Turing Completeness of Neural Networks

D'Ambrosi Denis

`dambrosi.denis@spes.uniud.it`

`147681`

February 28, 2023

Abstract

Neural networks are a powerful machine learning technique that can be applied to a variety of applications, from image classification to natural language processing. At a basic level, neural networks consist of interconnected nodes that transmit and locally process data, but this seamingly simple architecture allows for an impressive degree of adaptability. Since nowadays they are broadly used to solve virtually any form of task, we must question they actual expressive power: are they actually Turing complete? In this paper, we will examine neural networks' Turing completeness and how it affects how they can be used for computation.

Contents

1 Introduction

A neural network is the second best way to solve any problem. The best way is to actually understand the problem.

Unknown

In 1989, Cybenko [Cyb89] showed that for each continuous function there exists at least one neural network with a single hidden layer capable of approximating it to arbitrary accuracy. This result is significant because it implies that neural networks can be used to model a wide range of complex functions, including those with non-linear relationships between input and output variables. Multiple variations to the standard architecture have been proposed and implemented (for example increasing the number of layers, including skip connections and adding loops to the computational graph) during the years, but we must assess whenever these alternatives actually increase the expressive power of the basic topology and if so, where is the boundary of this data structure's computability power. The current essay is structured in the following way: the rest of this section will introduce the foundamental concepts and notation for the rest of the material. In section 2 we'll analyze the Turing completeness of (recurrent) neural networks from a theoretical perspective, without caring about actual implementability of the systems described. In the following section we will instead take a look at concrete architectures that were proposed to simulate memory-bounded Turing machines. Finally, in the conclusions, we'll sum up the results and present some expected future work within this research field.

1.1 Turing Machines

Turing machines are a fundamental concept in the theory of computation, introduced by the mathematician Alan Turing in 1937 [Tur37]. They provide a formal definition of what it means to compute a function, and have been instrumental in advancing our understanding of the limits of what can be computed by a mechanical process.

A Turing machine consists of a tape divided into discrete cells, a read/write head that can move along the tape, and a set of rules that govern how the head interacts with the tape. The tape is initially populated with a finite sequence of symbols, and the machine is in a particular (starting) state. At each step, the machine reads the symbol under the head, performs a specified action (such as writing a new symbol, moving the head left or right, or changing its state), and then moves to the next cell on the tape. The output of the machine is determined by the final state and the symbols on the tape.

Formally, a Turing machine can be represented using a quadruple

$$(Q \cup \{q_{\text{accept}}, q_{\text{reject}}\}, \Gamma, \delta, q_0) \tag{1}$$

where:

- Q is a finite set of states.

- Γ is a finite set of tape symbols, which includes the blank symbol #.

- δ is a transition function that maps $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, where L and R represent moving the head left or right on the tape.

- $q_0 \in Q$ is the initial state.

- $q_{\text{accept}} \in Q$ is the accepting state.

- $q_{\text{reject}} \in Q$ is the rejecting state.

Before continuing, it's best that we also define the concept of istantaneous description of a Turing machine, since it will represent the starting (and ending) point of a simulation cycle by the neural networks that will be constructed.

An instantaneous description of a Turing machine is a snapshot of its current state. It provides a complete description of the machine's configuration at a given point in time, including the contents of the tape, the position of the head, and the current state of the machine.

Formally, an instantaneous description can be represented as a 3-tuple (q, l, r), where:

- q is the current state of the machine.

- l is the contents of the tape to the left of the head.

- r is the contents of the tape to the right of the head.

For example, if a Turing machine is in state q_0, with the tape contents "00101" and the read/write head positioned over the first symbol, the instantaneous description can be represented as $(q_0, \epsilon, 00101)$, where ϵ represents the empty string.

The instantaneous description of a Turing machine is important because it allows us to reason about its behavior at a particular point in time. By examining the current state and tape contents, we can determine which transition the machine will take next, and how it will update its configuration. This allows us to analyze the computation of the machine step by step, and to understand how it processes input and produces output.

1.2 Neural Networks

Neural networks are a class of machine learning models that are designed to learn and recognize patterns in data. At their core, neural networks are composed of a large number of interconnected processing nodes, which are designed to simulate the behavior of neurons in the brain. As computer scientists, we are able to execute sophisticated computations on input data by connecting these nodes into intricate, layered structures, enabling the data structures themselves

to deduce intricate patterns in the information provides and forecast future predictions on unseen examples.

At the most basic level, a node in a neural network is a mathematical function that takes one or more inputs and produces a single scalar output. Each node is connected to one or more other nodes, composing a network of interconnected processing elements. The computed output of one node serves as the input to the next (that may be just one or multiple ones), allowing information to flow in a predetermined way through the network itself and be processed at each step.

Following the explaination written by David Kriesel [Kri07], we can define a neuron as a node that computes the composition of three separate functions:

1. A **net** function, that aggregates the input values from the input neurons

2. An **activation** function, that maps the net value and a threshold value called **bias** into a single scalar value

3. An **output** function, that transforms the "activated value" of the neuron into an output scalar that can be forwarded to the following neurons

In practice, a weighted sum is istantiated as net function, a sigmoid ($\sigma(z) = 1/(1 + e^{-z})$) or $ReLU$ ($\sigma(z) = max(0, z)$) function is chosen as activation and the output is left as the identity. Although most of the real architectures do not differ much from this last blueprint, keeping in mind the more general definition will allow us to better investigate the expressive power of these structures.

The weights in the net function are learned through a process called training, which involves adjusting the weights to minimize the error between the network's output and the desired output. This allows the network to learn to recognize patterns in the data and make accurate predictions. In order to do so, we generally use the backpropagation algorithm [quote], that requires all of the operations computed within the network to be differentiable in order to compute a gradient.

It's also important to keep in mind that at least one of the three composed functions (generally the activation) needs to introduce non-linearity to the system: otherwise, the network would be limited to performing linear transformations on the input data, which would severely limit its expressiveness.

Taking a closer look to the types of connection within a network, we can discriminate between two main kinds of architectures:

- **Feedforward Neural Networks** (FFNNs) are networks that have a single flow of input, where the data is processed from the input layer through one or more hidden layers to the output layer. In FFNNs, the information flows in one direction, from the input to the output layer, with no feedback connections: the output of one layer serves as the input for the next layer.

- On the other hand, **Recurrent Neural Networks** (RNNs) are designed to process sequential data, where the order of the data points matters.

RNNs have loops in the network, which allow the output of a given layer to be fed back as input to the same layer or to a previous layer in the network. RNNs can maintain a sort of "memory" of previous inputs, which enables them to handle sequential data such as speech, text, and time series data. The main advantage of RNNs is their ability to model sequences of arbitrary length and process input data of variable size. We will shortly see how this property will be crucial to obtain Turing completeness.

2 Theoretical Turing completeness of RNNs

In this section we will further investigate the expressive power of Recurrent Neural Networks without taking into consideration the actual realizability of the systems taken into consideration. This branch of research has been initially explored by Siegelmann and Sontag in 1995 [SS95], when they presented a theoretical framework for understanding the computational power of neural networks, and in particular, their ability to simulate the behavior of Turing machines.

This result has significant implications for the field of artificial intelligence and computer science, as it shows that neural networks are not just powerful tools for solving specific problems, but can also serve as a universal computational substrate capable of performing any computation that can be performed by a Turing machine.

Unfortunately, this result is not applicable to real-life recurrent architectures since we would require registers to have unlimited precision to store the content of the tape using rational numbers through a fractal encoding function. Still, in 2021 Chung and Siegelmann published another article [CS21] about the expressiveness of recurrent networks: this paper did not introduce any practical implications about the Turing completeness of these data structures but clarified some aspects about the previous proof from 1995 and partially solved the finite precision problem by introducing other caveauts. In the rest of this section we will take a brief look at the results provided by this article, as well as their issues.

2.1 Unbounded precision

The first theorem presented in [CS21] contributes with only very few additional implications with respect to the proof from 1995 from a practical standpoint, but it provides a clearer explaination and a slighty faster (only a linear speedup) simulation; it states:

Theorem 1 *Given a Turing Machine $\mathcal{M}$, there exists an injective function $\rho : \mathcal{X} \rightarrow \mathbb{Q}^N$ and an n-neuron unbounded-precision RNN $\mathcal{T}_{W,b} : \mathbb{Q}^n \rightarrow \mathbb{Q}^n$, where $n = 2|\Gamma| + \lceil \log_2 |Q| \rceil |Q||\Gamma| + 5$ such that for all istantaneous descriptions*

$$\rho^{-1}(\mathcal{T}_{W,b}^3(\rho(x))) = \mathcal{P}_{\mathcal{M}}(x)$$

Let's dissect this statement.

ρ The construction of ρ is provided earlier in the article: it includes a version of the fractal encoding function already introduced in [SS95], that allows us to save an instant configuration into a vector of size $2|\Gamma| + \lceil \log_2 |Q| \rceil |Q||\Gamma| + 5$ of rational numbers. ρ concatenates the encodings of the state, the left tape, the right tape and the first symbols to the sides of the head (even if it's actually explained that these last two informations aren't strictly necessary). We still need to define the encoding functions for the state, the tapes and the symbols, but before we need to note that for this construction to work properly we must encode the symbol alphabet Γ into a set of odd numbers.

- $\rho^{(q)} : Q \to \{0,1\}^{\lceil \log_2 Q \rceil}$ maps the states into a binary enumeration (for example, assuming we have $Q = \{q_0, q_1, q_2\}$, we can define a state encoding as $\rho^{(q)}(q_0) = [0,0]$, $\rho^{(q_1)}(q_0) = [0,1]$, $\rho^{(q_2)}(q_0) = [1,0]$)

- $\rho^{(s)} : \Gamma* \to \mathbb{Q}$ is used to encode the left and right tape using the following fractal encoding function:

$$\rho^{(s)}(y) := \left(\sum_{i=1}^{|y|} \frac{y_{(i)}}{(2|\Gamma|)^i} \right) + \frac{1}{(2|\Gamma|)^{|y|}(2|\Gamma|-1)} \qquad (2)$$

  Note that using fractal encoding we can manipulate the top symbols of the encoded stack and check for emptyness through simple arithmetical operations as explained in [SS95] and [Sie95].

- $\rho^r : \Gamma \to \{0,1\}^{|\Gamma|-1}$ allows for the encoding of a single symbol $s \in \Gamma$. The formula that defines each of the coordinates is

$$\rho_i^{(r)}(s) = 1\{s > 2i\} \; \forall i \in 1,..,|\Gamma|-1 \qquad (3)$$

  That is, the i-th coordinate of the encoding is equal to 1 if the symbol is bigger then $2i$ and 0 otherwise. A simple example of this encoding would be that if we had $\Gamma = \{1,3,5\}$, then $\rho^{(r)}(1) = [0,0]$, $\rho^{(r)}(3) = [1,0]$, $\rho^{(r)}(5) = [1,1]$.

Finally, note that ρ is injective, thus we can define an inverse function ρ^-1 that allows us to compute the configuration encoded by a valid vector.

$\mathcal{P}_\mathcal{M}(x)$ With this notation, the authors simply refer to the transition step between the configuration x of the Turing machine $\mathcal{M}$ and it's successor according to $\mathcal{M}$'s δ function (represented by $\mathcal{P}_\mathcal{M}$).

$\mathcal{T}^3_{W,b}$ The simulation of one step the Turing machine $\mathcal{M}$ requires 3 cycles of computation by the RNN $\mathcal{T}_{W,b}$, which we consider already trained on with weights W and biases b. We need to note that, within this paper, we consider a single computation of the RNN as an affine transformation of the values of the previous state (thus the implicit introduction of time) followed by the linear saturated activation function σ:

$$\sigma(x) = \begin{cases} 0 & \text{if} x < 0 \\ x & \text{if} 0 \le x \le 1 \\ 1 & \text{if} x > 1 \end{cases} \tag{4}$$

In practice, given a state $x(t)$ at time t, the following state $x(t+1)$ will be recursively computed as:

$$x(t+1) = \sigma(Wx(t) + b) \tag{5}$$

It's interesting to notice that, with this definition, we apparently do not have any inputs nor outputs, only relations between internal states. This is not a problem since we assume to be able to access and modify the internal representation of the network by keeping track of the activations of the neurons: before the first step of the computation we need to encode a valid configuration x into the neurons of $\mathcal{T}_{W,b}$ by applying ρ to x and then assigning its coordinates to different neurons according to their function. A more precise explaination will be provided shortly, but, before continuing, we wanted to also note that this proof can be easily expanded to standard input/output RNNs by applying the construction provided by section 4.4 of [SS95], which showed the equivalence between these two classes of networks. Finally, we can appreciate now how recurrence is crucial in order to obtain Turing completeness: without this ingredient, we would only have a finite amount of steps to calculate the transitions between configurations. We are thus able to conclude that feed forward neural networks can, at most, simulate time-bounded Turing machines.

Actual proof   The article provides a full-detailed proof within its supplementary material, but for the sake of briefness we will now only highlight its key concepts.

The network is subdivided into 6 main groups of neurons:

1. Stage neurons, that encode in which of the 3 states the RNN is (remember that we need 3 steps of computation to simulate a Turing machine transition).

2. Entry neurons, which compute the combination of state and symbol under the head to determine the right transition rule according to the δ function of the Turing machine.

3. Temporary tape neurons, which serve as a buffer to compute the transition of the head on the tape during the second stage.

4. Tape neurons, that encode the left and right tape in fractal encoding through equation 2

5. Readout neurons, which encode the first symbol to the left and to the right of the head (note that within this proof, the first symbol to the left is considered as the cell under the cursor).

6. State neurons, which encode the Turing machine's state

As previously said, we need to initialise these neurons with different coordinates of the encoded configuration $\rho(x)$ according to their function. The actual simulation is executed through a series of differentiable equations that follow the general structure of formula 5. The said three steps can be summarized as following:

1. In the first step, entry neurons, which are initialized with **0** compute the state-symbol combination in order to determine the next transition according to $\mathcal{M}$'s δ.

2. In the second step, state neurons and temporary tape neurons are updated according to the transition determined in the previous stage. Note that temporary tape neurons server as a buffer for tape neurons when shifting the tape

3. In the third stage, tape neurons are updated with the temporary neurons values

Note that readout and stage neurons are never quoted in this summary since are "used" multiple times. Stage neurons inhibit the update in neurons during the wrong timestamps (for example, we want tape neurons to remain unchanged during the first two steps) by subtracting its coordinates within the relative affine transformations. Similarly, readout neurons are used during the update of entry and temporary tape neurons, that happen at different stages of the simulation.

For more details about the proof, please refer to appendix A of [CS21]

Corollaries By applying 1 multiple times, we can simulate each step of the computation of a Turing machine $\mathcal{M}$ and compute the same function (if defined). This result is explained in corollary 1.1 of the paper, where they state that if $\mathcal{P}*_M$ is defined, then

$$\rho^{-1}(\mathcal{T}_{W,b}^*(\rho(x))) = \mathcal{P}_{\mathcal{M}}^*(x)$$

and $\mathcal{T}_{W,b}^*(\rho(x))$ is undefined otherwise.

Furthermore, since Neary and Woods in 2009 introduced a 6 states and 4 symbols universal Turing machine [NW09], we can state that there exists a 40-neuron unbounded-precision RNN that can simulate any Turing machine in time $\mathcal{O}(T^6)$.

2.2 Growing memory modules

The second addend within formula 2 allows to remove the unlimited precision requirement introduced in [SS95] to encode the infinite number of blank symbols on the tape, but the proof of theorem 1 still needs registers with unbounded precision, making any practical implementation impossible.

In the article they also introduced the idea of a *growing memory module*, that can be used as an external stack of neurons to store part of the tape of a Turing machine. The dynamics of this stack is controlled by two neurons called u and o, which are responsible for the popping and the pushing operations. Keeping into consideration the notion of time already introduced to define the evolution of an RNN, we can describe the update of the stack as follows:

- if $u(t) > 0$, then a new neuron with the value $u(t)$ is pushed onto the stack and $u(t + 1) = 0$.

- if $o(t) = 0$ and the stack is not empty, then the top neuron n is popped from the stack and $o(t + 1) = n$.

- if $o(t) = 0$ and the stack is empty, then $o(t + 1) = c$ where c is a default value.

We can define an RNN with two growing memory modules (following the key idea of simulating a Turing machine with a 2-stack pushdown atomata) as a mapping $\mathcal{T}_{W,b} : (\mathbb{Q}^N, \mathbb{Q}^*, \mathbb{Q}^*) \to (\mathbb{Q}^N, \mathbb{Q}^*, \mathbb{Q}^*)$.

The following result, presented in the article, aims at removing the unbounded precision requirement previously introduced with theorem 1:

Theorem 2 *Given a Turing Machine $\mathcal{M}$, there exists an injective function $\rho : \mathcal{X} \to (\mathbb{Q}^n, \mathbb{Q}^*, \mathbb{Q}^*)$ and an n-neuron p-precision (in base $2|\Gamma|$) RNN with two growing memory modules $\mathcal{T}_{W,b} : (\mathbb{Q}^n, \mathbb{Q}^*, \mathbb{Q}^*) \to (\mathbb{Q}^n, \mathbb{Q}^*, \mathbb{Q}^*)$, where $n = 2|\Gamma| + \lceil \log_2 |Q| \rceil |Q||\Gamma| + 19$ and $p \geq 2$, such that for all instantaneous descriptions $x \in \mathcal{X}$,*

$$\rho^{-1}(\mathcal{T}_{W,b}^3(\rho(x))) = \mathcal{P}_{\mathcal{M}}(x)$$

The idea is the same as behind theorem 1, but the construction of ρ need to undergo some changes:

- Since (as we will soon explain) we only need to keep a part of the tape within the RNN, $\rho^{(s)}$ does not need to take care of the infinite series of blanks anymore, thus it can be simply replaced by

$$\rho^{(s)}(y) := \left(\sum_{i=1}^{|y|} \frac{y_{(i)}}{(2|\Gamma|)^i} \right) \tag{6}$$

9

- We need to keep track the number of symbols currently stored within the RNN neurons: this information will be defined as $h(|s_j|)$, where j indicates whenever we are referring to the left tape s_L or the right tape s_R. $h(|s_j|)$ is then encoded through $\rho^{(h)}$, defined as:

$$\rho^{(h)}(y) = \frac{y}{p+1} \tag{7}$$

- Finally, to save the content of the tape in the memory modules, we define the $\rho^{(M)}$ function, which, starting from the symbols furthest from the head, encodes p symbols at a time through $\rho^{(s)}$ in each direction and then pushes the obtained value in the relative stack.

As a whole, ρ will be computed similarly to how it was defined in the previous section: the first coordinate of $\rho(q, s_L, s_R)$ will be the concatenation of the encodings of the state, the first $h(|s_L|)$ and $h(|s_R|)$ symbols, the following p symbols for each side of the tape, the guard symbols on the sides of the head, $h(|s_L|)$ and $h(|s_R|)$. The second and the third coordinates will be the encodings (as growing memory modules) of the two sides of the tape. Similarly to the previous section, ρ is injective, thus we can define the decoder function ρ^{-1} : $\rho(\mathcal{X}) \to \mathcal{X}$.

Actual proof Again, in the supplementary material we can find the full-detailed proof of the simulation of a Turing machine using an RNN. We will only discuss succintly the key ideas behind the construction.

First of all, we must define how $h(|s_j|)$ evolves during time, to understand which is the criterion that the RNNs uses to pop and push neurons in the stacks. In the case of $h(|s_L|)$, we have that:

$$h(|s_L'|) = \begin{cases} h(|s_L|) - 1 & \text{if } d = L \text{ and } h(|s_L|) \geq 2 \\ p & \text{if } d = L \text{ and } h(|s_L|) = 1 \\ h(|s_L|) + 1 & \text{if } d = R \text{ and } h(|s_L|) \leq p - 1 \\ 1 & \text{if } d = R \text{ and } h(|s_L|) = p \end{cases} \tag{8}$$

where d is the direction in which the head is moving and s_L' is the shifted tape. The idea is simple for the first and the third case (if we are moving left, we are consuming one symbol from the tape, if we are moving right we are adding one symbol to the tape), while can be a little less intuitive for the other two cases. If we are moving left and there is only one symbol left on the left tape ($h(|s_L|) = 1$), we consume it and thus we have to pop a neuron from the stack. Consequently, we have a new set of p symbols at our disposal in the neuron that encodes of the left tape. On the other hand, if we are moving right and we have p symbols for the left tape in the RNN, we aren't able to add another symbol to the neuron, thus we have to push it to the left stack and start encoding the tape again with only the new symbol. Of course, $h(|s_R|)$ is defined analogously.

Having defined how this quantity varies through execution, we can proceed by analyzing the structure of the RNN. The general idea follows the proof of

theorem 1 (we have all the 6 categories of neurons again), but we need to add a few more neurons to keep track of the pushing and popping operations:

8. Guard neurons, which will keep track of $h(|s_L|)$ and $h(|s_R|)$.

9. Buffer neurons, that are required to compute intermediate values for tape updates: when active (thus, not equal to 0), they hold the values popped from the stacks.

10. Push-pop neurons (u_L, o_L) and (u_R, o_R), which access the memory modules, as described previously.

The simulation is executed again during three main steps and the newly introduced neurons evolve during the second stage. The nodes presented in the previous section are updated with an alternative version of the original equations, that take into consideration the external memory modules:

- Tape neurons not only take into consideration temporary tapes, but also the values of the buffer neurons.

- Similarly, readout neurons take into consideration also buffer neurons in the case a popping operation was executed.

At the end of the third step, the updated RNN's state encodes the configuration of the simulated Turing machine after the relative transition according to its δ function.

For more details and the actual equations that descrive the dynamics of the system, please refer to the supplementary material of [CS21].

Corollary As for the previous section, applying multiple times this result leads to the corollary we're mostly interested in: if $\mathcal{P}*_M$ is defined, then

$$\rho^{-1}(\mathcal{T}^*_{W,b}(\rho(x))) = \mathcal{P}^*_{\mathcal{M}}(x)$$

and $\mathcal{T}^*_{W,b}(\rho(x))$ is undefined otherwise. Furthermore, in analogy with the previous section, we can state that there exists a 54-neuron p-precision RNN with two growing memory modules that can simulate any Turing Machine in $\mathcal{O}(T^6)$, where $p \geq 2$. Moreover, since growing memory modules represent the foundation form of stack-augmented RNNs, it's possible to simulate any latter RNN with the architecture described in the paper (the details of the proof are explained again in [CS21]), thus this proof can be extended to the class of stack-augmented recurrent neural networks.

This proof shows a (technically) possible implementation of the actual RNN (assuming we have enough memory to run any Turing machine we want to emulate) which allows for fast manipulation of the data: although the simulation of a single transition requires three stages of computation in both constructions, the network with the growing memory module only requires updating the symbols near the head at each step, while the unbounded precision architecture

needed to take into consideration all the tape. Unfortunately, as the authors explain in the conclusion, the RNN presented in this section is not trainable as of today, since the non-differentiability of the growing memory modules doesn't allow us exploit error backpropagation. Possible solutions would be to construct a differentiable version of the memory modules or to use a different learning rule to deal with the discrete pushing and popping operations.

2.3   Unbounded neurons

Siegelmann and Chung's article presented in conclusion a third final theorem, which explored another (arguably even less implementable) path to obtain Turing completeness: having at disposal an unbounded number of neurons of finite precision in order to encode the tape of the machine. The authors state that this could be either achieved by having an illimitate number of neurons at the beginning of the simulation, or by increasing the numerosity of the nodes of the networks on the fly based on the states reached by the RNN during the computation. Although this result (explained in corollary 3.1 of the paper) has apparently limited practical implications (at least for now), the path that the authors undertook to state it produced some concrete conclusions about the topic: before generalizing to an unbounded network, they actually proved that any memory-bounded Turing machine can be simulated by an RNN of predetermined size. The theorem is stated as follows:

Theorem 3 *Given a Turing machine $\mathcal{M}$ with a bounded tape of size F, there exists an injective function $\rho : \mathcal{X} \to \mathbb{Q}^n$ and an n-neuron p-precision (in base $|2\Gamma|$) RNN $\mathcal{T}_{W,b} : \mathbb{Q}^n \to \mathbb{Q}^n$, where $n = \mathcal{O}(\lceil F/p \rceil)$ and $p \geq 2$, such that for all instantaneous descriptions $x \in \mathcal{X}$,*

$$\rho^{-1}(\mathcal{T}_{W,b}^3(\rho(x))) = \mathcal{P}_{\mathcal{M}}(x)$$

Again, it is sensible to take a closer look to the definition of ρ before continuing with the proof overview. In this case, the encoding function is a combination of the constructions presented in the previous sections: on one side, it must encode the tape into different sections of at most p symbols; on the other hand, it has to map the entire configuration onto a single array instead of a triple since we want all the information to remain within the RNN itself. ρ is thus defined as $\rho : \mathcal{X} \to \mathbb{Q}^n$, where $n = 2|\Gamma| + \lceil \log_2 |Q| \rceil |Q||\Gamma| + 10f + 11$. As before, it will be constructed by concatenation of different encodings, of which we will now show the computation:

- $\rho^{(q)}, \rho^{(s)}, \rho^{(r)}$ and $\rho^{(h)}$ are the same functions as seen in the previous section.

- $\rho^{(M)}$ is defined in a very similar way to the tape encoding function introduced in the construction of theorem 2: it first divides the sequence of symbols to process into lists of length at most p and then transforms them using the fractal encoding function defined in equation 6. The difference

12

with the previously defined $\rho^{(M)}$ is that this version, instead of pushing the processed sequences into a stack, concatenates them into a vector of length $f = \lceil F/p \rceil$. Note that this array may feature trailing 0s to account for unnecessary tape space filled with blanks.

- $\rho^{(d)} : Q^* \to \{0,1\}^f$ encodes the position of the last non-zero element in $\rho^{(M)}(y)$. It is defined as

$$\rho_i^{(d)}(y) = \begin{cases} 1 & \text{if } = \lceil |y|/p - 1 \rceil \\ 0 & \text{else} \end{cases} \tag{9}$$

that is, the i-th coordinate $(i \in 1, ..., f)$ of $\rho^{(d)}(y)$ is equal to 1 if and only if it is the position of the last non-zero element of the encoding of the tape. We need this information to keep track of how many sections of the tape contain actual information (before the blank symbols).

Actual proof The construction of this RNNs recycles most of the work done for theorem 2 but, instead of push and pop neurons, it introduces yet again two new types of nodes:

- Stack neurons are introduced to replace growing memory modules: they are initialised through an application the freshly described $\rho^{(M)}$ and must encode the tape to the left and to the right of the head.

- Pointer neurons, which are istantiated with values calculated by a computation of the $\rho^{(d)}$ function; these nodes must keep track of which neurons contain relevant information. Note that these neurons can be seen as a one-hot-encoding of the sequences closest to the head, thus can be used to selectively execute operations on the stacks.

Moreover, buffer neurons do not access values within the external modules, but, instead, read values from the last non-zero values of the stack neurons.

As explained in supplementary material of the article, the objective is to remove the growing memory modules introduced in the previous section by storing all neurons encoding the tape within the RNN itself. The rest of the operations are only slighty changed: if the tape neuron holds less than 1 symbol after the update, we pop the stack node (that is, setting the last non-zero stack neuron to 0, changing the relative pointer nodes and updating the tape and readout values). On the other hand, if the tape neuron holds more than p symbols after the update, we push the bottom p symbols of the tape node onto the stack neurons (by setting the first zero neuron pointed by the pointer nodes to the value to be pushed).

As before, the simulation of a transition of $\mathcal{M}$ is executed in 3 cycles of computation of $\mathcal{T}_{W,b}$. All the stages remain the same to the ones explained in section 2.2, with additional care for the newly introduced neurons, that are updated during the third (and last) step.

Again, we encourage taking a look at appendix C of [CS21] for all the details about the equations involved.

Corollaries   By proving the previous theorem, Siegelmann and Chung implicitly demonstrated that any space-bounded Turing machine can be simulated by a carefully constructed RNN: by applying theorem 3 repeatedly, we can yet again reach the conclusion that, given a Turing machine $\mathcal{M}$ and a starting instantaneous description x, we can simulate $\mathcal{P}_{\mathcal{M}}(x)$ through a n-neuron p-precision recurrent network $\mathcal{T}_{W,b}$:

$$\rho^{-1}(\mathcal{T}_{W,b}^3(\rho(x))) = \mathcal{P}_{\mathcal{M}}(x)$$

such that $\mathcal{T}_{W,b}^3(\rho(x))$ is defined if and only if $\mathcal{P}_{\mathcal{M}}(x)$ is defined. Consequently, by unbounding the number of neurons, we can simulate any possible Turing machine. Since we do not take into consideration the size of the network anymore, we can exploit the construction provided by Hennie and Stearns [HS66] to show that there exists an unbounded-neuron, bounded-precision RNN capable of simulating any Turing machine in time $\mathcal{O}(T \log T)$.

References

[CS21] Stephen Chung and Hava T Siegelmann. Turing completeness of bounded-precision recurrent neural networks. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

[HS66] Frederick C Hennie and Richard E Stearns. Two-tape simulation of multitape turing machines. *Journal of the ACM (JACM)*, 13(4):533–546, 1966.

[Kri07] David Kriesel. *A brief introduction to neural networks*. BoD–Books on Demand, Norderstedt, Germany, 1 edition, 2007.

[NW09] Turlough Neary and Damien Woods. Four small universal turing machines. *Theoretical Computer Science*, 410(50):5105–5118, 2009.

[Sie95] Hava T Siegelmann. Computation beyond the turing limit. *Science*, 268(5210):545–548, 1995.

[SS95] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50:132–150, 1995.

[Tur37] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1937.