise 3.11:
Fill the dots in the definition of \( \varphi_{\text{mult}}(x, y, z) \) in the proof of Lemma 3.7.

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

We are now ready to prove Proposition 3.7 stating that \( \text{REF} \leq \text{NNL}^* \):

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

\[
\exists (z_t)_{t \in T}. (z_{t_1} \leq z_{t_2} \land \psi_{t_1} \land \psi_{t_2})
\]

where \( \psi_{t_1} \) and \( \psi_{t_2} \) are defined inductively by

\[
\begin{align*}
\psi_a &= (z_a = a) \\
\psi_x &= (z_x = x) \\
\psi_{t+t'} &= (z_t + z_{t'} = z_{t+t'} \land \psi_t \land \psi_{t'}) \\
\psi_{t \cdot t'} &= (z_t \cdot z_{t'} = z_{t \cdot t'} \land \psi_t \land \psi_{t'}) \\
\psi_{e^t} &= (e^{z_t} = z_{e^t} \land \psi_t)
\end{align*}
\]

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 start with a simple class of recurrent neural networks, also known as Elman neural networks \[13\]. We will later consider more advanced architectures such as long short-term memory (LSTM) \[19\].

<table>
<thead>
<tr>
<th>Definition 4.1: Recurrent Neural Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>A recurrent neural network (RNN) is a triple ( \mathcal{R} = (\mathcal{L}^\text{in}, \mathcal{L}^\text{out}, h^{(0)}) ) where, for some ( m, n, o \in \mathbb{N}_+ ),</td>
</tr>
<tr>
<td>( - \mathcal{L}^\text{in} ) is a layer with ( \text{in}(\mathcal{L}^\text{in}) = n + m ) and ( \text{out}(\mathcal{L}^\text{in}) = n ),</td>
</tr>
<tr>
<td>( - \mathcal{L}^\text{out} ) is a layer with ( \text{in}(\mathcal{L}^\text{out}) = n ) and ( \text{out}(\mathcal{L}^\text{out}) = o ),</td>
</tr>
<tr>
<td>( - h^{(0)} \in \mathbb{Q}^n ) is the initial (hidden) state vector.</td>
</tr>
</tbody>
</table>

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

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

When we see \( \mathcal{R} \) as a sequence-to-sequence transducer, we consider the mapping

\[
\left[\mathcal{R}\right] : \left\{ \begin{array}{c}
(\mathbb{R}^m)^+ \to (\mathbb{R}^o)^+ \\
\mathbf{x}^{(1)} \ldots \mathbf{x}^{(\ell)} \mapsto \mathbf{y}^{(1)} \ldots \mathbf{y}^{(\ell)}
\end{array} \right. 
\]
where, for all discrete time points $\tau \in \{1, \ldots, \ell\}$,

$$
\begin{align*}
\mathbf{h}(\tau) &= \left[ \mathcal{L}^{\text{in}} \right](\mathbf{h}(\tau-1) \mathbin{\boxplus} \mathbf{x}(\tau)) \\
\mathbf{y}(\tau) &= \left[ \mathcal{L}^{\text{out}} \right](\mathbf{h}(\tau))
\end{align*}
$$

Recall that $\mathbf{h}(\tau-1) \mathbin{\boxplus} \mathbf{x}(\tau)$ 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 $\mathbf{h}(1) \ldots \mathbf{h}(\ell) \in (\mathbb{R}^m)^\ast$. Note that $[\mathcal{R}]$ is length preserving, i.e., an input sequence containing $\ell$ vectors is always mapped to an output sequence containing $\ell$ 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 $\delta_{\mathcal{R}}(\mathbf{h}, \mathbf{x}) = \left[ \mathcal{L}^{\text{in}} \right](\mathbf{h} \mathbin{\boxplus} \mathbf{x})$, which is canonically extended to sequences. When considering $\mathcal{R}$ as a sequence-to-vector transducer, we would be interested in the mapping

$$
\langle \mathcal{R} \rangle : \left\{ \begin{array}{l}
(\mathbb{R}^m)^+ \to \mathbb{R}^o \\
w \mapsto \left[ \mathcal{L}^{\text{out}} \right](\delta_{\mathcal{R}}(\mathbf{h}(0), w))
\end{array} \right.
$$

**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.

**Definition 4.2: One-Hot Encoding**

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

42
The corresponding one-hot decoding is the mapping $dec_\Sigma : \mathbb{R}^m \to \Sigma$ defined by $dec_\Sigma(x) = \alpha$ where $\alpha$ is such that $\text{argmax}(enc_\Sigma(\alpha)) = \text{min}(\text{argmax}(x))$. 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 $\Sigma$ 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 $\theta$. This is particularly common in the output layer in combination with activation function $\sigma$ and threshold $\theta = 0.5$.

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

Let $\Sigma$ and $\Gamma$ be finite alphabets with associated one-hot encodings $enc_\Sigma$ and $enc_\Gamma$, and decodings $dec_\Sigma$ and $dec_\Gamma$. Furthermore, assume $|\Sigma| = \text{in}(\mathcal{R})$ and $|\Gamma| = \text{out}(\mathcal{R})$. Then, RNN $\mathcal{R}$ defines the mappings

$$\left[\mathcal{R}\right]_{\Sigma, \Gamma} : \left\{ \Sigma^+ \to \Gamma^+ \right\} \quad w \mapsto dec_\Gamma(\left[\mathcal{R}\right](enc_\Sigma(w)))$$

$$\left\langle \mathcal{R} \right\rangle_{\Sigma, \Gamma} : \left\{ \Sigma^+ \to \Gamma \right\} \quad w \mapsto dec_\Gamma(\left\langle \mathcal{R} \right\rangle(enc_\Sigma(w)))$$

**Remark 4.2: Alternative Semantics**

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

$$\left\langle \mathcal{R} \right\rangle^{\text{prob}}_{\Sigma, \Gamma} : \left\{ \Sigma^+ \to \mathcal{D}(\Gamma) \right\} \quad w \mapsto \left\langle \mathcal{R} \right\rangle(enc_\Sigma(w))$$

where $\mathcal{D}(\Gamma)$ is the set of probability distributions over $\Gamma$. The idea is that, for the vector $y = \left\langle \mathcal{R} \right\rangle^{\text{prob}}_{\Sigma, \Gamma}(w)$, the probability of selecting $\beta \in \Gamma$ as the next letter is $y_{\text{argmax}(enc_\Gamma(\beta))}$. 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 $\left[\mathcal{R}\right]$ and $\left\langle \mathcal{R} \right\rangle$, which are of the following form and whose definitions are as expected:

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

$$\left[\mathcal{R}\right]^{\omega}_{\Sigma, \Gamma} : \Sigma^{\omega} \to \Gamma^{\omega}$$

**RNNs as String Classifiers.** Let $\Sigma$ be a finite alphabet. An RNN $\mathcal{R}$ is called an RNN over $\Sigma$ if $\text{in}(\mathcal{L}^{\text{in}}) = |\Sigma|$. Like above, $\mathcal{R}$ can then take one-hot encoded letters from $\Sigma$ as inputs. If, furthermore, $\mathcal{L}^{\text{out}}$ uses $\sigma$ as activation function and $\text{out}(\mathcal{L}^{\text{out}}) = 1$, we can interpret $\mathcal{R}$ as a classifier for strings over $\Sigma$. For $\bowtie \in \{\geq, >, =\}$ and $\theta \in [0, 1]$, we define $L^{\bowtie, \theta}_{\Sigma}(\mathcal{R}) = \{w \in \Sigma^+ \mid \left\langle \mathcal{R} \right\rangle(enc_\Sigma(w)) \bowtie \theta\}$
to be the *language* of $\mathcal{R}$, i.e., the set of words (or strings) that have a sufficient “score”, or *acceptance probability*, and are therefore classified by $\mathcal{R}$ as positive.

**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 [22]. Consider, for example, an email spam classifier that processes a text, i.e., a sequence over a finite alphabet $\Sigma$ (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 $\theta$ 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 Emptiness**

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

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

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

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, 21, 24, 36, 45].

The rest of this section is dedicated to the proof of Theorem [11]. We first show that (ReLU, $\sigma$)-RNNs can simulate all *probabilistic finite automata* (PFAs). PFAs were introduced and studied by Rabin [35]. Their (non)emptiness problem is undecidable and they are strictly more expressive than finite automata.

**Definition 4.3: Probabilistic Finite Automaton (PFA)**

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

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

Note that $\mathcal{P}^a$ is a *left stochastic matrix*. We call it the *transition-probability matrix* of
\( \alpha \in \Sigma \). The intuition is that \( A \) has an (implicit) set of states \( Q = \{q_1, \ldots, q_n\} \), and \( p_{j,i}^{\alpha} \) is the probability that, when reading \( \alpha \) in state \( q_i \), we go to state \( q_j \). \(^1\) Thus, \( A \) has an equivalent representation as a state-transition graph \( (Q, \iota, \Delta, F) \), 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 \( F = \{q_i \mid i \in \{1, \ldots, n\} \text{ with } \gamma_i = 1\} \). 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 \( i, j \in \{1, \ldots, n\} \) such that \( p_{j,i}^{\alpha} > 0 \), the transition \((q_i, \alpha, p_{j,i}^{\alpha}, q_j)\). We will switch between these representations at discretion.

PFA \( A \) defines the mapping 
\[
\delta_{A} : \left\{ \mathbb{R}^n \times \Sigma \rightarrow \mathbb{R}^n \right\} \ni (x, \alpha) \mapsto P^{\alpha} \cdot x .
\]

Suppose that \( \delta_{A} \) gets \((x, \alpha)\) as arguments and that \( x \) represents a probability distribution over \( \{q_1, \ldots, q_n\} \), i.e., \( x_i \) is the probability of being in state \( q_i \). Then, letting \( x' = \delta_{A}(x, \alpha) \), \( x' \) can naturally be interpreted as the probability of being in state \( q_i \) after reading \( \alpha \). Using the extension of \( \delta_{A} \) to strings, we obtain the mapping 
\[
\left[ A \right] : \left\{ \Sigma^* \rightarrow [0,1] \right\} \ni w \mapsto \gamma \cdot \delta_{A}(\lambda, w) .
\]

Thus, starting from the initial state, \( \left[ A \right](w) \) is the probability of reaching a final state after reading \( w \). Finally, for \( \propto \in \{\geq, >, =\} \) and \( \theta \in [0,1] \), we define the language 
\[
L^\propto\theta(A) = \{ w \in \Sigma^* \mid \left[ A \right](w) \propto \theta \} .
\]

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

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 \(^40\)) 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 \( \delta : Q \times \Sigma \rightarrow D(Q) \), whereas generative PFAs have a transition function of type \( \delta : Q \rightarrow D(\Sigma \times Q) \).

Example 4.2: Probabilistic Finite Automaton

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

\[
\lambda = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad P^{\alpha} = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} \end{pmatrix} \quad P^{\beta} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \gamma = \begin{pmatrix} 1 \\ 0 \end{pmatrix}
\]

Alternatively, \( A \) can be represented by its state-transition diagram:

\(^1\)We choose \( p_{j,i}^{\alpha} \) rather than the more standard \( p_{i,j}^{\alpha} \), as it reflects 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.
For $n \in \mathbb{N}$, we have

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

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

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

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

**Lemma 4.1: Closure Properties of PFAs**

Let $\Sigma$ be a finite alphabet, $p \in [0,1]$ be rational, and $A$ and $B$ be PFAs over $\Sigma$. We can effectively construct PFAs $C_1, C_2, C_3$ over $\Sigma$ such that, for all $w \in \Sigma^+$,

- $[C_1](w) = 1 - [A](w)$,
- $[C_2](w) = p \cdot [A](w) + (1 - p) \cdot [B](w)$, and
- $[C_3](w) = [A](w) \cdot [B](w)$.

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

\[
L^{\preceq \frac{1}{2}}_\Sigma(R) = L^{\leq \theta}(A).
\]
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 \( L^{in} \) such that \( |in(L^{in})| = |\Sigma| \) (with one-hot encoded letters as inputs) and \( state(L^{in}) = |Q| \) as state dimension. The corresponding matrix \( A^{in} \) would then have dimension \( |Q| \times (|Q| + |\Sigma|) \). However, in \( A \), we have \( |\Sigma| \cdot |Q|^2 \) transition probabilities, which thus may not fit into \( A^{in} \).

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 \( A \) from Example 4.2. Recall that \( \Sigma = \{\alpha, \beta\} \) and suppose \( \text{enc}_\Sigma(\alpha) = (1, 0)^\top \) and \( \text{enc}_\Sigma(\beta) = (0, 1)^\top \). The PFA \( A' \) illustrated in Figure 4.2a is equivalent to the PFA \( A \) from Example 4.2, i.e., \( J_A K(w) = J_{A'} K(w) \) for all \( w \in \Sigma^* \). However, every state of \( A' \) now has only incoming transitions labeled with a dedicated letter \( \alpha \) or \( \beta \) (which we accordingly call an \( \alpha \)- or \( \beta \)-state). Note that \( A' \) is given by the following ingredients:

\[
\lambda = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad P^\alpha = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} & \frac{2}{3} \\ 0 & 0 & 0 \end{pmatrix}, \quad P^\beta = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}, \quad \gamma = (1 \ 0 \ 1 \ 0)
\]

Thanks to the transformation, the transition-probability matrices \( P^\alpha \) and \( P^\beta \) 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 \( P^\alpha \) and \( P^\beta \) accommodates all probabilities occurring in \( A' \) in a single matrix of dimension \( \mathbb{R}^{4 \times 4} \). The layer \( L^{in} \) of the desired RNN is illustrated in Figure 4.2b for a neuron belonging to the \( \alpha \)-state \( q_2 \). If the input letter is \( \alpha \), i.e., the input vector is \( (1, 0)^\top \), the output neuron indeed receives the updated probability of being in \( q_2 \) after processing the input. If, on the other hand, the input is \( \beta \) and, respectively, \( (0, 1)^\top \), then the result before applying ReLU will be \( \leq 0 \) so that the overall result is 0. As \( q_2 \) is an \( \alpha \)-state, the probability of being in \( q_2 \) after processing \( \beta \) 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 [31]. In this context, a prominent result by Siegelmann and Sontag states that RNNs can simulate all Turing machines [41]. A general technique of transforming finite-state machines into equivalent RNNs goes back to Minsky [32]. It has recently been generalized to *generative* PFAs [42], which define probability distributions over \( \Sigma^* \) (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 \( \Sigma \) be a finite alphabet and \( m = |\Sigma| \). Let \( A = (P^\alpha)_{\alpha \in \Sigma}, \lambda, \gamma \) be a PFA over \( \Sigma \), with state-transition representation \((Q, \iota, \Delta, F)\) where \( Q = \{q_1, \ldots, q_n\} \). Moreover, fix \( \Rightarrow \in \{\geq, >, =\} \) 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 \( \alpha = \beta \). We call state \( q_j \) whose incoming transitions all carry letter \( \alpha \) an \( \alpha \)-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 \( p^{\alpha}_{i,j} > 0 \) and \( p^{\beta}_{i,j} > 0 \), we have \( \alpha = \beta \).

We now formally construct the corresponding \((\text{ReLU}, \sigma)\)-RNN \( \mathcal{R} = (\mathcal{L}^{\text{in}}, \mathcal{L}^{\text{out}}, h^{(0)}) \) over \( \Sigma \) such that \( \text{out}(\mathcal{L}^{\text{out}}) = 1 \) and \( L_{\Sigma}^{\mathcal{H}}(\mathcal{R}) = L(\mathcal{A}) \).

- We let \( h^{(0)} = \lambda \).

- Define \( \mathcal{L}^{\text{in}} = (A^{\text{in}}, b^{\text{in}}, \text{ReLU}) \) as follows: Let \( A^{\text{in}} = P \Box C \) be the concatenation of the matrices \( P \in [0, 1]^{n \times n} \) and \( C \in \{0, 1\}^{n \times m} \) defined by

\[
P = \sum_{\alpha \in \Sigma} P^{\alpha}
\]

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

\[
c_{j,k} = \begin{cases} 
1 & \text{if } \arg\max(\text{enc}_{\Sigma}(\alpha)) = k \\
0 & \text{otherwise.}
\end{cases}
\]

Finally, for all \( j \in \{1, \ldots, n\} \), we set \( b^{\text{in}}_j = -1 \).

- We define \( \mathcal{L}^{\text{out}} = (A^{\text{out}}, b^{\text{out}}, \sigma) \) by \( A^{\text{out}} = \gamma \) and \( b^{\text{out}} = (-\theta) \).
Correctness Proof. Let us show that, for all \( h \in [0, 1]^n \) and \( \alpha \in \Sigma \), we have \( \delta_R(h, \text{enc}_\Sigma(\alpha)) = \delta_A(h, \alpha) \). Let \( x = \text{enc}_\Sigma(\alpha) \). We have

\[
\delta_R(h, \text{enc}_\Sigma(\alpha)) = \delta_R(h, x) = [L^{in}](h \boxplus x)
\]

\[
= \text{ReLU}(A^{in} \cdot (h \boxplus x) + b^{in})
\]

\[
= \text{ReLU}((P \boxtimes C) \cdot (h \boxplus x) + b^{in})
\]

\[
= \text{ReLU}(P \cdot h + C \cdot x + b^{in})
\]

\[
= \text{ReLU}((\sum_{\beta \in \Sigma} P^{\beta}) \cdot h + C \cdot x + b^{in})
\]

Let \( y = (y_1, \ldots, y_n)^\top = C \cdot x + b^{in} \). 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 \( z = (\sum_{\beta \in \Sigma} P^{\beta}) \cdot h + y \in \mathbb{R}^n \), we get, for all \( j \in \{1, \ldots, n\} \),

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

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

\[
\delta_R(h, \text{enc}_\Sigma(\alpha)) = \text{ReLU}(z) = P^{\alpha} \cdot h = \delta_A(h, \alpha).
\]

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

\[
\langle \langle \mathcal{R} \rangle \rangle(\text{enc}_\Sigma(w)) \gg \frac{1}{2}
\]

iff \( [L^{out}](\delta_R(h^{(0)}, \text{enc}_\Sigma(w))) \gg \frac{1}{2} \)

iff \( [L^{out}](\delta_A(\lambda, w)) \gg \frac{1}{2} \)

iff \( \sigma(\gamma \cdot \delta_A(\lambda, w) - \theta) \gg \frac{1}{2} \)

iff \( \gamma \cdot \delta_A(\lambda, w) - \theta \gg 0 \)

iff \( \gamma \cdot \delta_A(\lambda, w) \gg \theta \)

iff \( [A](w) \gg \theta \)

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

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

The following undecidability results for PFAs are due to [5] and [34]. The proofs were later simplified and strengthened in [15]. For a concise overview of what is decidable and undecidable in PFAs, we refer to [14].
The following three decision problems are undecidable:

**Input:** A finite alphabet $\Sigma$ and a PFA $A$ over $\Sigma$.

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

**Question 2:** Do we have $L_{= \frac{1}{2}}(A) \neq \emptyset$?

**Question 3:** Do we have $L_{> \frac{1}{2}}(A) \neq \emptyset$?

**Proof.** We first consider Question 1. The proof is a reduction from the following modified Post’s correspondence problem (modified PCP).\(^3\)

**Input:** A finite alphabet $\Sigma$ and morphisms $f_1, f_2 : \Sigma^* \rightarrow \{0,1\}^*$ 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 $A$ over $\Sigma$ such that, for all $w \in \Sigma^+$, we have $f_1(w) = f_2(w)$ iff $[A](w) = \frac{1}{2}$, which implies the theorem.

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

$$u_1 \ldots u_k = 0.u_ku_{k-1} \ldots u_1 \text{ (in binary)}$$

$$= \frac{u_k}{2^1} + \frac{u_{k-1}}{2^2} + \ldots + \frac{u_1}{2^k}.$$

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

Towards the PFA $A$, we will construct two PFAs $A_1$ and $A_2$ over $\Sigma$ such that, for all $i \in \{1,2\}$ and $w \in \Sigma^*$, we get

$$[A_i](w) = f_i(w). \quad (4.1)$$

We then combine $A_1$ and $A_2$ using Lemma 4.1 and obtain a PFA $A$ over $\Sigma$ such that, for all $w \in \Sigma^*$,

$$[A](w) = \frac{1}{2}([A_1](w) + (1 - [A_2](w))).$$

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

**PFA Evaluating Binary Numbers.** The main building block in the construction of both $A_1$ and $A_2$ will be a PFA $B = ((P^0, P^1), \lambda, \gamma)$ over $\{0,1\}$ such that, for all

\(^3\)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 $g_1, g_2 : \Sigma \rightarrow \{0,1\}^+$, we can translate it into $f_1, f_2 : \Sigma \rightarrow \{0,1\}^*$ defined by $f_i = f \circ g_i$ where $f : \{0,1\}^* \rightarrow \{0,1\}^*$ 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 $w \in \Sigma^+$.
\( \nu \in \{0, 1\}^* \), \( [\mathcal{B}] (\nu) = \nu \). Thus, \( \mathcal{B} \) “evaluates” \( \nu \). It is given by

\[
\lambda = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad P^0 = \begin{pmatrix} 1 & \frac{1}{2} \\ 0 & \frac{1}{2} \end{pmatrix} \quad P^1 = \begin{pmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & 1 \end{pmatrix} \quad \gamma = (0 \ 1)
\]

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

\[
[\mathcal{B}] (0) = p_{2,1}^0 = 0.0 = 0 \\
\text{and} \quad [\mathcal{B}] (1) = p_{2,1}^1 = 0.1 = 1
\]

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

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

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

\[
p = [\mathcal{B}] (\nu) = \nu. \tag{4.2}
\]

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

\[
[\mathcal{B}] (u_1 \ldots u_k u) = (0 \ 1) \cdot P^u \cdot \begin{pmatrix} 1 - p \\ p \end{pmatrix} = \begin{cases} \frac{p}{2} & \text{if } u = 0 \\ \frac{1}{2} + \frac{p}{2} & \text{if } u = 1 \end{cases}
\]

\[
= \frac{u}{2} + \frac{p}{2} \quad 0.u + 0.0u_k \ldots u_1 = 0.uu_k \ldots u_1 = \overline{u_1 \ldots u_k u}
\]

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

\[
[\mathcal{A}_i] (w) = \gamma \cdot P^{\alpha_k} \ldots P^{\alpha_1} \cdot \lambda \\
= \gamma \cdot P_{f_i (\alpha_k)} \ldots P_{f_i (\alpha_1)} \cdot \lambda \\
= \gamma \cdot P_{f_i (w)} \cdot \lambda = f_i (w).
\]

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 \( \Sigma \) such that, for all \( w \in \Sigma^* \), we have

\[
[B] (w) = [A] (w) \cdot (1 - [A] (w)).
\]

\[\text{Note that } P^0 \text{ and } P^1 \text{ should not be confused with powers of some matrix } P.\]
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 $[B](w) \geq \frac{1}{4}$ iff $[A](w) = \frac{1}{2}$.

Undecidability of Question 3 can be obtained by analyzing the transition probabilities in the automata considered so far. We refer the reader to [15].

We now have shown Theorem 4.1 using the effective constructions from Theorems 4.2 and 4.3 to reduce PFA emptiness problems for $\infty \in \{\geq, >, =\}$ to RNN emptiness. Specifically, for every PFA $A$ over $\Sigma$, we constructed $(\text{ReLU}, \sigma)$-RNNs $R_1, R_2, R_3$ over $\Sigma$ whose output layers have output dimension 1 and such that the following hold:

\[
L_{\Sigma}^{\geq \frac{1}{2}}(R_1) = L^{\geq \frac{1}{2}}(A)
\]

\[
L_{\Sigma}^{> \frac{1}{2}}(R_2) = L^{> \frac{1}{2}}(A)
\]

\[
L_{\Sigma}^{> \frac{1}{2}}(R_3) = L^{> \frac{1}{8}}(A)
\]
Bibliography


