Journal of Machine Learning Research 22 (2021) 1-35 Submitted 3/20; Revised 10/20; Published 3/21

Attention is Turing Complete

Jorge Pérez JPEREZ@DCC.UCHILE.CL
Department of Computer Science
Universidad de Chile
IMFD Chile

Pablo Barceló PBARCELO@UC.CL
Institute for Mathematical and Computational Engineering
School of Engineering, Faculty of Mathematics
Universidad Católica de Chile
IMFD Chile

Javier Marinkovic JMARINKOVIC@DCC.UCHILE.CL
Department of Computer Science
Universidad de Chile
IMFD Chile

Editor: Luc de Raedt

Abstract

Alternatives to recurrent neural networks, in particular, architectures based on *self-attention*, are gaining momentum for processing input sequences. In spite of their relevance, the computational properties of such networks have not yet been fully explored. We study the computational power of the *Transformer*, one of the most paradigmatic architectures exemplifying self-attention. We show that the Transformer with *hard-attention* is Turing complete exclusively based on their capacity to compute and access internal dense representations of the data. Our study also reveals some minimal sets of elements needed to obtain this completeness result.

Keywords: Transformers, Turing completeness, self-Attention, neural networks, arbitrary precision

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 and Mikolov, 2015; Kaiser and 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*. Most of the networks proposed for learning algorithms are Turing complete simply by definition, 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 and 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 and 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 and 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. A prominent example of this approach corresponds to architectures based on *attention* and *self-attention* mechanisms. In this work we look at the problem of Turing completeness à la Siegelmann and Sontag for one of the most paradigmatic models exemplifying attention: the *Transformer* (Vaswani et al., 2017).

The main contribution of our paper is to show that the Transformer is Turing complete à la Siegelmann and Sontag, that is, based on its capacity to compute and access internal dense representations of the data it processes and produces. To prove this we assume that internal activations are represented as rational numbers with arbitrary precision. The proof of Turing completeness of the Transformer is based on a direct simulation of a Turing machine which we believe to be quite intuitive. Our study also reveals some minimal sets of elements needed to obtain these completeness results.

Background work The study of the computational power of neural networks can be traced back to McCulloch and Pitts (1943), which established an analogy between neurons with hard-threshold activations and threshold logic formulae (and, in particular, Boolean propositional formulae), 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 and 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 and Mikolov, 2015), and the (De)Queue-RNNs (Grefenstette et al., 2015). By Siegelmann and 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, Dehghani et al. (2018) have advocated the need for enriching its architecture with several new features as a way to learn general procedures in practice. 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.

For instance, Dehghani et al. (2018) assume fixed precision, while we allow arbitrary internal precision during computation. As we show in this paper, Transformers with fixed precision are not Turing complete. 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 *hard attention* while the Transformer is often trained with *soft attention* (Vaswani et al., 2017).

Recently, Hahn (2019) has studied the Transformer encoder (see Section 3) as a language recognizer and shown several limitations of this architecture. Also, Yun et al. (2020) studied Transformers showing that they are universal approximators of continuous functions over strings. None of these works studied the completeness of the Transformer as a general computational device which is the focus of our work.

Related article A related version of this paper was previously presented at the International Conference on Learning Representations, ICLR 2019, in which we announced results on two modern neural network architectures: Transformers and Neural GPUs; see (Pérez et al., 2019). For the sake of uniformity this submission focuses only on the former.

Organization of the paper The rest of the paper is organized as follows. We begin by introducing notation and terminology in Section 2. In Section 3 we formally introduce the Transformer architecture and prove a strong invariance property (Section 3.1) that motivates the addition of positional encodings (Section 3.2). In Section 4 we prove our main result on the Turing completeness of the Transformer (Theorem 6). As the proof need several technicalities we divide it in three parts: overview of the main construction (Section 4.1), implementation details for every part of the construction (Section 4.2) and proof of intermediate lemmas (Appendix A). We finally discuss on some of the characteristics of the Transformer needed to obtain Turing completeness (Section 5) and finish with the possible future work (Section 6).

2. Preliminaries

We adopt the notation of Goodfellow et al. (2016) and use letters x, y, z, etc. for scalars, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$, etc. for vectors, and $\boldsymbol{A}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, etc. for matrices and sequences of vectors. We assume that all vectors are row vectors, and thus they are multiplied as, e.g., $\boldsymbol{x}\boldsymbol{A}$ for a vector $\boldsymbol{x}$ and matrix $\boldsymbol{A}$. Moreover, $\boldsymbol{A}_{i,:}$ denotes the i-th row of matrix $\boldsymbol{A}$.

We assume all weights and activations to be rational numbers of arbitrary precision. 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

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

Observe that $\sigma(x)$ can actually be constructed from standard ReLU activation functions. In fact, recall that $\text{relu}(x) = \max(0, x)$. Hence,

$$\sigma(x) = \text{relu}(x) - \text{relu}(x - 1).$$

Then there exist matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, and a bias vector $\boldsymbol{b}$, such that for every vector $\boldsymbol{x}$ it holds that $\sigma(\boldsymbol{x}) = \text{relu}(\boldsymbol{x}\boldsymbol{A}+\boldsymbol{b})\boldsymbol{B}$. This observation implies that in all our results, whenever we use $\sigma(\cdot)$ as an activation function, we can alternatively use $\text{relu}(\cdot)$ but at the price of an additional network layer.

Sequence-to-sequence neural networks We are interested in *sequence-to-sequence* (seq-to-seq) neural network architectures that we formalize next. 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 architectures of this type 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 interpret a seq-to-seq network as a *language recognizer* of strings as follows.

Definition 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 polynomial time 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 polynomial-time; given a vector $\boldsymbol{f}$, the membership $\boldsymbol{f} \in \mathbb{F}$ should be decided by a Turing machine working in polynomial 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 stopping condition, while being permissive enough to construct meaningful embeddings and stopping criterions.

Turing machine computations Let us recall that (deterministic) Turing machines are tuples of the form $M = (Q, \Sigma, \delta, q_{\text{init}}, F)$, where:

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{1, -1\}$ is the transition function,
- $q_{\text{init}} \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of final states.

We assume that M is formed by a single tape that consists of infinitely many cells (or positions) to the right, and also that the special symbol $\# \in \Sigma$ is used to mark blank

positions in such a tape. Moreover, M has a single head that can move left and right over the tape, reading and writing symbols from Σ.

We do not provide a formal definition of the notion of computation of M on input string $w \in \Sigma^*$, but it can be found in any standard textbook on the theory of computation; c.f. Sipser (2006). Informally, we assume that the input $w = a_1 \cdots a_n$ is placed symbol-by-symbol in the first n cells of the tape. The infinitely many other cells to the right of w contain the special blank symbol #. The computation of M on w is defined in steps $i = 1, 2, \ldots$. In the first step the machine M is in the initial state q_{init} with its head reading the first cell of the tape. If at any step i, for $i > 0$, the machine is in state $q \in Q$ with its head reading a tape that contains symbol $a \in \Sigma$, the machine proceeds to do the following. Assume that $\delta(q, a) = (q', b, d)$, for $q' \in Q$, $b \in \Sigma$, and $d \in \{1, -1\}$. Then M writes symbol b in the cell that is reading, updates its state to q', and moves its head in the direction d, with $d = 1$ meaning move one cell to the right and $d = -1$ one cell to the left.

If the computation of M on w ever reaches a final state $q \in F$, we say that M *accepts* w. The language of all strings in Σ^* that are accepted by M is denoted $L(M)$. A language L is *recognizable*, or *decidable*, if there exists a TM M with $L = L(M)$.

Turing completeness of seq-to-seq neural network architectures 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 *A class $\mathcal{N}$ of seq-to-seq neural network architectures is Turing Complete if $\mathcal{L}_{\mathcal{N}}$ contains all decidable languages (i.e., all those that are recognizable by Turing machines).*

3. The Transformer architecture

In this section we present a formalization of the Transformer architecture (Vaswani et al., 2017), abstracting away from some 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 formal way.

The Transformer is heavily based on the attention mechanism introduced next. Consider a scoring function score : $\mathbb{Q}^d \times \mathbb{Q}^d \to \mathbb{Q}$ and a normalization function $\rho : \mathbb{Q}^n \to \mathbb{Q}^n$, for $d, n > 0$. 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.

$$(s_1, \ldots, s_n) = \rho(\text{score}(\boldsymbol{q}, \boldsymbol{k}_1), \text{score}(\boldsymbol{q}, \boldsymbol{k}_2), \ldots, \text{score}(\boldsymbol{q}, \boldsymbol{k}_n)) \quad (2)$$

$$\boldsymbol{a} = s_1 \boldsymbol{v}_1 + s_2 \boldsymbol{v}_2 + \cdots + s_n \boldsymbol{v}_n. \quad (3)$$

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 functions, but we do pose some restrictions over the normalization function to ensure that it produces a probability distribution over the positions. We require the normalization function to satisfy that for each $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Q}^n$ there is a function f_ρ from $\mathbb{Q}$ to $\mathbb{Q}^+$ such that it is the case that the i-th component $\rho_i(\boldsymbol{x})$ of $\rho(\boldsymbol{x})$ is equal

to $f_\rho(x_i)/\sum_{j=1}^{n} f_\rho(x_j)$. We note that, for example, one can define the softmax function in this way by simply selecting $f_\rho(x)$ to be the exponential function e^x, but we allow other possibilities as well as we next explain.

When proving possibility results, we will need to pick specific scoring and normalization functions. A usual choice for the scoring function is a non linear function defined by a *feed forward network* with input $(\boldsymbol{q}, \boldsymbol{k}_i)$, sometimes called *additive attention* (Bahdanau et al., 2014). Another possibility is to use the dot product $\langle \boldsymbol{q}, \boldsymbol{k}_i \rangle$, called *multiplicative attention* (Vaswani et al., 2017). We actually use a combination of both: multiplicative attention plus a feed forward network defined as the composition of functions of the form $\sigma(g(\cdot))$, where g is an affine transformation and σ is the piecewise-linear sigmoidal activation defined in equation (1). For the normalization function, softmax is a standard choice. Nevertheless, in our proofs we use the hardmax function, which is obtained by setting $f_{\text{hardmax}}(x_i) = 1$ if x_i is the maximum value in $\boldsymbol{x}$, and $f_{\text{hardmax}}(x_i) = 0$ otherwise. Thus, for a vector $\boldsymbol{x}$ in which the maximum value occurs r times, we have that $\text{hardmax}_i(\boldsymbol{x}) = \frac{1}{r}$ if x_i is the maximum value of $\boldsymbol{x}$, and $\text{hardmax}_i(\boldsymbol{x}) = 0$ otherwise. We call it *hard attention* whenever hardmax is used as normalization function.

Let us observe that the choice of hardmax is crucial for our proofs to work in their current shape, as it allows to simulate the process of "accessing" specific positions in a sequence of vectors. Hard attention has been previously used specially for processing images (Xu et al., 2015; Elsayed et al., 2019) but, as far as we know, it has not been used in the context of self-attention architectures to proces sequences. See Section 5 for further discussion on our choices for functions in positive results. As it is customary, for a function $F : \mathbb{Q}^d \to \mathbb{Q}^d$ and a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$, with $\boldsymbol{x}_i \in \mathbb{Q}^d$, we write $F(\boldsymbol{X})$ to denote 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 $\text{Enc}(\boldsymbol{\theta})$, with $\boldsymbol{\theta}$ being the parameters, that receives as input a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ of vectors in $\mathbb{Q}^d$ and returns a sequence $\text{Enc}(\boldsymbol{X}; \boldsymbol{\theta}) = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ of vectors in $\mathbb{Q}^d$ of the same length as $\boldsymbol{X}$. In general, we consider the parameters in $\boldsymbol{\theta}$ to be parameterized 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

$$\boldsymbol{a}_i = \text{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) + \boldsymbol{x}_i \tag{4}$$

$$\boldsymbol{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i \tag{5}$$

Notice that in equation 4 we apply functions Q and V, separately, over each entry in $\boldsymbol{X}$. In practice $Q(\cdot)$, $K(\cdot)$, $V(\cdot)$ are typically linear transformations specified as matrices of dimension $(d \times d)$, and $O(\cdot)$ is a feed-forward network. The $+\ \boldsymbol{x}_i$ and $+\ \boldsymbol{a}_i$ summands are usually called *residual connections* (He et al., 2016; He et al.). When the particular functions used as parameters are not important, we simply write $\boldsymbol{Z} = \text{Enc}(\boldsymbol{X})$.

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 $\boldsymbol{X}^1 = \boldsymbol{X}$):

$$\boldsymbol{X}^{\ell+1} = \text{Enc}(\boldsymbol{X}^\ell; \boldsymbol{\theta}_\ell), \quad \boldsymbol{K} = K(\boldsymbol{X}^L), \quad \boldsymbol{V} = V(\boldsymbol{X}^L). \tag{6}$$

We write $(\boldsymbol{K}, \boldsymbol{V}) = \text{TEnc}_L(\boldsymbol{X})$ to denote that $(\boldsymbol{K}, \boldsymbol{V})$ is the result of an L-layer Transformer encoder over the input 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 $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k)$ of the same length as $\boldsymbol{Y}$. When defining a decoder layer we denote by $\boldsymbol{Y}_i$ the sequence $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_i)$, for $1 \leq i \leq k$. The output $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k)$ of the layer is also parameterized, this time by four functions $Q(\cdot)$, $K(\cdot)$, $V(\cdot)$ and $O(\cdot)$ from $\mathbb{Q}^d$ to $\mathbb{Q}^d$, and is defined as follows for each $1 \leq i \leq k$:

$$\boldsymbol{p}_i = \text{Att}(Q(\boldsymbol{y}_i), K(\boldsymbol{Y}_i), V(\boldsymbol{Y}_i)) + \boldsymbol{y}_i \tag{7}$$

$$\boldsymbol{a}_i = \text{Att}(\boldsymbol{p}_i, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) + \boldsymbol{p}_i \tag{8}$$

$$\boldsymbol{z}_i = O(\boldsymbol{a}_i) + \boldsymbol{a}_i \tag{9}$$

Notice that the first (self) attention over $(K(\boldsymbol{Y}_i), V(\boldsymbol{Y}_i))$ considers the subsequence of $\boldsymbol{Y}$ only until index i and is used to generate a query $\boldsymbol{p}_i$ to attend the external pair $(\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$. We denote the output of the single-decoder layer over $\boldsymbol{Y}$ and $(\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$ as $\text{Dec}((\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}), \boldsymbol{Y}; \boldsymbol{\theta})$.

The Transformer *decoder* is a repeated application of single-layer decoders, plus a transformation function $F : \mathbb{Q}^d \to \mathbb{Q}^d$ applied to the final vector of the decoded sequence. Thus, the output of the decoder is a single vector $\boldsymbol{z} \in \mathbb{Q}^d$. Formally, the L-layer Transformer decoder is defined as

$$\boldsymbol{Y}^{\ell+1} = \text{Dec}((\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}), \boldsymbol{Y}^{\ell}; \boldsymbol{\theta}_\ell), \quad \boldsymbol{z} = F(\boldsymbol{y}_k^L) \qquad (1 \leq \ell \leq L-1 \text{ and } \boldsymbol{Y}^1 = \boldsymbol{Y}). \tag{10}$$

We use $\boldsymbol{z} = \text{TDec}_L((\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}), \boldsymbol{Y})$ to denote that $\boldsymbol{z}$ is the output of this L-layer Transformer decoder on input $\boldsymbol{Y}$ and $(\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$.

An important restriction of Transformers is that the output of a Transformer decoder always corresponds to the encoding of a letter in some finite alphabet Γ. Formally speaking, it is required that there exists a finite alphabet Γ and an embedding function $g : \Gamma \to \mathbb{Q}^D$, such that the final transformation function F of the Transformer decoder maps any vector in $\mathbb{Q}^d$ to a vector in the finite set $g(\Gamma)$ of embeddings of letters in Γ.

The complete Transformer 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

$$\boldsymbol{y}_{t+1} = \text{TDec}(\text{TEnc}(\boldsymbol{X}), (\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_t)), \qquad \text{for } 0 \leq t \leq r-1. \tag{11}$$

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

3.1 Invariance under proportions

Transformer networks, as defined above, are quite weak in terms of its abilities to capture languages. This is due to the fact that Transformers are *order-invariant*, i.e., they do not have access to the relative order of the elements in the input. More formally, two input sequences that are permutations of each other produce exactly the same output. This is a consequence of the following 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 $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Att}(\boldsymbol{q}, \pi(\boldsymbol{K}), \pi(\boldsymbol{V}))$ for every query $\boldsymbol{q}$.

Based on order-invariance we can show that Transformers, as currently defined, are quite weak in their ability to recognize basic string languages. As a standard yardstick we use the well-studied class of regular languages, i.e., languages recognized by finite automata; see, e.g., Sipser (2006). Order-invariance implies that not every regular language can be recognized by a Transformer network. For 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 themselves 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*, and that we present next. For a string $w \in \Sigma^*$ and a symbol $a \in \Sigma$, we use $\text{prop}(a, w)$ to denote the ratio between the number of times that a appears in w and the length of w. Consider now the set $\text{PropInv}(w) = \{u \in \Sigma^* \mid \text{prop}(a, w) = \text{prop}(a, u) \text{ for every } a \in \Sigma\}$. Then:

Proposition 3 *Let* Trans *be a Transformer,* $\boldsymbol{s}$ *a seed,* $r \in \mathbb{N}$*, and* $f : \Sigma \to \mathbb{Q}^d$ *an embedding function. Then* $\text{Trans}(f(w), \boldsymbol{s}, r) = \text{Trans}(f(u), \boldsymbol{s}, r)$*, for each* $u, w \in \Sigma^*$ *with* $u \in \text{PropInv}(w)$*.*

The proof of this result is quite technical and we have relegated it to the appendix. As an immediate corollary we obtain the following.

Corollary 4 *Consider the order-invariant regular language* $L = \{w \in \{a, b\}^* \mid w$ *has an even number of* a *symbols*$\}$*. Then* L *cannot be recognized by a Transformer network.*

Proof 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 $w_1 \in \text{PropInv}(w_2)$ by Proposition 3 we have that $w_1 \in L(A)$ if and only if $w_2 \in L(A)$. This is a contradiction since $w_1 \in L$ but $w_2 \notin L$. ■

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

Proposition 5 *There is a Transformer network that recognizes the non-regular language* $S = \{w \in \{a, b\}^* \mid w$ *has strictly more symbols* a *than symbols* $b\}$*.*

Proof 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 $w_1 \in \text{PropInv}(w_2)$ by Proposition 3 we have that $w_1 \in L(A)$ if and only if $w_2 \in L(A)$. This is a contradiction since $w_1 \in L$ but $w_2 \notin L$. ■

That is, the computational power of Transformer networks as defined in this section 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).

3.2 Positional Encodings

The weakness in terms of expressive power that Transformers exhibit due to order- and proportion-invariance 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 come to remedy the order-invariance issue by providing information about the absolute positions of the symbols in the input. A positional encoding is just a function $\text{pos} : \mathbb{N} \to \mathbb{Q}^d$. Function pos combined with an embedding function $f : \Sigma \to \mathbb{Q}^d$ give rise to a new embedding function $f_{\text{pos}} : \Sigma \times \mathbb{N} \to \mathbb{Q}^d$ such that $f_{\text{pos}}(a, i) = f(a) + \text{pos}(i)$. Thus, given an input string $w = a_1 a_2 \cdots a_n \in \Sigma^*$, the result of the embedding function $f_{\text{pos}}(w)$ provides a "new" input

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

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

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

As for the case of the embedding functions, we require the positional encoding $\text{pos}(i)$ to be computable by a Turing machine in polynomial time with respect to the size (in bits) of i.

As we show in the next section, positional encodings not only solve the aforementioned weaknesses of Transformer networks in terms of order- and proportion-invariance, but actually ensure a much stronger condition: There is a natural class of positional encoding functions that provide Transformer networks with the ability to encode any language accepted by a Turing machine.

4. Turing completeness of the Transformer with positional encodings

In this section we prove our main result.

Theorem 6 *The class of Transformer networks with positional encodings is Turing complete. Moreover, Turing completeness holds even in the restricted setting in which the only non-constant values in positional embedding* $\text{pos}(n)$ *of* n*, for* $n \in \mathbb{N}$*, are* n*,* $1/n$*, and* $1/n^2$*, and Transformer networks have a single encoder layer and three decoder layers.*

Actually, the proof of this result shows something stronger: Not only Transformers can recognize all languages accepted by Turing machines, i.e., the so-called *recognizable* or *decidable* languages; they can recognize all *recursively enumerable* or *semi-decidable* languages, which are those languages L for which there exists a TM that enumerates all strings in L.

We now provide a complete proof of Theorem 6. For readability, the proofs of some intermediate lemmas are relegated to the appendix.

Let $M = (Q, \Sigma, \delta, q_{\text{init}}, F)$ be a Turing machine with a tape that is infinite to the right and assume that the special symbol $\# \in \Sigma$ is used to mark blank positions in the tape. We make the following assumptions about how M works when processing an input string:

- M begins at state q_{init} pointing to the first cell of the tape reading the blank symbol #. The input is written immediately to the right of this first cell.

- Q has a special state q_{read} used to read the complete input.
- Initially (step 0), M makes a transition to state q_{read} and move its head to the right.
- While in state q_{read} it moves to the right until symbol # is read.
- There are no transitions going out from accepting states (states in F).

It is easy to prove that every general Turing machine is equivalent to one that satisfies the above assumptions. We prove that one can construct a transformer network Trans_M that is able to simulate M on every possible input string; or, more formally, $L(M) = L(\text{Trans}_M)$.

The construction is somehow involved and makes use of several auxiliary definitions and intermediate results. To make the reading easier we divide the construction and proof in three parts. We first give a high-level view of the strategy we use. Then we give some details on the architecture of the encoder and decoder needed to implement our strategy, and finally we formally prove that every part of our architecture can be actually implemented.

4.1 Overview of the construction and high-level strategy

In the encoder part of Trans_M we receive as input the string $w = s_1 s_2 \cdots s_n$. We first use an embedding function to represent every s_i as a one-hot vector and add a positional encoding for every index. The encoder produces output $(\boldsymbol{K}^{\boldsymbol{e}}, \boldsymbol{V}^{\boldsymbol{e}})$, where $\boldsymbol{K}^{\boldsymbol{e}} = (\boldsymbol{k}_1^{\boldsymbol{e}}, \ldots, \boldsymbol{k}_n^{\boldsymbol{e}})$ and $\boldsymbol{V}^{\boldsymbol{e}} = (\boldsymbol{v}_1^{\boldsymbol{e}}, \ldots, \boldsymbol{v}_n^{\boldsymbol{e}})$ are sequences of keys and values such that $\boldsymbol{v}_i^{\boldsymbol{e}}$ contains the information of s_i and $\boldsymbol{k}_i^{\boldsymbol{e}}$ contains the information of the i-th positional encoding. We later show that this allows us to attend to every specific position and copy every input symbol from the encoder to the decoder (see Lemma 7).

In the decoder part of Trans_M we simulate a complete execution of M over $w = s_1 s_2 \cdots s_n$. For this we define the following sequences (for $i \geq 0$):

$q^{(i)}$	:	state of M at step i of the computation
$s^{(i)}$	:	symbol read by the head of M during step i
$v^{(i)}$	:	symbol written by M during step i
$m^{(i)}$	:	direction in which the head is moved in the transition of M during step i

For the case of $m^{(i)}$ we assume that -1 represents a movement to the left and 1 represents a movement to the right. In our construction we show how to build a decoder that computes all the above values, for every step i, using self attention plus attention over the encoder part. Since the above values contain all the needed information to reconstruct the complete history of the computation, we can effectively simulate M.

In particular, our construction produces the sequence of output vectors $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ such that, for every i, the vector $\boldsymbol{y}_i$ contains both $q^{(i)}$ and $s^{(i)}$ encoded as one-hot vectors. The construction and proof goes by induction. We begin with an initial vector $\boldsymbol{y}_0$ that represents the state of the computation before it has started, that is $q^{(0)} = q_{\text{init}}$ and $s^{(0)} = \#$. For the induction step we assume that we have already computed $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_r$ such that $\boldsymbol{y}_i$ contains information about $q^{(i)}$ and $s^{(i)}$, and we show how on input $(\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_r)$ the decoder produces the next vector $\boldsymbol{y}_{r+1}$ containing $q^{(r+1)}$ and $s^{(r+1)}$.

The overview of the construction is as follows. First, by definition the transition function δ of Turing machine M relates the above values with the following equation:

$$\delta(q^{(i)}, s^{(i)}) = (q^{(i+1)}, v^{(i)}, m^{(i)}). \tag{12}$$

We prove that we can use a two-layer feed-forward network to mimic the transition function δ (Lemma 8). Thus, given that the input vector $\boldsymbol{y}_i$ contains $q^{(i)}$ and $s^{(i)}$, we can produce the values $q^{(i+1)}$, $v^{(i)}$ and $m^{(i)}$ (and store them as values in the decoder). In particular, since $\boldsymbol{y}_r$ is in the input, we can produce $q^{(r+1)}$ which is part of what we need for $\boldsymbol{y}_{r+1}$. In order to complete the construction we also need to compute the value $s^{(r+1)}$, that is, we need to compute the symbol read by the head of machine M during the next step (step $r+1$). We next describe at a high level, how this symbol can be computed with two additional decoder layers.

We first make some observations about $s^{(i)}$ that are fundamental for our construction. Assume that during step i of the computation the head of M is pointing at the cell with index k. Then we have three possibilities:

1. If $i \leq n$, then $s^{(i)} = s_i$ since M is still reading its input string.
2. If $i > n$ and M has never written at index k, then $s^{(i)} = \#$, the blank symbol.
3. Otherwise, that is, if $i > n$ and step i is not the first step in which M is pointing to index k, then $s^{(i)}$ is the last symbol written by M at index k.

For the case (1) we can produce $s^{(i)}$ by simply attending to position i in the encoder part. Thus, if $r+1 \leq n$ to produce $s^{(r+1)}$ we can just attend to index $r+1$ in the encoder and copy this value to $\boldsymbol{y}_{r+1}$. For cases (2) and (3) the solution is a bit more complicated, but almost all the important work is done by computing the index of the cell that M will be pointing during step $r+1$.

To formalize this computation, let us denote by $c^{(i)} \in \mathbb{N}$ the following value:

$$c^{(i)} \quad : \quad \text{the index of the cell the head of } M \text{ is pointing to during step } i.$$

Notice that the value $c^{(i)}$, for $i > 0$, satisfies that $c^{(i)} = c^{(i-1)} + m^{(i-1)}$. If we unroll this equation by using $c^{(0)} = 0$, we obtain that

$$c^{(i)} = m^{(0)} + m^{(1)} + \cdots + m^{(i-1)}.$$

Then, since we are assuming that the decoder stores each value of the form $m^{(j)}$, for $0 \leq j \leq i$, at step i the decoder has all the necessary information to compute not only value $c^{(i)}$ but also $c^{(i+1)}$. We actually show that the computation (of a representation) of $c^{(i)}$ and $c^{(i+1)}$ can be done by using a single layer of self attention (Lemma 9).

We still need to define a final notion. With $c^{(i)}$ one can define the auxiliary value $\ell(i)$ as follows. If the set $\{j \mid j < i \text{ and } c^{(j)} = c^{(i)}\}$ is nonempty, then

$$\ell(i) := \max\{\, j \mid j < i \text{ and } c^{(j)} = c^{(i)} \,\}.$$

Otherwise, $\ell(i) = (i-1)$. Thus, if the cell $c^{(i)}$ has been visited by the head of M before step i, then $\ell(i)$ denotes the last step in which this happened. Since, by assumption, at

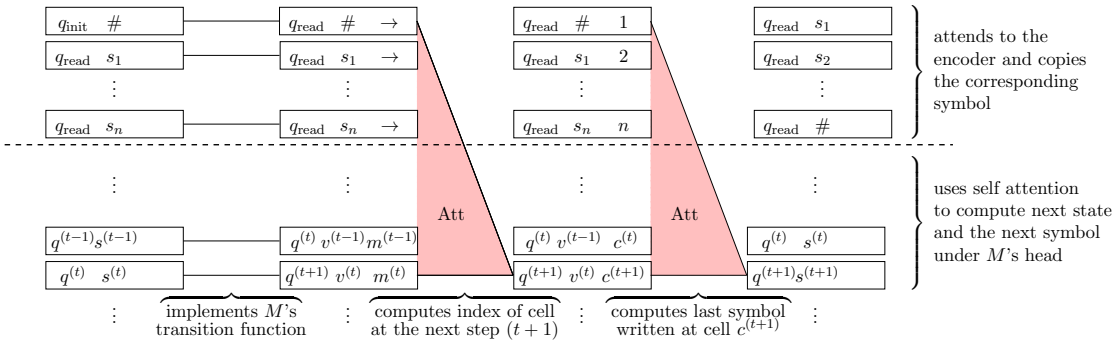

Figure 1: High-level structure of the decoder part of Trans_M.

every step of the computation M moves its head either to the right or to the left (it never stays in the same cell), for every i it holds that $c^{(i)} \neq c^{(i-1)}$, from which we obtain that, in this case, $\ell(i) < i-1$. This implies that $\ell(i) = i-1$ if and only if cell $c^{(i)}$ is visited by the head of M for the first time at time step i. This allows us to check that $c^{(i)}$ is visited for the first time at time step i by just checking whether $\ell(i) = i-1$.

We now have all the necessary ingredients to explain how to compute the value $s^{(r+1)}$, i.e., the symbol read by the head of M during step i of the computation. Assume that $r+1 > n$ (the case $r+1 \leq n$ was already covered before). We first note that if $\ell(r+1) = r$ then $s^{(r+1)} = \#$ since this is the first time that cell $c^{(r+1)}$ is visited. On the other hand, if $\ell(r+1) < r$ then $s^{(r+1)}$ is the value written by M at step $\ell(r+1)$, i.e., $s^{(r+1)} = v^{(\ell(r+1))}$. Thus, in this case we only need to attend to position $\ell(r+1)$ and copy the value $v^{(\ell(r+1))}$ to produce $s^{(r+1)}$. We show that all this can be done with an additional self-attention decoder layer (Lemma 10).

We have described at a high-level a decoder that, with input $(\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_r)$, computes the values $q^{(r+1)}$ and $s^{(r+1)}$ which are needed to produce $\boldsymbol{y}_{r+1}$. A pictorial description of this high-level idea is depicted in Figure 1. In the following we explain the technical details of our construction.

4.2 Details of the architecture of Trans_M

In this section we give more details on the architecture of the encoder and decoder needed to implement our strategy.

Attention mechanism For our attention mechanism we make use of the non-linear function $\varphi(x) = -|x|$ to define a scoring function $\text{score}_\varphi : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ such that

$$\text{score}_\varphi(\boldsymbol{u}, \boldsymbol{v}) = \varphi(\langle \boldsymbol{u}, \boldsymbol{v} \rangle) = -|\langle \boldsymbol{u}, \boldsymbol{v} \rangle|.$$

We note that this function can be computed as a small feed-forward network that has the dot product as input. Recall that $\text{relu}(x) = \max(0, x)$. Function $\varphi(x)$ can be implemented as $\varphi(x) = -\text{relu}(x) - \text{relu}(-x)$. Then, let $\boldsymbol{w}_1$ be the (row) vector $[1, -1]$ and $\boldsymbol{w}_2$ the vector $[-1, -1]$. We have that $\varphi(x) = \text{relu}(x\boldsymbol{w}_1)\boldsymbol{w}_2^T$ and then $\text{score}_\varphi(\boldsymbol{u}, \boldsymbol{v}) = \text{relu}(\boldsymbol{u}\boldsymbol{v}^T\boldsymbol{w}_1)\boldsymbol{w}_2^T$.

Now, let $\boldsymbol{q} \in \mathbb{Q}^d$, and $\boldsymbol{K} = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n)$ and $\boldsymbol{V} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$ be tuples of elements in $\mathbb{Q}^d$. We describe how $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})$ is computed when hard attention is considered

(i.e., when hardmax is used as a normalization function). Assume first that there is a single $j^\star \in \{1, \ldots, n\}$ that maximizes the value $\text{score}_\varphi(\boldsymbol{q}, \boldsymbol{k}_j)$. In such a case we have that $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{v}_{j^\star}$ with

$$\begin{aligned} j^\star &= \arg\max_{1 \leq j \leq n} \ \text{score}_\varphi(\boldsymbol{q}, \boldsymbol{k}_j) \\ &= \arg\max_{1 \leq j \leq n} \ -|\langle \boldsymbol{q}, \boldsymbol{k}_j \rangle| \\ &= \arg\min_{1 \leq j \leq n} \ |\langle \boldsymbol{q}, \boldsymbol{k}_j \rangle| \end{aligned} \tag{13}$$

Thus, when computing hard attention with the function $\text{score}_\varphi(\cdot)$ we essentially select the vector $\boldsymbol{v}_j$ such that the dot product $\langle \boldsymbol{q}, \boldsymbol{k}_j \rangle$ is as close to 0 as possible. If there is more than one index, say indexes $j_1, j_2, \ldots, j_r$, that minimize the dot product $\langle \boldsymbol{q}, \boldsymbol{k}_j \rangle$, we then have

$$\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \frac{1}{r}\big(\boldsymbol{v}_{j_1} + \boldsymbol{v}_{j_2} + \cdots + \boldsymbol{v}_{j_r}\big).$$

Thus, in the extreme case in which all dot products of the form $\langle \boldsymbol{q}, \boldsymbol{k}_j \rangle$ are equal, for $1 \leq j \leq n$, attention behaves simply as the average of all value vectors, that is $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{v}_j$. We use all these properties of the hard attention in our proof.

Vectors and encodings We now describe the vectors that we use in the encoder and decoder parts of Trans_M. All such vectors are of dimension $d = 2|Q|+4|\Sigma|+11$. To simplify the exposition, whenever we use a vector $\boldsymbol{v} \in \mathbb{Q}^d$ we write it arranged in four groups of values as follows

$$\begin{aligned} \boldsymbol{v} = [\ & \boldsymbol{q}_1, \boldsymbol{s}_1, x_1, \\ & \boldsymbol{q}_2, \boldsymbol{s}_2, x_2, x_3, x_4, x_5, \\ & \boldsymbol{s}_3, x_6, \boldsymbol{s}_4, x_7 \\ & x_8, x_9, x_{10}, x_{11} \qquad\quad], \end{aligned}$$

where for each i we have that $\boldsymbol{q}_i \in \mathbb{Q}^{|Q|}$, $\boldsymbol{s}_i \in \mathbb{Q}^{|\Sigma|}$, and $x_i \in \mathbb{Q}$. Whenever in a vector of the above form any of the four groups of values is composed only of 0's, we simply write '$0, \ldots, 0$' assuming that the length of this sequence is implicitly determined by the length of the corresponding group. Finally, we denote by $\mathbf{0}_q$ and $\mathbf{0}_s$ the vectors in $\mathbb{Q}^{|Q|}$ and $\mathbb{Q}^{|\Sigma|}$, respectively, that consist exclusively of 0s.

For a symbol $s \in \Sigma$, we use $[\![\ s\ ]\!]$ to denote a one-hot vector in $\mathbb{Q}^{|\Sigma|}$ that represents s. That is, given an injective function $\pi : \Sigma \to \{1, \ldots, |\Sigma|\}$, the vector $[\![\ s\ ]\!]$ has a 1 in position $\pi(s)$ and a 0 in all other positions. Similarly, for $q \in Q$, we use $[\![\ q\ ]\!]$ to denote a one-hot vector in $\mathbb{Q}^{|Q|}$ that represents q.

Embeddings and positional encodings We can now introduce the embedding and positional encodings used in our construction. We use an embedding function $f : \Sigma \to \mathbb{Q}^d$ defined as

$$\begin{aligned} f(s) = [\ & 0, \ldots, 0, \\ & 0, \ldots, 0, \\ & [\![\ s\ ]\!], 0, \mathbf{0}_s, 0, \\ & 0, \ldots, 0 \qquad\quad] \end{aligned}$$

Our construction uses the positional encoding $\text{pos} : \mathbb{N} \to \mathbb{Q}^d$ such that

$$\text{pos}(i) \;=\; [\;\; 0,\ldots,0,\; 0,\ldots,0,\; 0,\ldots,0,\; 1,i,1/i,1/i^2 \;\;]$$

Thus, given an input sequence $s_1 s_2 \cdots s_n \in \Sigma^*$, we have for each $1 \leq i \leq n$ that

$$f_{\text{pos}}(s_i) \;=\; f(s_i) + \text{pos}(i) \;=\; [\;\; 0,\ldots,0,\; 0,\ldots,0,\; [\![\; s_i \;]\!], 0, \mathbf{0}_s, 0,\; 1,i,1/i,1/i^2 \;\;]$$

We denote this last vector by $\boldsymbol{x}_i$. That is, if M receives the input string $w = s_1 s_2 \cdots s_n$, then the input for Trans_M is the sequence $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$. The need for using a positional encoding having values $1/i$ and $1/i^2$ will be clear when we formally prove the correctness of our construction.

We need a final preliminary notion. In the formal construction of Trans_M we also use the following auxiliary sequences:

$$\alpha^{(i)} \;=\; \begin{cases} s_i & 1 \leq i \leq n \\ s_n & i > n \end{cases}$$

$$\beta^{(i)} \;=\; \begin{cases} i & i \leq n \\ n & i > n \end{cases}$$

These are used to identify when M is still reading the input string.

Construction of TEnc_M The encoder part of Trans_M is very simple. For TEnc_M we use a single-layer encoder, such that $\text{TEnc}_M(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = (\boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}})$, where $\boldsymbol{K}^{\mathbf{e}} = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n)$ and $\boldsymbol{V}^{\mathbf{e}} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$ satisfy the following for each $1 \leq i \leq n$:

$$\boldsymbol{k}_i \;=\; [\;\; 0,\ldots,0,\; 0,\ldots,0,\; 0,\ldots,0,\; i,-1,0,0 \;\;]$$

$$\boldsymbol{v}_i \;=\; [\;\; 0,\ldots,0,\; 0,\ldots,0,\; [\![\; s_i \;]\!], i, \mathbf{0}_s, 0,\; 0,\ldots,0 \;\;]$$

It is straightforward to see that these vectors can be produced with a single encoder layer by using a trivial self attention, taking advantage of the residual connections in equations (4) and (5), and then using linear transformations for $V(\cdot)$ and $K(\cdot)$ in equation (6).

When constructing the decoder we use the following property.

Lemma 7 *Let $\boldsymbol{q} \in \mathbb{Q}^d$ be a vector such that $\boldsymbol{q} = [\_, \ldots, \_, 1, j, \_, \_]$, where $j \in \mathbb{N}$ and '$\_$' denotes an arbitrary value. Then*

$$\begin{array}{rcll} \mathrm{Att}(\boldsymbol{q}, \boldsymbol{K^e}, \boldsymbol{V^e}) & = & [& 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & [\![\ \alpha^{(j)}\ ]\!], \beta^{(j)}, \mathbf{0}_s, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\quad] \end{array}$$

Construction of TDec_M We next show how to construct the decoder part of Trans_M to produce the sequence of outputs $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$, where $\boldsymbol{y}_i$ is given by:

$$\begin{array}{rcll} \boldsymbol{y}_i & = & [& [\![\ q^{(i)}\ ]\!], [\![\ s^{(i)}\ ]\!], m^{(i-1)}, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\quad] \end{array}$$

That is, $\boldsymbol{y}_i$ contains information about the state of M at step i, the symbol under the head of M at step i, and the last direction followed by M (the direction of the movement of the head at step $i-1$). The need to include $m^{(i-1)}$ will become clear in the construction. Notice that this respects the restriction on the vectors produced by Transformer decoders: there is only a finite number of vectors $\boldsymbol{y}_i$ the decoder can produce, and thus such vectors can be understood as embeddings of some finite alphabet Γ.

We consider as the initial (seed) vector for the decoder the vector

$$\begin{array}{rcll} \boldsymbol{y}_0 & = & [& [\![\ q_{\mathrm{init}}\ ]\!], [\![\ \#\ ]\!], 0, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\quad] \end{array}$$

We are assuming that $m^{(0)} = 0$ to represent that previous to step 1 there was no head movement. Our construction resembles a proof by induction; we describe the architecture piece by piece and at the same time we show how for every $r \geq 0$ our architecture constructs $\boldsymbol{y}_{r+1}$ from the previous vectors $(\boldsymbol{y}_0, \ldots, \boldsymbol{y}_r)$.

Thus, assume that $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_r$ satisfy the properties stated above. Since we are using positional encodings, the actual input for the first layer of the decoder is the sequence

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

We denote by $\overline{\boldsymbol{y}}_i$ the vector $\boldsymbol{y}_i$ plus its positional encoding. Thus,

$$\begin{array}{rcll} \overline{\boldsymbol{y}}_i & = & [& [\![\ q^{(i)}\ ]\!], [\![\ s^{(i)}\ ]\!], m^{(i-1)}, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & 1, (i+1), 1/(i+1), 1/(i+1)^2 \quad] \end{array}$$

For the first self attention in equation (7) we just produce the identity which can be easily implemented as in the proof of Proposition 5. Thus, we produce the sequence of vectors $(\boldsymbol{p}_0^1, \boldsymbol{p}_1^1, \ldots, \boldsymbol{p}_r^1)$ such that $\boldsymbol{p}_i^1 = \overline{\boldsymbol{y}}_i$.

Since $\boldsymbol{p}_i^1$ is of the form $[\_, \ldots, \_, 1, i+1, \_, \_]$, by Lemma 7 we know that if we use $\boldsymbol{p}_i^1$ to attend over the encoder we obtain

$$\begin{array}{rcll} \mathrm{Att}(\boldsymbol{p}_i^1, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) & = & [& 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\qquad\qquad] \end{array}$$

Thus, in equation (8) we finally produce the vector $\boldsymbol{a}_i^1$ given by

$$\begin{array}{rcll} \boldsymbol{a}_i^1 = \mathrm{Att}(\boldsymbol{p}_i^1, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) + \boldsymbol{p}_i^1 & = & [& [\![\ q^{(i)}\ ]\!], [\![\ s^{(i)}\ ]\!], m^{(i-1)}, \\ & & & 0, \ldots, 0, \\ & & & [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ & & & 1, (i+1), 1/(i+1), 1/(i+1)^2 \quad] \end{array} \tag{14}$$

As the final piece of the first decoder layer we use a function $O_1(\cdot)$ in equation (9) that satisfies the properties stated in the following lemma.

Lemma 8 *There exists a two-layer feed-forward network $O_1 : \mathbb{Q}^d \to \mathbb{Q}^d$ such that, on input vector $\boldsymbol{a}_i^1$ as defined in equation (14), it produces as output*

$$\begin{array}{rcll} O_1(\boldsymbol{a}_i^1) & = & [& -[\![\ q^{(i)}\ ]\!], -[\![\ s^{(i)}\ ]\!], -m^{(i-1)}, \\ & & & [\![\ q^{(i+1)}\ ]\!], [\![\ v^{(i)}\ ]\!], m^{(i)}, m^{(i-1)}, 0, 0 \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\qquad\qquad] \end{array}$$

That is, function $O_1(\cdot)$ simulates transition $\delta(q^{(i)}, s^{(i)})$ to construct $[\![\ q^{(i+1)}\ ]\!]$, $[\![\ v^{(i)}\ ]\!]$, and $m^{(i)}$ besides some other linear transformations.

We finally produce as the output of the first decoder layer, the sequence $(\boldsymbol{z}_0^1, \boldsymbol{z}_1^1, \ldots, \boldsymbol{z}_r^1)$ such that

$$\begin{array}{rcll} \boldsymbol{z}_i^1 = O_1(\boldsymbol{a}_i^1) + \boldsymbol{a}_i^1 & = & [& 0, \ldots, 0, \\ & & & [\![\ q^{(i+1)}\ ]\!], [\![\ v^{(i)}\ ]\!], m^{(i)}, m^{(i-1)}, 0, 0, \\ & & & [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ & & & 1, (i+1), 1/(i+1), 1/(i+1)^2 \qquad] \end{array} \tag{15}$$

Notice that $\boldsymbol{z}_r^1$ already holds info about $q^{(r+1)}$ and $m^{(r)}$ which we need for constructing vector $\boldsymbol{y}_{r+1}$. The single piece of information that we still need to construct is $s^{(r+1)}$, that is, the symbol under the head of machine M at the next time step (time $r+1$). We next describe how this symbol can be computed with two additional decoder layers.

Recall that $c^{(i)}$ is the cell to which M is pointing to at time i, and that it satisfies that $c^{(i)} = m^{(0)} + m^{(1)} + \cdots + m^{(i-1)}$. We can take advantage of this property to prove the following lemma.

Lemma 9 *Let $\boldsymbol{Z}_i^1 = (\boldsymbol{z}_0^1, \boldsymbol{z}_1^1, \ldots, \boldsymbol{z}_i^1)$. There exist functions $Q_2(\cdot)$, $K_2(\cdot)$, and $V_2(\cdot)$ defined by feed-forward networks such that*

$$\begin{array}{rcll} \mathrm{Att}(Q_2(\boldsymbol{z}_i^1), K_2(\boldsymbol{Z}_i^1), V_2(\boldsymbol{Z}_i^1)) & = & [& 0, \ldots, 0, \\ & & & \mathbf{0}_q, \mathbf{0}_s, 0, 0, \frac{c^{(i+1)}}{(i+1)}, \frac{c^{(i)}}{(i+1)}, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\qquad] \end{array} \tag{16}$$

Lemma 9 essentially shows that one can construct a representation for values $c^{(i)}$ and $c^{(i+1)}$ for every possible index i. In particular, if $i = r$ we will be able to compute the value $c^{(r+1)}$ that represents the cell to which the head of M is pointing to during the next time step.

Continuing with the second decoder layer, after applying the self attention defined by Q_2, K_2, and V_2, and adding the residual connection in equation (7), we obtain the sequence of vectors $(\boldsymbol{p}_0^2, \boldsymbol{p}_1^2, \ldots, \boldsymbol{p}_r^2)$ such that for each $0 \leq i \leq r$:

$$\begin{aligned} \boldsymbol{p}_i^2 &= \mathrm{Att}(Q_2(\boldsymbol{z}_i^1), K_2(\boldsymbol{Z}_i^1), V_2(\boldsymbol{Z}_i^1)) + \boldsymbol{z}_i^1 \\ &= [\ 0, \ldots, 0, \\ &\quad [\![\ q^{(i+1)}\ ]\!], [\![\ v^{(i)}\ ]\!], m^{(i)}, m^{(i-1)}, \tfrac{c^{(i+1)}}{(i+1)}, \tfrac{c^{(i)}}{(i+1)}, \\ &\quad [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ &\quad 1, (i+1), 1/(i+1), 1/(i+1)^2 \qquad\qquad] \end{aligned}$$

From vectors $(\boldsymbol{p}_0^2, \boldsymbol{p}_1^2, \ldots, \boldsymbol{p}_r^2)$, and by using the residual connection in equation (8) plus a null output function $O(\cdot)$ in equation (9), one can produce the sequence of vectors $(\boldsymbol{z}_0^2, \boldsymbol{z}_1^2, \ldots, \boldsymbol{z}_r^2)$ such that $\boldsymbol{z}_i^2 = \boldsymbol{p}_i^2$, for each $0 \leq i \leq r$, as the output of the second decoder layer. That is,

$$\begin{aligned} \boldsymbol{z}_i^2 = \boldsymbol{p}_i^2 &= [\ 0, \ldots, 0, \\ &\quad [\![\ q^{(i+1)}\ ]\!], [\![\ v^{(i)}\ ]\!], m^{(i)}, m^{(i-1)}, \tfrac{c^{(i+1)}}{(i+1)}, \tfrac{c^{(i)}}{(i+1)}, \\ &\quad [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ &\quad 1, (i+1), 1/(i+1), 1/(i+1)^2 \qquad\qquad] \end{aligned}$$

We now describe how we can use a third, and final, decoder layer to produce our desired $s^{(r+1)}$ value, i.e., the symbol read by the head of M over the next time step. Recall that $\ell(i)$ is the last time (previous to i) in which M was pointing to position $c^{(i)}$, or it is $i-1$ if this is the first time that M is pointing to $c^{(i)}$. We can prove the following lemma.

Lemma 10 *There exist functions $Q_3(\cdot)$, $K_3(\cdot)$, and $V_3(\cdot)$ defined by feed-forward networks such that*

$$\begin{aligned} \mathrm{Att}(Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{Z}_i^2), V_3(\boldsymbol{Z}_i^2)) &= [\ 0, \ldots, 0, \\ &\quad 0, \ldots, 0, \\ &\quad \mathbf{0}_s, 0, [\![\ v^{(\ell(i+1))}\ ]\!], \ell(i+1), \\ &\quad 0, \ldots, 0 \qquad\qquad] \end{aligned}$$

We prove Lemma 10 by showing that, for every i, one can attend exactly to position $\ell(i+1)$ and then copy both values: $\ell(i+1)$ and $[\![\ v^{(\ell(i+1))}\ ]\!]$. We do this by taking advantage of the previously computed values $c^{(i)}$ and $c^{(i+1)}$. Then we have that $\boldsymbol{p}_i^3$ is given by

$$\begin{aligned} \boldsymbol{p}_i^3 &= \mathrm{Att}(Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{Z}_i^2), V_3(\boldsymbol{Z}_i^2)) + \boldsymbol{z}_i^2 \\ &= [\ 0, \ldots, 0 \\ &\quad [\![\ q^{(i+1)}\ ]\!], [\![\ v^{(i)}\ ]\!], m^{(i)}, m^{(i-1)}, \tfrac{c^{(i+1)}}{(i+1)}, \tfrac{c^{(i)}}{(i+1)}, \\ &\quad [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, [\![\ v^{(\ell(i+1))}\ ]\!], \ell(i+1), \\ &\quad 1, (i+1), 1/(i+1), 1/(i+1)^2 \qquad\qquad] \end{aligned} \tag{17}$$

From vectors $(\boldsymbol{p}_0^3, \boldsymbol{p}_1^3, \ldots, \boldsymbol{p}_r^3)$, and by using the residual connection in Equation (8) plus a null output function $O(\cdot)$ in equation (9), we can produce the sequence of vectors $(\boldsymbol{z}_0^3, \boldsymbol{z}_1^3, \ldots, \boldsymbol{z}_r^3)$ such that $\boldsymbol{z}_i^3 = \boldsymbol{p}_i^3$, for each $1 \leq i \leq r$, as the output of the third and final decoder layer. We then have

$$\begin{array}{rcll} \boldsymbol{z}_i^3 \;=\; \boldsymbol{p}_i^3 & = & [& 0, \ldots, 0, \\ & & & [\![\, q^{(i+1)} \,]\!], [\![\, v^{(i)} \,]\!], m^{(i)}, m^{(i-1)}, \frac{c^{(i+1)}}{(i+1)}, \frac{c^{(i)}}{(i+1)}, \\ & & & [\![\, \alpha^{(i+1)} \,]\!], \beta^{(i+1)}, [\![\, v^{(\ell(i+1))} \,]\!], \ell(i+1), \\ & & & 1, (i+1), 1/(i+1), 1/(i+1)^2 \qquad\qquad\qquad\qquad] \end{array}$$

We finish our construction by applying a final transformation function $F(\cdot)$ in equation (10) with the properties specified in the following lemma.

Lemma 11 *There exists a function $F : \mathbb{Q}^d \to \mathbb{Q}^d$ defined by a feed-forward network such that*

$$\begin{array}{rcll} F(\boldsymbol{z}_r^3) & = & [& [\![\, q^{(r+1)} \,]\!], [\![\, s^{(r+1)} \,]\!], m^{(r)}, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\qquad\qquad] \\ & & & \\ & = & & \boldsymbol{y}_{r+1} \end{array}$$

Final step Recall that $M = (Q, \Sigma, \delta, q_{\text{init}}, F)$. We can now use our Trans_M network to construct the seq-to-seq language recognizer

$$A \;=\; (\Sigma, f_{\text{pos}}, \text{Trans}_M, \boldsymbol{y}_0, \mathbb{F}),$$

where $\mathbb{F}$ is the set of all vectors in $\mathbb{Q}^d$ that correspond to one-hot encodings of final states in F. Formally, $\mathbb{F} = \{[\![\, q \,]\!] \mid q \in F\} \times \mathbb{Q}^{d-|Q|}$. It is straightforward to see that membership in $\mathbb{F}$ can be checked in linear time.

It is easy to observe that $L(A) = L(M)$, i.e., for every $w \in \Sigma^*$ it holds that A accepts w if and only if M accepts w. In fact, if M accepts w then the computation of M on w reaches an accepting state $q_f \in F$ at some time step, say $t^\star$. It is clear from the construction above that, on input $f_{\text{pos}}(w)$, our network Trans_M produces a vector $\boldsymbol{y}_{t^\star}$ that contains q_f as a one-hot vector; this implies that A accepts w. In turn, if M does not accept w then the computation of M on w never reaches an accepting state. Therefore, Trans_M on input $f_{\text{pos}}(w)$ never produces a vector $\boldsymbol{y}_r$ that contains a final state in F as a one-hot vector. This finishes the proof of the theorem.

5. Requirements for Turing completeness

In this section we describe some of the main features used in the proof of Turing completeness for Transformer networks presented in the previous section, and discuss to what extent they are required for such a result to hold.

Choices of functions and parameters Although the general architecture that we presented closely follows the original presentation of the Transformer by Vaswani et al. (2017), some choices for functions and parameters in our Turing completeness proof are different to the standard choices in practice. For instance, for the output function $O(\cdot)$ in equation (9) our proof uses a feed-forward network with various layers, while Vaswani et al. (2017) use only two layers. A more important difference is that we use hard attention which allow us to attend directly to specific positions. Different forms of attention, including hard attention, has been used in previous work (Bahdanau et al., 2014; Xu et al., 2015; Luong et al., 2015; Elsayed et al., 2019; Gülçehre et al.; Ma et al., 2019; Katharopoulos et al., 2020). The decision of using hard attention in our case is fundamental for our current proof to work. In contrast, the original formulation by Vaswani et al. (2017), as well as most of the subsequent work on Transformers and its applications (Dehghani et al., 2018; Devlin et al., 2019; Shiv and Quirk, 2019; Yun et al., 2020) uses soft attention by considering softmax as the normalization function in Equation (2). Although softmax is not a rational function, and thus, is forbidden in our formalization, one can still try to analyze if our results can be extended by for example allowing a bounded-precision version of it. Weiss et al. (2018) have presented an analysis of the computational power of different types of RNNs with bounded precision. They show that when precision is restricted, then different types of RNNs have different computational capabilities. One could try to develop a similar approach for Transformers, and study to what extent using softmax instead of hardmax has an impact in its computational power. This is a challenging problem that deserves further study.

The need of arbitrary precision Our Turing-complete proof relies on having arbitrary precision for internal representations, in particular, for storing and manipulating positional encodings. Although having arbitrary precision is a standard assumption when studying the expressive power of neural networks (Cybenko (1989); Siegelmann and Sontag (1995)), practical implementations rely on fixed precision hardware. If fixed precision is used, then positional encodings can be seen as functions of the form $\text{pos} : \mathbb{N} \to A$ where A is a finite subset of $\mathbb{Q}^d$. Thus, the embedding function f_{pos} can be seen as a regular embedding function $f' : \Sigma' \to \mathbb{Q}^d$ where $\Sigma' = \Sigma \times A$. Thus, whenever fixed precision is used, the net effect of having positional encodings is to just increase the size of the input alphabet. Then from Proposition 3 we obtain that the Transformer with positional encodings and fixed precision is not Turing complete.

An interesting way to bound the precision used by Transformer networks is to make it depend on the size of the input only. An immediate corollary of our Turing completeness proof in Theorem 6 is that by uniformly bounding the precision used in positional encodings we can capture every deterministic complexity class defined by Turing machines. Let us formalize this idea. We assume now that Turing machines have both accepting and rejecting states, and none of these states have outgoing transitions. A language $L \subseteq \Sigma^*$ is accepted by a Turing machine M *in time* $T(n)$, for $T : \mathbb{N} \to \mathbb{N}$, if for every $w \in \Sigma^*$ we have that the computation of M on w reaches an accepting or rejecting state in at most $T(|w|)$ steps, and $w \in L(M)$ iff it is an accepting state that is reached by the computation. We write $\textsc{Time}(T(n))$ for the class of all languages L that are accepted by Turing machines in time $f(n)$. This definition covers some important complexity class; e.g., the famous class P of

languages that can be accepted in polynomial time is defined as

$$\text{P} \ := \ \textsc{Time}(n^{O(1)}) \ = \ \bigcup_{k \geq 0} \text{TIME}(n^k),$$

while the class of Exptime of languages that can be accepted in single exponential time is defined as

$$\textsc{Exptime} \ := \ \textsc{Time}(2^{n^{O(1)}}) \ = \ \bigcup_{k \geq 0} \text{TIME}(2^{n^k}).$$

Since a Turing machine M that accepts a language in time $T(n)$ does not need to move its head beyond cell $T(|w|)$, assuming that w is the input given to M, the positional encodings used in the proof of Theorem 6 need at most $\log(T(|w|))$ bits to represent $\text{pos}(n)$, for n the index of a cell visited by the computation of M on w. Moreover, in order to detect whether M accepts or not the input w the decoder Trans_M does not need to produce more than $T(|w|)$ vectors in the output. Let us then define a Transformer network Trans to be $T(n)$-*bounded* if the following conditions hold:

- The precision allowed for the internal representations of Trans on input w is of at most $\log(T(|w|)$ bits.
- If A is a seq-to-seq language recognizer based on Trans such that A accepts input w, then A accepts w without producing more than $T(|w|)$ output vectors, i.e., there is $r \leq T(|w|)$ such that $\text{Trans}(f(w), \boldsymbol{s}, r) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_r)$ and $\boldsymbol{y}_r \in \mathbb{F}$.

The following corollary, which is obtained by simple inspection of the proof of Theorem 6, establishes that to recognize languages in $\textsc{Time}(T(n))$ it suffices to use $T(n)$-bounded Transformer networks.

Corollary 12 *Let $T : \mathbb{N} \to \mathbb{N}$. For every language $L \in \textsc{Time}(T(n))$ there is a seq-to-seq language recognizer A, based on a $T(n)$-bounded Transformer network Trans_M, such that $L = L(A)$.*

Residual connections First, it would be interesting to refine our analysis by trying to draw a complete picture of the minimal sets of features that make the Transformer architecture Turing complete. As an example, our current proof of Turing completeness requires the presence of *residual connections*, i.e., the $+\boldsymbol{x}_i$, $+\boldsymbol{a}_i$, $+\boldsymbol{y}_i$, and $+\boldsymbol{p}_i$ summands in the definition of the single-layer encoder. We would like to sort out whether such connections are in fact essential to obtain Turing completeness.

6. Future work

We have already mentioned some interesting open problems in the previous section. It would be interesting, in addition, to extract practical implications from our theoretical results. For example, the undecidability of several practical problems related to probabilistic language modeling with RNNs has recently been established Chen et al. (2018). 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 and Sontag (1995). We plan to study to what extent our analogous undecidability results for Transformers imply undecidability for language modeling problems based on them.

Another very important issue is if our choices of functions and parameters may have a practical impact, in particular when learning algorithms from examples. There is some evidence that using a piece-wise linear activation function, similar to the one used in this paper, substantially improves the generalization in algorithm learning for Neural GPUs (Freivalds and Liepins, 2018). This architecture is of interest, as it is the first one able to learn decimal multiplication from examples. In a more recent result, Yan et al. (2020) show that Transformers with a special form of *masked attention* are better suited to learn numerical sub-routines compared with the usual soft attention. Masked attention is similar to hard attention as it forces the model to ignore a big part of the sequence in every attention step, thus allowing only a few positions to be selected. A thorough analysis on how our theoretical results and assumptions may have a practical impact is part of our future work.

Acknowledgments

Barceló and Pérez are funded by the Millennium Institute for Foundational Research on Data (IMFD Chile) and Fondecyt grant 1200967.

Appendix A. Missing proofs from Section 3

Proof [**of Proposition 3**] We extend the definition of the function PropInv to sequences of vectors. Given a sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ of n vectors, we use $\text{vals}(\boldsymbol{X})$ to denote the set of all vectors occurring in $\boldsymbol{X}$. Similarly as for strings, for a vector $\boldsymbol{v}$ we write $\text{prop}(\boldsymbol{v}, \boldsymbol{X})$ for the number of times that $\boldsymbol{v}$ occurs in $\boldsymbol{X}$ divided by n. To extend the notion of PropInv to any sequence $\boldsymbol{X}$ of vectors we use the following definition:

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

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

$$\text{Trans}(\boldsymbol{X}, \boldsymbol{s}, r) = \text{Trans}(\boldsymbol{X}', \boldsymbol{s}, r), \text{ for every } \boldsymbol{X}' \in \text{PropInv}(\boldsymbol{X}). \tag{18}$$

To further simplify the exposition of the proof we introduce some extra notation. We denote by $p_{\boldsymbol{v}}^{\boldsymbol{X}}$ the number of times that vector $\boldsymbol{v}$ occurs in $\boldsymbol{X}$. Thus we have that $\boldsymbol{X}' \in \text{PropInv}(\boldsymbol{X})$ if and only if, there exists a value $\gamma \in \mathbb{Q}^+$ such that for every $\boldsymbol{v} \in \text{vals}(\boldsymbol{X})$ it holds that $p_{\boldsymbol{v}}^{\boldsymbol{X}'} = \gamma p_{\boldsymbol{v}}^{\boldsymbol{X}}$. We now have all the necessary ingredients to proceed with the proof of Proposition 3; more in particular, with the proof of equation 18. Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be an arbitrary sequence of vectors, and let $\boldsymbol{X}' = (\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_m) \in \text{PropInv}(\boldsymbol{X})$. Moreover, let $\boldsymbol{Z} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \text{Enc}(\boldsymbol{X}; \boldsymbol{\theta})$ and $\boldsymbol{Z}' = (\boldsymbol{z}'_1, \ldots, \boldsymbol{z}'_m) = \text{Enc}(\boldsymbol{X}'; \boldsymbol{\theta})$. We first prove the following property:

$$\text{For every pair of indices } (i, j) \in \{1, \ldots, n\} \times \{1, \ldots, m\}, \text{ if } \boldsymbol{x}_i = \boldsymbol{x}'_j \text{ then } \boldsymbol{z}_i = \boldsymbol{z}'_j. \tag{19}$$

Let (i, j) be a pair of indices such that $\boldsymbol{x}_i = \boldsymbol{x}'_j$. From Equations (4-5) we have that $\boldsymbol{z}_i = O(\boldsymbol{a}_i)+\boldsymbol{a}_i$, where $\boldsymbol{a}_i = \text{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X}))+\boldsymbol{x}_i$. Thus, since $\boldsymbol{x}_i = \boldsymbol{x}'_j$, in order to prove $\boldsymbol{z}_i = \boldsymbol{z}'_j$ it is enough to prove that $\text{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) = \text{Att}(Q(\boldsymbol{x}'_j), K(\boldsymbol{X}'), V(\boldsymbol{X}'))$. By equations (2-3) and the restriction over the form of normalization functions,

$$\text{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) \;=\; \frac{1}{\alpha}\sum_{\ell=1}^{n} s_\ell V(\boldsymbol{x}_\ell),$$

where $s_\ell = f_\rho(\text{score}(Q(\boldsymbol{x}_i), K(\boldsymbol{x}_\ell)))$, for some normalization function f_ρ, and $\alpha = \sum_{\ell=1}^{n} s_\ell$. The above equation can be rewritten as

$$\text{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}), V(\boldsymbol{X})) \;=\; \frac{1}{\alpha'}\sum_{\boldsymbol{v}\in\text{vals}(\boldsymbol{X})} p^{\boldsymbol{X}}_{\boldsymbol{v}} f_\rho(\text{score}(Q(\boldsymbol{x}_i), K(\boldsymbol{v}))) \cdot V(\boldsymbol{v})$$

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

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

with $\beta = \sum_{\boldsymbol{v}\in\text{vals}(\boldsymbol{X}')} p^{\boldsymbol{X}'}_{\boldsymbol{v}} f_\rho(\text{score}(Q(\boldsymbol{v}), K(\boldsymbol{v})))$.

Now, since $\boldsymbol{X}' \in \text{PropInv}(\boldsymbol{X})$ we know that $\text{vals}(\boldsymbol{X}) = \text{vals}(\boldsymbol{X}')$ and there exists a $\gamma \in \mathbb{Q}^+$ such that $p^{\boldsymbol{X}'}_{\boldsymbol{v}} = \gamma p^{\boldsymbol{X}}_{\boldsymbol{v}}$ for every $\boldsymbol{v} \in \text{vals}(\boldsymbol{X})$. From this property, plus the fact that $\boldsymbol{x}_i = \boldsymbol{x}'_j$ we have

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

This completes the proof of property (19) above.

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

$$\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) \;=\; \text{Att}(\boldsymbol{q}, \boldsymbol{K}', \boldsymbol{V}'). \tag{20}$$

By following a similar reasoning as for proving Property (19) plus induction on the layers of TEnc, one can first show 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 mappings $M_K : \text{vals}(\boldsymbol{X}) \to \text{vals}(\boldsymbol{K})$ and $M_V : \text{vals}(\boldsymbol{X}) \to \text{vals}(\boldsymbol{V})$ such that, for every $\boldsymbol{X}'' = (x''_1, \ldots, x''_p)$ with $\text{vals}(\boldsymbol{X}'') = \text{vals}(\boldsymbol{X})$, it holds that $\text{TEnc}(\boldsymbol{X}'') = (\boldsymbol{K}'', \boldsymbol{V}'')$ for $\boldsymbol{K}'' = (M_K(x''_1), \ldots, M_K(x''_p))$ and $\boldsymbol{V}'' = (M_V(x''_1), \ldots, M_V(x''_p))$. Let us then focus on $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})$. It holds that

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

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

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

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

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

Finally, using $\boldsymbol{X}' \in \text{PropInv}(\boldsymbol{X})$, we obtain

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

which is what we wanted.

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

$$\boldsymbol{y}_{k+1} = \text{TDec}(\text{TEnc}(\boldsymbol{X}), (\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_k)), \quad \text{for } 0 \leq k < r.$$

To prove that $\text{Trans}(\boldsymbol{X}, \boldsymbol{y}_0, r) = \text{Trans}(\boldsymbol{X}', \boldsymbol{y}_0, r)$ we use an inductive argument on k and show that if $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_r) = \text{Trans}(\boldsymbol{X}, \boldsymbol{y}_0, r)$ and $(\boldsymbol{y}'_1, \ldots, \boldsymbol{y}'_r) = \text{Trans}(\boldsymbol{X}', \boldsymbol{y}_0, r)$, then $\boldsymbol{y}_k = \boldsymbol{y}'_k$ for each $1 \leq k \leq r$. For the basis case it holds that

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

Now $\text{TDec}((\boldsymbol{K}, \boldsymbol{V}), (\boldsymbol{y}_0))$ is completely determined by $\boldsymbol{y}_0$ and the values of the form $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V})$, where $\boldsymbol{q}$ is query that only depends on $\boldsymbol{y}_0$. But $\text{Att}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Att}(q, \boldsymbol{K}', \boldsymbol{V}')$, for any such a query q from equation 20, and thus

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

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

Appendix B. Missing proofs from Section 4

Proof [**of Lemma 7**] Let $\boldsymbol{q} \in \mathbb{Q}^d$ be a vector such that $\boldsymbol{q} = [\_, \ldots, \_, 1, j, \_, \_]$, where $j \in \mathbb{N}$ and '_' is an arbitrary value. We next prove that

$$\begin{array}{rcll} \text{Att}(\boldsymbol{q}, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) & = & [& 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & [\![\ \alpha^{(j)}\ ]\!], \beta^{(j)}, \mathbf{0}_s, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\quad] \end{array}$$

where $\alpha^{(j)}$ and $\beta^{(j)}$ are defined as

$$\alpha^{(j)} = \begin{cases} s_j & 1 \leq j \leq n \\ s_n & j > n \end{cases}$$

$$\beta^{(j)} = \begin{cases} j & j \leq n \\ n & j > n \end{cases}$$

Recall that $\boldsymbol{K}^{\mathbf{e}} = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n)$ is such that $\boldsymbol{k}_i = [\ 0, \ldots, 0, i, -1, 0, 0\ ]$. Then

$$\text{score}_{\varphi}(\boldsymbol{q}, \boldsymbol{k}_i) = \varphi(\langle \boldsymbol{q}, \boldsymbol{k}_i \rangle) = -|\langle \boldsymbol{q}, \boldsymbol{k}_i \rangle| = -|i - j|.$$

Notice that, if $j \leq n$, then the above expression is maximized when $i = j$. Otherwise, if $j > n$ then the expression is maximized when $i = n$. Then $\text{Att}(\boldsymbol{q}, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) = \boldsymbol{v}_{i^\star}$, where

$$i^\star = \begin{cases} j & j \leq n \\ n & j > n \end{cases}$$

Therefore, $i^\star$ as just defined is exactly $\beta^{(j)}$. Thus, given that $\boldsymbol{v}_i$ is defined as

$$\begin{array}{rcll} \boldsymbol{v}_i & = & [& 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & [\![\ s_i\ ]\!], i, \mathbf{0}_s, 0, \\ & & & 0, \ldots, 0 \qquad\quad] \end{array}$$

we obtain that

$$\begin{array}{rcl} \text{Att}(\boldsymbol{q}, \boldsymbol{K}^{\mathbf{e}}, \boldsymbol{V}^{\mathbf{e}}) = \boldsymbol{v}_{i^\star} & = & [\ 0, \ldots, 0, \\ & & \quad 0, \ldots, 0, \\ & & \quad [\![\ s_{i^\star}\ ]\!], i^\star, \mathbf{0}_s, 0, \\ & & \quad 0, \ldots, 0 \qquad\qquad] \\ & & \\ & = & [\ 0, \ldots, 0, \\ & & \quad 0, \ldots, 0, \\ & & \quad [\![\ \alpha^{(j)}\ ]\!], \beta^{(j)}, \mathbf{0}_s, 0, \\ & & \quad 0, \ldots, 0 \qquad\qquad] \end{array}$$

which is what we wanted to prove. ■

Proof [**of Lemma 8**] In order to prove the lemma we need some intermediate notions and properties. Assume that the injective function $\pi_1 : \Sigma \to \{1, \ldots, |\Sigma|\}$ is the one used to construct the one-hot vectors $[\![s]\!]$ for $s \in \Sigma$, and that $\pi_2 : Q \to \{1, \ldots, |Q|\}$ is the one used to construct $[\![q]\!]$ for $q \in Q$. Using π_1 and π_2 one can construct one-hot vectors for pairs in the set $Q \times \Sigma$. Formally, given $(q, s) \in Q \times \Sigma$ we denote by $[\![(q, s)]\!]$ a one-hot vector with a 1 in position $(\pi_1(s) - 1)|Q| + \pi_2(q)$ and a 0 in every other position. To simplify the notation, we define

$$\pi(q, s) := (\pi_1(s) - 1)|Q| + \pi_2(q).$$

One can similarly construct an injective function π' from $Q \times \Sigma \times \{-1, 1\}$ such that $\pi'(q, s, m) = \pi(q, s)$ if $m = -1$, and $\pi'(q, s, m) = |Q||\Sigma| + \pi(q, s)$ otherwise. We denote by $[\![(q, s, m)]\!]$ the corresponding one-hot vector for each $(q, s, m) \in Q \times \Sigma \times \{-1, 1\}$.

We prove three useful properties below. In every case we assume that $q \in Q$, $s \in \Sigma$, $m \in \{-1, 1\}$, and $\delta(\cdot, \cdot)$ is the transition function of the Turing machine M.

1. There exists $f_1 : \mathbb{Q}^{|Q|+|\Sigma|} \to \mathbb{Q}^{|Q||\Sigma|}$ such that $f_1([\![q]\!], [\![s]\!]) = [\![(q, s)]\!]$.
2. There exists $f_\delta : \mathbb{Q}^{|Q||\Sigma|} \to \mathbb{Q}^{2|Q||\Sigma|}$ such that $f_\delta([\![(q, s)]\!]) = [\![\delta(q, s)]\!]$.
3. There exists $f_2 : \mathbb{Q}^{2|Q||\Sigma|} \to \mathbb{Q}^{|Q|+|\Sigma|+1}$ such that $f_2([\![(q, s, m)]\!]) = [\, [\![q]\!], [\![s]\!], m \,]$.

To show (1), let us denote by $\boldsymbol{S}_i$, with $i \in \{1, \ldots, |\Sigma|\}$, a binary matrix of dimensions $|\Sigma| \times |Q|$ such that every cell of the form $(i, \cdot)$ in $\boldsymbol{S}_i$ contains a 1, and every other cell contains a 0. We note that for every $s \in \Sigma$ it holds that $[\![s]\!]\boldsymbol{S}_i = [1, \ldots, 1]$ if and only if $\pi_1(s) = i$; otherwise $[\![s]\!]\boldsymbol{S}_i = [0, \ldots, 0]$. Now, consider the vector $\boldsymbol{v}_{(q,s)}$

$$\boldsymbol{v}_{(q,s)} = [\, [\![q]\!] + [\![s]\!]\boldsymbol{S}_1, [\![q]\!] + [\![s]\!]\boldsymbol{S}_2, \ldots, [\![q]\!] + [\![s]\!]\boldsymbol{S}_{|\Sigma|} \,].$$

We first note that for every $i \in \{1, \ldots, |\Sigma|\}$, if $\pi_1(s) \neq i$ then

$$[\![q]\!] + [\![s]\!]\boldsymbol{S}_i = [\![q]\!] + [0, \ldots, 0] = [\![q]\!].$$

Moreover,

$$[\![q]\!] + [\![s]\!]\boldsymbol{S}_{\pi_1(s)} = [\![q]\!] + [1, \ldots, 1]$$

is a vector that contains a 2 exactly at index $\pi_2(q)$, and a 1 in all other positions. Thus, the vector $\boldsymbol{v}_{(q,s)}$ contains a 2 exactly at position $(\pi_1(s) - 1)|Q| + \pi_2(q)$ and either a 0 or a 1 in every other position.

Now, let us denote by $\boldsymbol{o}$ a vector in $\mathbb{Q}^{|Q||\Sigma|}$ that has a 1 in every position and consider the following affine transformation

$$g_1([\, [\![q]\!], [\![s]\!] \,]) = \boldsymbol{v}_{(q,s)} - \boldsymbol{o}. \tag{21}$$

Vector $g_1([\, [\![q]\!], [\![s]\!] \,])$ contains a 1 exclusively at position $(\pi_1(s) - 1)|Q| + \pi_2(q) = \pi(q, s)$, and a value less than or equal to 0 in every other position. Thus, to construct $f_1(\cdot)$ we can apply the piecewise-linear sigmoidal activation $\sigma(\cdot)$ (see Equation (1)) to obtain

$$f_1([\, [\![q]\!], [\![s]\!] \,]) = \sigma(g_1([\, [\![q]\!], [\![s]\!] \,])) = \sigma(\boldsymbol{v}_{(q,s)} - \boldsymbol{o}) = [\![(q, s)]\!], \tag{22}$$

which is what we wanted.

Now, to show (2), let us denote by $\boldsymbol{M}^{\delta}$ a matrix of dimensions $(|Q||\Sigma|) \times (2|Q||\Sigma|)$ constructed as follows. For $(q, s) \in Q \times \Sigma$, if $\delta(q, s) = (p, r, m)$ then $\boldsymbol{M}^{\delta}$ has a 1 at position $(\pi(q, s), \pi'(p, r, m))$ and it has a 0 in every other position. That is,

$$\boldsymbol{M}^{\delta}_{\pi(q,s),:} = [\![(p, r, m)]\!] = [\![\delta(q, s)]\!].$$

It is straightforward to see then that $[\![(q, s)]\!]\boldsymbol{M}^{\delta} = [\![\delta(q, s)]\!]$, and thus we can define $f_2(\cdot)$ as

$$f_2([\![(q, s)]\!]) = [\![(q, s)]\!]\boldsymbol{M}^{\delta} = [\![\delta(q, s)]\!]. \tag{23}$$

To show (3), consider the matrix $\boldsymbol{A}$ of dimensions $(2|Q||\Sigma|) \times (|Q| + |\Sigma| + 1)$ such that

$$\boldsymbol{A}_{\pi'(q,s,m),:} = [\, [\![q]\!], [\![s]\!], m \,].$$

Then we can define $f_3(\cdot)$ as

$$f_3([\![(q, s, m)]\!]) = [\![(q, s, m)]\!]\boldsymbol{A} = [\, [\![q]\!], [\![s]\!], m \,]. \tag{24}$$

We are now ready to begin with the proof of the lemma. Recall that $\boldsymbol{a}^1_i$ is given by

$$\begin{array}{rcll} \boldsymbol{a}^1_i & = & [& [\![q^{(i)}]\!], [\![s^{(i)}]\!], m^{(i-1)}, \\ & & & 0, \ldots, 0, \\ & & & [\![\alpha^{(i+1)}]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ & & & 1, (i+1), 1/(i+1), 1/(i+1)^2 \quad] \end{array}$$

We need to construct a function $O_1 : \mathbb{Q}^d \to \mathbb{Q}^d$ such that

$$\begin{array}{rcll} O_1(\boldsymbol{a}^1_i) & = & [& -[\![q^{(i)}]\!], -[\![s^{(i)}]\!], -m^{(i-1)}, \\ & & & [\![q^{(i+1)}]\!], [\![v^{(i)}]\!], m^{(i)}, m^{(i-1)}, 0, 0 \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad] \end{array}$$

We first apply function $h_1(\cdot)$ defined as follows. Let us define

$$\hat{m}^{(i-1)} := \frac{1}{2} m^{(i-1)} + \frac{1}{2}.$$

Note that $\hat{m}^{(i-1)}$ is 0 if $m^{(i-1)} = -1$, it is $\frac{1}{2}$ if $m^{(i-1)} = 0$ (which occurs only when $i = 1$), and it is 1 if $m^{(i-1)} = 1$. We use this transformation exclusively to represent $m^{(i-1)}$ with a value in $[0, 1]$.

Now, consider $h_1(\boldsymbol{a}^1_i)$ defined by

$$h_1(\boldsymbol{a}^1_i) = [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], \hat{m}^{(i-1)}, g_1([[\![q^{(i)}]\!], [\![s^{(i)}]\!]]) \,],$$

where $g_1(\cdot)$ is the function defined above in Equation (21). It is clear that $h_1(\cdot)$ is an affine transformation. Moreover, we note that except for $g_1([[\![q^{(i)}]\!], [\![s^{(i)}]\!]])$ all values in $h_1(\boldsymbol{a}^1_i)$ are in the interval $[0, 1]$. Thus, if we apply function $\sigma(\cdot)$ to $h_1(\boldsymbol{a}^1_i)$ we obtain

$$\begin{array}{rcl} \sigma(h_1(\boldsymbol{a}^1_i)) & = & [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], \hat{m}^{(i-1)}, \sigma(g_1([[\![q^{(i)}]\!], [\![s^{(i)}]\!]])) \,] \\ & = & [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], \hat{m}^{(i-1)}, [\![(q^{(i)}, s^{(i)})]\!] \,], \end{array}$$

where the second equality is obtained from equation (22). We can then define $h_2(\cdot)$ such that

$$\begin{aligned} h_2(\sigma(h_1(\boldsymbol{a}_i^1))) &= [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], 2\hat{m}^{(i-1)} - 1, f_2([\![(q^{(i)}, s^{(i)})]\!]) \,] \\ &= [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], m^{(i-1)}, [\![\delta(q^{(i)}, s^{(i)})]\!] \,] \\ &= [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], m^{(i-1)}, [\![(q^{(i+1)}, v^{(i)}, m^{(i)})]\!] \,], \end{aligned}$$

where the second equality is obtained from equation (23). Now we can define $h_3(\cdot)$ as

$$\begin{aligned} h_3(h_2(\sigma(h_1(\boldsymbol{a}_i^1)))) &= [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], m^{(i-1)}, f_3([\![(q^{(i+1)}, v^{(i)}, m^{(i)})]\!]) \,] \\ &= [\, [\![q^{(i)}]\!], [\![s^{(i)}]\!], m^{(i-1)}, [\![q^{(i+1)}]\!], [\![v^{(i)}]\!], m^{(i)} \,], \end{aligned}$$

where the second equality is obtained from equation (24).

Finally we can apply a function $h_4(\cdot)$ to just reorder the values and multiply some components by -1 to complete our construction

$$\begin{aligned} O_1(\boldsymbol{a}_i^1) \;=\; h_4(h_3(h_2(\sigma(h_1(\boldsymbol{a}_i^1))))) \;=\; [\; & -[\![q^{(i)}]\!], -[\![s^{(i)}]\!], -m^{(i-1)}, \\ & [\![q^{(i+1)}]\!], [\![v^{(i)}]\!], m^{(i)}, m^{(i-1)}, 0, 0 \\ & 0, \ldots, 0, \\ & 0, \ldots, 0 \;] \end{aligned}$$

We note that we applied a single non-linearity and all other functions are affine transformations. Thus $O_1(\cdot)$ can be implemented with a two-layer feed-forward network. ■

Proof [of Lemma 9] Recall that $\boldsymbol{z}_i^1$ is the following vector

$$\begin{aligned} \boldsymbol{z}_i^1 \;=\; [\; & 0, \ldots, 0, \\ & [\![q^{(i+1)}]\!], [\![v^{(i)}]\!], m^{(i)}, m^{(i-1)}, 0, 0, \\ & [\![\alpha^{(i+1)}]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ & 1, (i+1), 1/(i+1), 1/(i+1)^2 \;] \end{aligned}$$

We consider $Q_2 : \mathbb{Q}^d \to \mathbb{Q}^d$ and $K_2 : \mathbb{Q}^d \to \mathbb{Q}^d$ to be trivial functions that for every input produce an output vector composed exclusively of 0s. Moreover, we consider $V_2 : \mathbb{Q}^d \to \mathbb{Q}^d$ such that for every $j \in \{0, 1, \ldots, i\}$,

$$\begin{aligned} V_2(\boldsymbol{z}_j^1) \;=\; [\; & 0, \ldots, 0, \\ & \mathbf{0}_q, \mathbf{0}_s, 0, 0, m^{(j)}, m^{(j-1)}, \\ & 0, \ldots, 0, \\ & 0, \ldots, 0 \;] \end{aligned}$$

Then, since $K_2(\boldsymbol{z}_j^1) = [0, \ldots, 0]$, it holds that $\text{score}_\varphi(Q_2(\boldsymbol{z}_i^1), K_2(\boldsymbol{z}_j^1)) = 0$ for every $j \in \{0, \ldots, i\}$. Thus, we have that the attention $\text{Att}(Q_2(\boldsymbol{z}_i^1), K_2(\boldsymbol{Z}_i^1), V_2(\boldsymbol{Z}_i^1))$ corresponds precisely to the average of the vectors in $V_2(\boldsymbol{Z}_i^1) = V_2(\boldsymbol{z}_0^1, \ldots, \boldsymbol{z}_i^1)$, i.e.,

$$\begin{aligned} \text{Att}(Q_2(\boldsymbol{z}_i^1), K_2(\boldsymbol{Z}_i^1), V_2(\boldsymbol{Z}_i^1)) &= \tfrac{1}{(i+1)} \textstyle\sum_{j=0}^{i} V_2(\boldsymbol{z}_j^1) \\ &= [\; 0, \ldots, 0, \\ &\qquad \mathbf{0}_q, \mathbf{0}_s, 0, 0, \tfrac{1}{(i+1)} \textstyle\sum_{j=0}^{i} m^{(j)}, \tfrac{1}{(i+1)} \textstyle\sum_{j=0}^{i} m^{(j-1)}, \\ &\qquad 0, \ldots, 0, \\ &\qquad 0, \ldots, 0 \;] \end{aligned}$$

Then, since $m^{(0)} + \cdots + m^{(i)} = c^{(i+1)}$ and $m^{(0)} + \cdots + m^{(i-1)} = c^{(i)}$ it holds that

$$\begin{array}{rcll} \mathrm{Att}(Q_2(\boldsymbol{z}_i^1), K_2(\boldsymbol{Z}_i^1), V_2(\boldsymbol{Z}_i^1)) & = & [& 0, \ldots, 0, \\ & & & \mathbf{0}_q, \mathbf{0}_s, 0, 0, \frac{c^{(i+1)}}{(i+1)}, \frac{c^{(i)}}{(i+1)}, \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0 \qquad\qquad\qquad\qquad] \end{array}$$

which is exactly what we wanted to show. ■

Proof [**of Lemma 10**] Recall that $\boldsymbol{z}_i^2$ is the following vector

$$\begin{array}{rcll} \boldsymbol{z}_i^2 & = & [& 0, \ldots, 0, \\ & & & [\![\ q^{(i+1)}\ ]\!], [\![\ v^{(i)}\ ]\!], m^{(i)}, m^{(i-1)}, \frac{c^{(i+1)}}{(i+1)}, \frac{c^{(i)}}{(i+1)}, \\ & & & [\![\ \alpha^{(i+1)}\ ]\!], \beta^{(i+1)}, \mathbf{0}_s, 0, \\ & & & 1, (i+1), 1/(i+1), 1/(i+1)^2 \qquad\qquad] \end{array}$$

We need to construct functions $Q_3(\cdot)$, $K_3(\cdot)$, and $V_3(\cdot)$ such that

$$\begin{array}{rcll} \mathrm{Att}(Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{Z}_i^2), V_3(\boldsymbol{Z}_i^2)) & = & [& 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & \mathbf{0}_s, 0, [\![\ v^{(\ell(i+1))}\ ]\!], \ell(i+1), \\ & & & 0, \ldots, 0 \qquad\qquad\qquad\qquad] \end{array}$$

We first define the query function $Q_3 : \mathbb{Q}^d \to \mathbb{Q}^d$ such that

$$\begin{array}{rcll} Q_3(\boldsymbol{z}_i^2) & = & [& 0, \ldots, 0 \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & 0, \frac{c^{(i+1)}}{(i+1)}, \frac{1}{(i+1)}, \frac{1}{3(i+1)^2} \quad] \end{array}$$

Now, for every $j \in \{0, 1, \ldots, i\}$ we define $K_3 : \mathbb{Q}^d \to \mathbb{Q}^d$ and $V_3 : \mathbb{Q}^d \to \mathbb{Q}^d$ such that

$$\begin{array}{rcll} K_3(\boldsymbol{z}_j^2) & = & [& 0, \ldots, 0 \\ & & & 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & 0, \frac{1}{(j+1)}, \frac{-c^{(j)}}{(j+1)}, \frac{1}{(j+1)^2} \quad] \end{array}$$

$$\begin{array}{rcll} V_3(\boldsymbol{z}_j^2) & = & [& 0, \ldots, 0, \\ & & & 0, \ldots, 0, \\ & & & \mathbf{0}_s, 0, [\![\ v^{(j)}\ ]\!], j, \\ & & & 0, \ldots, 0 \qquad\qquad] \end{array}$$

It is clear that the three functions are linear transformations and thus they can be defined by feed-forward networks.

Consider now the attention $\mathrm{Att}(Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{Z}_i^2), V_3(\boldsymbol{Z}_i^2))$. In order to compute this value, and since we are considering hard attention, we need to find a value $j \in \{0, 1, \ldots, i\}$ that maximizes

$$\mathrm{score}_\varphi(Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{z}_j^2)) = \varphi(\langle Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{z}_j^2)\rangle).$$

Let $j^\star$ be any such a value. Then

$$\mathrm{Att}(Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{Z}_i^2), V_3(\boldsymbol{Z}_i^2)) \;=\; V_3(\boldsymbol{z}_{j^\star}^2).$$

We show next that given our definitions above, it always holds that $j^\star = \ell(i+1)$, and hence $V_3(\boldsymbol{z}_{j^\star}^2)$ is exactly the vector that we wanted to obtain. Moreover, this implies that $j^\star$ is actually unique.

To simplify the notation, we denote by χ_j^i the dot product $\langle Q_3(\boldsymbol{z}_i^2), K_3(\boldsymbol{z}_j^2)\rangle$. Thus, we need to find $j^\star = \arg\max_j \varphi(\chi_j^i)$. Recall that, given the definition of φ (see equation (13)), it holds that

$$\arg\max_{j\in\{0,\ldots,i\}} \varphi(\chi_j^i) \;=\; \arg\min_{j\in\{0,\ldots,i\}} |\chi_j^i|.$$

Then, it is enough to prove that

$$\arg\min_{j\in\{0,\ldots,i\}} |\chi_j^i| \;=\; \ell(i+1).$$

Now, by our definition of $Q_3(\cdot)$ and $K_3(\cdot)$ we have that

$$\begin{aligned}\chi_j^i &= \frac{c^{(i+1)}}{(i+1)(j+1)} - \frac{c^{(j)}}{(i+1)(j+1)} + \frac{1}{3(i+1)^2(j+1)^2} \\ &= \varepsilon_i\varepsilon_j \cdot \left(c^{(i+1)} - c^{(j)} + \frac{\varepsilon_i\varepsilon_j}{3}\right)\end{aligned}$$

where $\varepsilon_k = \frac{1}{(k+1)}$. We next prove the following auxiliary property.

If j_1 is such that $c^{(j_1)} \neq c^{(i+1)}$ and j_2 is such that $c^{(j_2)} = c^{(i+1)}$, then $|\chi_{j_2}^i| < |\chi_{j_1}^i|$. (25)

In order to prove (25), assume first that $j_1 \in \{0,\ldots,i\}$ is such that $c^{(j_1)} \neq c^{(i+1)}$. Then we have that $|c^{(i+1)} - c^{(j_1)}| \geq 1$ since $c^{(i+1)}$ and $c^{(j_1)}$ are integer values. From this we have two possibilities for $\chi_{j_1}^i$:

- If $c^{(i+1)} - c^{(j_1)} \leq -1$, then

  $$\chi_{j_1}^i \;\leq\; -\varepsilon_i\varepsilon_{j_1} + \frac{(\varepsilon_i\varepsilon_{j_1})^2}{3}.$$

  Notice that $1 \geq \varepsilon_{j_1} \geq \varepsilon_i > 0$. Then we have that $\varepsilon_i\varepsilon_{j_1} \geq (\varepsilon_i\varepsilon_{j_1})^2 > \frac{1}{3}(\varepsilon_i\varepsilon_{j_1})^2$, and thus

  $$|\chi_{j_1}^i| \;\geq\; \varepsilon_i\varepsilon_{j_1} - \frac{(\varepsilon_i\varepsilon_{j_1})^2}{3}$$

  Finally, and using again that $1 \geq \varepsilon_{j_1} \geq \varepsilon_i > 0$, from the above equation we obtain that

  $$|\chi_{j_1}^i| \;\geq\; \varepsilon_i\varepsilon_i - \frac{(\varepsilon_i\varepsilon_{j_1})^2}{3} \;\geq\; (\varepsilon_i)^2 - \frac{(\varepsilon_i)^2}{3} \;\geq\; \frac{2(\varepsilon_i)^2}{3}.$$

- If $c^{(i+1)} - c^{(j_1)} \geq 1$, then $\chi_{j_1}^i \geq \varepsilon_i\varepsilon_{j_1} + \frac{1}{3}(\varepsilon_i\varepsilon_{j_1})^2$. Since $1 \geq \varepsilon_{j_1} \geq \varepsilon_i > 0$ we obtain that $|\chi_{j_1}^i| \geq \varepsilon_i\varepsilon_{j_1} \geq \varepsilon_i\varepsilon_i \geq \frac{2}{3}(\varepsilon_i)^2$.

Thus, we have that if $c^{(j_1)} \neq c^{(i+1)}$ then $|\chi^i_{j_1}| \geq \frac{2}{3}(\varepsilon_i)^2$.

Now assume $j_2 \in \{0, \ldots, i\}$ is such that $c^{(j_2)} = c^{(i+1)}$. In this case we have that

$$|\chi^i_{j_2}| \;=\; \frac{(\varepsilon_i \varepsilon_{j_2})^2}{3} \;=\; \frac{(\varepsilon_i)^2(\varepsilon_{j_2})^2}{3} \;\leq\; \frac{(\varepsilon_i)^2}{3},$$

where the last inequality holds since $0 \leq \varepsilon_{j_2} \leq 1$.

We showed that if $c^{(j_1)} \neq c^{(i+1)}$ then $|\chi^i_{j_1}| \geq \frac{2}{3}(\varepsilon_i)^2$ and if $c^{(j_2)} = c^{(i+1)}$ then $|\chi^i_{j_2}| \leq \frac{1}{3}(\varepsilon_i)^2$, which implies that $|\chi^i_{j_2}| < |\chi^i_{j_1}|$. This completes the proof of the property in (25).

We now prove that $\arg\min_j |\chi^i_j| = \ell(i+1)$. Recall first that $\ell(i+1)$ is defined as

$$\ell(i+1) = \begin{cases} \max\{j \mid j \leq i \text{ and } c^{(j)} = c^{(i+1)}\} & \text{if there exists } j \leq i \text{ s.t. } c^{(j)} = c^{(i+1)}, \\ i & \text{otherwise.} \end{cases}$$

Assume first that there exists $j \leq i$ such that $c^{(j)} = c^{(i+1)}$. By (25) we know that

$$\begin{aligned} \arg\min_{j \in \{0,\ldots,i\}} |\chi^i_j| &= \arg\min_{j \text{ s.t. } c^{(j)} = c^{(i+1)}} |\chi^i_j| \\ &= \arg\min_{j \text{ s.t. } c^{(j)} = c^{(i+1)}} \frac{(\varepsilon_i \varepsilon_j)^2}{3} \\ &= \arg\min_{j \text{ s.t. } c^{(j)} = c^{(i+1)}} \varepsilon_j \\ &= \arg\min_{j \text{ s.t. } c^{(j)} = c^{(i+1)}} \frac{1}{j+1} \\ &= \max_{j \text{ s.t. } c^{(j)} = c^{(i+1)}} j \\ &= \max\{j \mid c^{(j)} = c^{(i+1)}\} \end{aligned}$$

On the contrary, assume that for every $j \leq i$ it holds that $c^{(j)} \neq c^{(i+1)}$. We will prove that in this case $|\chi^j_i| > |\chi^i_i|$, for every $j < i$, and thus $\arg\min_{j \in \{0,\ldots,i\}} |\chi^i_j| = i$. Now, since $c^{(j)} \neq c^{(i+1)}$ for every $j \leq i$, then $c^{(i+1)}$ is a cell that has never been visited before by M. Given that M never makes a transition to the left of its initial cell, then cell $c^{(i+1)}$ is a cell that appears to the right of every other previously visited cell. This implies that $c^{(i+1)} > c^{(j)}$ for every $j \leq i$. Thus, for every $j \leq i$ we have $c^{(i+1)} - c^{(j)} \geq 1$. Hence,

$$|\chi^i_j| \;=\; \chi^i_j \;\geq\; \varepsilon_i \varepsilon_j + \frac{1}{3}(\varepsilon_i \varepsilon_j)^2.$$

Moreover, notice that if $j < i$ then $\varepsilon_j > \varepsilon_i$ and thus, if $j < i$ we have that

$$|\chi^i_j| \;\geq\; \varepsilon_i \varepsilon_j + \frac{(\varepsilon_i \varepsilon_j)^2}{3} \;>\; \varepsilon_i \varepsilon_i + \frac{(\varepsilon_i \varepsilon_i)^2}{3} \;=\; |\chi^i_i|.$$

Therefore, $\arg\min_{j \in \{0,\ldots,i\}} |\chi^i_j| = i$.

Summing it up, we have shown that

$$\arg\min_{j \in \{0,\ldots,i\}} |\chi^i_j| \;=\; \begin{cases} \max\{j \mid c^{(j)} = c^{(i+1)}\} & \text{if there exists } j \leq i \text{ s.t. } c^{(j)} = c^{(i+1)}, \\ i & \text{otherwise.} \end{cases}$$

This is exactly the definition of $\ell(i+1)$, which completes the proof of the lemma. ■

Proof [**Lemma 11**] Before proving Lemma 11 we establish the following auxiliary claim that allows us to implement a particular type of *if* statement with a feed-forward network.

Claim 1 *Let $\boldsymbol{x} \in \{0,1\}^m$ and $\boldsymbol{y}, \boldsymbol{z} \in \{0,1\}^n$ be binary vectors, and let $b \in \{0,1\}$. There exists a two-layer feed-forward network $f : \mathbb{Q}^{m+2n+1} \to \mathbb{Q}^{m+n}$ such that*

$$f([\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, b]) = \begin{cases} [\boldsymbol{x}, \boldsymbol{y}] & \text{if } b = 0, \\ [\boldsymbol{x}, \boldsymbol{z}] & \text{if } b = 1. \end{cases}$$

Proof Consider the function $f_1 : \mathbb{Q}^{m+2n+1} \to \mathbb{Q}^{m+2n}$ such that

$$f_1([\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, b]) = [\boldsymbol{x}, \boldsymbol{y} - b\mathbf{1}, \boldsymbol{z} + b\mathbf{1} - \mathbf{1}],$$

where $\mathbf{1}$ is the n-dimensional vector $[1, 1, \ldots, 1]$. Thus, we have that

$$f_1([\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, b]) = \begin{cases} [\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} - \mathbf{1}] & \text{if } b = 0, \\ [\boldsymbol{x}, \boldsymbol{y} - \mathbf{1}, \boldsymbol{z}] & \text{if } b = 1. \end{cases}$$

Now, since $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ are all binary vectors, it is easy to obtain by applying the piecewise-linear sigmoidal activation function σ that

$$\sigma(f_1([\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, b])) = \begin{cases} [\boldsymbol{x}, \boldsymbol{y}, \mathbf{0}] & \text{if } b = 0, \\ [\boldsymbol{x}, \mathbf{0}, \boldsymbol{z}] & \text{if } b = 1, \end{cases}$$

where $\mathbf{0}$ is the n-dimensional vector $[0, 0, \ldots, 0]$. Finally, consider the function $f_2 : \mathbb{Q}^{m+2n} \to \mathbb{Q}^{m+n}$ such that $f_2([\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}]) = [\boldsymbol{x}, \boldsymbol{y} + \boldsymbol{z}]$. Then we have that

$$f_2(\sigma(f_1([\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, b]))) = \begin{cases} [\boldsymbol{x}, \boldsymbol{y}] & \text{if } b = 0, \\ [\boldsymbol{x}, \boldsymbol{z}] & \text{if } b = 1. \end{cases}$$

We note that $f_1(\cdot)$ and $f_2(\cdot)$ are affine transformations, and thus $f(\cdot) = f_2(\sigma(f_1(\cdot)))$ is a two-layer feed-forward network. This completes our proof. ■

We can now continue with the proof of Lemma 11. Recall that $\boldsymbol{z}_r^3$ is the following vector

$$\begin{aligned} \boldsymbol{z}_r^3 \quad = \quad [\quad & 0, \ldots, 0, \\ & [\![\, q^{(r+1)} \,]\!], [\![\, v^{(r)} \,]\!], m^{(r)}, m^{(r-1)}, \tfrac{c^{(r+1)}}{(r+1)}, \tfrac{c^{(r)}}{(r+1)}, \\ & [\![\, \alpha^{(r+1)} \,]\!], \beta^{(r+1)}, [\![\, v^{(\ell(r+1))} \,]\!], \ell(r+1), \\ & 1, (r+1), 1/(r+1), 1/(r+1)^2 \qquad\qquad\qquad\qquad] \end{aligned}$$

Let us denote by $[\![\, m^{(r)} \,]\!]$ a vector such that

$$[\![\, m^{(r)} \,]\!] = \begin{cases} [1, 0] & \text{if } m^{(r)} = 1, \\ [0, 1] & \text{if } m^{(r)} = -1. \end{cases}$$

We first consider the function $f_1(\cdot)$ such that

$$f_1(\boldsymbol{z}_r^3) \;=\; [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!], (r+1)-\beta^{(r+1)}, [\![\, v^{(\ell(r+1))} \,]\!], [\![\, \# \,]\!], \ell(r+1)-(r-1)\,].$$

It is straightforward that $f_1(\cdot)$ can be implemented as an affine transformation. Just notice that $[\![\, \# \,]\!]$ is a fixed vector, $\ell(r+1)-(r-1) = \ell(r+1)-(r+1)+2$ and that $[\![\, m^{(r)} \,]\!] = [\frac{m^{(r)}}{2}, \frac{-m^{(r)}}{2}]+[\frac{1}{2}, \frac{1}{2}]$. Moreover, all values in $f_1(\boldsymbol{z}_r^3)$ are binary values except for $(r+1)-\beta^{(r+1)}$ and $\ell(r+1)-(r-1)$. Thus, if we apply function $\sigma(\cdot)$ to $f_1(\boldsymbol{z}_r^3)$ we obtain

$$\sigma(f_1(\boldsymbol{z}_r^3)) \;=\; [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!], b_1, [\![\, v^{(\ell(r+1))} \,]\!], [\![\, \# \,]\!], b_2\,],$$

where $b_1 = \sigma((r+1)-\beta^{(r+1)})$ and $b_2 = \sigma(\ell(r+1)-(r-1))$. By the definition of $\beta^{(r+1)}$ we know that $\beta^{(r+1)} = r+1$ whenever $r+1 \leq n$, and $\beta^{(r+1)} = n$ if $r+1 > n$. Thus we have that

$$b_1 \;=\; \begin{cases} 0 & \text{if } r+1 \leq n \\ 1 & \text{if } r+1 > n \end{cases}$$

For the case of b_2, since $\ell(r+1) \leq r$ we have that $b_2 = 1$ if $\ell(r+1) = r$ and it is 0 otherwise. Therefore,

$$b_2 \;=\; \begin{cases} 1 & \text{if } \ell(r+1) = r \\ 0 & \text{if } \ell(r+1) \neq r \end{cases}$$

Then, we can use the *if* function in Claim 1 to implement a function $f_2(\cdot)$ such that

$$f_2(\sigma(f_1(\boldsymbol{z}_r^3))) \;=\; \begin{cases} [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!], b_1, [\![\, v^{(\ell(r+1))} \,]\!]\,] & \text{if } b_2 = 0, \\ [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!], b_1, [\![\, \# \,]\!]\,] & \text{if } b_2 = 1. \end{cases}$$

We can use again the *if* function in Claim 1 to implement a function $f_3(\cdot)$ such that

$$f_3(f_2(\sigma(f_1(\boldsymbol{z}_r^3)))) \;=\; \begin{cases} [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!]\,] & \text{if } b_2 = 0 \text{ and } b_1 = 0, \\ [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, v^{(\ell(r+1))} \,]\!]\,] & \text{if } b_2 = 0 \text{ and } b_1 = 1, \\ [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!]\,] & \text{if } b_2 = 1 \text{ and } b_1 = 0, \\ [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \# \,]\!]\,] & \text{if } b_2 = 1 \text{ and } b_1 = 1, \end{cases}$$

which can be rewritten as

$$f_3(f_2(\sigma(f_1(\boldsymbol{z}_r^3)))) \;=\; \begin{cases} [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \alpha^{(r+1)} \,]\!]\,] & \text{if } r+1 \leq n, \\ [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, \# \,]\!]\,] & \text{if } r+1 > n \text{ and } \ell(r+1) = r, \\ [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, v^{(\ell(r+1))} \,]\!]\,] & \text{if } r+1 > n \text{ and } \ell(r+1) \neq r. \end{cases}$$

From this, it is easy to prove that

$$f_3(f_2(\sigma(f_1(\boldsymbol{z}_r^3)))) = [\,[\![\, q^{(r+1)} \,]\!], [\![\, m^{(r)} \,]\!], [\![\, s^{(r+1)} \,]\!]\,].$$

This can be obtained from the following observation. If $r+1 \leq n$, then $\alpha^{(r+1)} = s_{r+1} = s^{(r+1)}$. If $r+1 > n$ and $\ell(r+1) = r$, then we know that $c^{(r+1)}$ is visited by M for the

first time at time $r+1$ and this cell is to the right of the original input, which implies that $s^{(r+1)} = \#$. Finally, if $r+1 > n$ and $\ell(r+1) \neq r$, then we know that $c^{(r+1)}$ has been visited before at time $\ell(r+1)$, and thus $s^{(r+1)} = v^{(\ell(r+1))}$.

The final piece of the proof consists in converting $[\![\, m^{(r)} \,]\!]$ back to its value $m^{(r)}$, reorder the values, and add 0s to obtain $\boldsymbol{y}_{r+1}$. We do all this with a final linear transformation $f_4(\cdot)$ such that

$$\begin{aligned} f_4(f_3(f_2(\sigma(f_1(\boldsymbol{z}_r^3))))) &= [\; [\![\, q^{(r+1)} \,]\!], [\![\, s^{(r+1)} \,]\!], m^{(r)}, \\ &\qquad 0, \ldots, 0, \\ &\qquad 0, \ldots, 0, \\ &\qquad 0, \ldots, 0 \;] \\ &= \boldsymbol{y}_{r+1} \end{aligned}$$

This completes the proof of the lemma. ■

References

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

Yining Chen, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight. Recurrent neural networks as weighted language recognizers. In *Conference of the North American Chapter of the Association for Computational Linguistics, NAACL-HLT*, pages 2261–2271, 2018.

George Cybenko. Approximation by superpositions of a sigmoidal function. *MCSS*, 2(4): 303–314, 1989.

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

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Conference of the North American Chapter of the Association for Computational Linguistics, NAACL-HLT*, pages 4171–4186. Association for Computational Linguistics, 2019.

Gamaleldin F. Elsayed, Simon Kornblith, and Quoc V. Le. Saccader: Improving accuracy of hard attention models for vision. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Conference on Neural Information Processing Systems, NeurIPS*, pages 700–712, 2019.

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

Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

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 *Conference on Neural Information Processing Systems, NIPS*, pages 1828–1836, 2015.

Çaglar Gülçehre, Misha Denil, Mateusz Malinowski, Ali Razavi, Razvan Pascanu, Karl Moritz Hermann, Peter W. Battaglia, Victor Bapst, David Raposo, Adam Santoro, and Nando de Freitas. Hyperbolic attention networks. In *International Conference on Learning Representations, ICLR*.

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *arXiv preprint arXiv:1906.06755*, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision, ECCV*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition, CVPR*, pages 770–778, 2016.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Conference on Neural Information Processing Systems, NIPS*, pages 190–198, 2015.

Lukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *International Conference on Learning Representations, ICLR*, 2016.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning, ICML*, 2020.

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

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *International Conference on Learning Representations, ICLR*, 2016.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A tensorized transformer for language modeling. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Conference on Neural Information Processing Systems, NeurIPS*, pages 2229–2239, 2019.

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

Jorge Pérez, Javier Marinkovic, and Pablo Barceló. On the turing completeness of modern neural network architectures. In *International Conference on Learning Representations, ICLR*, 2019.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Conference of the North American Chapter of the Association for Computational Linguistics, NAACL-HLT*, pages 464–468, 2018.

Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. In *Conference on Neural Information Processing Systems, NeurIPS*, pages 12081–12091. Curran Associates, Inc., 2019.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *Conference on Computational Learning Theory, COLT*, pages 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.

Michael Sipser. *Introduction to the Theory of Computation*. Second edition, 2006.

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

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

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning, ICML*, pages 2048–2057, 2015.

Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines. In *Conference on Neural Information Processing Systems, NeurIPS*, 2020.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations, ICLR*, 2020.