ON THE TURING COMPLETENESS OF MODERN NEURAL NETWORK ARCHITECTURES

Anonymous authors
Paper under double-blind review

ABSTRACT

Alternatives to recurrent neural networks, in particular, architectures based on *attention* or *convolutions*, have been gaining momentum for processing input sequences. In spite of their relevance, the computational properties of these alternatives have not yet been fully explored. We study the computational power of two of the most paradigmatic architectures exemplifying these mechanisms: the *Transformer* (Vaswani et al., 2017) and the *Neural GPU* (Kaiser & Sutskever, 2016). We show both models to be Turing complete exclusively based on their capacity to compute and access internal dense representations of the data. In particular, neither the Transformer nor the Neural GPU requires access to an external memory to become Turing complete. Our study also reveals some minimal sets of elements needed to obtain these completeness results.

1 INTRODUCTION

There is an increasing interest in designing neural network architectures capable of learning algorithms from examples (Graves et al., 2014; Grefenstette et al., 2015; Joulin & Mikolov, 2015; Kaiser & Sutskever, 2016; Kurach et al., 2016; Dehghani et al., 2018). A key requirement for any such an architecture is thus to have the capacity of implementing arbitrary algorithms, that is, to be *Turing complete*. Turing completeness often follows for these networks as they can be seen as a control unit with access to an unbounded memory; as such, they are capable of simulating any Turing machine.

On the other hand, the work by Siegelmann & Sontag (1995) has established a different way of looking at the Turing completeness of neural networks. In particular, their work establishes that recurrent neural networks (RNNs) are Turing complete even if only a bounded number of resources (i.e., neurons and weights) is allowed. This is based on two conditions: (1) the ability of RNNs to compute *internal* dense representations of the data, and (2) the mechanisms they use for accessing such representations. Hence, the view proposed by Siegelmann & Sontag shows that it is possible to release the full computational power of RNNs without arbitrarily increasing its model complexity.

Most of the early neural architectures proposed for learning algorithms correspond to extensions of RNNs – e.g., Neural Turing Machines (Graves et al., 2014) –, and hence they are Turing complete in the sense of Siegelmann & Sontag. However, a recent trend has shown the benefits of designing networks that manipulate sequences but do not directly apply a recurrence to sequentially process their input symbols. Architectures based on *attention* or *convolutions* are two prominent examples of this approach. In this work we look at the problem of Turing completeness à la Siegelmann & Sontag for two of the most paradigmatic models exemplifying these features: the *Transformer* architecture (Vaswani et al., 2017) and the *Neural GPU* (Kaiser & Sutskever, 2016).

The main contribution of our paper is to show that the Transformer and the Neural GPU are Turing complete exclusively based on their capacity to compute and access internal dense representations of the data. In particular, neither the Transformer nor the Neural GPU requires access to an external memory to become Turing complete. We prove both results by simulating standard *sequence-to-sequence* RNNs. Our study also reveals some minimal sets of elements needed to obtain these completeness results. This might be of help in understanding how such elements can be exploited to obtain full computational power at a minimal computational cost.

Background work The study of the computational power of neural networks can be traced back to McCulloch & Pitts (1943) which established an analogy between neurons with hard-threshold

activations and first order logic sentences, and Kleene (1956) that draw a connection between neural networks and finite automata. As mentioned earlier, the first work showing the Turing completeness of finite neural networks with linear connections was carried out by Siegelmann & Sontag (1992; 1995). Since being Turing complete does not ensure the ability to actually learn algorithms in practice, there has been an increasing interest in enhancing RNNs with mechanisms for supporting this task. One strategy has been the addition of inductive biases in the form of external memory, being the *Neural Turing Machine* (NTM) (Graves et al., 2014) a paradigmatic example. To ensure that NTMs are differentiable, their memory is accessed via a soft attention mechanism (Bahdanau et al., 2014). Other examples of architectures that extend RNNs with memory are the Stack-RNN (Joulin & Mikolov, 2015), and the (De)Queue-RNNs (Grefenstette et al., 2015). By Siegelmann & Sontag's results, all these architectures are Turing complete.

The Transformer architecture (Vaswani et al., 2017) is almost exclusively based on the attention mechanism, and it has achieved state of the art results on many language-processing tasks. While not initially designed to learn general algorithms, a parallel study has advocated the need for enriching its architecture with several new features as a way to learn general procedures in practice (Dehghani et al., 2018). This enrichment is motivated by the empirical observation that the original Transformer architecture struggles to generalize to input of lengths not seen during training. We, in contrast, show that the original Transformer architecture is Turing complete, based on different considerations. These results do not contradict each other, but show the differences that may arise between theory and practice. In fact, we think that both approaches can be complementary as our theoretical results can shed light on what are the intricacies of the original architecture, which aspects of it are candidates for change or improvement, and which others are strictly needed. For instance, our proof uses *additive hard attention* while the Transformer is often trained with *multiplicative soft attention* (Vaswani et al., 2017). See Section 3.3 for a discussion on these differences.

The Neural GPU is an architecture that mixes convolutions and *gated recurrences* over tridimensional tensors. It has been shown that NeuralGPUs are powerful enough to learn decimal multiplication from examples (Freivalds & Liepins, 2018), being the first neural architecture capable of solving this problem end-to-end. The similarity of Neural GPUs and cellular automata has been used as an argument to state the Turing completeness of the architecture (Kaiser & Sutskever, 2016; Price et al., 2016). Cellular automata are Turing complete (Smith III, 1971; Ollinger, 2012) and their completeness is established assuming an unbounded number of cells. In the Neural GPU architecture, in contrast, the number of cells that can be used during a computation is proportional to the size of the input sequence (Kaiser & Sutskever, 2016). One can cope with the need for more cells by padding the Neural GPU input with additional (dummy) symbols, as much as needed for a particular computation. Nevertheless, this is only a partial solution, as for a Turing-complete model of computation, one cannot decide a priori how much memory is needed to solve a particular problem. Our results in this paper are somehow orthogonal to the previous argument; we show that one can leverage the dense representations of the Neural GPU cells to obtain Turing completeness without requiring to add cells beyond the ones used to store the input.

2 PRELIMINARIES

We assume all weights to be rational numbers in $\mathbb{Q}$. Moreover, we only allow the use of rational functions with rational coefficients. Most of our positive results make use of the *piecewise-linear sigmoidal activation* function $\sigma : \mathbb{Q} \to \mathbb{Q}$, which is defined as

$$\sigma(x) = \begin{cases} 0 & x < 0, \\ x & 0 \leq x \leq 1, \\ 1 & x > 1. \end{cases}$$

We are mostly interested in *sequence-to-sequence* (seq-to-seq) neural network architectures that we next formalize. A seq-to-seq network N receives as input a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ of vectors $\boldsymbol{x}_i \in \mathbb{Q}^d$, for some $d > 0$, and produces as output a sequence $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m)$ of vectors $\boldsymbol{y}_i \in \mathbb{Q}^d$. Most of these types of architectures require a *seed* vector $\boldsymbol{s}$ and some stopping criterion for determining the length of the output. The latter is usually based on the generation of a particular output vector called an *end of sequence* mark. In our formalization instead, we allow a network to produce a fixed number $r \geq 0$ of output vectors. Thus, for convenience we see a general seq-to-seq network as a function N such that the value $N(\boldsymbol{X}, \boldsymbol{s}, r)$ corresponds to an output sequence of

the form $\boldsymbol{Y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_r)$. With this definition, we can view every seq-to-seq network as a *language recognizer* of strings as follows.

Definition 2.1. *A seq-to-seq language recognizer is a tuple $A = (\Sigma, f, N, \boldsymbol{s}, \mathbb{F})$, where Σ is a finite alphabet, $f : \Sigma \to \mathbb{Q}^d$ is an* embedding function*, N is a seq-to-seq network, $\boldsymbol{s} \in \mathbb{Q}^d$ is a* seed vector*, and $\mathbb{F} \subseteq \mathbb{Q}^d$ is a set of* final vectors*. We say that A* accepts the string *$w \in \Sigma^*$, if there exists an integer $r \in \mathbb{N}$ such that $N(f(w), \boldsymbol{s}, r) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_r)$ and $\boldsymbol{y}_r \in \mathbb{F}$. The language accepted by A, denoted by $L(A)$, is the set of all strings accepted by A.*

We impose two additional restrictions over recognizers. The embedding function $f : \Sigma \to \mathbb{Q}^d$ should be computed by a Turing machine in time linear w.r.t. the size of Σ. This covers the two most typical ways of computing input embeddings from symbols: the *one-hot encoding*, and embeddings computed by fixed feed-forward networks. Moreover, the set $\mathbb{F}$ should also be recognizable in linear-time; given a vector $\boldsymbol{f}$, the membership $\boldsymbol{f} \in \mathbb{F}$ should be decided by a Turing machine working in linear time with respect to the size (in bits) of $\boldsymbol{f}$. This covers the usual way of checking equality with a fixed end-of-sequence vector. We impose these restrictions to disallow the possibility of *cheating* by encoding arbitrary computations in the input embedding or the stop condition, while being permissive enough to construct meaningful embeddings and stoping criterions.

Finally, a class $\mathcal{N}$ of seq-to-seq neural network architectures defines the class $\mathcal{L}_{\mathcal{N}}$ composed of all the languages accepted by language recognizers that use networks in $\mathcal{N}$. From these notions, the formalization of Turing completeness of a class $\mathcal{N}$ naturally follows.

Definition 2.2. *A class $\mathcal{N}$ of seq-to-seq neural network architectures is Turing Complete if $\mathcal{L}_{\mathcal{N}}$ is exactly the class of languages recognized by Turing machines.*

Given an input sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, a seed vector $\boldsymbol{y}_0$, and $r \in \mathbb{N}$, an *encoder-decoder* RNN is given by the following two recursions

$$\boldsymbol{h}_0 = \boldsymbol{0}, \qquad \boldsymbol{h}_i = f_1(\boldsymbol{x}_i \boldsymbol{W} + \boldsymbol{h}_{i-1} \boldsymbol{V} + \boldsymbol{b}_1) \quad (\text{with } 1 \leq i \leq n) \tag{1}$$

$$\boldsymbol{g}_0 = \boldsymbol{h}_n, \qquad \boldsymbol{g}_t = f_2(\boldsymbol{g}_{t-1} \boldsymbol{U} + \boldsymbol{y}_{t-1} \boldsymbol{R} + \boldsymbol{b}_2), \qquad \boldsymbol{y}_t = O(\boldsymbol{g}_t) \quad (\text{with } 1 \leq t \leq r) \tag{2}$$

where $\boldsymbol{V}, \boldsymbol{W}, \boldsymbol{U}, \boldsymbol{R}$ are matrices, $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are vectors, $O(\cdot)$ is an output function, and f_1 and f_2 are activations functions. Equation (1) is called the RNN-*encoder* and (2) the RNN-*decoder*.

The next Theorem follows by inspection of the proof by Siegelmann & Sontag (1992; 1995) after adapting it to our formalization of encoder-decoder RNNs.

Theorem 2.3 (Siegelmann & Sontag (1992; 1995))**.** *The class of encoder-decoder RNNs is Turing complete. Turing completeness holds even if we restrict to the class in which $\boldsymbol{R}$ is the zero matrix, $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are the zero vector, $O(\cdot)$ is the identity function, and f_1 and f_2 are the piecewise-linear sigmoidal activation σ.*

3 THE TRANSFORMER ARCHITECTURE

In this section we present a formalization of the Transformer architecture (Vaswani et al., 2017), abstracting away from specific choices of functions and parameters. Our formalization is not meant to produce an efficient implementation of the Transformer, but to provide a simple setting over which its mathematical properties can be established in a clean way.

The Transformer is heavily based on the attention mechanism introduced next. Consider a scoring function $\text{score} : \mathbb{Q}^d \times \mathbb{Q}^d \to \mathbb{Q}$ and a normalization function $\rho : \mathbb{Q}^n \to \mathbb{Q}^n$, for $d, n > 0$. Let us assume that $\boldsymbol{q} \in \mathbb{Q}^d$, and that $\boldsymbol{K} = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n)$ and $\boldsymbol{V} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$ are tuples of elements in $\mathbb{Q}^d$. A *$\boldsymbol{q}$-attention* over $(\boldsymbol{K}, \boldsymbol{V})$, denoted by $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})$, is a vector $\boldsymbol{a} \in \mathbb{Q}^d$ defined as follows.

$$(s_1, \ldots, s_n) = \rho(\text{score}(\boldsymbol{q}, \boldsymbol{k}_1), \text{score}(\boldsymbol{q}, \boldsymbol{k}_2), \ldots, \text{score}(\boldsymbol{q}, \boldsymbol{k}_n)) \tag{3}$$

$$\boldsymbol{a} = s_1 \boldsymbol{v}_1 + s_2 \boldsymbol{v}_2 + \cdots + s_n \boldsymbol{v}_n. \tag{4}$$

Usually, $\boldsymbol{q}$ is called the *query*, $\boldsymbol{K}$ the *keys*, and $\boldsymbol{V}$ the *values*. We do not pose any restriction on the scoring and normalization functions, as some of our results hold in general. We only require the normalization function to satisfy that there is a function f_ρ from $\mathbb{Q}$ to $\mathbb{Q}^+$ such that for each $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Q}^n$ it is the case that the i-th component $\rho_i(\boldsymbol{x})$ of $\rho(\boldsymbol{x})$ is equal to $f_\rho(x_i) / \sum_{j=1}^n f_\rho(x_j)$. Thus, $\boldsymbol{a}$ in Equation (4) is a convex combination of the vectors in $\boldsymbol{V}$.

When proving possibility results, in particular Turing completeness, we will need to pick specific scoring and normalization functions. A usual choice for the scoring function is a feed forward network with input $(\boldsymbol{q}, \boldsymbol{k}_i)$ and a single neuron as output (Bahdanau et al., 2014), sometimes called *additive attention*. Another possibility is to use the dot product $\langle \boldsymbol{q}, \boldsymbol{k}_i \rangle$ (Vaswani et al., 2017), called *multiplicative attention*. For the normalization function, $\mathrm{softmax}$ is a standard choice. Nevertheless, in our possibility proofs we mostly use the $\mathrm{hardmax}$ function, which is obtained by setting $f_{\mathrm{hardmax}}(x_i) = 1$ if x_i is the maximum value, and $f_{\mathrm{hardmax}}(x_i) = 0$ otherwise. Thus, for a vector $\boldsymbol{x}$ in which the maximum value occurs r times, we have that $\mathrm{hardmax}_i(\boldsymbol{x}) = \frac{1}{r}$ if x_i is the maximum value of $\boldsymbol{x}$, and $\mathrm{hardmax}_i(\boldsymbol{x}) = 0$ otherwise. We call it *hard attention* whenever $\mathrm{hardmax}$ is used as normalization function. As customary, if $F : \mathbb{Q}^d \to \mathbb{Q}^d$ and $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$, with $\boldsymbol{x}_i \in \mathbb{Q}^d$, we write $F(\boldsymbol{X})$ for the sequence $(F(\boldsymbol{x}_1), \ldots, F(\boldsymbol{x}_n))$.

Transformer Encoder and Decoder A *single-layer encoder* of the Transformer is a parametric function $\mathrm{Enc}(\boldsymbol{X}; \boldsymbol{\theta})$ receiving a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ of vectores in $\mathbb{Q}^d$ and returning a sequence $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ of the same length of vectors in $\mathbb{Q}^d$. In general, we consider the parameters in $\mathrm{Enc}(\boldsymbol{X}; \boldsymbol{\theta})$ as functions $Q(\cdot), K(\cdot), V(\cdot)$, and $O(\cdot)$, all of them from $\mathbb{Q}^d$ to $\mathbb{Q}^d$. The single-layer encoder is then defined as follows.

$$\boldsymbol{a}_i = \mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) + \boldsymbol{x}_i \tag{5}$$

$$\boldsymbol{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i \tag{6}$$

In practice $Q(\cdot)$, $K(\cdot)$, $V(\cdot)$ are typically matrix multiplications, and $O(\cdot)$ a feed-forward network. The $+\ \boldsymbol{x}_i$ and $+\ \boldsymbol{a}_i$ summands are usually called *residual connections* (He et al., 2016a;b). When the particular functions used as parameters are not important, we simply write $\boldsymbol{Z} = \mathrm{Enc}(\boldsymbol{X})$.

The Transformer *encoder* is defined simply as the repeated application of single-layer encoders (with independent parameters), plus two final transformation functions $K(\cdot)$ and $V(\cdot)$ applied to every vector in the output sequence of the final layer. Thus the L-layer Transformer encoder is defined by the following recursion (with $1 \leq \ell \leq L-1$ and $\boldsymbol{X}^1 = \boldsymbol{X}$).

$$\boldsymbol{X}^{\ell+1} = \mathrm{Enc}(\boldsymbol{X}^\ell; \boldsymbol{\theta}_\ell), \quad \boldsymbol{K} = K(\boldsymbol{X}^L), \quad \boldsymbol{V} = V(\boldsymbol{X}^L). \tag{7}$$

We use $(\boldsymbol{K}, \boldsymbol{V}) = \mathrm{TEnc}_L(\boldsymbol{X})$ to denote an L-layer Transformer encoder over the sequence $\boldsymbol{X}$.

A *single-layer decoder* is similar to a single-layer encoder but with additional attention to an external pair of key-value vectors $(\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$. The input for the single-layer decoder is a sequence $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_k)$ plus the external pair $(\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$, and the output is a sequence $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k)$. This layer is also parameterized by four functions $Q(\cdot)$, $K(\cdot)$, $V(\cdot)$ and $O(\cdot)$ and is defined as follows.

$$\boldsymbol{p}_i = \mathrm{Att}(Q(\boldsymbol{y}_i), K(\boldsymbol{Y}), V(\boldsymbol{Y})) + \boldsymbol{y}_i \tag{8}$$

$$\boldsymbol{a}_i = \mathrm{Att}(\boldsymbol{p}_i, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) + \boldsymbol{p}_i \tag{9}$$

$$\boldsymbol{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i \tag{10}$$

Notice that the first (self) attention over $(K(\boldsymbol{Y}), V(\boldsymbol{Y}))$ is used to generate a query $\boldsymbol{p}_i$ to attend the external pair $(\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$. We denote this single-decoder layer by $\mathrm{Dec}((\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}), \boldsymbol{Y}; \boldsymbol{\theta})$.

The Transformer *decoder* is a repeated application of single-layer decoders, plus a transformation function $F : \mathbb{Q}^d \to \mathbb{Q}^d$ which is applied over the final vector of the decoded sequence. Thus, the output of the decoder is a single vector $\boldsymbol{z} \in \mathbb{Q}^d$, and not a sequence of them as in the case of the encoder. Formally, the L-layer Transformer decoder is defined as

$$\boldsymbol{Y}^{\ell+1} = \mathrm{Dec}((\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}), \boldsymbol{Y}^\ell; \boldsymbol{\theta}_\ell), \quad \boldsymbol{z} = F(\boldsymbol{y}_k^L) \qquad (1 \leq \ell \leq L-1 \text{ and } \boldsymbol{Y}^1 = \boldsymbol{Y}). \tag{11}$$

We use $\boldsymbol{z} = \mathrm{TDec}_L((\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}), \boldsymbol{Y})$ to denote an L-layer Transformer decoder.

The complete Tansformer A *Transformer network* receives an input sequence $\boldsymbol{X}$, a seed vector $\boldsymbol{y}_0$, and a value $r \in \mathbb{N}$. Its output is a sequence $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_r)$ defined as

$$\boldsymbol{y}_{t+1} = \mathrm{TDec}(\mathrm{TEnc}(\boldsymbol{X}), (\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_t)), \qquad \text{for } 0 \leq t \leq r-1. \tag{12}$$

We denote the output sequence of the transformer as $\boldsymbol{Y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_r) = \mathrm{Trans}(\boldsymbol{X}, \boldsymbol{y}_0, r)$.

3.1 INVARIANCE UNDER PROPORTIONS

The Transformer, as defined above, is *order-invariant*: two input sequences that are permutations of each other produce exactly the same output. This is a consequence of the following straightforward property of the attention function: if $\boldsymbol{K} = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n)$, $\boldsymbol{V} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$, and $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a permutation, then $\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{Att}(\boldsymbol{q}, \pi(\boldsymbol{K}), \pi(\boldsymbol{V}))$ for every query $\boldsymbol{q}$. This weakness has motivated the need for including information about the order of the input sequence by other means; in particular, this is often achieved by using the so-called *positional encodings* (Vaswani et al., 2017; Shaw et al., 2018), which we study below.

But before going into positional encodings, a natural question is what languages the Transformer can recognize without them. As a standard yardstick we use the well-studied class of regular languages, i.e., languages recognized by finite automata. Order-invariance implies that not every regular language can be recognized by a Transformer network. As an example, there is no Transformer network that can recognize the regular language $(ab)^*$, as the latter is not order-invariant. A reasonable question then is whether the Transformer can express all regular languages which are order-invariant. It is possible to show that this is not the case by proving that the Transformer actually satisfies a stronger invariance property, which we call *proportion invariance*.

For a string $w \in \Sigma^*$ and a symbol $a \in \Sigma$, we use $\mathrm{prop}(a, w)$ to denote the ratio between the number of times that a appears in w and the length of w. Consider now the set $\mathrm{PropInv}(w) = \{u \in \Sigma^* \mid \mathrm{prop}(a, w) = \mathrm{prop}(a, u) \text{ for every } a \in \Sigma\}$. Then:

Proposition 3.1. *Let* Trans *be a Transformer,* $\boldsymbol{s}$ *a seed,* $r \in \mathbb{N}$*, and* $f : \Sigma \to \mathbb{Q}^d$ *an embedding function. Then* $\mathrm{Trans}(f(w), \boldsymbol{s}, r) = \mathrm{Trans}(f(u), \boldsymbol{s}, r)$*, for each* $u, w \in \Sigma^*$ *with* $u \in \mathrm{PropInv}(w)$*.*

As an immediate corollary we obtain the following.

Corollary 3.2. *Consider the order-invariant regular language* $L = \{w \in \{a, b\}^* \mid w \text{ has an even number of } a \text{ symbols}\}$*. Then* L *cannot be recognized by a Transformer network.*

On the other hand, languages recognized by Transformer networks are not necessarily regular.

Proposition 3.3. *There is a Transformer network that recognizes the non-regular language* $S = \{w \in \{a, b\}^* \mid w \text{ has strictly more symbols } a \text{ than symbols } b\}$*.*

That is, the computational power of Transformer networks without positional encoding is both rather weak (they do not even contain order-invariant regular languages) and not so easy to capture (as they can express counting properties that go beyond regularity). As we show in the next section, the inclusion of positional encodings radically changes the picture. In fact, with such positional encodings the Tranformer is able to recognize all computable languages.

3.2 POSITIONAL ENCODINGS AND COMPLETENESS OF THE TRANSFORMER

Order-invariance states that Transformer networks lack information about how the input is ordered. Positional encodings come to remedy this issue by providing information about the absolute positions of the symbols in the input. A positional encoding is just a function $\mathrm{pos} : \mathbb{N} \to \mathbb{Q}^d$. Function pos combined with an embedding function $f : \Sigma \to \mathbb{Q}^d$ give rise to a new embedding function $f_{\mathrm{pos}} : \Sigma \times \mathbb{N} \to \mathbb{Q}^d$ such that $f_{\mathrm{pos}}(a, i) = f(a) + \mathrm{pos}(i)$. Thus, given an input string $w = a_1 a_2 \cdots a_n \in \Sigma$, the result of the embedding function $f_{\mathrm{pos}}(w)$ provides a "new" input

$$\big(f_{\mathrm{pos}}(a_1, 1), f_{\mathrm{pos}}(a_2, 2), \ldots, f_{\mathrm{pos}}(a_n, n)\big)$$

to the Transformer encoder. Similarly, the Transformer decoder instead of receiving the sequence $\boldsymbol{Y} = (\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_t)$ as input, it receives now the sequence

$$\boldsymbol{Y}' \;=\; \big(\boldsymbol{y}_0 + \mathrm{pos}(1), \boldsymbol{y}_1 + \mathrm{pos}(2), \ldots, \boldsymbol{y}_t + \mathrm{pos}(t+1)\big)$$

As for the case of the embedding functions, we require the positional encoding $\mathrm{pos}(i)$ to be computable by a Turing machine working in linear time w.r.t. the size (in bits) of i.

The main result of this section is the completeness of Transformers with positional encodings.

Theorem 3.4. *The class of Transformer networks with positional encodings is Turing complete.*

Proof Sketch. For the sake of space we only present a sketch of the proof. The full proof can be found in the appendix. Let $A = (\Sigma, f, N, \boldsymbol{s}, \mathbb{F})$ be an RNN encoder-decoder language recognizer, such that N is of dimension d and its encoder and decoder are defined by the equations $\boldsymbol{h}_i = \sigma(\boldsymbol{x}_i\boldsymbol{W} + \boldsymbol{h}_{i-1}\boldsymbol{V})$ and $\boldsymbol{g}_t = \sigma(\boldsymbol{g}_{t-1}\boldsymbol{U})$, respectively, where $\boldsymbol{g}_0 = \boldsymbol{h}_n$ and n is the length of the input. We explain how to construct a Transformer network $A' = (\Sigma, f', T, \boldsymbol{s}', \mathbb{F}')$ such that $L(A) = L(A')$. By Theorem 2.3 this yields Turing-completeness for Transformer networks.

We use a simple positional encoding $\mathrm{pos}(i) = [\mathbf{0}, \mathbf{0}, \mathbf{0}, i] \in \mathbb{Q}^{3d+1}$, where $\mathbf{0}$ is the vector composed exclusively by 0's. For each $a \in \Sigma$ we define $f'(a) = [f(a), \mathbf{0}, \mathbf{0}, 0]$. Therefore, if for an input $w \in \Sigma^*$ it is the case that $f(w) = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, then $f'_{\mathrm{pos}}(w) = (\overline{\boldsymbol{x}}_1, \ldots, \overline{\boldsymbol{x}}_n)$, where $\overline{\boldsymbol{x}}_i = [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, i]$. We construct a Tranformer encoder which consists of a single-layer that takes each input vector $\overline{\boldsymbol{x}}_i = [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, i]$ and produces $\boldsymbol{k}_i = [\mathbf{0}, \mathbf{0}, \mathbf{0}, i]$ as keys and $\boldsymbol{v}_i = [\mathbf{0}, \boldsymbol{x}_i, \mathbf{0}, 0]$ as values.

The first n steps of the Transformer decoder emulate the work of the RNN encoder. We maintain the following invariant: for every $i \leq n$ the i-th output of the decoder is $\boldsymbol{y}_i = [\mathbf{0}, \boldsymbol{h}_i, \mathbf{0}, 0]$. We prove this by an inductive argument. Let us assume by induction hypothesis that $\boldsymbol{y}_i = [\mathbf{0}, \boldsymbol{h}_i, \mathbf{0}, 0]$, and thus that its positional encoding is $\overline{\boldsymbol{y}}_i = [\mathbf{0}, \boldsymbol{h}_i, \mathbf{0}, i+1]$. We omit the first self attention (Equation (8)) so that $\boldsymbol{p}_i = \overline{\boldsymbol{y}}_i$, and use $\boldsymbol{p}_i$ to attend – exactly – at the $(i+1)$-th pair $(\boldsymbol{k}_{i+1} = [\mathbf{0}, \mathbf{0}, \mathbf{0}, i+1], \boldsymbol{v}_{i+1} = [\mathbf{0}, \boldsymbol{x}_{i+1}, \mathbf{0}, 0])$ of the Transformer encoder. This can be done with a feed-forward network with three layers plus hard attention. Hence, $\boldsymbol{a}_i$ in Equation (9) will be $\boldsymbol{v}_{i+1} + \boldsymbol{p}_i = [\boldsymbol{x}_{i+1}, \boldsymbol{h}_i, \mathbf{0}, i+1]$. We then define the $O(\cdot)$ function in Equation (10) in such a way that $O([\boldsymbol{x}_{i+1}, \boldsymbol{h}_i, \mathbf{0}, i+1]) = [\mathbf{0}, \mathbf{0}, \sigma(\boldsymbol{x}_{i+1}\boldsymbol{W} + \boldsymbol{h}_i\boldsymbol{V}), 0] = [\mathbf{0}, \mathbf{0}, \boldsymbol{h}_{i+1}, 0]$, and thus $\boldsymbol{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i = [\boldsymbol{x}_{i+1}, \boldsymbol{h}_i, \boldsymbol{h}_{i+1}, i+1]$. We produce $\boldsymbol{y}_{i+1} = [\mathbf{0}, \boldsymbol{h}_{i+1}, \mathbf{0}, 0]$ as the output of another decoder layer and use function $F(\cdot)$ in Equation (11) as the identity to produce the final output.

Simulating the decoder of the RNN N, on the other hand, is a bit more cumbersome. We need to extend the previous construction so that the Transformer decoder can stop reading the input and begin implementing the recursion $\boldsymbol{g}_t = \sigma(\boldsymbol{g}_{t-1}\boldsymbol{U})$. For this, we maintain a *state value* b_i in the decoder such that $b_i = 0$ if $i = n+1$, and $b_i = 1$ otherwise. We do this by combining an attention over the key-value pairs $(\boldsymbol{k}_i, \boldsymbol{v}_i)$ generated by the Transformer encoder together with a self-attention in the Transformer decoder as follows. The attention we use over the encoder satisfies that, whenever one tries to attend over an index bigger than n, it yields the same value, say $\boldsymbol{v}^*$. On the other hand, the self-attention over the decoder decides if the value obtained from the encoder in this iteration is the same as in the previous one. In this way we can generate a sequence $a_1, a_2, a_3, \ldots$ of the form $000 \cdots 0111 \cdots$ that changes from 0 to 1 exactly when $i = n+1$. With a decoder layer we produce the desired b_i simply as $1 - (a_i - a_{i-1})$. We maintain another vector $\boldsymbol{\delta}_i$ and set $\boldsymbol{\delta}_i := \sigma(\boldsymbol{h}_{i-1} - b_i\mathbf{1})$. Since $\mathbf{0} \leq \boldsymbol{h}_i \leq \mathbf{1}$ for every i, we have that $\boldsymbol{\delta}_i = \boldsymbol{h}_n$ exactly when $i = n+1$, and $\boldsymbol{\delta}_i = \mathbf{0}$ otherwise. Combining this with the idea developed in the previous paragraph for simulating the encoder of N, we can maintain yet another vector $\boldsymbol{\gamma}_i$ defined as $\boldsymbol{\gamma}_i := \sigma(\boldsymbol{\gamma}_{i-1}\boldsymbol{U}) + \boldsymbol{\delta}_i$. We obtain then that $\boldsymbol{g}_i = \boldsymbol{\gamma}_{n+1+i}$, which implies that we can also simulate the decoder part of N. Putting it all together, we have that the Transformer needs a vector in $\mathbb{Q}^{6d+8}$ to store the vectors that simulate $\boldsymbol{h}_i$ and $\boldsymbol{g}_i$, the positional encoding, the values b_i, a_i, and some additional working space. $\square$

3.3 Differences with Vaswani et al. (2017)'s framework

Although the general architecture that we presented closely follows that of Vaswani et al. (2017), some choices for functions and parameters in our positive results are different to the usual choices in practice. For instance, we use additive hard attention which allow us to attend directly to specific positions. In contrast, Vaswani et al. (2017) use dot-product for scores, $\mathrm{softmax}$ to attend, plus $\sin$-$\cos$ functions as positional encodings. The $\mathrm{softmax}$, $\sin$ and $\cos$ are not rational functions, and thus, are forbidden in our formalization. An interesting line for future work is to consider arbitrary functions but with additional restrictions, such as finite precision as done by Weiss et al. (2018). Another difference is that for the function $O(\cdot)$ in Equation (10) our proof uses a feed-forward network with various layers, while in Vaswani et al. (2017) only two layers are used. But perhaps the most important difference is the use of function $F(\cdot)$ in Equation (11). Vaswani et al. (2017) uses a function F from $\mathbb{Q}^d$ to $\{0,1\}^d$; actually, a one-hot function produced by a $\mathrm{softmax}$. This one-hot vector is then passed back as input for the next iteration of the decoder. Although the function we use is simpler (we use the identity), it allows us to pass more information to the next decoder step. We leave as an open question whether the Transformer remains Turing complete when a one-hot output function is used instead of the identity function that we use in our proof.

4 NEURAL GPUS

The Neural GPU (Kaiser & Sutskever, 2016) is an architecture that mixes convolutions and gated recurrences over tridimensional tensors. It is parameterized by three functions $U(\cdot)$ (update function), $R(\cdot)$ (reset function), and $F(\cdot)$. Given a tensor $\mathbf{S} \in \mathbb{Q}^{h \times w \times d}$ and a value $r \in \mathbb{N}$, it produces a sequence $\mathbf{S}^1, \mathbf{S}^2, \ldots, \mathbf{S}^r$ given by the following recursive definition (with $\mathbf{S}^0 = \mathbf{S}$).

$$\mathbf{U}^t = U(\mathbf{S}^{t-1}), \qquad \mathbf{R}^t = R(\mathbf{S}^{t-1}), \qquad \mathbf{S}^t = \mathbf{U}^t \odot \mathbf{S}^{t-1} \, + \, (\mathbf{1} - \mathbf{U}) \odot F(\mathbf{R}^t \odot \mathbf{S}^{t-1}).$$

where $\odot$ denotes the element-wise product, and $\mathbf{1}$ is a tensor with only 1's. Neural GPUs force functions $U(\cdot)$ and $R(\cdot)$ to produce a tensor of the same shape as its input with all values in $[0, 1]$. Thus, a Neural GPU resembles a gated recurrent unit (Cho et al., 2014), with $\mathbf{U}$ working as the update gate and $\mathbf{R}$ as the reset gate. Functions $U(\cdot)$, $R(\cdot)$, and $F(\cdot)$ are defined as a convolution of its input with a *4-dimensional kernel bank* with shape (k_H, k_W, d, d) plus a bias tensor, followed by a point-wise transformation

$$f(\mathbf{K} * \mathbf{S} + \mathbf{B}) \tag{13}$$

with different kernels and biases for $U(\cdot)$, $R(\cdot)$, and $F(\cdot)$.

To have an intuition on how the convolution $\mathbf{K} * \mathbf{S}$ works, it is illustrative to think of $\mathbf{S}$ as an $(h \times w)$-grid of (row) vectors and $\mathbf{K}$ as a $(k_H \times k_W)$-grid of $(d \times d)$ matrices. More specifically, let $\boldsymbol{s}_{ij} = \mathbf{S}_{i,j,:}$, and $\boldsymbol{K}_{ij} = \mathbf{K}_{i,j,:,:}$, then $\mathbf{K} * \mathbf{S}$ is a regular two-dimensional convolution in which scalar multiplication has been replaced by vector-matrix multiplication as in the following expression

$$(\mathbf{K} * \mathbf{S})_{i,j,:} \; = \; \sum_u \sum_v \boldsymbol{s}_{i+\Delta_1(u), j+\Delta_2(v)} \, \boldsymbol{K}_{uv}, \tag{14}$$

where $\Delta_1(u) = u - \lfloor k_H/2 \rfloor - 1$ and $\Delta_2(v) = v - \lfloor k_W/2 \rfloor - 1$. This intuition makes evident the similarity between Neural GPUs and cellular automata: $\mathbf{S}$ is a grid of cells, and in every iteration each cell is updated considering the values of its neighbors according to a fixed rule given by $\mathbf{K}$ (Kaiser & Sutskever, 2016). As customary, we assume zero-padding when convolving outside $\mathbf{S}$.

4.1 THE COMPUTATIONAL POWER OF NEURAL GPUS

To study the computational power of Neural GPUs, we cast them as a standard seq-to-seq architecture. Given an input sequence, we put every vector in the first column of the tensor $\mathbf{S}$. We also need to pick a special cell of $\mathbf{S}$ as the *output cell* from which we read the output vector in every iteration. We pick the last cell of the first column of $\mathbf{S}$. Formally, given a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ with $\boldsymbol{x}_i \in \mathbb{Q}^d$, and a fixed value $w \in \mathbb{N}$, we construct the tensor $\mathbf{S} \in \mathbb{Q}^{n \times w \times d}$ by leting $\mathbf{S}_{i,1,:} = \boldsymbol{x}_i$ and $\mathbf{S}_{i,j,:} = \mathbf{0}$ for $j > 1$. The output of the Neural GPU, denoted by $\text{NGPU}(\boldsymbol{X}, r)$, is the sequence of vectors $\boldsymbol{Y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_r)$ such that $\boldsymbol{y}_t = \mathbf{S}^t_{n,1,:}$. Given this definition, we can naturally view the Neural GPUs as language recognizers (as formalized them in Section 2).

Since the bias tensor $\mathbf{B}$ in Equation (13) is of the same size than $\mathbf{S}$, the number of parameters in a Neural GPU grows with the size of the input. Thus, a Neural GPU cannot be considered as a fixed architecture. To tackle this issue we introduce the notion of *uniform Neural GPU*, as one in which for every bias $\mathbf{B}$ there exists a matrix $\boldsymbol{B} \in \mathbb{Q}^{w \times d}$ such that $\mathbf{B}_{i,:,:} = \boldsymbol{B}$ for each i. Thus, uniform Neural GPUs can be finitely specified (as they have a constant number of parameters, not depending on the length of the input). We now establish the completeness of this model.

Theorem 4.1. *The class of uniform Neural GPUs is Turing complete.*

Proof sketch. The proof is based on simulating a seq-to-seq RNN; thus, completeness follows from Theorem 2.3. Consider an RNN encoder-decoder language recognizer, such that N is of dimension d and its encoder and decoder are defined by the equations $\boldsymbol{h}_i = \sigma(\boldsymbol{x}_i \boldsymbol{W} + \boldsymbol{h}_{i-1} \boldsymbol{V})$ and $\boldsymbol{g}_t = \sigma(\boldsymbol{g}_{t-1} \boldsymbol{U})$, respectively, where $\boldsymbol{g}_0 = \boldsymbol{h}_n$ and n is the length of the input. We use a Neural GPU with input tensor $\mathbf{S} \in \mathbb{Q}^{n \times 1 \times 3d+3}$. Let $\mathbf{E}_i = \mathbf{S}_{i,1,1:d}$ and $\mathbf{D}_i = \mathbf{S}_{i,1,d+1:2d}$. The idea is to use $\mathbf{E}$ for the encoder and $\mathbf{D}$ for the decoder. We use kernel banks of shape $(2, 1, 3d+3, 3d+3)$ with uniform bias tensors to simulate the following computation. In every step t, we first compute the value of $\sigma(\mathbf{E}_t \boldsymbol{W} + \mathbf{E}_{t-1} \boldsymbol{V})$ and store it in $\mathbf{E}_t$, and then reset $\mathbf{E}_{t-1}$ to zero. Similarly, in step t we update the vector in position $\mathbf{D}_{t-1}$ storing in it the value $\sigma(\mathbf{D}_{t-1} \boldsymbol{U} + \mathbf{E}_{t-1} \boldsymbol{U})$ (for the value of $\mathbf{E}_{t-1}$ before the reset). We use the gating mechanism to ensure a sequential update of the cells such that at time t

we update only positions $\mathbf{E}_i$ and $\mathbf{D}_j$ for $i \leq t$ and $j \leq t-1$. Thus the updates on the $\mathbf{D}$ are always one iteration behind the update of $\mathbf{E}$. Since the vectors in $\mathbf{D}$ are never reset to zero, they keep being updated which allows us to simulate an arbitrary long computation. In particular we prove that at iteration t it holds that $\mathbf{E}_t = \boldsymbol{h}_t$, and at iteration $n+t$ it holds that $\mathbf{D}_n = \boldsymbol{g}_t$. We require $3d+3$ components, as we need to implement several gadgets for properly using the update and reset gates. In particular, we need to store the value of $\mathbf{E}_{t-1}$ before we reset it. The detailed construction and the correctness proof can be found in the appendix. □

The proof above makes use of kernels of shape $(2, 1, d, d)$ to obtain Turing completeness. This is, in a sense, optimal, as one can easily prove that Neural GPUs with kernels of shape $(1, 1, d, d)$ are not Turing complete, regardless of the size of d. In fact, for kernels of this shape the value of a cell of $\mathbf{S}$ at time t depends only on the value of the same cell in time $t-1$.

Zero padding vs circular convolution The proof of Theorem 4.1 requires the application of *zero padding* in convolution. This allows us to clearly differentiate internal cells from cells corresponding to the endpoints of the input sequence. Interestingly, Turing-completeness is lost if we replace zero padding with *circular convolution*. Formally, given $\mathbf{S} \in \mathbb{Q}^{h \times w \times d}$, a circular convolution is obtained by defining $\mathbf{S}_{h+1,:,:} = \mathbf{S}_{1,:,:}$ and $\mathbf{S}_{0,:,:} = \mathbf{S}_{h,:,:}$ One can prove that uniform Neural GPUs with circular convolutions cannot differentiate among periodic sequences of different length; in particular, they cannot check if a periodic input sequence is of even or odd length. This yields the following:

Proposition 4.2. *Uniform Neural GPUs with circular convolutions are not Turing complete.*

Related to this last result is the empirical observation by Price et al. (2016) that Neural GPUs that learn to solve *hard* problems, e.g., binary multiplication, and which generalizes to most of the inputs, struggle with highly symmetric (and nearly periodic) inputs. Actually, Price et al. (2016) exhibit examples of the form 11111111×11111111 failing for all inputs with eight or more 1s. We leave as future work to explore the implications of our theoretical results on this practical observation.

Bidimensional tensors and piecewise linear activations Freivalds & Liepins (2018) simplified Neural GPUs and proved that, by considering piecewise linear activations and bidimensional input tensors instead of the original smooth activations and tridimensional tensors used by Kaiser & Sutskever (2016), it is possible to achieve substantially better results in terms of training time and generalization. Our Turing completeness proof also relies on a bidimensional tensor and uses piecewise linear activations, thus providing theoretical evidence that these simplifications actually retain the full expressiveness of Neural GPUs while simplifying its practical applicability.

5 Final Remarks and Future Work

We have presented an analysis of the Turing completeness of two popular neural architectures for sequence-processing tasks; namely, the Transformer, based on attention, and the Neural GPU, based on recurrent convolutions. We plan to further refine this analysis in the future. For example, our proof of Turing completeness for the Transformer requires the presence of *residual connections*, i.e., the $+\boldsymbol{x}_i$, $+\boldsymbol{a}_i$, $+\boldsymbol{y}_i$, and $+\boldsymbol{p}_i$ summands in Equations (5-10), while our proof for Neural GPUs heavily relies on the gating mechanism. We will study whether these features are actually essential to obtain completeness.

Although our results are mostly of theoretical interest, they might lead to observations of practical interest. For example, Chen et al. (2018) have established the undecidability of several practical problems related to probabilistic language modeling with RNNs. This means that such problems can only be approached in practice via heuristics solutions. Many of the results in Chen et al. (2018) are, in fact, a consequence of the Turing completeness of RNNs as established by Siegelmann & Sontag (1995). We plan to study to what extent our analogous undecidability results for Transformers and Neural GPUs imply undecidability for language modeling problems based on these architectures.

Finally, our results rely on being able to compute internal representations of arbitrary precision. It would be interesting to perform a theoretical study of the main properties of both architectures in a setting in which only finite precision is allowed, as have been recently carried out for RNNs (Weiss et al., 2018). We also plan to tackle this problem in our future work.

REFERENCES

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

Yining Chen, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight. Recurrent neural networks as weighted language recognizers. In *NAACL-HLT 2018*, pp. 2261–2271, 2018.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pp. 1724–1734, 2014.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. *CoRR*, abs/1807.03819, 2018. URL https://arxiv.org/abs/1807.03819.

Karlis Freivalds and Renars Liepins. Improving the neural GPU architecture for algorithm learning. In *Neural Abstract Machines & Program Induction (NAMPI)*, 2018.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1828–1836, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, pp. 630–645, 2016b.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pp. 190–198, 2015.

Lukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *ICLR*, 2016.

S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy (eds.), *Automata Studies*, pp. 3–41. Princeton University Press, 1956.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *ICLR*, 2016.

Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

Nicolas Ollinger. Universalities in cellular automata. In *Handbook of Natural Computing*, pp. 189–229. 2012.

Eric Price, Wojciech Zaremba, and Ilya Sutskever. Extensions and limitations of the neural GPU. *CoRR*, abs/1611.00736, 2016. URL http://arxiv.org/abs/1611.00736.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT*, pp. 464–468, 2018.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT*, pp. 440–449, 1992.

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

Alvy Ray Smith III. Simple computation-universal cellular spaces. *Journal of the ACM (JACM)*, 18 (3):339–353, 1971.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pp. 5998–6008, 2017.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision RNNs for language recognition. In *ACL 2018*, pp. 740–745, 2018.

A PROOFS FOR SECTION 2

A.1 PROOF OF THEOREM 2.3

We first sketch the main idea of Siegelmann & Sontag's proof. We refer the reader to the original paper for details. Siegelmann & Sontag show how to simulate a two-stack machine M (and subsequently, a Turing machine) with a single RNN N with σ as activation. They first construct a network N_1 that, with $\mathbf{0}$ as initial state ($\boldsymbol{h}_0^{N_1} = \mathbf{0}$) and with a binary string $w \in \{0,1\}^*$ as input sequence, produces a representation of w as a rational number and stores it as one of its internal values. Their internal representation of strings encodes every w as a rational number between 0 and 1. In particular, they use base 4 such that, for example, a string $w = 100110$ is encoded as $(0.311331)_4$ that is, its encoding is

$$3 \times 4^{-1} + 1 \times 4^{-2} + 1 \times 4^{-3} + 3 \times 4^{-4} + 3 \times 4^{-5} + 1 \times 4^{-6}.$$

This representation allows one to easily simulate stack operations as affine transformations plus σ activations. For instance, if x_w is the value representing string $w = b_1 b_2 \cdots b_n$ seen as a stack, then the $\mathrm{top}(w)$ operation can be defined as simply $y = \sigma(4x_w - 2)$, since $y = 1$ if and only if $b_1 = 1$, and $y = 0$ if and only if $b_1 = 0$. Other stack operations can de similarly simulated. Using this representation, they construct a second network N_2 that simulates the two-stacks machine by using one neuron value to simulate each stack. The input w for the simulated machine M is assumed to be at an internal value given to N_2 as an initial state ($\boldsymbol{h}_0^{N_2}$). Thus, N_2 expects only zeros as input. Actually, to make N_2 work for r steps, an input of the form 0^r should be provided.

Finally, they combine N_1 and N_2 to construct a network N which expects an input of the following form: $(b_1, 1, 0)(b_2, 1, 0) \cdots (b_n, 1, 0)(0, 0, 1)(0, 0, 0)(0, 0, 0) \cdots (0, 0, 0)$. The idea is that the first component contains the input string $w = b_1 b_2 \cdots b_n$, the second component states when the input is *active*, and the third component is 1 only when the input is inactive for the first time. Before the input vector $(0, 0, 1)$ the network N_1 is working. The input $(0, 0, 1)$ is used to simulate a change from N_1 to N_2, and the rest of input vectors $(0, 0, 0)$ are provided to continue with N_2 for as many steps as needed. The number neurons that this construction needs to simulate a machine M with m states, is $10m + 30$. [1]

It is clear that Siegelmann & Sontag's proof resembles a modern encoder-decoder RNN architecture, where N_1 is the encoder and N_2 is the decoder, thus it is straightforward to use the same construction to provide an RNN encoder-decoder N' and a language recognizer A that uses N' and simulates the two-stacks machine M. There are some details that is important to notice. Assume that N' is given by the formulas in Equations (1) and (2). First, since N_2 in the above construction expects no input, we can safely assume that $\boldsymbol{R}$ in Equation (2) is the null matrix. Moreover, since A defines its own embedding function, we can ensure that every vector that we provide for the encoder part of N' has a 1 in a fixed component, and thus we do not need the bias $\boldsymbol{b}_1$ in Equation (1) since it can be simulated with one row of matrix $\boldsymbol{V}$. We can do a similar construction for the bias $\boldsymbol{b}_2$ (Equation (2)). Finally, Siegelmann & Sontag show that its construction can be modified such that a particular neuron of N_2, say $n^\star$, is always 0 except for the first time an accepting state of M is reached, in which case $n^\star = 1$. Thus, one can consider $O(\cdot)$ (Equation (2)) as the identity function and add to A the stopping criterion that just checks if $n^\star$ is 1. This completes the proof sketch of Theorem 2.3.

[1] The idea presented above allows one to linearly simulate M, that is, each step of M is simulated with a constant number of steps of the corresponding RNN. Siegelmann & Sontag show that, with a refinement of the above encoding one can simulate M in *real-time*, that is, a single step of M is simulated with a single step of the recurrent network. The $10m + 30$ is the bound given by a simulation with slow-down of two. See the original paper for details (Siegelmann & Sontag, 1995).

B PROOFS FOR SECTION 3

B.1 PROOF OF PROPOSITION 3.1

Proof. We extend the definition of the function $\mathrm{PropInv}$ to sequences of vectors. Given a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ we use $\mathrm{vals}(\boldsymbol{X})$ to denote the set of all vectors occurring in $\boldsymbol{X}$. Similarly as for strings, we use $\mathrm{prop}(\boldsymbol{v}, \boldsymbol{X})$ as the number of times that $\boldsymbol{v}$ occurs in $\boldsymbol{X}$ divided by the length of $\boldsymbol{X}$. Now we are ready to extend $\mathrm{PropInv}$ with the following definition:

$$\mathrm{PropInv}(\boldsymbol{X}) = \{\boldsymbol{X}' \mid \mathrm{vals}(\boldsymbol{X}') = \mathrm{vals}(\boldsymbol{X}) \text{ and } \mathrm{prop}(\boldsymbol{v}, \boldsymbol{X}) = \mathrm{prop}(\boldsymbol{v}, \boldsymbol{X}') \text{ for all } \boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})\}$$

Notice that for every embedding function $f : \Sigma \to \mathbb{Q}^d$ and string $w \in \Sigma^*$, we have that if $u \in \mathrm{PropInv}(w)$ then $f(u) \in \mathrm{PropInv}(f(w))$. Thus in order to prove that $\mathrm{Trans}(f(w), \boldsymbol{s}, r) = \mathrm{Trans}(f(u), \boldsymbol{s}, r)$ for every $u \in \mathrm{PropInv}(w)$, it is enough to prove that

$$\mathrm{Trans}(\boldsymbol{X}, \boldsymbol{s}, r) = \mathrm{Trans}(\boldsymbol{X}', \boldsymbol{s}, r) \text{ for every } \boldsymbol{X}' \in \mathrm{PropInv}(\boldsymbol{X}) \tag{15}$$

To further simplify the exposition of the proof we introduce another notation. We denote by $p_{\boldsymbol{v}}^{\boldsymbol{X}}$ as the number of times that vector $\boldsymbol{v}$ occurs in $\boldsymbol{X}$. Thus we have that $\boldsymbol{X}' \in \mathrm{PropInv}(\boldsymbol{X})$ if and only if, there exists a value $\gamma \in \mathbb{Q}^+$ such that for every $\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})$ it holds that $p_{\boldsymbol{v}}^{\boldsymbol{X}'} = \gamma p_{\boldsymbol{v}}^{\boldsymbol{X}}$.

We now have all the necessary to proceed with the proof of Proposition 3.1. We will prove it by proving the property in (15). Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be an arbitrary sequence of vectors, and let $\boldsymbol{X}' = (\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_m) \in \mathrm{PropInv}(\boldsymbol{X})$. Moreover, let $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \mathrm{Enc}(\boldsymbol{X}; \boldsymbol{\theta})$ and $\boldsymbol{Z}' = (\boldsymbol{z}'_1, \ldots, \boldsymbol{z}'_m) = \mathrm{Enc}(\boldsymbol{X}'; \boldsymbol{\theta})$. We first prove the following property:

$$\text{For every pair of indices } (i, j) \in \{1, \ldots, n\} \times \{1, \ldots, m\}, \text{ if } \boldsymbol{x}_i = \boldsymbol{x}'_j \text{ then } \boldsymbol{z}_i = \boldsymbol{z}'_j. \tag{16}$$

Lets (i, j) be a pair of indices such that $\boldsymbol{x}_i = \boldsymbol{x}'_j$. From Equations (5-6) we have that $\boldsymbol{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i$ where $\boldsymbol{a}_i = \mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) + \boldsymbol{x}_i$. Thus, since $\boldsymbol{x}_i = \boldsymbol{x}'_j$, in order to prove $\boldsymbol{z}_i = \boldsymbol{z}'_j$ it is enough to prove that $\mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) = \mathrm{Att}(Q(\boldsymbol{x}'_j), K(\boldsymbol{X}'), V(\boldsymbol{X}'))$. By equations (3-4) and the restriction over the form of normalization functions we have that

$$\mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) = \frac{1}{\alpha} \sum_{\ell=1}^{n} f_\rho(\mathrm{score}(Q(\boldsymbol{x}_i), K(\boldsymbol{x}_\ell)))V(\boldsymbol{x}_\ell)$$

where $\alpha = \sum_{\ell=1}^{n} f_\rho(\mathrm{score}(Q(\boldsymbol{x}_\ell), K(\boldsymbol{x}_\ell)))$. The above equation can be rewritten as

$$\mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) = \frac{1}{\alpha} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} p_{\boldsymbol{v}}^{\boldsymbol{X}} f_\rho(\mathrm{score}(Q(\boldsymbol{x}_i), K(\boldsymbol{v})))V(\boldsymbol{v})$$

with $\alpha = \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} p_{\boldsymbol{v}}^{\boldsymbol{X}} f_\rho(\mathrm{score}(Q(\boldsymbol{v}), K(\boldsymbol{v})))$. By a similar reasoning we can write

$$\mathrm{Att}(Q(\boldsymbol{x}'_j), K(\boldsymbol{X}'), V(\boldsymbol{X}')) = \frac{1}{\beta} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X}')} p_{\boldsymbol{v}}^{\boldsymbol{X}'} f_\rho(\mathrm{score}(Q(\boldsymbol{x}'_j), K(\boldsymbol{v})))V(\boldsymbol{v})$$

with $\beta = \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X}')} p_{\boldsymbol{v}}^{\boldsymbol{X}'} f_\rho(\mathrm{score}(Q(\boldsymbol{v}), K(\boldsymbol{v})))$. Now, since $\boldsymbol{X}' \in \mathrm{PropInv}(\boldsymbol{X})$ we know that $\mathrm{vals}(\boldsymbol{X}) = \mathrm{vals}(\boldsymbol{X}')$ and there exists a $\gamma \in \mathbb{Q}^+$ such that $p_{\boldsymbol{v}}^{\boldsymbol{X}'} = \gamma p_{\boldsymbol{v}}^{\boldsymbol{X}}$ for every $\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})$. Finally, from this last property, plus the fact that $\boldsymbol{x}_i = \boldsymbol{x}'_j$ we have

$$\begin{aligned}
\mathrm{Att}(Q(\boldsymbol{x}'_j), K(\boldsymbol{X}'), V(\boldsymbol{X}')) &= \frac{1}{\gamma\alpha} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} \gamma p_{\boldsymbol{v}}^{\boldsymbol{X}} f_\rho(\mathrm{score}(Q(\boldsymbol{x}'_j), K(\boldsymbol{v})))V(\boldsymbol{v}) \\
&= \frac{1}{\alpha} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} p_{\boldsymbol{v}}^{\boldsymbol{X}} f_\rho(\mathrm{score}(Q(\boldsymbol{x}_i), K(\boldsymbol{v})))V(\boldsymbol{v}) \\
&= \mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X}))
\end{aligned}$$

Which completes the proof of Property (16) above.

Consider now the complete encoder TEnc. Let $(\boldsymbol{K}, \boldsymbol{V}) = \mathrm{TEnc}(\boldsymbol{X})$ and $(\boldsymbol{K}', \boldsymbol{V}') = \mathrm{TEnc}(\boldsymbol{X}')$, and let $\boldsymbol{q}$ be an arbitrary vector. We will prove now that $\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}', \boldsymbol{V}')$. By

following a similar reasoning as for proving Property (16) (plus induction on the layers of TEnc) we obtain that if $\boldsymbol{x}_i = \boldsymbol{x}'_j$ then $\boldsymbol{k}_i = \boldsymbol{k}'_j$ and $\boldsymbol{v}_i = \boldsymbol{v}'_j$, for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. Thus, there exists a mapping $M_K : \mathrm{vals}(\boldsymbol{X}) \to \mathrm{vals}(\boldsymbol{K})$ such that $M_K(\boldsymbol{x}_i) = \boldsymbol{k}_i$ and $M_K(\boldsymbol{x}'_j) = \boldsymbol{k}'_j$ and similarly a mapping $M_V : \mathrm{vals}(\boldsymbol{X}) \to \mathrm{vals}(\boldsymbol{V})$ such that $M_V(\boldsymbol{x}_i) = \boldsymbol{v}_i$ and $M_V(\boldsymbol{x}'_j) = \boldsymbol{v}'_j$, for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. Lets focus now on $\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})$. We have:

$$\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) \quad = \quad \frac{1}{\alpha} \sum_{i=1}^{n} f_\rho(\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_i))\boldsymbol{v}_i$$

with $\alpha = \sum_{i=1}^{n} f_\rho(\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_i))$. Similarly as before, we can rewrite this as

$$\begin{aligned}
\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) &= \frac{1}{\alpha} \sum_{i=1}^{n} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{x}_i)))M_V(\boldsymbol{x}_i) \\
&= \frac{1}{\alpha} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} p^{\boldsymbol{X}}_{\boldsymbol{v}} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{v})))M_V(\boldsymbol{v})
\end{aligned}$$

with $\alpha = \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} p^{\boldsymbol{X}}_{\boldsymbol{v}} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{v})))$. Similarly for $\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}', \boldsymbol{V}')$ we have

$$\begin{aligned}
\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}', \boldsymbol{V}') &= \frac{1}{\beta} \sum_{j=1}^{m} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{x}'_j)))M_V(\boldsymbol{x}'_j) \\
&= \frac{1}{\beta} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X}')} p^{\boldsymbol{X}'}_{\boldsymbol{v}} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{v})))M_V(\boldsymbol{v})
\end{aligned}$$

And finally using that $\boldsymbol{X}' \in \mathrm{PropInv}(\boldsymbol{X})$ we obtain

$$\begin{aligned}
\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}', \boldsymbol{V}') &= \frac{1}{\beta} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X}')} p^{\boldsymbol{X}'}_{\boldsymbol{v}} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{v})))M_V(\boldsymbol{v}) \\
&= \frac{1}{\gamma\alpha} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} \gamma p^{\boldsymbol{X}}_{\boldsymbol{v}} f_\rho(\mathrm{score}(\boldsymbol{q}, M_K(\boldsymbol{v})))M_V(\boldsymbol{v}) \\
&= \frac{1}{\alpha} \sum_{\boldsymbol{v} \in \mathrm{vals}(\boldsymbol{X})} p^{\boldsymbol{X}}_{\boldsymbol{v}} f_\rho(\mathrm{score}(\boldsymbol{q}, K(\boldsymbol{v}))V(\boldsymbol{v}) \\
&= \mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})
\end{aligned}$$

which is what we wanted.

To complete the rest proof, consider $\mathrm{Trans}(\boldsymbol{X}, \boldsymbol{y}_0, r)$ which is defined by the recursion

$$\boldsymbol{y}_{k+1} \quad = \quad \mathrm{TDec}(\mathrm{TEnc}(\boldsymbol{X}), (\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_k))$$

To prove that $\mathrm{Trans}(\boldsymbol{X}, \boldsymbol{y}_0, r) = \mathrm{Trans}(\boldsymbol{X}', \boldsymbol{y}_0, r)$ we use an inductive argument. We know that

$$\begin{aligned}
\boldsymbol{y}_1 &= \mathrm{TDec}(\mathrm{TEnc}(\boldsymbol{X}), (\boldsymbol{y}_0)) \\
&= \mathrm{TDec}((\boldsymbol{K}, \boldsymbol{V}), (\boldsymbol{y}_0)).
\end{aligned}$$

Now TDec only access $(\boldsymbol{K}, \boldsymbol{V})$ via attentions of the form $\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})$ and for the case of $\boldsymbol{y}_1$ the vector $\boldsymbol{q}$ can only depend on $\boldsymbol{y}_0$, thus, from $\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}', \boldsymbol{V}')$ we have that

$$\begin{aligned}
\boldsymbol{y}_1 &= \mathrm{TDec}((\boldsymbol{K}, \boldsymbol{V}), (\boldsymbol{y}_0)) \\
&= \mathrm{TDec}((\boldsymbol{K}', \boldsymbol{V}'), (\boldsymbol{y}_0)) \\
&= \mathrm{TDec}(\mathrm{TEnc}(\boldsymbol{X}'), (\boldsymbol{y}_0)).
\end{aligned}$$

The rest of the steps follow by a simple induction on k. □

B.2 Proof of Corollary 3.2

To obtain a contradiction, assume that there is a language recognizer A that uses a Transformer network and such that $L = L(A)$. Now consider the strings $w_1 = aabb$ and $w_2 = aaabbb$. Since $w_1 \in \mathrm{PropInv}(w_2)$ by Proposition 3.1 we have that $w_1 \in L(A)$ if and only if $w_2 \in L(A)$ which is a contradiction since $w_1 \in L$ but $w_2 \notin L$. This completes the proof of the corollary.

B.3 PROOF OF PROPOSITION 3.3

We construct a language recognizer $A = (\Sigma, f, \mathrm{Trans}, \boldsymbol{s}, \mathbb{F})$ with Trans a very simple Transformer network with dimension $d = 2$ and using just one layer of encoder and one layer of decoder, such that $L(A) = \{w \in \{a, b\}^* \mid w \text{ has strictly more symbols } a \text{ than symbols } b\}$. As embedding function, we use $f(a) = [0, 1]$ and $f(b) = [0, -1]$.

Assume that the output for the encoder part of the transformer is $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. First we use an encoder layer that implements the identity function. This can be trivially done using null functions for the self attention and through the residual connections this encoder layer shall preserve the original $\boldsymbol{x}_i$ values. For the final $V(\cdot)$ and $K(\cdot)$ functions of the Transformer encoder (Equation (7)), we use $V(\boldsymbol{x}) = \boldsymbol{x}$ the identity function and $K(\boldsymbol{x}) = [0, 0]$, giving $\boldsymbol{V^e} = \boldsymbol{X}$ and $\boldsymbol{K^e} = ([0, 0], [0, 0], \ldots, [0, 0])$.

For the decoder we use a similar approach. We consider the identity in the self attention plus the residual (which can be done by just using the null functions for the self attention). Considering the external attention, that is the attention over $(\boldsymbol{K^e}, \boldsymbol{V^e})$, we let score and ρ be arbitrary scoring and normalization functions. And finally for the function $O(\cdot)$ (Equation (10)) we use a single layer neural network implementing the affine transformation $O([x, y]) = [y - x, -y]$ such that $O([x, y]) + [x, y] = [y, 0]$. The final function $F(\cdot)$ is just the identity function.

In order to complete the proof we introduce some notation. Lets denote by $\#_a(w)$ as the number of a's in w, and similarly $\#_b(w)$ for the number of b's in w. Lets call c_w as the value $\frac{\#_a(w) - \#_b(w)}{n}$. We now prove that, for any string $w \in \{a, b\}^*$ if we consider $f(w) = \boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ as the input sequence for Trans and we use initial value $\boldsymbol{s} = [0, 0]$ for the decoder, the complete network shall compute a sequence $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_r$ such that:

$$\boldsymbol{y}_i = \begin{cases} [0, 0] & i = 0 \\ [c_w, 0] & i > 0 \end{cases}$$

We proceed by induction. The base case trivially holds since $\boldsymbol{y}_0 = \boldsymbol{s} = [0, 0]$. Assume now that we are at step r and the input for the decoder is $(\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_r)$. We will show that $\boldsymbol{y}_{r+1} = [c_w, 0]$. Since we consider the identity in the self attention (Equation (8)), we have that $\boldsymbol{p}_i = \boldsymbol{y}_i$ for every i in $\{0, \ldots, i\}$. Now considering the external attention, that is the attention over $(\boldsymbol{K^e}, \boldsymbol{V^e})$, Since all key vectors in $\boldsymbol{K^e}$ are $[0, 0]$, the external attention will produce the same score value for all positions. That is, $\mathrm{score}(\boldsymbol{p}_i, \boldsymbol{k}_{j_1}) = \mathrm{score}(\boldsymbol{p}_i, \boldsymbol{k}_{j_2})$ for every j_1, j_2. Lets call this value $s^\star$. Thus we have that

$$\begin{aligned} \rho(\mathrm{score}(\boldsymbol{p}_i, \boldsymbol{k}_1), \ldots, \mathrm{score}(\boldsymbol{p}_i, \boldsymbol{k}_n)) &= \rho(s^\star, s^\star, \ldots, s^\star) \\ &= \left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right). \end{aligned}$$

Then, since $\boldsymbol{V^e} = \boldsymbol{X}$ we have that

$$\begin{aligned} \mathrm{Att}(\boldsymbol{p}_i, \boldsymbol{K^e}, \boldsymbol{V^e}) &= \frac{1}{n} \sum_{\ell=1}^{n} \boldsymbol{x}_\ell \\ &= \frac{1}{n} [0, \#_a(w) - \#_b(w)] \end{aligned}$$

for every $i \in \{0, \ldots, r\}$. The last equality holds since our embedding are $f(a) = [0, 1]$ and $f(b) = [0, -1]$, and so every a in w sums one and every b subtracts one. Thus, we have that

$$\mathrm{Att}(\boldsymbol{p}_i, \boldsymbol{K^e}, \boldsymbol{V^e}) = [0, c_w].$$

for every $i \in \{0, \ldots, r\}$. In the next step, after the external attention plus the residual connection (Equation (9)) we have

$$\begin{aligned} \boldsymbol{a}_i &= \mathrm{Att}(\boldsymbol{p}_i, \boldsymbol{K^e}, \boldsymbol{V^e}) + \boldsymbol{p}_i \\ &= \mathrm{Att}(\boldsymbol{p}_i, \boldsymbol{K^e}, \boldsymbol{V^e}) + \boldsymbol{y}_i \\ &= [0, c_w] + [c_w, 0] \\ &= [c_w, c_w] \end{aligned}$$

Applying function $O(\cdot)$ plus the residual connection (Equation (10)) we have

$$\begin{aligned} \boldsymbol{z}_i &= O(\boldsymbol{a}_i) + \boldsymbol{a}_i \\ &= O([c_w, c_w]) + [c_w, c_w] \\ &= [c_w - c_w, -c_w] + [c_w, c_w] \\ &= [c_w, 0] \end{aligned}$$

Finally, $\boldsymbol{y}_{r+1} = F(\boldsymbol{z}_r) = \boldsymbol{z}_r = [c_w, 0]$ which is exactly what we wanted to prove.

To complete the proof, notice that $\#_a(w) > \#_b(w)$ if and only if $c_w > 0$. If we define $\mathbb{F}$ as $\mathbb{Q}^+ \times \mathbb{Q}$, the recognizer $A = (\Sigma, f, \mathrm{Trans}, \boldsymbol{s}, \mathbb{F})$ will accept the string w exactly when $c_w > 0$, that is, $w \in L(A)$ if and only if $\#_a(w) > \#_b(w)$. That is exactly the language S, and so the proof is complete.

B.4 Proof of Theorem 3.4

The main idea is to simulate an RNN encoder-decoder architecture using a Transformer network. Let $A = (\Sigma, f, N, \boldsymbol{s}, \mathbb{F})$ be a recognizer, with N an RNN encoder-decoder of dimension d and composed of the equations

$$\begin{aligned} \boldsymbol{h}_i &= \sigma(\boldsymbol{x}_i \boldsymbol{V} + \boldsymbol{h}_{i-1} \boldsymbol{V}) \\ \boldsymbol{g}_t &= \sigma(\boldsymbol{g}_{t-1} \boldsymbol{U}) \end{aligned}$$

with $\boldsymbol{h}_0 = \boldsymbol{0}$ (the vector in $\mathbb{Q}^d$ with only zeros), and $\boldsymbol{g}_0 = \boldsymbol{h}_n$ where n is the length of the input. We shall construct a recognizer $A' = (\Sigma, f', \mathrm{Trans}, \boldsymbol{s}', \mathbb{F}')$ with Trans a Transformer network such that $L(A) = L(A')$. In our proof we will use vectors in $\mathbb{Q}^{6d+8}$. The idea of every component of this vector will be clear when we describe the details of the construction.

Preliminaries

We first introduce the embedding and positional encoding functions. We use an embedding function $f' : \Sigma \to \mathbb{Q}^{6d+8}$ such that

$$f'(a) = [f(a), \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, 0, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, 0]$$

Our construction uses a very simple positional encoding $\mathrm{pos} : \mathbb{N} \to \mathbb{Q}^{6d+8}$ such that $\mathrm{pos}(i)$ is a vector that only has the value i in its last position and 0 everywhere else. Thus if $f(a_i) = \boldsymbol{x}_i$ we have that

$$f'_{\mathrm{pos}}(a_i) = [\boldsymbol{x}_i, \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, 0, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, i]$$

We denote this last vector by $\overline{\boldsymbol{x}}_i$. That is, the input for Trans is the sequence $(\overline{\boldsymbol{x}}_1, \overline{\boldsymbol{x}}_2, \ldots, \overline{\boldsymbol{x}}_n)$. We need a final notation to begin with the proof. We will usually have the last component of all our vectors with the positional encoding. Thus, given an arbitrary vector $\boldsymbol{v} \in \mathbb{Q}^{6d+8}$ we denote by $e(\boldsymbol{v})$ the value of its last component. That is, for the vector $\overline{\boldsymbol{x}}_i$ above we have $e(\overline{\boldsymbol{x}}_i) = i$.

The attention mechanism used

In what follows we will use a scoring function $\mathrm{score} : \mathbb{Q}^{6d+8} \times \mathbb{Q}^{6d+8} \to \mathbb{Q}$ that essentially performs the following computation for a pair of vectors $(\boldsymbol{q}, \boldsymbol{k})$:

$$\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}) = -|e(\boldsymbol{q}) - e(\boldsymbol{k})|$$

This function can be implemented with an additive attention as follows. We use a feed-forward network of three layers that receives as input a vector of the form $[\boldsymbol{q}, \boldsymbol{k}] \in \mathbb{Q}^{2(6d+8)}$. The first layer will be a function $f_1 : \mathbb{Q}^{2(6d+8)} \to \mathbb{Q}^2$ that just picks the values at positions $6d+8$ and at $2(6d+8)$, thus, with input $[\boldsymbol{q}, \boldsymbol{k}]$ it produces $f_1([\boldsymbol{q}, \boldsymbol{k}]) = [e(\boldsymbol{q}), e(\boldsymbol{k})]$. It is clear that this transformation can be performed with a single-layer neural network (with an identity activation) since this is just a linear transformation. The next two layers are defined by functions $f_2 : \mathbb{Q}^2 \to \mathbb{Q}^2$, and $f_3 : \mathbb{Q}^2 \to \mathbb{Q}$ defined as follows:

$$\begin{aligned} f_2([x, y]) &= \mathrm{relu}([x - y, y - x]) \\ f_3([x, y)] &= -x - y \end{aligned}$$

Thus, $\mathrm{score}(\boldsymbol{q}, \boldsymbol{k})$ will be

$$\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}) = f_3(f_2(f_1([\boldsymbol{q}, \boldsymbol{k}])))$$

It is not difficult to see that the result will be

$$\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}) = -\operatorname{relu}(e(\boldsymbol{q}) - e(\boldsymbol{k})) - \operatorname{relu}(e(\boldsymbol{k}) - e(\boldsymbol{q})) = -|e(\boldsymbol{q}) - e(\boldsymbol{k})|$$

which is exactly what we wanted to obtain.

The most important property of this score function is that, when combined with hard attention, it has the effect of attending to a specific position of the attended sequence of vectors. More formally, assume that $\boldsymbol{q}$ is a vector such that $1 \leq e(\boldsymbol{q}) \leq n$, and $\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_n$ are all such that $e(\boldsymbol{k}_i) = i$. Then we have that the following holds:

$$(\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_1), \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_2), \ldots, \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_n)) \quad = \quad (-k, -k+1, \ldots, -1, 0, -1, \ldots, -(n-k))$$

where the 0 appears exactly in the k-th position of the above vector. Thus we have that

$$\mathrm{hardmax}(\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_1), \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_2), \ldots, \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_n)) \quad = \quad (0, 0, \ldots, 0, 1, 0, \ldots, 0)$$

Where the 1 appears exactly in the k-th position. Moreover, if we assume that $e(\boldsymbol{q}) = k$ such that $k > n$ then

$$(\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_1), \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_2), \ldots, \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_n)) \quad = \quad (-k, -k+1, \ldots, -k+n)$$

and thus the maximum value occurs in the last position which implies that

$$\mathrm{hardmax}(\mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_1), \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_2), \ldots, \mathrm{score}(\boldsymbol{q}, \boldsymbol{k}_n)) \quad = \quad (0, 0, \ldots, 0, 1)$$

That is, it is the sequence with a 1 in its last position.

Putting all this in the Att function, we obtain the following. Assume that we have a sequence $\boldsymbol{K} = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n)$ of key vectors such that $e(\boldsymbol{k}_i) = i$ for every i. Let $\boldsymbol{V} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$ be a sequence of arbitrary value vectors, and assume that we have a query $\boldsymbol{q}$ such that $1 \leq e(\boldsymbol{q}) \leq n$. Then we have

$$\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{v}_{e(\boldsymbol{q})}.$$

Moreover, whenever we have a query vector $\boldsymbol{q}$ such that $e(\boldsymbol{q}) > n$ then

$$\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{v}_n.$$

Given this property and also given that all vectors that we use will have a positional encoding in its last component, to simplify the notation we can use the function $\mathrm{Att}(i, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{v}_i$, and thus

$$\mathrm{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{Att}(e(\boldsymbol{q}), \boldsymbol{K}, \boldsymbol{V})$$

Notice that

$$\mathrm{Att}(i, \boldsymbol{K}, \boldsymbol{V}) = \begin{cases} \boldsymbol{v}_i & i \leq n \\ \boldsymbol{v}_n & i > n \end{cases}$$

We extensively use these properties of the attention function in our construction. We now have all the necessary to begin with the details of our proof.

CONSTRUCTION OF TEnc

The encoder part of Trans is very simple. For TEnc we use a single-layer encoder, such that $\mathrm{TEnc}(\overline{\boldsymbol{x}}_1, \ldots, \overline{\boldsymbol{x}}_n) = (\boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}})$ where

$$\boldsymbol{K}^{\boldsymbol{e}} = (\overline{\boldsymbol{x}}_1, \ldots, \overline{\boldsymbol{x}}_n)$$

that is, it implements the identity function for keys, and

$$\boldsymbol{V}^{\boldsymbol{e}} = (\overline{\boldsymbol{v}}_1, \ldots, \overline{\boldsymbol{v}}_n)$$

where

$$\overline{\boldsymbol{v}}_i = [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, 0, 0, 0, i, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0]$$

That is, $\overline{\boldsymbol{v}}_i$ is just $\overline{\boldsymbol{x}}_i$ where the last component has been moved to the center component. It is straightforward that this can be done with a single layer encoder by just implementing the identity (that can be done by taking advantage of the residual connection in Equations (5) and (6)), and then using affine transformations for $V(\cdot)$ and $K(\cdot)$ in Equation (7). The most important part of the construction is in the decoder that we next describe.

OVERVIEW OF THE CONTRUCTION OF TDec

We base our proof in the construction the following sequences of vectors and values:

$$\begin{aligned}
\boldsymbol{\alpha}_i &= \begin{cases} \boldsymbol{x}_i & 1 \leq i \leq n \\ \boldsymbol{x}_n & i > n \end{cases} \\
\boldsymbol{\beta}_i &= \begin{cases} \boldsymbol{h}_i & 0 \leq i \leq n \\ \sigma(\boldsymbol{W}\boldsymbol{\beta}_{i-1} + \boldsymbol{V}\boldsymbol{\alpha}_i) & i > n \end{cases} \\
\boldsymbol{\gamma}_i &= \begin{cases} \boldsymbol{0} & i \leq n \\ \boldsymbol{\beta}_n & i = n+1 \\ \sigma(\boldsymbol{U}\boldsymbol{\gamma}_{i-1}) & i > n+1 \end{cases} \\
a_i &= \begin{cases} 0 & 0 \leq i \leq n \\ 1 & i > n \end{cases} \\
b_i &= \begin{cases} 0 & i = n+1 \\ 1 & i \neq n+1 \end{cases} \\
c_i &= \begin{cases} i & 0 \leq i \leq n \\ n & i > n \end{cases}
\end{aligned}$$

It is easy to see that since $\boldsymbol{\beta}_0 = \boldsymbol{h}_0$ and $\boldsymbol{\alpha}_i = \boldsymbol{x}_i$ for $1 \leq i \leq n$, then $\boldsymbol{\beta}$ follows the recurrence $\boldsymbol{\beta}_i = \sigma(\boldsymbol{\beta}_{i-1}\boldsymbol{W} + \boldsymbol{x}_i\boldsymbol{V})$ with the same starting value and equations as the $\boldsymbol{h}_i$'s , and so we have that $\boldsymbol{\beta}_i = \boldsymbol{h}_i$ for $i \leq n$. Then, since $\boldsymbol{\gamma}_{n+1} = \boldsymbol{\beta}_n = \boldsymbol{h}_n = \boldsymbol{g}_0$, and after that $\boldsymbol{\gamma}_i = \sigma(\boldsymbol{U}\boldsymbol{\gamma}_{i-1})$, we conclude that $\boldsymbol{\gamma}_{n+1+i} = \boldsymbol{g}_i$. We will show that these sequences can be computed in a transformer network, and so we can simulate the RNN N.

In what follows we show an overview on how to compute all these sequences with the decoder part of Trans. In the decoder, we show how to produce the sequence of outputs $(\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_r)$ where $\boldsymbol{y}_i$ is given by:

$$\boldsymbol{y}_i = [\boldsymbol{0}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, a_i, b_i, c_i, 0, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, 0]$$

We consider as the starting vector for the decoder the vector

$$\boldsymbol{y}_0 = [\boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, 0, 1, 0, 0, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, 0].$$

Thus, what we are actually stating is that $\boldsymbol{\beta}_0 = \boldsymbol{\gamma}_0 = \boldsymbol{0}$, $a_0 = c_0 = 0$ and $b_0 = 1$, which coincide with the values for the sequences described above. Recall that we are using positional encodings, thus, when producing the output, the actual input for the first layer of the decoder will be the sequence

$$\boldsymbol{y}_0 + \mathrm{pos}(1), \boldsymbol{y}_1 + \mathrm{pos}(2), \ldots, \boldsymbol{y}_r + \mathrm{pos}(i+1)$$

We denote by $\overline{\boldsymbol{y}}_i$ the vector $\boldsymbol{y}_i$ plus its positional encoding (which is always at its last component).

We now roughly describe a two-layer decoder network that, with input $\boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}$ and sequence $(\overline{\boldsymbol{y}}_0, \ldots, \overline{\boldsymbol{y}}_r)$ will produce $\boldsymbol{y}_{r+1}$ as output (the details are in the following section). Our construction will resemble a proof by induction; we show how to construct $\boldsymbol{y}_{r+1}$ from the previous vector $\boldsymbol{y}_r$.

The overview of the construction is very simple. With input $\boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}$ and sequence $(\overline{\boldsymbol{y}}_0, \ldots, \overline{\boldsymbol{y}}_r)$ the first decoder layer produces a sequence of vectors $\boldsymbol{z}_0^1, \boldsymbol{z}_1^1, \ldots, \boldsymbol{z}_r^1$ such that

$$\boldsymbol{z}_i^1 = [\boldsymbol{\alpha}_{i+1}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, a_i, b_i, c_i, c_{i+1}, \boldsymbol{\beta}_{i+1}, \boldsymbol{\gamma}_{i+1}, \boldsymbol{0}, a_{i+1}, b_{i+1}, c_{i+1}, i+1] \tag{17}$$

Notice that the first decoder layer will do almost all the heavy work. We actually use a feed-forward network with 3 layers to obtain this output. After that, the second decoder layer, just do some cleaning work by moving some parts of the vector and deleting others to produce the sequence of output vectors $\boldsymbol{z}_0^2, \boldsymbol{z}_1^2, \ldots, \boldsymbol{z}_r^2$ where

$$\boldsymbol{z}_i^2 = [\boldsymbol{0}, \boldsymbol{\beta}_{i+1}, \boldsymbol{\gamma}_{i+1}, a_{i+1}, b_{i+1}, c_{i+1}, 0, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, 0, 0, 0, 0]. \tag{18}$$

This is exactly what we want as output, and thus, we just consider the final output transformation $F(\cdot)$ as the identity to produce $\boldsymbol{y}_{r+1} = F(\boldsymbol{z}_r^2)$ (Equation (11)) which completes our construction.

DETAILED CONSTRUCTION OF TDec

First, we have the following useful identities for $\boldsymbol{\alpha}_i$, $\boldsymbol{\beta}_i$ and $\boldsymbol{\gamma}_i$ that can be obtained by simply inspecting the definitions above and the attention function that we defined:

$$\begin{aligned}
\boldsymbol{\alpha}_i &= \mathrm{Att}(i, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}})_{1:d} \\
\boldsymbol{\beta}_i &= \begin{cases} \mathbf{0} & i = 0 \\ \sigma(\boldsymbol{\beta}_{i-1}\boldsymbol{W} + \boldsymbol{\alpha}_i \boldsymbol{V}) & i > 0 \end{cases} \\
\boldsymbol{\gamma}_i &= \begin{cases} \mathbf{0} & i = 0 \\ \sigma(\boldsymbol{\gamma}_{i-1}\boldsymbol{U}) + (1 - b_i)\boldsymbol{\beta}_{i-1} & i > 0 \end{cases}
\end{aligned}$$

We also have some identities for a_i, b_i and c_i:

$$\begin{aligned}
c_i &= \begin{cases} 0 & i = 0 \\ \mathrm{Att}(i, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}})_{3d+4} & i > 0 \end{cases} \\
a_i &= \begin{cases} 0 & i = 0 \\ 1 - (c_i - c_{i-1}) & i > 0 \end{cases} \\
b_i &= \begin{cases} 1 & i = 0 \\ 1 - (a_i - a_{i-1}) & i > 0 \end{cases}
\end{aligned}$$

We also note that b_i can be rewritten as:

$$b_i = \begin{cases} 1 & i = 0 \\ c_i - c_{i-1} + a_{i-1} & i > 0 \end{cases}$$

The first layer of the decoder, that we call Dec_1, is defined by the functions $Q_1(\cdot)$, $V_1(\cdot)$, $K_1(\cdot)$, and $O_1(\cdot)$ that we now describe. We use the function $V_1 : \mathbb{Q}^{6d+8} \to \mathbb{Q}^{6d+8}$ such that $V_1(\boldsymbol{v})$ is the vector with only zeroes, thus, we just *nullify* the self-attention in the first decoder layer. This plus the residual connection in Equation (8) has the net effect of producing the identity and thus we obtain $\boldsymbol{p}_i^1 = \overline{\boldsymbol{y}}_i$ for every $i \in \{0, \ldots, r\}$. Notice that $e(\boldsymbol{p}_i^1) = e(\overline{\boldsymbol{y}}_i) = i + 1$, and thus, if $i + 1 \leq n$ we have that

$$\mathrm{Att}(\boldsymbol{p}_i^1, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}) = \mathrm{Att}(i+1, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}) = \overline{\boldsymbol{v}}_{i+1} = [\boldsymbol{x}_{i+1}, \mathbf{0}, \mathbf{0}, 0, 0, 0, i+1, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0]$$

and, otherwise, if $i + 1 > n$ we have

$$\mathrm{Att}(\boldsymbol{p}_i^1, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}) = \mathrm{Att}(i+1, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}) = \overline{\boldsymbol{v}}_{n} = [\boldsymbol{x}_{n}, \mathbf{0}, \mathbf{0}, 0, 0, 0, n, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0].$$

Thus, combining both cases we actually have that

$$\mathrm{Att}(\boldsymbol{p}_i^1, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}) = [\boldsymbol{\alpha}_{i+1}, \mathbf{0}, \mathbf{0}, 0, 0, 0, c_{i+1}, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0].$$

This last result is summed with $\boldsymbol{p}_i^1 = \overline{\boldsymbol{y}}_i$ (the residual connection in Equation (9)) to obtain $\boldsymbol{a}_i^1$ which is given by:

$$\begin{aligned}
\boldsymbol{a}_i^1 &= \mathrm{Att}(\boldsymbol{p}_i^1, \boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}}) + \boldsymbol{p}_i^1 \\
&= [\boldsymbol{\alpha}_{i+1}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, a_i, b_i, c_i, c_{i+1}, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, i+1]
\end{aligned}$$

We now describe the function $O_1(\cdot)$ which is going to be defined by a feed-forward neural network that we next describe. The first layer of the network will be given by a function $f_1 : \mathbb{Q}^{6d+8} \to \mathbb{Q}^{6d+8}$ such that with input $\boldsymbol{a}_i^1$ it will produce

$$f_1(\boldsymbol{a}_i^1) = [\boldsymbol{\alpha}_{i+1}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, 0, 0, 0, 0, \mathbf{0}, \mathbf{0}, \mathbf{0}, a_{i+1}, b_{i+1}, 0, 0].$$

It is clear that f_1 can be implemented with a single neural-network layer as it only has to apply the affine transformation given by the equations $a_{i+1} = 1 - (c_{i+1} - c_i)$ and $b_{i+1} = c_{i+1} - c_i + a_i$, and making some positions equal to 0.

After f_1 we apply another feed-forward layer $f_2 : \mathbb{Q}^{6d+8} \to \mathbb{Q}^{6d+8}$ given by

$$f_2(f_1(\boldsymbol{a}_i^1)) = \sigma([\mathbf{0}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, 0, 0, 0, 0, \boldsymbol{\beta}_i \boldsymbol{W} + \boldsymbol{\alpha}_{i+1}\boldsymbol{V}, \boldsymbol{\gamma}_i \boldsymbol{U}, \boldsymbol{\beta}_i - b_{i+1}\mathbf{1}, a_{i+1}, b_{i+1}, 0, 0])$$

where $\mathbf{1}$ is a vector in $\mathbb{Q}^d$ with only ones. It is clear that the internal vector can be computed with an affine transformation. Since a_{i+1}, b_{i+1} are all rationals between 0 and 1, and all the values in $\boldsymbol{\beta}_i$ and $\boldsymbol{\gamma}_i$ are also between 0 and 1, when we apply the function σ we obtain

$$f_2(f_1(\boldsymbol{a}_i^1)) = [\mathbf{0}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, 0, 0, 0, 0, \sigma(\boldsymbol{\beta}_i \boldsymbol{W} + \boldsymbol{\alpha}_{i+1}\boldsymbol{V}), \sigma(\boldsymbol{\gamma}_i \boldsymbol{U}), \sigma(\boldsymbol{\beta}_i - b_{i+1}\mathbf{1}), a_{i+1}, b_{i+1}, 0, 0]).$$

We note that $\sigma(\boldsymbol{\beta}_i \boldsymbol{W} + \boldsymbol{\alpha}_{i+1}\boldsymbol{V})$ is actually $\boldsymbol{\beta}_{i+1}$. Moreover, as we next describe, it holds that $\sigma(\boldsymbol{\beta}_i - b_{i+1}\mathbf{1}) = (1 - b_{i+1})\boldsymbol{\beta}_i$. To see this, notice that $\boldsymbol{\beta}_i$ has all its values between 0 and 1. Then, if $b_{i+1} = 1$ we have that $\boldsymbol{\beta}_i - b_{i+1}\mathbf{1}$ is a vector with all its components less than or equal to 0 which implies that $\sigma(\boldsymbol{\beta}_i - b_{i+1}\mathbf{1}) = \mathbf{0}$. On the other hand, if $b_{i+1} = 0$ then $\sigma(\boldsymbol{\beta}_i - b_{i+1}\mathbf{1}) = \sigma(\boldsymbol{\beta}_i) = \boldsymbol{\beta}_i$. Thus we can write the output of this layer as:

$$f_2(f_1(\boldsymbol{a}_i^1)) \quad = \quad [\mathbf{0}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, 0, 0, 0, 0, \boldsymbol{\beta}_{i+1}, \sigma(\boldsymbol{\gamma}_i \boldsymbol{U}), (1 - b_{i+1})\boldsymbol{\beta}_i, a_{i+1}, b_{i+1}, 0, 0]).$$

We need a final linear layer given by a function $f_3 : \mathbb{Q}^{6d+8} \rightarrow \mathbb{Q}^{6d+8}$ that does the following computation

$$f_3(f_2(f_1(\boldsymbol{a}_i^1))) \quad = \quad [\mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0, \boldsymbol{\beta}_{i+1}, \sigma(\boldsymbol{\gamma}_i \boldsymbol{U}) + (1 - b_{i+1})\boldsymbol{\beta}_i, \mathbf{0}, a_{i+1}, b_{i+1}, 0, 0]).$$

Notice that $\sigma(\boldsymbol{\gamma}_i \boldsymbol{U}) + (1 - b_{i+1})\boldsymbol{\beta}_i$ is exactly $\boldsymbol{\gamma}_{i+1}$ thus the output of f_3 can be written as

$$f_3(f_2(f_1(\boldsymbol{a}_i^1))) \quad = \quad [\mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0, \boldsymbol{\beta}_{i+1}, \boldsymbol{\gamma}_{i+1}, \mathbf{0}, a_{i+1}, b_{i+1}, 0, 0]).$$

This is the output of function $O_1(\cdot)$. Thus. we have that

$$\begin{aligned} O_1(\boldsymbol{a}_i^1) &= f_3(f_2(f_1(\boldsymbol{a}_i^1))) \\ &= [\mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0, \boldsymbol{\beta}_{i+1}, \boldsymbol{\gamma}_{i+1}, \mathbf{0}, a_{i+1}, b_{i+1}, 0, 0]). \end{aligned}$$

Now the output of this decoder layer is $\boldsymbol{z}_i^1 = O_1(\boldsymbol{a}_i^1) + \boldsymbol{a}_i^1$ (Equation (10)) and thus we have

$$\begin{aligned} \boldsymbol{z}_i^1 &= O_1(\boldsymbol{a}_i^1) + \boldsymbol{a}_i^1 \\ &= [\boldsymbol{\alpha}_{i+1}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i, a_i, b_i, c_i, c_{i+1}, \boldsymbol{\beta}_{i+1}, \boldsymbol{\gamma}_{i+1}, \mathbf{0}, a_{i+1}, b_{i+1}, 0, i+1]). \end{aligned}$$

For the second decoder layer Dec_2, it is straightforward to construct a layer that just do some cleaning and thus from $\boldsymbol{z}_0^1, \boldsymbol{z}_1^1, \ldots, \boldsymbol{z}_r^1$ produces the sequence $\boldsymbol{z}_0^2, \boldsymbol{z}_1^2, \ldots, \boldsymbol{z}_r^2$ such that

$$\boldsymbol{z}_i^2 \quad = \quad [\mathbf{0}, \boldsymbol{\beta}_{i+1}, \boldsymbol{\gamma}_{i+1}, a_{i+1}, b_{i+1}, c_{i+1}, 0, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0]).$$

We finally use $F(\cdot)$ as the identity function to produce our desired output from $\boldsymbol{z}_r^2$, that is

$$\boldsymbol{y}_{r+1} = F(\boldsymbol{z}_r^2) = [\mathbf{0}, \boldsymbol{\beta}_{r+1}, \boldsymbol{\gamma}_{r+1}, a_{r+1}, b_{r+1}, c_{r+1}, 0, \mathbf{0}, \mathbf{0}, \mathbf{0}, 0, 0, 0, 0]).$$

This completes the construction and the correctness of our construction.

FINAL STEP

We now can use our Trans network to construct the recognizer $A' = (\Sigma, f', \mathrm{Trans}, \boldsymbol{s}', \mathbb{F}')$. We use $\boldsymbol{s}'$ as simple $\boldsymbol{y}_0$ in our above construction. Finally, notice that in Trans the vector $\boldsymbol{\gamma}_i$ is actually the output of the RNN encoder-decoder that we are simulating but only for $i \geq n+1$. We take advantage of the value a_i which is 1 if and only if $i \geq n+1$. Thus we can define $\mathbb{F}' = \mathbb{Q}^{2d} \times \mathbb{F} \times \{1\} \times \mathbb{Q}^{3d+7}$. This completes the proof of the Theorem.

C PROOFS FOR SECTION 4

C.1 PROOF OF THEOREM 4.1

The formulas of the Neural GPU in detail are as follows (with $\mathbf{S}^0$ the initial input tensor):

$$\begin{aligned}
\mathbf{U}^t &= U(\mathbf{S}^{t-1}) \\
\mathbf{R}^t &= R(\mathbf{S}^{t-1}) \\
\mathbf{S}^t &= \mathbf{U}^t \odot \mathbf{S}^{t-1} \; + \; (\mathbf{1} - \mathbf{U}) \odot F(\mathbf{R}^t \odot \mathbf{S}^{t-1})
\end{aligned}$$

With $U(\cdot)$, $R(\cdot)$, and $F(\cdot)$ defined as

$$\begin{aligned}
U(\mathbf{X}) &= f_U(\mathbf{K}^U * \mathbf{X} + \mathbf{B}^U) \\
R(\mathbf{X}) &= f_R(\mathbf{K}^R * \mathbf{X} + \mathbf{B}^R) \\
F(\mathbf{X}) &= f_F(\mathbf{K}^F * \mathbf{X} + \mathbf{B}^F)
\end{aligned}$$

Consider now an RNN encoder-decoder N of dimension d and composed of the equations

$$\begin{aligned}
\boldsymbol{h}_i &= \sigma(\boldsymbol{x}_i \boldsymbol{W} + \boldsymbol{h}_{i-1} \boldsymbol{V}) \\
\boldsymbol{g}_t &= \sigma(\boldsymbol{g}_{t-1} \boldsymbol{U})
\end{aligned}$$

with $\boldsymbol{h}_0 = \mathbf{0}$ and $\boldsymbol{g}_0 = \boldsymbol{h}_n$ where n is the length of the input.

CONSTRUCTING THE NEURAL GPU TO SIMULATE N

We construct a Neural GPU network NGPU that simulates N as follows. Assume that the input of N is $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. Then we first construct the sequence $\boldsymbol{X}' = (\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_n)$ such that $\boldsymbol{x}'_i = [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, 1, 1, 0]$ with $\mathbf{0} \in \mathbb{Q}^d$ the vector with all values as 0. Notice that $\boldsymbol{x}'_i \in \mathbb{Q}^{3d+3}$, moreover it is straightforward that if $\boldsymbol{x}_i$ was constructed from an embedding function $f : \Sigma \to \mathbb{Q}^d$ applied to a symbol $a \in \Sigma$, then $\boldsymbol{x}'_i$ can also be constructed with an embedding function $f' : \Sigma \to \mathbb{Q}^{3d+3}$ such that $f'(a) = [f(a), \mathbf{0}, \mathbf{0}, 1, 1, 0]$.

We consider an input tensor $\mathbf{S} \in \mathbb{Q}^{n \times 1 \times 3d+3}$ such that for every $i \in \{1, \ldots, n\}$ it holds that $\mathbf{S}_{i,1,:} = \boldsymbol{x}'_i = [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, 1, 1, 0]$. Notice that since we picked $w = 1$, our tensor $\mathbf{S}$ is actually a $2D$ grid. Our proof shows that a bi-dimensional tensor is enough for simulating an RNN.

We now describe how to construct the kernel banks $\mathbf{K}^U$, $\mathbf{K}^R$ and $\mathbf{K}^F$ of shape $(2, 1, 3d+3, 3d+3)$. Notice that for each kernel $\mathbf{K}^X$ we essentially have to define two matrices $\mathbf{K}^X_{1,1,:,:}$ and $\mathbf{K}^X_{2,1,:,:}$ each one of dimension $(3d+3) \times (3d+3)$. We begin by defining every matrix in $\mathbf{K}^F$ as block matrices. When defining the matrices, all blank spaces are considered to be 0.

$$\mathbf{K}^F_{1,1,:,:} = \left[\begin{array}{c|c|c|c}
 & & & \\ \hline
\boldsymbol{V} & \boldsymbol{V} & & \\ \hline
 & & & \\ \hline
 & & & \boldsymbol{F}_1
\end{array}\right]$$

$$\mathbf{K}^F_{2,1,:,:} = \left[\begin{array}{c|c|c|c}
\boldsymbol{W} & \boldsymbol{W} & & \\ \hline
 & & \boldsymbol{U} & \\ \hline
 & & \boldsymbol{U} & \\ \hline
 & & & \boldsymbol{F}_2
\end{array}\right]$$

where $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$ are 3×3 matrices defined by

$$\boldsymbol{F}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \boldsymbol{F}_2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Tensors $\mathbf{K}^U$ and $\mathbf{K}^R$ are considerable simpler. For the case of $\mathbf{K}^U$ we have

$$\mathbf{K}^U_{1,1,:,:} = \left[\begin{array}{c|c|c|c} & & & \\ & & & \\ \hline \boldsymbol{A} & \boldsymbol{A} & & \boldsymbol{U}_1 \end{array}\right]$$

$$\mathbf{K}^U_{2,1,:,:} = \left[\begin{array}{c|c|c|c} & & & \\ & & & \\ \hline & & \boldsymbol{A} & \boldsymbol{U}_2 \end{array}\right]$$

where $\boldsymbol{U}_1$ and $\boldsymbol{U}_2$ are 3×3 matrices defined by

$$\boldsymbol{U}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \boldsymbol{U}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and where $\boldsymbol{A}$ is the $3 \times d$ matrix defined by

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Finally, we define $\mathbf{K}^R$ as

$$\mathbf{K}^R_{1,1,:,:} = \left[\begin{array}{c|c|c|c} & & & \\ & & & \\ \hline & & & \end{array}\right]$$

$$\mathbf{K}^R_{2,1,:,:} = \left[\begin{array}{c|c|c|c} & & & \\ & & & \\ \hline \boldsymbol{A} & \boldsymbol{B} & & \boldsymbol{R}_2 \end{array}\right]$$

where $\boldsymbol{R}_2$ is the 3×3 matrix defined by

$$\boldsymbol{R}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and where $\boldsymbol{B}$ is the $3 \times d$ matrix defined by

$$\boldsymbol{B} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

The bias tensors $\mathbf{B}^U$ and $\mathbf{B}^F$ are $\mathbf{0}$ (the tensor with all values 0). Finally, to construct tensor $\mathbf{B}^R$ we consider the matrix $\boldsymbol{D}$ of dimension $1 \times (3d+3)$ such that

$$\boldsymbol{D} = [\ \mathbf{0} \quad \mathbf{0} \quad \mathbf{1} \quad 0 \quad 0 \quad 1\ ]$$

Then we let $\mathbf{B}^R_{i,:,:} = \boldsymbol{D}$ for all i. Finally, we consider $f_U = f_R = f_F = \sigma$. The constructed Neural GPU is a uniform Neural GPU.

Before continuing with the proof we note that for every kernel $\mathbf{K}^X$ and tensor $\mathbf{S}$ we have that

$$(\mathbf{K}^X * \mathbf{S})_{i,1,:} = \mathbf{S}_{i-1,1,:}\mathbf{K}^X_{1,1,:,:} + \mathbf{S}_{i,1,:}\mathbf{K}^X_{2,1,:,:}$$

Correctness of the construction

We now prove that the following properties hold for every $t \geq 0$:

$$\mathbf{S}^t_{i,1,:} = \begin{cases} [\mathbf{0}, \mathbf{0}, \boldsymbol{\alpha}^i_{t-i}, 0, 0, 0] & \text{for } i < t \\ [\boldsymbol{h}_i, \boldsymbol{h}_i, \mathbf{0}, 0, 1, 0] & \text{for } i = t \\ [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, 1, 1, 0] & \text{for } i > t \end{cases} \tag{19}$$

where $\boldsymbol{\alpha}_j^k$ is given by the recurrence $\boldsymbol{\alpha}_0^k = \boldsymbol{h}_k$ and $\boldsymbol{\alpha}_j^k = \sigma(\boldsymbol{\alpha}_{j-1}^k \boldsymbol{U})$. Notice that $\boldsymbol{g}_j = \boldsymbol{\alpha}_j^n$. That is, we are going to prove that our construction actually simulates N. By (19) one can see that the intuition in our construction is to use the first d components to simulate the encoder part, the next d components to communicate data between the encoder and decoder simulation, and the next d components to simulate the decoder part. The last three components are needed as gadgets for the gates to actually simulate a sequencial read of the input, and to ensure that the hidden state of the encoder and decoder are updated properly.

We prove the above statement by induction in t. First notice that the property trivially holds for $\mathbf{S}^0$. Now assume that this holds for $t-1$ and lets prove it for t. We know that $\mathbf{U}^t$ is computed as

$$\mathbf{U}^t = \sigma(\mathbf{K}^U * \mathbf{S}^{t-1} + \mathbf{B}^U) = \sigma(\mathbf{K}^U * \mathbf{S}^{t-1})$$

Thus we have that:

$$\begin{aligned} \mathbf{U}^t_{i,1,:} &= \sigma((\mathbf{K}^U * \mathbf{S}^{t-1})_{i,1,:}) \\ &= \sigma(\mathbf{S}^{t-1}_{i-1,1,:}\mathbf{K}^U_{1,1,:,:} + \mathbf{S}^{t-1}_{i,1,:}\mathbf{K}^U_{2,1,:,:}) \end{aligned}$$

By the induction hypothesis we have

$$\mathbf{S}^{t-1}_{i,1,:} = \begin{cases} [\mathbf{0}, \mathbf{0}, \boldsymbol{\alpha}^i_{t-1-i}, 0, 0, 0] & \text{for } i < t-1 \\ [\boldsymbol{h}_i, \boldsymbol{h}_i, \mathbf{0}, 0, 1, 0] & \text{for } i = t-1 \\ [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, 1, 1, 0] & \text{for } i > t-1 \end{cases} \tag{20}$$

Now, notice that $\mathbf{K}^U_{1,1,:,:}$ and $\mathbf{K}^U_{2,1,:,:}$ are not zero only in its three last rows, thus we can focus on the three last components of the vectors in $\mathbf{S}^{t-1}$, and then we can compute $\mathbf{U}^t_{i,1,:}$ as

$$\begin{aligned} \mathbf{U}^t_{i,1,:} &= \begin{cases} \sigma([_,_,_,0,0,0]\mathbf{K}^U_{1,1,:,:} + [_,_,_,0,0,0]\mathbf{K}^U_{2,1,:,:}) & \text{for } i < t-1 \\ \sigma([_,_,_,0,0,0]\mathbf{K}^U_{1,1,:,:} + [_,_,_,0,1,0]\mathbf{K}^U_{2,1,:,:}) & \text{for } i = t-1 \\ \sigma([_,_,_,0,1,0]\mathbf{K}^U_{1,1,:,:} + [_,_,_,1,1,0]\mathbf{K}^U_{2,1,:,:}) & \text{for } i = t \\ \sigma([_,_,_,1,1,0]\mathbf{K}^U_{1,1,:,:} + [_,_,_,1,1,0]\mathbf{K}^U_{2,1,:,:}) & \text{for } i > t \end{cases} \\ &= \begin{cases} \sigma([\mathbf{0},\mathbf{0},\mathbf{0},0,0,0]) & \text{for } i < t-1 \\ \sigma([\mathbf{0},\mathbf{0},\mathbf{0},0,0,0]) & \text{for } i = t-1 \\ \sigma([\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) & \text{for } i = t \\ \sigma([\mathbf{1},\mathbf{1},\mathbf{1},1,1,1]) & \text{for } i > t \end{cases} \\ &= \begin{cases} [\mathbf{0},\mathbf{0},\mathbf{0},0,0,0] & \text{for } i < t \\ [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1] & \text{for } i = t \\ [\mathbf{1},\mathbf{1},\mathbf{1},1,1,1] & \text{for } i > t \end{cases} \end{aligned}$$

Now, for $\mathbf{R}^t$ we have

$$\mathbf{R}^t = \sigma(\mathbf{K}^R * \mathbf{S}^{t-1} + \mathbf{B}^R)$$

and thus for $\mathbf{R}^t_{i,1,:}$ we have

$$\begin{aligned} \mathbf{R}^t_{i,1,:} &= \sigma((\mathbf{K}^R * \mathbf{S}^{t-1})_{i,1,:} + \mathbf{B}^R_{i,1,:}) \\ &= \sigma(\mathbf{S}^{t-1}_{i-1,1,:}\mathbf{K}^R_{1,1,:,:} + \mathbf{S}^{t-1}_{i,1,:}\mathbf{K}^R_{2,1,:,:} + \mathbf{B}^R_{i,1,:}) \\ &= \sigma(\mathbf{S}^{t-1}_{i,1,:}\mathbf{K}^R_{2,1,:,:} + \mathbf{B}^R_{i,1,:}) \\ &= \sigma(\mathbf{S}^{t-1}_{i,1,:}\mathbf{K}^R_{2,1,:,:} + [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) \end{aligned}$$

where we deleted the term with $\mathbf{K}^R_{1,1,:,:}$ since it is the null matrix. Then by using the definition of $\mathbf{S}^{t-1}_{i,1,:}$ above (Equation (20)) we have

$$\begin{aligned}
\mathbf{R}^t_{i,1,:} &= \begin{cases} \sigma([_,_,_,0,0,0]\mathbf{K}^R_{2,1,:,:} + [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) & \text{for } i < t-1 \\ \sigma([_,_,_,0,1,0]\mathbf{K}^R_{2,1,:,:} + [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) & \text{for } i = t-1 \\ \sigma([_,_,_,1,1,0]\mathbf{K}^R_{2,1,:,:} + [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) & \text{for } i = t \\ \sigma([_,_,_,1,1,0]\mathbf{K}^R_{2,1,:,:} + [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) & \text{for } i > t \end{cases} \\
&= \begin{cases} \sigma([\mathbf{0},\mathbf{0},\mathbf{1},0,0,1]) & \text{for } i < t-1 \\ \sigma([\mathbf{0},\mathbf{1},\mathbf{1},0,1,1]) & \text{for } i = t-1 \\ \sigma([\mathbf{1},\mathbf{1},\mathbf{1},1,1,1]) & \text{for } i = t \\ \sigma([\mathbf{1},\mathbf{1},\mathbf{1},1,1,1]) & \text{for } i > t \end{cases} \\
&= \begin{cases} [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1] & \text{for } i < t-1 \\ [\mathbf{0},\mathbf{1},\mathbf{1},0,1,1] & \text{for } i = t-1 \\ [\mathbf{1},\mathbf{1},\mathbf{1},1,1,1] & \text{for } i \geq t \end{cases}
\end{aligned}$$

We can now compute $\mathbf{S}^t_{i,1,:}$. By the definition of $\mathbf{S}^t$ we have

$$\mathbf{S}^t_{i,1,:} = \mathbf{U}^t_{i,1,:} \odot \mathbf{S}^{t-1}_{i,1,:} + (\mathbf{1}_{i,1,:} - \mathbf{U}_{i,1,:}) \odot \sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:})$$

where we dropped $\mathbf{B}^F$ that has only zeros. First, by using what we already computed for $\mathbf{U}^t_{i,1,:}$ we have that

$$\mathbf{S}^t_{i,1,:} = \begin{cases} \sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}) & \text{for } i < t \\ [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1] \odot \mathbf{S}^{t-1}_{i,1,:} + [\mathbf{1},\mathbf{1},\mathbf{0},1,1,0] \odot \sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}) & \text{for } i = t \\ \mathbf{S}^{t-1}_{i,1,:} & \text{for } i > t \end{cases}$$

When $i = t$ we have that $\mathbf{S}^{t-1}_{i,1,:} = [\boldsymbol{x}_i, \mathbf{0}, \mathbf{0}, 1, 0, 0]$ (Equation (20)), thus $[\mathbf{0},\mathbf{0},\mathbf{1},0,0,1] \odot \mathbf{S}^{t-1}_{i,1,:} = [\mathbf{0},\mathbf{0},\mathbf{0},0,0,0]$. Then

$$\mathbf{S}^t_{i,1,:} = \begin{cases} \sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}) & \text{for } i < t \\ [\mathbf{1},\mathbf{1},\mathbf{0},1,1,0] \odot \sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}) & \text{for } i = t \\ [\boldsymbol{x}_i,\mathbf{0},\mathbf{0},1,1,0] & \text{for } i > t \end{cases} \tag{21}$$

We are almost done with the inductive step, we only need to compute $\sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:})$. Given what we have for $\mathbf{R}^t$ and $\mathbf{S}^{t-1}$ we have that $\mathbf{R}^t \odot \mathbf{S}^{t-1}$ is

$$\begin{aligned}
(\mathbf{R}^t \odot \mathbf{S}^{t-1})_{i,1,:} &= \begin{cases} [\mathbf{0},\mathbf{0},\mathbf{1},0,0,1] \odot [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^i_{t-1-i},0,0,0] & \text{for } i < t-1 \\ [\mathbf{0},\mathbf{1},\mathbf{1},0,1,1] \odot [\boldsymbol{h}_i,\boldsymbol{h}_i,\mathbf{0},0,1,0] & \text{for } i = t-1 \\ [\mathbf{1},\mathbf{1},\mathbf{1},1,1,1] \odot [\boldsymbol{x}_i,\mathbf{0},\mathbf{0},1,1,0] & \text{for } i \geq t \end{cases} \\
&= \begin{cases} [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^i_{t-1-i},0,0,0] & \text{for } i < t-1 \\ [\mathbf{0},\boldsymbol{h}_i,\mathbf{0},0,1,0] & \text{for } i = t-1 \\ [\boldsymbol{x}_i,\mathbf{0},\mathbf{0},1,1,0] & \text{for } i \geq t \end{cases}
\end{aligned}$$

Lets $\mathbf{T}^t = \sigma(\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))$. Notice that from Equation (21) we actually need to know the values in $\mathbf{T}^t_{i,1,:}$ only for $i \leq t$. Now we have that

$$
\begin{aligned}
\mathbf{T}^t_{i,1,:} &= \sigma((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}) \\
&= \sigma((\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}\mathbf{K}^F_{1,1,:,:} + (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,1,:}\mathbf{K}^F_{2,1,:,:}) \\
&= \begin{cases} \sigma([\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i-1}_{t-i},0,0,0]\mathbf{K}^F_{1,1,:,:} + [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i}_{t-1-i},0,0,0]\mathbf{K}^F_{2,1,:,:}) & \text{for } i < t-1 \\ \sigma([\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i-1}_{t-i},0,0,0]\mathbf{K}^F_{1,1,:,:} + [\mathbf{0},\boldsymbol{h}_i,\mathbf{0},0,1,0]\mathbf{K}^F_{2,1,:,:}) & \text{for } i = t-1 \\ \sigma([\mathbf{0},\boldsymbol{h}_{i-1},\mathbf{0},0,1,0]\mathbf{K}^F_{1,1,:,:} + [\boldsymbol{x}_i,\mathbf{0},\mathbf{0},1,1,0]\mathbf{K}^F_{2,1,:,:}) & \text{for } i = t \end{cases} \\
&= \begin{cases} \sigma([\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i}_{t-1-i}\boldsymbol{U},0,0,0]) & \text{for } i < t-1 \\ \sigma([\mathbf{0},\mathbf{0},\boldsymbol{h}_i\boldsymbol{U},0,0,0]) & \text{for } i = t-1 \\ \sigma([\boldsymbol{x}_i\boldsymbol{W} + \boldsymbol{h}_{i-1}\boldsymbol{V},\boldsymbol{x}_i\boldsymbol{W} + \boldsymbol{h}_{i-1}\boldsymbol{V},\mathbf{0},0,1,0]) & \text{for } i = t \end{cases} \\
&= \begin{cases} [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i}_{t-i},0,0,0] & \text{for } i < t-1 \\ [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i}_{1},0,0,0] & \text{for } i = t-1 \\ [\boldsymbol{h}_i,\boldsymbol{h}_i,\mathbf{0},0,1,0] & \text{for } i = t \end{cases} \\
&= \begin{cases} [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i}_{t-i},0,0,0] & \text{for } i < t \\ [\boldsymbol{h}_i,\boldsymbol{h}_i,\mathbf{0},0,1,0] & \text{for } i = t \end{cases}
\end{aligned}
$$

Putting the value of $\mathbf{T}^t_{i,1,:}$ in Equation (21) we obtain

$$
\begin{aligned}
\mathbf{S}^t_{i,1,:} &= \begin{cases} \mathbf{T}^t_{i,1,:} & \text{for } i < t \\ [\mathbf{1},\mathbf{1},\mathbf{0},1,1,0] \odot \mathbf{T}^t_{i,1,:} & \text{for } i = t \\ [\boldsymbol{x}_i,\mathbf{0},\mathbf{0},1,1,0] & \text{for } i > t \end{cases} \\
&= \begin{cases} [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{i}_{t-i},0,0,0] & \text{for } i < t \\ [\boldsymbol{h}_i,\boldsymbol{h}_i,\mathbf{0},0,1,0] & \text{for } i = t \\ [\boldsymbol{x}_i,\mathbf{0},\mathbf{0},1,1,0] & \text{for } i > t \end{cases}
\end{aligned}
$$

which is exactly what we needed to prove (Equation (19)).

Now, lets focus on $\mathbf{S}^{n+t}_{n,1,:}$ for $t \geq 1$. By what we have just proved, we obtain that

$$\mathbf{S}^{n+t}_{n,1,:} = [\mathbf{0},\mathbf{0},\boldsymbol{\alpha}^{n}_{t},0,0,0] = [\mathbf{0},\mathbf{0},\boldsymbol{g}_t,0,0,0]$$

which is the decoder part of the RNN N. Thus, we can simulate the complete network N with a Neural GPU.

C.2 Proof of Proposition 4.2

We first prove the following claim: Assume that $\mathbf{S} \in \mathbb{Q}^{h \times w \times d}$ is a tensor that satisfies the following property: there exists a $p \geq 1$ that divides h and such that, for every $i \in \{1, 2, \ldots, h-p\}$ it holds that

$$\mathbf{S}_{i,:,:} = \mathbf{S}_{i+p,:,:}.$$

Given that we will be considering circular convolutions, then we have that for every $\ell \leq 0$ we have that $\mathbf{S}_{\ell,j,:} = \mathbf{S}_{h+\ell,j,:}$ and for every $\ell > h$ we have that $\mathbf{S}_{\ell,j,:} = \mathbf{S}_{h-\ell,j,:}$. With this we have that for every $i \in \mathbb{N}$ it holds that $\mathbf{S}_{i,:,:} = \mathbf{S}_{i+p,:,:}$ that is, we do not need to restrict to values in $i \in \{1, 2, \ldots, h-p\}$.

Now lets $\mathbf{K}$ be an arbitrary kernel bank of shape (k_H, k_W, d, d). Let $\mathbf{T} = \mathbf{K} \circledast \mathbf{S}$ where $\circledast$ denotes the circular convolution. We prove next that

$$\mathbf{T}_{i,:,:} = \mathbf{T}_{i+p,:,:}$$

for every i. This is a simple fact that follows from the way in which the convolution is defined. We use the notation in the body of the paper for $\mathbf{T} = \mathbf{K} \circledast \mathbf{S}$, that is, we denote by $\boldsymbol{s}_{ij}$ the vector $\mathbf{S}_{i,j,:}$ and $\boldsymbol{K}_{ij}$ the matrix $\mathbf{K}_{i,j,:,:}$. Notice that $\boldsymbol{s}_{ij} = \boldsymbol{s}_{i+p,j}$ for every $i \in \mathbb{N}$. Now the circular convolution is

$$\mathbf{T}_{i,j,:} = (\mathbf{K} \circledast \mathbf{S})_{i,j,:} = \sum_{u=1}^{k_H} \sum_{v=1}^{k_W} \boldsymbol{s}_{i+\Delta_1(u),j+\Delta_2(v)} \, \boldsymbol{K}_{uv}$$

where $\Delta_1(u) = u - \lfloor k_H/2 \rfloor - 1$ and $\Delta_2(v) = v - \lfloor k_W/2 \rfloor - 1$. Then, given that $\boldsymbol{s}_{ij} = \boldsymbol{s}_{i+p,j}$ for every $i \in \mathbb{N}$ we have that

$$\mathbf{T}_{i,j,:} = (\mathbf{K} \circledast \mathbf{S})_{i,j,:} = \sum_{u=1}^{k_H} \sum_{v=1}^{k_W} \boldsymbol{s}_{i+\Delta_1(u)+p,j+\Delta_2(v)} \, \boldsymbol{K}_{uv} = (\mathbf{K} \circledast \mathbf{S})_{i+p,j,:} = \mathbf{T}_{i+p,j,:}$$

and then, $\mathbf{T}_{i,:,:} = \mathbf{T}_{i+p,:,:}$.

Consider now an arbitrary uniform Neural GPU that processes tensor $\mathbf{S}$ above, and assume that $\mathbf{S}^1, \mathbf{S}^2, \ldots, \mathbf{S}^r$ is the sequence produced by it. Next we prove that for every t and for every i it holds that $\mathbf{S}^t_{i,:,:} = \mathbf{S}^t_{i+p,:,:}$. We prove it by induction in t. For the case $\mathbf{S}^0$ it holds by definition. Thus assume that $\mathbf{S}^{t-1}$ satisfies the property. Let

$$\begin{aligned}
\mathbf{U}^t &= f_U(\mathbf{K}^U * \mathbf{S}^{t-1} + \mathbf{B}^U) \\
\mathbf{R}^t &= f_R(\mathbf{K}^R * \mathbf{S}^{t-1} + \mathbf{B}^R) \\
\mathbf{S}^t &= \mathbf{U}^t \odot \mathbf{S}^{t-1} \; + \; (\mathbf{1} - \mathbf{U}) \odot f_F(\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}) + \mathbf{B}^F)
\end{aligned}$$

Since we are considering uniform Neural GPUs, we know that there exist three matrices $\boldsymbol{B}^U$, $\boldsymbol{B}^R$ and $\boldsymbol{B}^F$ such that for every i it holds that $\mathbf{B}^U_{i,:,:} = \boldsymbol{B}^U$, $\mathbf{B}^R_{i,:,:} = \boldsymbol{B}^R$, and $\mathbf{B}^F_{i,:,:} = \boldsymbol{B}^F$. It is easy to prove that $\mathbf{U}^t_{i,:,:} = \mathbf{U}^t_{i+p,:,:}$. First note that by inductive hypothesis, we have that $\mathbf{S}^{t-1}_{i,:,:} = \mathbf{S}^{t-1}_{i+p,:,:}$ and thus by the property proved above we have that $(\mathbf{K}^U * \mathbf{S}^{t-1})_{i,:,:} = (\mathbf{K}^U * \mathbf{S}^{t-1})_{i+p,:,:}$. Thus we have that

$$\mathbf{U}^t_{i,:,:} = f_U((\mathbf{K}^U * \mathbf{S}^{t-1})_{i,:,:} + \boldsymbol{B}^U) = f_U((\mathbf{K}^U * \mathbf{S}^{t-1})_{i+p,:,:} + \boldsymbol{B}^U) = \mathbf{U}^t_{i+p,:,:}$$

With a similar argument we can prove that $\mathbf{R}^t_{i,:,:} = \mathbf{R}^t_{i+p,:,:}$. Moreover, notice that $(\mathbf{R}^t \odot \mathbf{S}^{t-1})_{i,:,:} = (\mathbf{R}^t \odot \mathbf{S}^{t-1})_{i+p,:,:}$, and thus $(\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,:,:} = (\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i+p,:,:}$. With all this we finally we have that

$$\begin{aligned}
\mathbf{S}^t_{i,:,:} &= \mathbf{U}^t_{i,:,:} \odot \mathbf{S}^{t-1}_{i,:,:} \; + \; (\mathbf{1}_{i,:,:} - \mathbf{U}_{i,:,:}) \odot f_F((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i,:,:} + \boldsymbol{B}^F) \\
&= \mathbf{U}^t_{i+p,:,:} \odot \mathbf{S}^{t-1}_{i+p,:,:} \; + \; (\mathbf{1}_{i+p,:,:} - \mathbf{U}_{i+p,:,:}) \odot f_F((\mathbf{K}^F * (\mathbf{R}^t \odot \mathbf{S}^{t-1}))_{i+p,:,:} + \boldsymbol{B}^F) \\
&= \mathbf{S}^t_{i+p,:,:}
\end{aligned}$$

This completes the first part of the proof.

We have shown that if the input of a uniform neural GPU is periodic, then the output is also periodic. We make a final observation. Let N be a uniform Neural GPU, and $\mathbf{S} \in \mathbb{Q}^{kp \times w \times d}$ be a tensor such that $\mathbf{S}_{i,:,:} = \mathbf{S}_{i+p,:,:}$ for every i. Moreover, let $\mathbf{T} \in \mathbb{Q}^{k'p \times w \times d}$ be a tensor such that $\mathbf{T}_{i,:,:} = \mathbf{T}_{i+p,:,:}$ for every i, and assume that $\mathbf{S}_{1:p,:,:} = \mathbf{T}_{1:p,:,:}$. Lets $\mathbf{S}^1, \mathbf{S}^2, \ldots$ and $\mathbf{T}^1, \mathbf{T}^2, \ldots$ be the sequences produced by N. Then with a similar argument as above it is easy to prove that for every t it holds that $\mathbf{S}^t_{1:p,:,:} = \mathbf{T}^t_{1:p,:,:}$.

From this it is easy to prove that uniform Neural GPUs will no be able to recognize the length of periodic inputs. Thus assume that there is a language recognizer A defined by of a uniform neural GPU N such that $L(A)$ contains all strings of even length. Assume that u is an arbitrary string in Σ such that $|u| = p$ with p an odd number, and let $w = uu$ and $w' = uuu$. Notice that $|w| = 2p$ and thus $w \in L(A)$, but $|w'| = 3p$ and thus $w' \notin L(A)$.

Let $f : \Sigma \to \mathbb{Q}^d$ and let $\boldsymbol{X} = f(w) = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{2p})$ and $\boldsymbol{X}' = f(w') = (\boldsymbol{x}'_1, \boldsymbol{x}'_2, \ldots, \boldsymbol{x}'_{3p})$. Consider now the tensor $\mathbf{S} \in \mathbb{Q}^{2p \times w \times d}$ such that $\mathbf{S}_{i,1,:} = \boldsymbol{x}_i$ for $i \in \{1, \ldots, 2p\}$, thus $\mathbf{S}_{i,:,:} = \mathbf{S}_{i+p,:,:}$. Similarly, consider $\mathbf{T} \in \mathbb{Q}^{3p \times w \times d}$ such that such that $\mathbf{T}_{i,1,:} = \boldsymbol{x}'_i$ for $i \in \{1, \ldots, 3p\}$, and thus $\mathbf{T}_{i,:,:} = \mathbf{T}_{i+p,:,:}$. Notice that $\mathbf{S}_{1:p,:,:} = \mathbf{T}_{1:p,:,:}$ then by the property above we have that for every t it holds that $\mathbf{S}^t_{1:p,:,:} = \mathbf{T}^t_{1:p,:,:}$. In particular, we have $\mathbf{S}^t_{p,:,:} = \mathbf{T}^t_{p,:,:}$. We also know that $\mathbf{S}^t_{p,:,:} = \mathbf{S}^t_{2p,:,:}$ and that $\mathbf{T}^t_{p,:,:} = \mathbf{T}^t_{2p,:,:} = \mathbf{T}^t_{3p,:,:}$. Thus we have that for every t it holds that $\mathbf{S}^t_{2p,1,:} = \mathbf{T}^t_{3p,1,:}$. From this we conclude that the outputs of N for both inputs $\boldsymbol{X}$ and $\boldsymbol{X}'$ are the same, and thus if A accepts w then A accepts w' which is a contradiction.