Turing Completeness of Neural Networks Architectures: a Review

D'Ambrosi Denis
dambrosi.denis@spes.uniud.it
147681

June 30, 2024

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 their 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 will 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 will 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 an unlimited 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 combination of specified action (such as writing a new symbol, moving the head left or right, and/or changing its state) according to a fixed transition rule, and then moves to the following cell on the tape. The output of the machine is determined by the final state and the symbols on the tape, if they are ever reached.

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 $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, where L and R represent moving the head left or right on the tape.
- $q_0 \in Q$ is the initial state.
- q_{accept} is the accepting state.
- q_{reject} is the rejecting state.

Before continuing, it is 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 complex, layered structures, enabling the networks themselves to deduce intricate patterns in the information provided 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 units. 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 Kiesel [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, that requires all of the operations computed within the network to be differentiable in order to compute a gradient.

It is 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 by introducing recursion. 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.

2.1 Turing completeness of general RNNs

Unfortunately, the conclusions explained in [SS95] are 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 new 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.

2.1.1 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 $\rho : \mathcal{X} \to \mathbb{Q}^N$ and an n-neuron unbounded-precision RNN $\mathcal{T}_{W,b} : \mathbb{Q}^n \to \mathbb{Q}^n$, where $n = 2|\Gamma| + \lceil \log_2 |Q| \rceil |Q||\Gamma| + 5$ such that for all istantaneous descriptions*

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

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 is actually explained that these last two informations are not 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 $\rho^{(q)}(q_0) = [0,0],\ \rho^{(q)}(q_1) = [0,1],\ \rho^{(q)}(q_2) = [1,0]$)
- $\rho^{(s)} : \Gamma^* \to \mathbb{Q}$ is used to encode the left and right tape using the following fractal encoding function:

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

  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

$$\rho^{(r)}_i(s) = 1\{s > 2i\}\ \forall i \in 1, .., |\Gamma| - 1 \tag{3}$$

  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:

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

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 is 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 explanation 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 $\mathbf{0}$ 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 $\mathcal{O}(T^6)$.

2.1.2 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 automata) 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 $x \in \mathcal{X}$,*

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

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

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

- 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 $\rho^{-1} : \rho(\mathcal{X}) \to \mathcal{X}$.

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 are not 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:

7. Guard neurons, which will keep track of $h(|s_L|)$ and $h(|s_R|)$.
8. 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.
9. 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 describe 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 are 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 is 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 does not 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.

2.1.3 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 $x \in \mathcal{X}$,*

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

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 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:

9. 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.

10. 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 2.1.2, 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 $\mathcal{T}_{W,b}$:

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

such that $\mathcal{T}^3_{W,b}(\rho(x))$ 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 $\mathcal{O}(T \log T)$.

To conclude this section, we would like to emphasize a small but important detail regarding the necessity of generalizing the function computed by neurons in a network to achieve the highest level of expressiveness. In his Ph.D dissertation, Pollack [Pol87] presented a recurrent network model, called "neuring machine", that achieved universality through the use of multiplications in the nodes. Altough he did not state that *high-order* neurons (that is, nodes that use products in the local computation) were definitely necessary to obtain Turing completeness, he conjectured that affine transformations did not provide enough flexibility to the system. On the contrary, the proofs in [SS95], along with all the following constructions, showed that linear combinations, if paired with proper non-linearity, are, in fact, enough powerful to compute any partial recursive function.

2.2 Turing completeness of modern architectures

Although the aim of this essay is to take into consideration the Turing completeness of general recurrent neural networks, we still wanted to present some interesting results about the expressiveness of more complex architectures that are currently used for real world tasks. In the remaining part of this section, we will briefly take a loot at the conclusions provided by [PMB19].

2.2.1 Models based on attention

The **Transformer** is a neural network architecture that has revolutionized the field of natural language processing (NLP). It was first introduced in 2017 in

a ground-breaking paper titled "Attention is All You Need" by Vaswani et al. [VSP+17], and has since become a widely used and highly influential architecture in NLP research.

At its core, the Transformer architecture is designed to process variable-length sequences of data, such as sentences or paragraphs, and is based on the use of multi-headed self-attention mechanisms, which will be briefly explained shortly. These attention mechanisms allow the model to capture long-range dependencies between different parts of the input, which is essential for many NLP tasks. This architectural choice allows for input vectorization, which is crucial to avoid lengthy training procedures by exploiting parallel computation over multiple examples.

A general Transformer network is composed of two similar, but in a sense opposite, components: an encoder and a decoder. The encoder takes the input sequence and produces a sequence of hidden states, that are analyzed by the decoder in order to compute the output sequence. These two actions are carried on by two distinct stacks of omogeneous, but independent, components, which, in turn, are made up of multiple network layers. Each layer of the each encoder/decoder is actually composed of two sub-layers: a multi-headed self-attention mechanism and a position-wise feedforward neural network. The self-attention mechanism allows the model to access to different parts of the input sequence simultaneously, while the feedforward network applies a non-linear transformation to each position independently.

One of the key innovations of the Transformer architecture is the use of self-attention mechanisms. Traditional recurrent neural networks and convolutional neural networks have limited ability to capture long-range dependencies in input sequences, which can lead to performance degradation on tasks that require modeling such dependencies. The self-attention mechanism allows the Transformer to attend to all positions in the input sequence simultaneously and to capture dependencies between distant positions.

The multi-headed self-attention mechanism in the Transformer allows the model to attend to multiple parts of the input sequence at once. This is achieved by splitting the input sequence into multiple heads, and applying self-attention to each head separately. This allows the model to capture multiple types of relationships between the input positions, and to weigh different relationships indipendently. Note that the last step of this mechanism requires the network to perform a convex combinations over embeddings with weights obtained through a softmax ($\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$) application.

Another important feature of the Transformer architecture is the use of positional encodings. Since the self-attention mechanism in the Transformer does not take into account the order of the input sequence, positional encodings are added to the input embeddings to give the model information about the order of the input. This allows the Transformer to capture both positional and content information from the input sequence. In the introductory paper, these encodings were computed through sin and cos operations.

The Transformer architecture has achieved state-of-the-art results on a range

of benchmark NLP datasets, including machine translation, question answering, and language modeling. It has also been adapted and extended for use in other domains, such as computer vision and speech recognition and synthesis [1]. The success of the Transformer has led to a renewed focus on self-attention mechanisms and has spurred further research in the area of NLP and deep learning.

As explained before, part of the wide success of this architecture is due to its non-recurrent nature, that allows for input vectorization and, thus, parallelized execution. This feature, although critical for speeding up the training procedure of various orders of magnitude with respect to classical sequence-to-sequence architectures (for example LSTMs or GRUs), makes real-world Transformers inherently not Turing-complete (the ability to loop is a required condition to simulate partial functions). In [PMB19], the authors thus considered an alternative version of the original architecture in order to overcome this (and the following) issue:

- The proof requires a system composed by one encoder and 3 decoders arranged sequentially. The output of the last component is then fed again as the input to the first decoder in order to build a loop in the computation.
- Since (analogously to [SS95]) in the paper they take into considerations only rational numbers of unbounded precision, they had to replace softmax attention with the hardmax function ($\sigma(x_i) = \frac{1}{m} \leftrightarrow x_i = \max(x)$, where m is the number of maximal values within x) and the positional encodings with an alternative mapping (where position i is encoded as $[1, (i+1), 1/(i+1), 1/(i+1)^2]$) explained in appendix B.4.2 of their supplementary material.

We will now provide a brief overlook of their inductive proof: our objective is, given a Turing machine $\mathcal{M}$ and an input string $w, |w| = n$, to use the system described above to show that any transition in the execution of $\mathcal{M}(w)$ can be simulated.

The encoder receives as input w and processes it through an embedding procedure and the addition with positional encodings. With the help of self-attention, it can trivially compute the a pair (K^e, V^e), where K^e is a vector of embeddings $k_1, ..., k_n$ that encode the positions on the string. V^e, on the other hand, contains information abouth both the index and the symbol contained in each cell of the tape.

The real simulation begins at the input of the first decoder, which uses the provided pair (K^e, V^e), along with an encoding of the pair (q_0, s_0) (the initial state and the symbol under the head at the beginning of the computation) as input to compute (q_1, s_1) within three steps. More generally, the triple of ordered decoders will exploit self-attention, residual connections and feed forward layers to simulate, given the sequence $(q_0, s_0), (q_1, s_1), ..., (q_{t-1}, s_{t-1}), (q_t, s_t)$, the

[1] To show a quick example of the practical capabilities of this architecture we hid a little *easter egg* within this essay: the abstract was actually generated by an AI model based on the Transformer architecture by asking it to "write an abstract for an article about the Turing completeness of neural networks in a catchy way"

transition to (q_{t+1}, s_{t+1}). This is done through 3 main steps (each one of them computed by a different decoder):

1. Given the above described sequence encoded as $y_0, ..., y_t$, we need first to augment it via positional encodings and then pass it through both an attention and a residual layer. This will allow us to process positional information to highlight relations between configurations at different steps, while keeping the positional encodings themselves for further computation. After a two layer feed forward network (described in detail in lemma B.2), this decoder outputs and embedding vector which encodes both q_{t+1}, v_{t+1} (the symbol written by the head before movement at timestamp $t+1$) and m_{t+1} (the movement at timestamp $t+1$). Effectively, this component is responsible of the implementation of the transition function δ of M.

2. The following decoder will use the previously computed information to compute the index of the head after the movement c_{t+1} by taking into consideration all the movements from the start of the computation ($\sum_{i \in \{0,...,t+1\}} m_i$ since $m_i \in -1, 0, 1 \forall i \in \{0, ..., t+1\}$). This is done, through the application of two attention layers and six feed forward neural networks (which construction is provided as additional lemmas in the supplementary material). The interesting part about this architecture is the use of residual connection to keep previously computed information intact during the computational flow: even if the data passes through multiple layers, information about q_i, v_i, m_i (for $i \in 1, ..., t$) and the relative positional encodings is kept through all the steps by carefully modifying only some coordinates of the embeddings.

3. Finally, the last decoder is able to compute s_{t+1} by calculating the last timestamp i during which the head was in position c_{t+1} and read the corrisponding written value v_i via the attention mechanism paired with three different parallel feed forward networks. After passing the computed data through a last neural layer, we can succesfully re-structure the data calculated as the embedding of (q_{t+1}, s_{t+1}).

Clearly, the new pair (q_{t+1}, s_{t+1}) is appended to the sequence $(q_0, s_0), ..., (q_t, s_t)$ and passed again through the first decoder. We can consider the execution as concluded when the embedding of a final state (q_{accept} or q_{reject}) is computed by the last decoder.

We encourage reading the supplementary material of [PMB19] to understand in detail the actual construction.

2.2.2 Models based on recurrent convolutions

Convolutional Neural Networks (CNNs) are a type of artificial neural network that have revolutionized the field of computer vision. CNNs are specifically designed to process images and other types of grid-like data by using a process called convolution, which involves sliding a set of filters over the input data to

extract features. Convolution is actually a mathematical operation that involves multiplying a small matrix of values, called a filter or kernel, with a portion of the input data and summing the results. By sliding the filter over the input data, the network can extract different features at different locations. The output of the convolutional layer is then fed into an activation function to introduce nonlinearity.

CNNs have been successful in a wide range of computer vision tasks, including image classification, object detection, facial recognition, and semantic segmentation. In recent years, CNNs have also been used in other fields, such as natural language processing, where they have been adapted to process sequential data, such as text, through the use of recurrent neural networks and attention mechanisms. In particular, some existing RNN architectures have been enhanced with the convolution capabilities to combine feature-extracting with sequence prediction. One example of this union is the **convolutional GRU** (CGRU), a recurrent network designed to process sequential data, such as time-series or video frames, by applying convolutions to capture spatial patterns and GRU units to capture temporal dependencies: while the convolutional layers are used to extract features from the input data, the GRU units are implemented to actually capture the temporal dependencies between these features.

In 2015, Kaiser and Sutskever [KS15] presented an alternative version of the convolutional GRU that allowed for fast parallel processing and (potentially) Turing complete capabilities: the **Neural GPU**. This architecture requires all the input sequence to be concatenated and encoded through an embedding matrix (similar to the input of the previously discussed Transformers) in order to avoid time-unfolding during the training phase: once the data is fed to the system, it is passed through a l-layer CGRU network without inputs, that exploits convolution to sequentially analyze the sequence. The state of the l^{th} CGRU is then transformed by an "inverse embedding" matrix, that provides the result of the computation. In the introductory paper, the authors showed that this architecture is able to recognise long-term dependencies in the data and to perform hard algorithmic tasks, such as long addition and multiplication, sequence copying, reversing and duplicating and counting by sorting bits. Because of the outstanding capabilities showed, the authors hypothesized that the Neural GPU's computational power is equivalent to cellular automatas', which are knowingly Turing complete [Coo04].

The expressive power of Neural GPUs is actually analyzed in the main article of this section, [PMB19]. The authors prove that a slight variant of this architecture, if given looping capabilities, is actually able to simulate any encoder-decoder recurrent neural network for sequence-to-sequence learning.

In particular, given an input $x, |x| = n$, an RNN encoder-decoder is defined by the following recursive equations:

$$h_i = \sigma(x_i W + h_{i-1} V) \tag{10}$$

$$g_t = \sigma(g_{t-1} U) \tag{11}$$

where W, V and U are weight matrices, $h_0 = \mathbf{0}$ is the input to the encoder and $g_0 = h_n$ is the input of the decoder.

Following the results provided by [SS95] (or, alternatively, [CS21]) we can conclude that this architecture is also Turing complete (always keeping in mind that the construction would require unbounded precision).

Proof sketch Before discussing an overview of the actual proof, it is important to note that the CGRU, during the update of its internal state, uses a bias vector of size directly proportional to the one of the input sequence, thus the Neural GPU can not be considered a fixed architecture. To solve this issue, the authors introduced the idea of a *uniform* Neural GPU, that is an analogous network such that for each bias $\mathbf{B} \in \mathbb{Q}^{n \times w \times d}$ there exists a matrix $B \in \mathbb{Q}^{w \times d}$ such that we have that $\mathbf{B}_i = B \ \forall i \in 1..n$. With this additional requirement we obtain a class of architectures that can be finitely specified (the number of parameters is constant, without depending on the length of the input).

The reduction of RNNs to Neural GPUs is quite long and cumbersome, but can be summarized as follows: given a encoder-decoder RNN (for language recognition) N, we simulate its execution through an ad-hoc constructed uniform Neural GPU, which multi-dimensional state is used for simulating the evolution of both the encoder and the decoder. To obtain the emulated state of these components at a timestamp t it's sufficient to project the Neural GPU's state $\mathbf{S}$ onto the t^{th} dimension: if the internal size of N is d, then the encoder state is encoded as $\mathbf{E}_t = \mathbf{S}_{t,1,1:d}$, while the decoders is embedded as $\mathbf{D}_t = \mathbf{S}_{t,1,d+1:2d}$ (using Python's standard slicing notation). By using convolutional kernels that space above multiple dimensions we can update $\mathbf{E}_t$ and $\mathbf{D}_t$ through a fixed number of CGRU applications on $\mathbf{S}_t$: at each timestamp t $\mathbf{E}_t$ is updated, while $\mathbf{E}_{t-1}$ is reset to 0 using the relative memory gate. This is necessary to ensure that each symbol of the input string is not processed by the convolutions more than once. On the other hand, vectors in $\mathbf{D}$ are never reset, thus they can keep being updated, which allows us to simulate an arbitrary long computation. The proof provided in the article shows that the architecture constructed allows to simulate N since, at each timestamp t, $\mathbf{E}_t$ equals to the output of the encoder h_t, while $\mathbf{D}_{n+t}$ is equivalent to the output of the decoder g_t. This discrepancy is needed because the update of $\mathbf{D}_t$ must delayed with respect to $\mathbf{E}_t$'s, since w must be completely processed before beginning the decoding part. While this simulation seems at first pretty straightforward, the construction of the actual kernels used, along with the actual proof of correctness of the emulation is quite long with a not-so-intuitive use of notation. For further understanding this argument we suggest to take a look at [PMB19]'s appendix C.

This proof (sketch) demonstrates that uniform Neural GPUs are capable of simulating any encoder-decoder RNN architecture. Theorem 2.3 of [PMB19] generalizes [SS95] result about RNN's computational power to the latter networks, thus, by transitive property, we can conclude that also Neural GPUs are Turing complete. Interestingly enough, the article by Pérez, Marinković

and Barceló also investigates how changing the way convolution kernel are constructed and slided over the state changes the power in computational power of the architecture:

- The construction in the proof requires *zero padding* to take into account filter overflow during convolution. If, on the other hand, we implemented *circular convolutions* that used the tensors stored at the first indexes of the input data to solve this issue, we would loose Turing completeness altogether as proved by the proposition 4.2 of [PMB19]. According to their article, "uniform Neural GPUs with circular convolutions cannot differentiate among period sequences of different length; in particular, they cannot check if a periodic input sequence is of even or odd length".
- The kernels used in the proof are of shape $(2, 1, d, d)$. These are the smallest kernels possible to achieve Turing completeness, since, with filters of shape $(1, 1, d, d)$, the value of a cell of $\mathbf{S}_t$ would only depend by the value of the same cell in $\mathbf{S}_{t-1}$.

3 Practical implementations of Turing-complete networks

In the previous part of this essay, we took into consideration ideal networks that were able to achieve Turing completeness through the use of unbounded precision or non-differentiable operations. To partially solve this problem, Google researchers published in 2014 an article [GWD14] that introduced a new perspective to the topic: instead of focusing on theoretical, but impractical universality, they tried to engineer a system able to simulate any memory-bound Turing machine. In the following part of this section, we will briefly discuss about the key concepts of this architecture, along with its later improved version, the Differentiable Neural Computer [GWR+16].

3.1 The Neural Turing Machine

Neural Turing Machines (NTMs) are a type of neural network architecture introduced in [GWD14] that incorporates an external memory bank to enable the system to perform complex computations that are beyond the capabilities of standard neural networks.

At a high level, an NTM consists of two main components: a neural network that is responsible for processing input data and producing output and a memory bank that stores structured data in a way that can be accessed and manipulated by the network. The memory bank is usually implemented as a large, two-dimensional array of memory cells, with each cell storing a vector of data. The network can read from and write to the memory bank using one or more read and write heads, which move across the array and interact with the data stored in each cell.

The external memory module is accessed both in writing and reading operations at every timestamp through differentiable operations: during each iteration t, a set of vectors $[k_t, \beta_t, g_t, s_t, \gamma_t]$ is output by the network controller.

- The first two values are used for content-based addressing of the memory matrix: the *content-addressing weighting* is computed as

$$w_t^c[i] = \frac{\exp(\beta_t K(k_t, M_t[i]))}{\sum_j \exp(\beta_t K(k_t, M_t[j]))} \tag{12}$$

  In the previous equation, we have that each row of the matrix $(M_t(i))$ is compared with the key k_t with the cosine similarity $(K(u, v) = \frac{u \cdot v}{||u|| ||v||})$. This measure is then passed through the softmax function to determine a probabilistic distribution over the memory cells. Note the use of β: this parameter can be altered to change the concentration of the distribution in order to have more or less exclusive access to the locations of the external module.

- g_t is used to determine an interpolation between a weighting calculated during the previous timestamp and the freshly computed content-addressing weighting

$$w_t^g = g_t w_t^c + (1 - g_t) w_{t-1} \tag{13}$$

  Note that, our objective will be to determine the current timestamp's weighting w_t in order to define the memory accessing operations.

- Finally, s_t is a vector used to define shift weightings: a vector of probabilities that allows to determine the degree of left or right shifts in the read and write operations of the memory matrix. The size k of the shift weighting vector s_t determines the degree of shift that the controller can apply to the memory matrix. In practice, a temporary weighting is computed as

$$\tilde{w}_t[i] = \sum_{j=0}^{N-1} w_t^g[j] s_t[i - j] \tag{14}$$

  where all index arithmetic is calculated modulo N, which is the number of locations in the memory matrix. In conclusion, the final weighting is calculated as a probability distribution through the use of a softmax application:

$$w_t[i] = \frac{\tilde{w}_t[i]^{\gamma_t}}{\sum_j \tilde{w}_t[j]^{\gamma_t}} \tag{15}$$

Note that the transformation applied could be slighty "blurry" over the locations. To overcome this, the scalar γ_t can be used to sharpen the final weighting (analogously to β_t in the previous point).

This weighting is then used both in reading and writing: the output of the system is given by a convex combination of the rows in memory weighted by w_t

$$r_t = \sum_i w_t[i] M_t[i] \tag{16}$$

while the update of the matrix is just a generalization of the **LSTM**'s long-term-memory update mechanism:

$$\tilde{M}_t[i] = M_{t-1}[i](\mathbf{1} - w_t[i]e_t) \tag{17}$$

$$M_t[i] = \tilde{M}_t[i] + w_t[i]a_t \tag{18}$$

where e_t and a_t are the *erase* and *add vector* output by the controller.

One of the key advantages of NTMs is that they can learn to store and retrieve information over multiple steps, allowing them to perform tasks that require access to structured, external data. For example, an NTM can be trained to perform handwriting recognition by receiving a sequence of input images representing a handwritten word, and then storing the sequence of images in memory. The network can then use the read and write heads to access the stored images and generate a transcription of the word.

Although the Neural Turing Machine represented a huge step ahead in the research field of memory-augmented networks, its capabilities are limited by its basic memory accessing operations: as the network iterates through consecutive locations, sequential information is preserved during reading. However, once the write head jumps to a different part of the memory using content addressing, the read head cannot recover the order of writes that occurred before and after the jump. In addition, the NTM lacks a mechanism to guarantee that allocated memory blocks do not overlap and cause interference. Lastly, this architecture does not provide an access method for freeing previously written locations and, as a result, cannot reuse memory when processing extended sequences..

3.2 The Differentiable Neural Computer

To overcome the above issues, in 2016 the same team of the previous article published [GWR+16], where they introduced an extended version of the NTM capable of better exploiting differentiable memory accessing methods to further generalize its learning capabilities. This new architecture was called the **Differentiable Neural Computer (DNC)** and inherited many ideas from its previous iteration, first of all the separated controller-memory bank structure. Again, in DNCs a neural network is used to compute from the input sequence a set of tensors (called the *interface vector*) that are used to access the external module through (mostly) differentiable operations. In addition to the interface

vector, the controller also computes a raw output tensor that is merged with the data retrieved from the memory at the same timestamp to produce the final network output.

In a DNC, memory is addressed through separate read and write weightings. In particular, in the paper they propose to have R different read heads and a single write head, that produce respectively R *read vectors* $r_t^1, ..., r_t^R$ and an updated memory matrix M_t from the previous timestamp's M_{t-1}. The external module is rewritten precisely in the same way of the NTM:

$$M_t = M_{t-1} \circ (E - w_t^w e_t^T) + w_t v_t^T \tag{19}$$

where $\circ$ is the pointwise multiplication, E is a matrix of ones of the same shape of M_{t-1} and e_t and v_t are, respectively, the *erase* and *write* vectors output by the controller. What changes with respect to the NTM is how w_t is computed. Memory is addressed in writing in two main modes:

1. Through content-based addressing (using the same exact mechanism as described before)

2. Through dynamic memory allocation. This procedure requires the computation of multiple intermediate vectors: at first, we have to define at each timestamp t the *memory retention vector* ψ_t, which represents how much of each location will be preserved by the R free gates f_t^i output in the interface vector

$$\psi_t = \prod_{i=1}^{R} (\mathbf{1} - f_t^i w_{t-1}^{r,i}) \tag{20}$$

where $w_{t-1}^{r,i}$ is the read weighting for the i^{th} head at the previous timestamp. ψ_t can be thus used to define the *usage vector*:

$$u_t = (u_{t-1} + w_{t-1}^w - u_{t-1} \circ w_{t-1}^w) \circ \psi_t \tag{21}$$

Please note that locations are defined used only if they have been preserved by the free gates ($\psi_t[i] \approx 1$) and are either already in use or have just been written to (w_{t-1}^w is the write weighting of the previous timestamp). Finally, u_t's indexes are sorted by ascending order of usage in the vector ϕ_t and this tensor is used to determine the *allocation weighting* a_t as:

$$a_t[\phi_t[j]] = (1 - u_t[\phi_t[j]]) \prod_{i=1}^{j-1} u_t[\phi_t[i]] \tag{22}$$

It's easy to convince ourselves that this vector can be used a continuous version of the free-list allocation scheme: if there exists an address l that represents a particularly allocable memory location, its usage will be low

(thus $l = \phi[1]$ and $1 - u_t[\phi_t[1]] \approx 1$) and the product will be empty, so $a_t[\phi_t[1]] \approx 1$. Even if there are other locations with a low usage, their allocability will be low since $u_t[\phi_t[1]] \approx 0$ will appear in the product. As a consequence, there will be at most one location with high allocability, while the rest of the vector will only contain small values: by calculating an outer product between this tensor and a write vector we will practically access only the allocable location, leaving all the others almost unmodified.

The network decides which of these two mechanism should be used by interpolating a_t with a content-based weighting c_t^w:

$$w_t^w = g_t^w(g_t^a a_t + (1 - g_t^a)c_t^w) \tag{23}$$

where g_t^a is used to discriminate between the two access modes, while g_t^w allows to the architecture to protect the memory from unnecessary updates. Note that both g_t^a and g_t^w are part of the interface vector.

Similarly, memory can be addressed in two main modes in reading mode:

1. Again, through content-based-addressing.
2. By a *memory linkage matrix*. We want to define at each timestamp a squared matrix L_t that has a row and a column for each location of the memory bank: we will use it to keep track of consecutively modified memory locations. At first, we have to recursively define the *precedence weighting* p_t as

$$p_0 = \mathbf{0} \tag{24}$$

$$p_t = (1 - \sum_i w_t^w[i])p_{t-1} + w_t^w \tag{25}$$

Intuitively, $p_t[i]$ represents the degree to which location i was the last one written to. This vector is used to define L_t as

$$L_0[i, j] = 0 \tag{26}$$

$$L_t[i, i] = 0 \tag{27}$$

$$L_t[i, j] = (1 - w_t^w[i] - w_t^w[j])L_{t-1}[i, j] + w_t^w[i]p_{t-1}[j] \tag{28}$$

We want to highlight a few things: first of all, L_0 only contains null values because there is no actual writing to keep track of. Secondly, the elements in the main diagonal are zeros because it's unclear how to follow the link from a location to itself. Finally, a cell is updated only in the case that at least one of its coordinates is being currently written to and the update is proportional to both the current and the last writing intensities.

Having computed L_t, we can compute the *forward* and *backward weightings* f_t^i and b_t^i for each read head i by multiplying L_t (or its transpose) by the previous read weighting $w_{t-1}^{r,i}$:

$$f_t^i = L_t w_{t-1}^{r,i} \tag{29}$$

$$b_t^i = L_t^T w_{t-1}^{r,i} \tag{30}$$

We can finally combine this weightings with a content-based one computed previously $c_t^{r,i}$ by computing a convex combination with a probability distribution vector $\pi^i \in [0,1]^3$:

$$w_t^{r,i} = \pi^i[1] b_t^i + \pi^i[2] c_t^{r,i} + \pi^i[3] f_t^i \tag{31}$$

The vector π^i allows, for each head, to discriminate between the memory addressing modes while reading.

Finally, we can use these weightings to access the memory and compute the read vectors $r_t^i = M_t^T w_t^{r,i}$. These tensors will be combined with the raw output v_t of the controller to determine the timestamp's network's output y_t:

$$y_t = v_t + W_r[r_t^1, ..., r_t^R] \tag{32}$$

where W_r is a learned parameter of the system.

The described architecture is almost completely differentiable (except for the sorting operation required to calculate the allocation equation 22, which did not exhibit any apparent issues during empirical evaluations), thus can be trained through standard backpropagation. The implementation of the DNC by Google researchers demonstrated its ability to generalize problems and learn from multiple tasks independently before combining them in a seamless manner: for example, it has been successfully trained on NLP and graph traversal and was then able to answer to questions like "Who is the first cousin of x?" after being given a written description of the family tree.

4 Conclusions

In this essay, we dealt with different perspectives about the expressive power of neural networks: at first, we discussed about the more theoretical results recently obtained about the Turing completeness of unpracticable RNNs. Then we showed how these proofs could be extended to newer and (apparently) more specific architectures currently used for sequence-to-sequence learning, such as the Transformer and the Neural GPU. Since these results did not provide any practical implementation of such networks, we finally explained the latest research efforts done to provide a truly universal artificial intelligence, capable of

combining both sub-symbolic (the neural network controller) and symbolic (the actual Turing machine simulated) capabilities.

In the future, we can expect to see more research towards new training methods or differentiable alternatives to stacks that would allow for architectures like the one presented in section 2.1.2 to be actually implemented. Similarly, by developing a parallelizable version of the DNC we would obtain an easily-trainable but incredibly powerful machine that would push the boundary towards a truly general AI.

References

[Coo04] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[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.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. In *Advances in Neural Information Processing Systems*, pages 2621–2629, 2014.

[GWR+16] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[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.

[KS15] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms, 2015.

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

[PMB19] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures, 2019.

[Pol87] James B Pollack. *On Connectionist Models of Natural Language Processing*. PhD thesis, Computer Science Department, University of Illinois, Urbana, 1987.

[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.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.