Simulating Weighted Automata over Sequences and Trees with Transformers

Michael Rizvi
Mila & DIRO
Université de Montréal

Maude Lizaire
Mila & DIRO
Université de Montréal

Clara Lacroce
McGill University
Mila, Montréal, Canada

Guillaume Rabusseau
Mila & DIRO
Université de Montréal,
CIFAR AI Chair

Abstract

Transformers are ubiquitous models in the natural language processing (NLP) community and have shown impressive empirical successes in the past few years. However, little is understood about how they reason and the limits of their computational capabilities. These models do not process data sequentially, and yet outperform sequential neural models such as RNNs. Recent work has shown that these models can compactly simulate the sequential reasoning abilities of deterministic finite automata (DFAs). This leads to the following question: can transformers simulate the reasoning of more complex finite state machines? In this work, we show that transformers can simulate weighted finite automata (WFAs), a class of models which subsumes DFAs, as well as weighted tree automata (WTA), a generalization of weighted automata to tree structured inputs. We prove these claims formally and provide upper bounds on the sizes of the transformer models needed as a function of the number of states the target automata. Empirically, we perform synthetic experiments showing that transformers are able to learn these compact solutions via standard gradient-based training.

1 INTRODUCTION

Transformers are the backbone of modern NLP systems (Vaswani et al., 2017). These models have shown impressive gains in the past few years. Large pretrained language models can translate texts, write code and can solve math problems, tasks which all require some level of sequential reasoning capabilities (Brown et al., 2020; Chen et al., 2021). However, unlike recurrent neural networks (RNNs) (Elman, 1990; Hochreiter and Schmidhuber, 1997), transformers do not perform their computations sequentially. Instead they process all input tokens in parallel. This defies our intuition as these models do not possess the inductive bias that naturally arises from treating a sequence from beginning to end.

In order to understand how transformers implement sequential reasoning, recent work by Liu et al. (2022) studied connections between deterministic finite automata (DFAs) and transformers. DFAs are simple models that perform deterministic sequential reasoning on strings from a given alphabet, which makes them a perfect candidate for exploring sequential reasoning in attention-based models. To do so, the authors consider the perspective of simulation. Informally, a transformer is said to simulate a DFA if for an input sequence of length T, it can output the sequence of states visited throughout the DFAs computation. The authors also consider how the complexity needed to perform this simulation task varies as a function of T. They find that it is possible to simulate all DFA at length T with a transformer of size $O(\log T)$. They also show that in the case where the automaton is solvable, it is possible to achieve such a result with $O(1)$ size. This sheds light on how transformers can *compactly* encode sequential behavior without explicitly performing sequential computation.

However this is not representative of the capacities of transformers. The type of reasoning they implement can indeed go much further than the simple reasoning done by DFAs. In this work, we propose to go further and consider (i) weighted finite automata (WFAs), a family of automata that generalize DFAs by computing a real-valued function over a sequence instead of simply accepting or rejecting it, and (ii) weighted tree automata (WTA), a generalization of weighted automata

Proceedings of the 27th International Conference on Artificial Intelligence and Statistics (AISTATS) 2024, Valencia, Spain. PMLR: Volume 238. Copyright 2024 by the author(s).

to tree structured inputs. We show that transformers can simulate both WFAs as well as WTAs, and that they can do so *compactly.*

More precisely, we show that, using hard attention and bilinear layers, transformers can *exactly* simulate all WFAs at length T with $O(\log T)$ layers. Moreover, we show that using a more standard transformer implementation, with soft attention and an MLP (multilayer perceptron), transformers can *approximately* simulate all WFAs at length T up to arbitrary precision with $O(\log T)$ layers and MLP width constant in T. This first set of results shows that transformers can learn shortcuts to sequence models significantly more complex than deterministic finite automata. Our second set of results is about computation over trees. For WTAs, the notion of simulation we introduce assumes that the transformer is fed a string representation of a tree and outputs the states of the WTA for each subtree of the input. We show that transformers can simulate WTA to arbitrary precision at length T with $O(\log T)$ layers over balanced trees. Since the class of WTA subsumes classical (non-weighted) tree automata, an important corollary we obtain is that transformers can also simulate tree automata. Our results thus extend the ones of Liu et al. (2022) for DFAs in two directions: from boolean to real weights and from sequences to trees.

Empirically, we study to which extent transformers can be trained to simulate WFAs. We first show that compact solutions can be found in practice using gradient-based optimization. To do so, we train transformers on simulation tasks using synthetic data. We also investigate if the number of layers and embedding size of such solutions scale as theory suggests.

2 PRELIMINARIES

2.1 Notation

We denote with $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ the set of natural, integers and real numbers, respectively. We use bold letters for vectors (*e.g.* $\mathbf{v} \in \mathbb{R}^{d_1}$), bold uppercase letters for matrices (*e.g.* $\mathbf{M} \in \mathbb{R}^{d_1 \times d_2}$) and bold calligraphic letters for tensors (*e.g.* $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$). All vectors considered are column vectors unless otherwise specified. We denote with $\mathbf{I}$ the identity matrix and write $\mathbf{I}_m$ to denote the $m \times m$ identity matrix. We will also denote $\mathbf{0}$ as the matrix full of zeros and use $\mathbf{0}_{m \times n}$ to denote the $m \times n$ zero matrix or simply $\mathbf{0}_m$ when said matrix is square. The i-th row and the j-th column of a matrix $\mathbf{M}$ are denoted by $\mathbf{M}_{i,:}$ and $\mathbf{M}_{:,j}$. We denote the Frobenius norm of a matrix as $\|\mathbf{M}\|_F$ Finally, we will use $\mathbf{e}_i$ to refer to the ith canonical basis vector.

Let Σ be a fixed finite alphabet of symbols, Σ^* the set of all finite strings (words) with symbols in Σ and Σ^n the set of all finite strings of length n. We use ε to denote the empty string. Given $p, s \in \Sigma^*$, we denote with ps their concatenation.

2.2 Weighted Finite Automata

Weighted finite automata are a generalization of finite state machines (Droste et al., 2009; Mohri, 2009; Salomaa and Soittola, 1978). This class of models subsumes deterministic and non-deterministic finite automata, as WFAs can calculate a function over strings in addition to accepting or rejecting a word. While general WFAs can have weights in arbitrary semi-rings, we focus our attention on WFAs with real weights, as they are more relevant to machine learning applications.

Definition 2.1. *A* weighted finite automaton *(WFA) of n states over Σ is a tuple $A = \langle \boldsymbol{\alpha}, \{\mathbf{A}^\sigma\}_{\sigma \in \Sigma}, \boldsymbol{\beta} \rangle$, where $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{R}^n$ are the initial and final weight vectors, respectively, and $\mathbf{A}^\sigma \in \mathbb{R}^{n \times n}$ is the matrix containing the transition weights associated with each symbol $\sigma \in \Sigma$. Every WFA A with real weights realizes a function $f_A : \Sigma^* \to \mathbb{R}$,* i.e. *given a string $x = x_1 \cdots x_t \in \Sigma^*$, it returns $f_A(x) = \boldsymbol{\alpha}^\top \mathbf{A}^{x_1} \cdots \mathbf{A}^{x_t} \boldsymbol{\beta} = \boldsymbol{\alpha}^\top \mathbf{A}^x \boldsymbol{\beta}$.*

To simplify the notation, we write the product of transition maps $\mathbf{A}^{x_1} \cdots \mathbf{A}^{x_t}$ as $\mathbf{A}^{x_1 \cdots x_t}$ or even $\mathbf{A}^{x_{1:t}}$ for longer sequences.

Similarly to ordinary DFAs, we define the *state* of a WFA on a word $x_1 \cdots x_t$ to be the product $\boldsymbol{\alpha}^\top \mathbf{A}^{x_1} \ldots \mathbf{A}^{x_t}$. In light of this, we define $A(x)$ to be the function that returns the sequence of states for a given word $x \in \Sigma^T$. More formally, we have

$$A(x) = (\boldsymbol{\alpha}^\top, \boldsymbol{\alpha}^\top \mathbf{A}^{x_1}, \boldsymbol{\alpha}^\top \mathbf{A}^{x_1 x_2}, \ldots, \boldsymbol{\alpha}^\top \mathbf{A}^{x_{1:T}})^\top$$

There exist many model families encompassed by WFAs, one of the most well-known subsets of these models are hidden Markov models (HMMs).

2.3 Weighted Tree Automata

Weighted tree automata (WTA) extend the notion of WFAs to the tree domain. In all generality, WTAs can be defined with weights over an arbitrary semi-ring and have as domain the set of ranked trees over an arbitrary ranked alphabet (Droste et al., 2009). Here, we only consider WTAs with real weights (for their relevance to machine learning applications) defined over binary trees (for simplicity of exposition). We first formally define the domain of WTAs we will consider.

Definition 2.2. *Given a finite alphabet Σ, the set of binary trees with leafs labeled by symbols in Σ is denoted by $\mathfrak{T}_\Sigma$ (or simply $\mathfrak{T}$ if the leaf alphabet is clear from context). Formally, $\mathfrak{T}_\Sigma$ is the smallest set such that $\Sigma \subset \mathfrak{T}_\Sigma$ and $(t_1, t_2) \in \mathfrak{T}_\Sigma$ for all $t_1, t_2 \in \mathfrak{T}_\Sigma$.*

A WTA (with real weights) computes a function mapping trees in $\mathcal{T}$ to real values. In the context of machine learning, they can thus be thought of as parameterized models for functions defined over trees (e.g. probability distributions or scoring functions). The computation of a WTA is performed in a bottom up fashion: (i) for each leaf, the state of a WTA with n states is an n-dimensional vector, (ii) the states for all subtrees are computed recursively from the ground up by applying a bilinear map to the left and right child of each internal nodes (iii) similarly to WFAs, the output of a WTA is then a linear function of the state of the root. Formally,

Definition 2.3. *A weighted tree automaton (WTA) A with n states on $\mathcal{T}$ is a tuple $\langle \boldsymbol{\alpha} \in \mathbb{R}^n, \boldsymbol{\mathcal{T}} \in \mathbb{R}^{n \times n \times n}, \{\mathbf{v}_\sigma \in \mathbb{R}^n\}_{\sigma \in \Sigma} \rangle$. A WTA A computes a function $f_A : \mathcal{T} \to \mathbb{R}$ defined by $f_A(t) = \langle \boldsymbol{\alpha}, \mu(t) \rangle$ where the mapping $\mu : \mathcal{T} \to \mathbb{R}^n$ is recursively defined by*

- *$\mu(\sigma) = \mathbf{v}_\sigma$ for all $\sigma \in \Sigma$,*
- *$\mu((t_1, t_2)) = \boldsymbol{\mathcal{T}} \times_2 \mu(t_1) \times_3 \mu(t_2)$ for all $t_1, t_2 \in \mathcal{T}$.*

The states of an n state WTA are thus the n-dimensional vectors $\mu(\tau)$ for each subtree τ of the input tree. The computation of states of a WTA on an exemple tree is illustrated in Figure 2 (left).

WTAs are naturally related to weighted (and probabilistic) context free grammars in the following ways. First, the set of derivation trees of a context free grammar is a regular tree language, that is a language that can be recognized by a tree automaton (Magidor and Moran, 1970; Comon et al., 2008). Furthermore, a weighted context free grammar (WCFG)) maps any sequence to a real value by summing the weights of all valid derivation trees of the input sequence, where the weight of a tree is the value computed by a given WTA (which defines the WCFG) (Droste et al., 2009).

2.4 Transformers

The transformer architecture used in our construction is very similar to the encoder in the original transformer architecture (Vaswani et al., 2017). First, we define the self-attention mechanism as

$$f(\mathbf{X}) = \mathrm{softmax}(\mathbf{X}\mathbf{W}_Q\mathbf{W}_K^\top\mathbf{X}^\top)\mathbf{X}\mathbf{W}_V,$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times k}$, d is the embedding dimension and k is some chosen dimension, usually with $k < d$. Note that the softmax is taken row-wise.

One can also define the self attention layer using *hard attention* by setting the largest value row-wise to 1 and all others to 0. This can be thought of as each row "selecting" a specific token to attend to.

By taking h copies of this structure, concatenating the outputs of each head and applying a linear layer, we obtain a multi-head attention block, which we denote f_{attn}.

We can now define the full transformer architecture. Given a sequence of length T with embedding dimension d, an *L-layer transformer* is a sequence to sequence network $f_{\mathsf{tf}} : \mathbb{R}^{T \times d} \to \mathbb{R}^{T \times d}$ where each layer is composed of a multi-head attention block followed by a feedforward block in the following manner

$$f_{\mathsf{tf}} = f_{\mathsf{mlp}}^{(L)} \circ f_{\mathsf{attn}}^{(L)} \circ f_{\mathsf{mlp}}^{(L-1)} \circ f_{\mathsf{attn}}^{(L-1)} \circ \cdots \circ f_{\mathsf{mlp}}^{(1)} \circ f_{\mathsf{attn}}^{(1)}.$$

The feedforward block is simply a multilayer perceptron (MLP). The parameters of this MLP are not necessarily the same from one layer to another.

2.5 Bilinear Layers

We now introduce a special feed-forward layer which will be used in the construction for exact simulation of WFAs.

Definition 2.4. *Given two vectors $\mathbf{x}_1 \in \mathbb{R}^{d_1}$ and $\mathbf{x}_2 \in \mathbb{R}^{d_2}$, a bilinear layer is a map from $\mathbb{R}^{d_1} \times \mathbb{R}^{d_2}$ to $\mathbb{R}^{d_3}$ such that*

$$\mathit{BilinearLayer}(\mathbf{x}_1, \mathbf{x}_2) = \boldsymbol{\mathcal{T}} \times_1 \mathbf{x}_1 \times_2 \mathbf{x}_2 + \mathbf{b}$$

where $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and $\mathbf{b} \in \mathbb{R}^{d_3}$ are learnable parameters.

It is easy to see that any bilinear map can be computed by such a layer given that all bilinear maps can be represented as a tensors (similarly to how linear maps are represented by matrices). Note that bilinear layers have been introduced and used for practical applications previously, especially in the context of multi-modal and multiview learning (Gao et al., 2016; Li et al., 2017; Lin et al., 2015).

3 SIMULATING WEIGHTED AUTOMATA OVER SEQUENCES

In this section, we start by introducing the definition of simulation for WFAs, which is crucial to understanding the results in this section. We then state and briefly analyze our main theorems.

3.1 Simulation Definition

Intuitively, simulation can be thought of as reproducing the intermediary steps of computation for a given algorithm. For a WFA, these intermediary steps correspond to the state vectors throughout the computation over a given word.

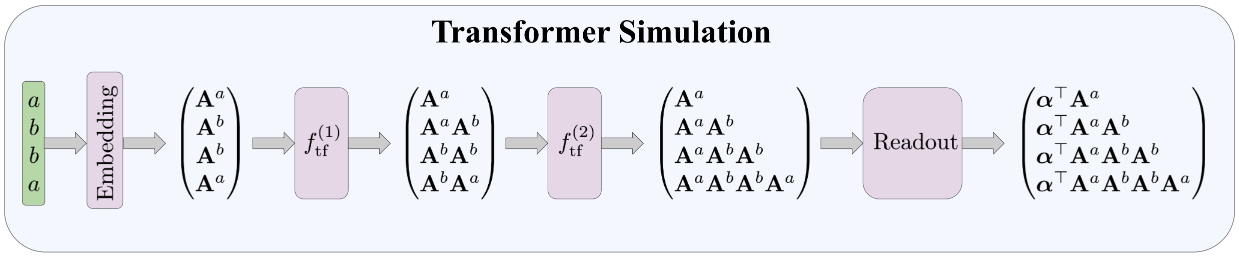

Transformer Simulation
a
b
b
a
Embedding
$\mathbf{A}^a$, $\mathbf{A}^b$, $\mathbf{A}^b$, $\mathbf{A}^a$
$f_{\text{tf}}^{(1)}$
$\mathbf{A}^a$, $\mathbf{A}^a\mathbf{A}^b$, $\mathbf{A}^b\mathbf{A}^b$, $\mathbf{A}^b\mathbf{A}^a$
$f_{\text{tf}}^{(2)}$
$\mathbf{A}^a$, $\mathbf{A}^a\mathbf{A}^b$, $\mathbf{A}^a\mathbf{A}^b\mathbf{A}^b$, $\mathbf{A}^a\mathbf{A}^b\mathbf{A}^b\mathbf{A}^a$
Readout
$\boldsymbol{\alpha}^\top\mathbf{A}^a$, $\boldsymbol{\alpha}^\top\mathbf{A}^a\mathbf{A}^b$, $\boldsymbol{\alpha}^\top\mathbf{A}^a\mathbf{A}^b\mathbf{A}^b$, $\boldsymbol{\alpha}^\top\mathbf{A}^a\mathbf{A}^b\mathbf{A}^b\mathbf{A}^a$

Figure 1: Simulation of the WFA computation over the input $w = abba$ with a transformer.

Definition 3.1. *Given a WFA A over some alphabet Σ, a function $f : \Sigma^T \ ! \ \mathbb{R}^{T \ n}$ exactly simulates A at length T if, for all $x \ 2 \ \Sigma^T$ as input, we have $f(x) = \mathsf{A}(x)$, where $\mathsf{A}(x) = (\boldsymbol{\alpha}^>, \boldsymbol{\alpha}^> \mathbf{A}^{x_1}, \ldots, \boldsymbol{\alpha}^> \mathbf{A}^{x_{1:T}})^>$.*

Additionally, we define the notion of approximate simulation. Intuitively, given some error tolerance ϵ, we can always find a function f which can simulate a WFA up to precision ϵ.

Definition 3.2. *Given a WFA A over some alphabet Σ, a function $f : \Sigma^T \ ! \ \mathbb{R}^{T \ n}$ approximately simulates A at length T with precision $\epsilon > 0$ if for all $x \ 2 \ \Sigma^T$, we have $\mathsf{k}f(x) \quad \mathsf{A}(x)\mathsf{k}_F < \epsilon$.*

Using a T layer transformer, it is easy to simulate a WFA over a sequence of length T. We simply use the transformer as we would an unrolled RNN; performing each step of the computation at the corresponding layer.

However, transformers are typically very shallow networks (Brown et al., 2022), which defies intuition given how deep models tend to be more expressive than their shallow counterparts (Eldan and Shamir, 2016; Cohen et al., 2016). This naturally leads to the following question: can transformers simulate WFAs using a number of layers that is less than linear (in the sequence length)? Following the work of Liu et al. (2022), we define the notion of shortcuts.

Definition 3.3. *Let A be a WFA. If for every $T \quad 0$, there exists a transformer f_T that simulates (exactly or approximately) A at length T with depth $L \quad o(T)$, then we say that there exists a shortcut solution to the problem of simulating A.*

3.2 Main theorems

In the following section, we state our main theorems and discuss their scope as well as their limitations. The proofs of these theorems can be found in Appendix B.

Theorem 1. *Transformers using bilinear layers in place of an MLP and hard attention can exactly simulate all WFAs with n states at length T, with depth $\mathsf{O}(\log T)$, embedding dimension $\mathsf{O}(n^2)$, attention width $\mathsf{O}(n^2)$, MLP width $\mathsf{O}(n^2)$ and $\mathsf{O}(1)$ attention heads.*

The proof of this theorem relies on hard attention as well as the use of bilinear layers. Since this does not correspond to the typical definition of transformer used in practice, we derive a second result. This consists in an approximate version of the previous theorem and relies on a more standard transformer implementation, using a softmax and a standard feedforward MLP.

Theorem 2. *Transformers can approximately simulate all WFAs with n states at length T, up to arbitrary precision $\epsilon > 0$, with depth $\mathsf{O}(\log T)$, embedding dimension $\mathsf{O}(n^2)$, attention width $\mathsf{O}(n^2)$, MLP width $\mathsf{O}(n^4)$ and $\mathsf{O}(1)$ attention heads.*

Notice that the size of the construction does not depend on the approximation error ϵ. This is one of the advantages of our approximate construction: we can achieve arbitrary precision without compromising the size of the model.

The proofs of both theorems rely on the *prefix sum algorithm* (Blelloch, 1990), an algorithm that can compute all T prefixes of a sequence in $\mathsf{O}(\log T)$ time (for more details we refer the reader to the appendix). Formally, this means that given a sequence $x_1, x_2, \ldots, x_T$ as input, the algorithm returns all partial sums $(x_1), (x_1 + x_2), \ldots, (x_1 + \ldots + x_T)$.

For both proofs in this section, we consider sequences of transition maps, and use composition as our "sum" operation. Using the definition of the transformer given in Section 2.4, we present a construction which implements this algorithm. Figure 1 illustrates the key elements of this construction. Here, we take $f_{\mathsf{tf}}^{(\ell)}$ to be the ℓth layer of an L layer transformer such that $f_{\mathsf{tf}}^{(\ell)} = f_{\mathsf{mlp}}^{(\ell)} \quad f_{\mathsf{attn}}^{(\ell)}$.

4 SIMULATING WEIGHTED TREE AUTOMATA

We now turn our focus to analyzing the capacity of transformers to efficiently simulate weighted tree au-

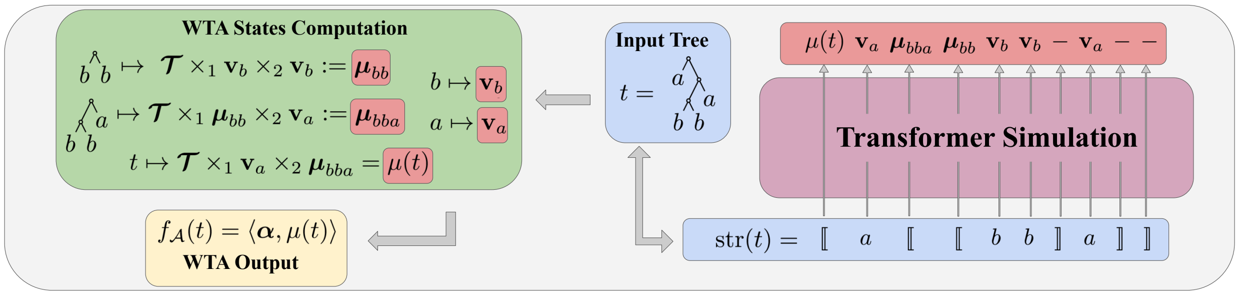

WTA States Computation
$\mapsto \boldsymbol{\mathcal{T}} \times_1 \mathbf{v}_b \times_2 \mathbf{v}_b := \boldsymbol{\mu}_{bb}$
$\mapsto \boldsymbol{\mathcal{T}} \times_1 \boldsymbol{\mu}_{bb} \times_2 \mathbf{v}_a := \boldsymbol{\mu}_{bba}$
$t \mapsto \boldsymbol{\mathcal{T}} \times_1 \mathbf{v}_a \times_2 \boldsymbol{\mu}_{bba} = \mu(t)$
$b \mapsto \mathbf{v}_b$
$a \mapsto \mathbf{v}_a$
Input Tree
$t =$
$f_{\mathcal{A}}(t) = \langle \boldsymbol{\alpha}, \mu(t) \rangle$
WTA Output
$\mu(t)$ $\mathbf{v}_a$ $\boldsymbol{\mu}_{bba}$ $\boldsymbol{\mu}_{bb}$ $\mathbf{v}_b$ $\mathbf{v}_b$ $-$ $\mathbf{v}_a$ $-$ $-$
Transformer Simulation
$\mathrm{str}(t) = [\![\; a \; [\![\; [\![\; b \; b \;]\!] \; a \;]\!] \;]\!]$

Figure 2: Computation of a WTA on the input tree $t = (a, ((b, b), b))$ (left) and simulation of the WTA computation over t with a transformer (right).

tomata.

4.1 Simulation definition

As mentioned in Section 2.3, the states of a WTA of size n are the n-dimensional vectors $\mu(\tau)$ for all subtrees τ of the input tree t. Intuitively, we will say that a transformer can simulate a given WTA if it can compute all subtree states $\mu(\tau)$ when fed as input a string representation of t. We now proceed to formalize this notion of simulation.

Given a tree $t \in \mathcal{T}$, we denote by $\mathrm{str}(t)$ its string representation, omitting commas. More formally, $\mathrm{str}(t) \in \Sigma^* \cup \{[\![,]\!]\}$ is recursively defined by

- $\mathrm{str}(\sigma) = \sigma$ for all $\sigma \in \Sigma$,
- $\mathrm{str}((t_1, t_2)) = [\![\; \mathrm{str}(t_1) \; \mathrm{str}(t_2) \;]\!]$ for all $t_1, t_2 \in \mathcal{T}$.

It is worth mentioning that the mapping $t \mapsto \mathrm{str}(t)$ is injective, thus any tree t can be recovered from its string representation (but not all strings in $(\Sigma \cup \{[\![,]\!]\})^*$ are valid representations of trees).

One can then observe that each opening parenthesis in $\mathrm{str}(t)$ can naturally be mapped to a subtree of t. Formally, given a tree t, let $I_t = \{i = 1, \cdots, |\mathrm{str}(t)| \mid \mathrm{str}(t)_i \neq]\!]\}$ be the set of positions of $\mathrm{str}(t)$ corresponding to opening parenthesis and leaf symbols. Then, the set of subtrees of t can be mapped (one-to-one) to indices in I_t:

Definition 4.1. *Given a tree $t \in \mathcal{T}$, we let $\tau : I_t \to \mathcal{T}$ be the mapping defined by*

$$\tau(i) = \mathrm{str}^{-1}(x_i x_{i+1} \cdots x_j)$$

where j is the unique index such that the string $x_i x_{i+1} \cdots x_j$ is a valid string representation of a tree (i.e. $s = x_i x_{i+1} \cdots x_j$ is the only substring starting in position i such that there exists a tree $\tau \in \mathcal{T}$ for which $\mathrm{str}(\tau) = s$).

Intuitively, we will say that a transformer can simulate a WTA if, given some input $\mathrm{str}(t)$, the output of the transformer for each position $i \in I_t$ is equal to the corresponding state $\mu(\tau(i))$ of the WTA after parsing the subtree τ_i. We can now formally define the notion of WTA simulation.

Definition 4.2. *Given a WTA $A = \langle \boldsymbol{\alpha}, \boldsymbol{\mathcal{T}}, \{\mathbf{v}_\sigma\}_{\sigma \in \Sigma} \rangle$ with n states on $\mathcal{T}_\Sigma$, we say that a function $f : (\Sigma \cup \{[\![,]\!]\})^T \to (\mathbb{R}^n)^T$ simulates A at length T if for all trees $t \in \mathcal{T}_\Sigma$ such that $|\mathrm{str}(t)| \leq T$, $f(\mathrm{str}(t))_i = \mu(\tau_i)$ for all $i \in I_t$ (where the subtrees τ_i are defined in Def. 4.1).*

Furthermore, we say that a family of functions $\mathcal{F}$ simulates WTAs with n states at length T if for any WTA A with n states there exists a function $f \in \mathcal{F}$ that simulates A at length T.

The overall notion of WTA simulation is illustrated in Figure 2. Note that this definition could be modified to encode a tree with the closing brackets instead of the opening ones. In this case, our results would still hold using attention layers with causal masking.

4.2 Results

We are now ready to state our main results for weighted tree automata:

Theorem 3. *Transformers can approximately simulate all WTAs A with n states at length T, up to arbitrary precision $\epsilon > 0$, with embedding dimension $O(n)$, attention width $O(n)$, MLP width $O(n^3)$ and $O(1)$ attention heads*[1]*. Simulation over arbitrary trees can be done with depth $O(T)$ and simulation over balanced trees (trees whose depth is of order $\log(T)$) with depth $O(\log(T))$.*

The construction used in the the proof (which can be found in Appendix C) revolves around two main

[1]The big O notation does not hide any large constant here: the depth is exactly $1 + \mathrm{depth(t)}$, the embedding dimension and the attention width are $n + 4 + p$ (where p is the size of the positional embedding) and the MLP width is $\frac{1}{2}(2n + 1)(2n + 2)$.

ideas: (i) leveraging the attention mechanism to have each node attend to the positions corresponding to its left and right subtrees and (ii) using the feed-forward layers to approximate the bilinear mapping $(\mu(t_1), \mu(t_2)) \mapsto \mu((t_1, t_2)) = \mathcal{T} \times_1 \mu(t_1) \times_2 \mu(t_2)$. In the intial embedding, each leaf position is initialized to the corresponding leaf state $\mathbf{v}_\sigma$. At each layer of computation, each position computes the output of the multilinear map applied to the embeddings of its respective left and right subtrees. Thus, after one layer the state of all the subtrees of depth up to 2 have been computed (leafs and subtrees of the form (σ_1, σ_2)). Similarly, after the ℓth layer, the transformer will have simulated all the states corresponding to subtrees of depth up to $\ell + 1$. Thus after as many layers as the depth of the input tree, all states have been computed.

We will conclude with a few interesting observations. First, note that for all balanced tree, all states will have been computed after $\mathcal{O}(\log T)$ layers of computation. Thus, if we only consider well-balanced trees, i.e. trees whose depth is in $\mathcal{O}(\log T)$, then our construction constitutes a shortcut. However, in the case of the most extremely unbalanced tree (which is a comb, i.e. a tree of the form $(\sigma_1, (\sigma_2, (\sigma_3, (\cdots)))))$, it will take $\mathcal{O}(T)$ layers to compute all states. At the same time, in this most extreme case, a tree is nothing else than a string/sequence, and the computation of the WTA on this tree can be as well be carried out by a WFA, which could be simulated by a transformer with $\mathcal{O}(\log T)$ layers from our previous results for WFA. This raises the question of whether there exist a construction interpolating between the WFA and WTA constructions proposed in this paper which could simulate WTA with $\mathcal{O}(\log T)$ layers for *arbitrary* trees. Still, this result shows that transformers can indeed learn shortcuts to WTAs when restricting the set of inputs to balanced trees.

Second, since WTAs subsume classical finite state tree automata, an important corollary of our result is that transformers can also simulate (non-weighted) tree automata. Classical tree automata can be defined as WTA with weights in the Boolean semi-ring. Intuitively, Boolean computations can, in some sense, be simulated with real weights by interpreting any non-zero value as true and any zero value as false, thus WTA can simulate classical tree automata. Since we just showed that WTA can in turn be simulated by transformers, we have the following corollary (whose proof can be found in appendix).

Corollary 1. *Transformers can approximately simulate all tree automata $\mathcal{A}$ with n states at length T, up to arbitrary precision $\epsilon > 0$, with the same hyperparameters and depth as for WTA given in Theorem 3.*

5 EXPERIMENTS

In this section, we investigate if logarithmic shortcut solutions can be found using gradient descent based learning. We train transformer models on sequence to sequence simulation tasks, where, for a given input sequence, the transformer must produce as output the corresponding sequence of states. We then study how varying certain parameters impacts model performance and compare this with results predicted by theory. We find that transformer models are indeed capable of approximately simulating WFAs and that the number of layers of the model has an important effect on performance, as predicted by theory.

5.1 Can logarithmic solutions be found?

We first investigate if, under ideal supervision, such solutions can even be found in the first place. We evaluate models on target WFAs taken from the Pautomac dataset (Verwer et al., 2014). We use the target automata to generate sequences of states with length $T = 64$.

In Table 1, we report the minimum number of layers necessary for a transformer to reach a test (mean squared) error below $\epsilon = 10^{-3}$. We report our findings for $L \in \{2, 4, 6, 8, 10\}$. If no model achieving ϵ was found, we indicate this using a dash. This table also includes all practical information about the target automaton such as its number of states and the size of its alphabet.

We see that for more than half of the considered automata, we are indeed able to find a solution which satisfies our error criterion. However, the minimum values attained are not always consistent with the logarithmic threshold for shortcuts we propose in theory (since $T = 64$, we would expect $L = 6$ to be the threshold). Furthermore, for certain automata, the maximum number of layers considered does not seem to be sufficient to achieve the target error. This behavior is not consistent with the measures of complexity of the problem given in the table (*i.e.*, number of states, alphabet size, symbol sparsity). In future work, it would be interesting to see if a more thorough hyperparameter search could lead to finding shortcut solutions for these automata as well.

5.2 Do solutions scale as theory suggests?

Next, we investigate how such solutions scale as key parameters in our construction increase. We study how the number of layers as well as the size of the embedding dimension influence the performance of transformer models trained to simulate synthetic WFAs.

Pautomac nb	4	12	14	20	30	31	33	38	39	45
num states	12	12	15	11	9	12	13	14	6	14
alphabet size	4	13	12	18	10	5	15	10	14	19
type	PFA	PFA	HMM	HMM	PFA	PFA	HMM	HMM	PFA	HMM
symbol sparsity	0.4375	0.3526	0.4944	0.3939	0.6555	0.3833	0.5949	0.7857	0.4167	0.8008
nb layers for ϵ	8	6	2	6	-	8	-	2	-	-

Table 1: Minimum number of layers to reach error $< \epsilon = 10^{3}$

For the experiments concerning the number of layers, the data is generated using a WFA with 2 states which counts the number of 0s in a binary string with $\Sigma =$ f$0, 1$g. Figure 3 shows how the mean squared error (MSE) varies as we increase the number of layers. We consider T 2 f$16, 32, 64$g, and each curve represents a sequence length for which we run the experiment. The dotted vertical lines represent the theoretical value for shortcuts to be found, *i.e.* $\log_2(T)$.

For all sequence lengths, the error curves display a pronounced elbow. We see that, at first, increasing the number of layers has a notable effect on decreasing the MSE. However, after a certain threshold, the MSE seems to stabilize and adding more layers has negligible effect. It is also interesting to note how close this stabilization point is to the number of layers necessary for shortcut solutions in practice.

For the experiments on the embedding dimension, we generate data using a WFA which counts k distinct symbols in sequences over some alphabet Σ. One can define such a WFA using $k+1$ states, where the first k components of the state correspond to the current counts for each of the k symbols, and the last component is constant and equal to one. As an example, consider $\Sigma =$ fa, b, cg, we could define a 2-counting automata which counts a and b. For the word $w = aaabb$ such an automaton would return the state $(3, 2, 1)$, where the two first dimensions count a and b respectively. More details about the considered automata can be found in the appendix.

For the purpose of this experiment, we consider $\Sigma =$ f$0, 1, 2, ..., 9$g and k 2 f$2, 4, 6, 8$g, where we count the k first characters in the alphabet (in numerical order). For simplicity, we fix both the embedding dimension and the hidden size of the model to the same value.

The results are presented in Figure 4, where we also notice that the curves show a pronounced elbow shape. Interestingly, we note that the stabilisation of the error does not seem to follow the value n^2 predicted by the theory as closely as for the number of layers. This may be due to the training procedure considered. For instance, training on a bigger dataset, for more epochs or using more varied hyperparameters, the results may

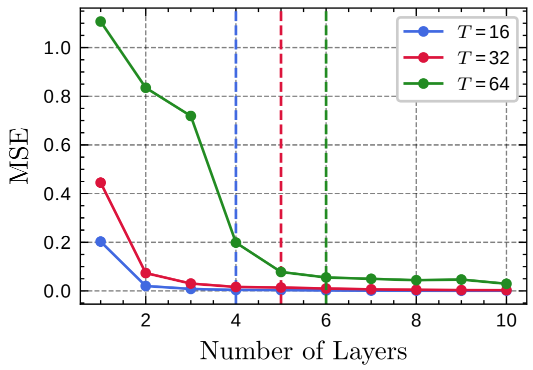


Figure 3: Average MSE vs. number of layers: For all considered sequence lengths, adding layers has an notable effect on the MSE at first, however past a certain point, the improvement is negligible. This stabilization is consistent with our theoretical results (shown as dotted lines).

match more closely the trend predicted by the theory.

6 RELATED WORKS

Formal languages and Neural Networks There has been extensive study of the relationship between formal languages and neural networks. Many studies investigate the empirical performances of sequential models on formal language recognition tasks. Delétang et al. (2022); Bhattamishra et al. (2020) show where transformers and RNN variants lie on the Chomsky hierarchy by training these models on simple language recognition tasks. Ebrahimi et al. (2020) study the empirical performance of transformers on Dyck languages, which describe balanced strings of brackets. Merrill et al. (2020) give an alternative hierarchy using space complexity and rational recurrence. There are also many theoretical results concerning connections between formal languages and neural models. Chiang and Cholak (2022); Daniely and Malach (2020) show constructions of feedforward neural networks and transformers for PARITY (a language of binary strings such that the number of 1s is even) and FIRST (a language of binary strings starting with a 1). Yao et al. (2021),

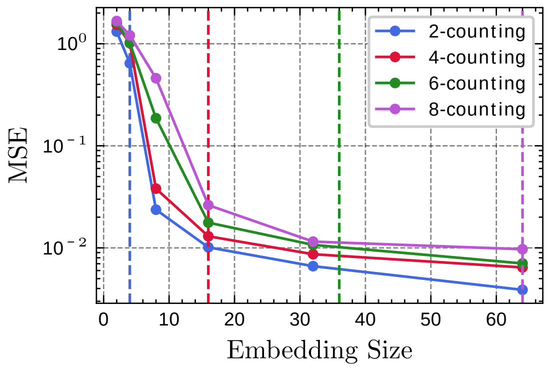


Figure 4: Average MSE (log scale) vs. embedding size: Increasing the embedding size also has a notable effect on the MSE. However the stabilization of the curves does not agree with as closely with our theoretical results (shown as dotted lines).

on the other hand, give constructions of transformers for Dyck languages.

Neural networks and models of computation Work has also been done investigating theoretical connections between classical computer science models and neural networks. The most widely-known result is certainly Chung and Siegelmann (2021), showing that RNNs are Turing-complete. However, this work assumes unbounded computation time and infinite precision and thus such a result is not very applicable in more practical settings. Several work focus on extracting DFAs (Weiss et al., 2018; Giles et al., 1992; Omlin and Giles, 1996; Merrill and Tsilivis, 2022; Muškardin et al., 2022) and WFAs (Weiss et al., 2019; Okudono et al., 2020; Zhang et al., 2021) from recurrent neural networks. A similar, more general approach can be found in a line of work focusing on extraction from black-box models on sequential data (Ayache et al., 2018; Eyraud and Ayache, 2020; Lacroce et al., 2021) and on knowledge distillation from RNNs and transformers (Eyraud et al., 2023). Interestingly, it is possible to formally show that WFA are equivalent to 2-RNNs, a family of RNNs which uses bilinear layers (Li et al., 2022). Moreover, connections have been shown between transformers and Boolean circuits. Merrill et al. (2022); Hao et al. (2022) and Chiang et al. (2023) show that transformers can implement Boolean circuits and show which circuit complexity classes transformers can recognize. On another line of thought, Weiss et al. (2021) show how transformers can realize declarative programs.

Closest to our work is Liu et al. (2022), which first introduced the notion of transformers simulating DFAs. This paper initially prompted us to investigate what other families of automata could be simulated by transformers. Moreover, many of the technical ideas used in Theorem 1 and Theorem 2 are inspired by the proof they present for the logarithmic case.

Lastly, Zhao et al. (2023) is also very relevant to our work. Their work shows that transformers can simulate probabilistic context free grammars (PCFGs). More precisely, they show that transformers can implement the inside-outside algorithm (Baker, 1979), which is a dynamic programming algorithm used to compute the probability of a given sequence under a PCFG. I.e., to compute the sum of the values returned by a given WTA (which defines the PCFG) on all possible derivation trees of the input sequence: a task orthogonal (and more difficult) than the one of simulating a WTA we consider here. Their construction (understandably) requires more parameters than ours ($O(T)$ layers with $O(n)$ attention heads each and embedding size of $O(nT)$).

7 CONCLUSION

In this paper, we theoretically demonstrate that transformers can simulate both WFAs and WTAs. For WFAs, simulation can be achieved using a number of layers logarithmic in the sequence length, whereas for WTAs, the number of layers must be equivalent to the depth of the tree given as input. Our results extend the ones of Liu et al. (2022) showing that transformers can learn shortcuts to models significantly more complex than deterministic finite automata. We verified on simple synthetic experiments that such shortcut solutions to WFAs can indeed be found using gradient based training methods. We hope that our results may shed some light on the success of transformers for sequential reasoning tasks, and give practical considerations in terms of depth and width of such models for given tasks.

There are many theoretical and empirical directions in which our work could be extended. A first question is whether such shortcut solutions can be found *in the wild*. Do transformer models trained on downstream tasks implement exactly or approximately the algorithmic reasoning capabilities of WFAs or WTAs? It may be interesting to analyze to what extent transformers natively implement the algorithmic reasoning used in our constructions. Concerning theoretical results, we only provide upper bounds to the simulation capacities of the considered models. One could extend the results presented in this paper by deriving lower bounds for either WFA or WTA. We posit that there should exist languages for which these bounds are tight. Another interesting question would be to analyze how such solutions scale with sample complexity and optimization schemes. Our results are about the existence of shortcut solutions, but the question of learnability for such solutions remains open. Empirically or theoretically, it would be interesting to show how the quantity of data, optimization procedure, or various aspects of the target structure can affect the quality of found shortcuts. In the same line of thought, an analysis of the training dynamics may be interesting.

Acknowledgements

This research is supported by the Canadian Institute for Advanced Research (CIFAR AI chair program) and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Jeffrey L Elman. Finding Structure in Time. *Cognitive science*, 14(2):179–211, 1990.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata.* Springer Science & Business Media, 2009.

Mehryar Mohri. *Weighted Automata Algorithms.* Springer-Verlag, 2009.

Arto Salomaa and Matti Soittola. *Automata-Theoretic Aspects of Formal Power Series.* Texts and Monographs in Computer Science. Springer, 1978. ISBN 978-0-387-90282-1. doi: 10.1007/978-1-4612-6264-0. URL `https://doi.org/10.1007/978-1-4612-6264-0`.

M Magidor and G Moran. Probabilistic tree automata and context free languages. *Israel Journal of Mathematics*, 8(4):340–348, 1970.

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2008.

Yang Gao, Oscar Beijbom, Ning Zhang, and Trevor Darrell. Compact bilinear pooling. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 317–326, 2016.

Yanghao Li, Naiyan Wang, Jiaying shortcuts, and Xiaodi Hou. Factorized bilinear models for image recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2079–2087, 2017.

Tsung-Yu Lin, Aruni RoyChowdhury, and Subhransu Maji. Bilinear cnn models for fine-grained visual recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 1449–1457, 2015.

Jason Ross Brown, Yiren Zhao, Ilia Shumailov, and Robert D Mullins. Wide attention is the way forward for transformers. *arXiv preprint arXiv:2210.00640*, 2022.

Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on learning theory*, pages 907–940. PMLR, 2016.

Nadav Cohen, Or Sharir, and Amnon Shashua. On the expressive power of deep learning: A tensor analysis. In *Conference on learning theory*, pages 698–728. PMLR, 2016.

Guy E Blelloch. Prefix sums and their applications. 1990.

Sicco Verwer, Rémi Eyraud, and Colin de la Higuera. Pautomac: a probabilistic automata and hidden markov models learning competition. *Mach. Learn.*, 96(1-2):129–154, 2014. doi: 10.1007/s10994-013-5409-9. URL `https://doi.org/10.1007/s10994-013-5409-9`.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, et al. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How can self-attention networks recognize dyck-n languages? *arXiv preprint arXiv:2010.04303*, 2020.

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A Smith, and Eran Yahav. A formal hierarchy of rnn architectures. *arXiv preprint arXiv:2004.08500*, 2020.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. *arXiv preprint arXiv:2202.12172*, 2022.

Amit Daniely and Eran Malach. Learning parities with neural networks. *Advances in Neural Information Processing Systems*, 33:20356–20365, 2020.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. *arXiv preprint arXiv:2105.11115*, 2021.

Stephen Chung and Hava Siegelmann. Turing completeness of bounded-precision recurrent neural networks. *Advances in Neural Information Processing Systems*, 34:28431–28441, 2021.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5244–5253. PMLR, 2018. URL `http://proceedings.mlr.press/v80/weiss18a.html`.

Clyde Lee Giles, Clifford B. Miller, Dong Chen, Hsing-Hen Chen, Guo-Zheng Sun, and Yee-Chun Lee. Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks. *Neural Comput.*, 4(3):393–405, 1992. doi: 10.1162/neco.1992.4.3.393. URL `https://doi.org/10.1162/neco.1992.4.3.393`.

Christian W. Omlin and Clyde Lee Giles. Constructing Deterministic Finite-State Automata in Recurrent Neural Networks. *J. ACM*, 43(6):937–972, 1996. doi: 10.1145/235809.235811.

William Merrill and Nikolaos Tsilivis. Extracting Finite Automata from RNNs Using State Merging, 2022.

Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, and Martin Tappler. Learning finite state models fromrecurrent neural networks. In *Integrated Formal Methods: 17th International Conference, IFM 2022, Lugano, Switzerland, June 7–10, 2022, Proceedings*, page 229–248, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-07726-5. doi: 10.1007/978-3-031-07727-2_13. URL `https://doi.org/10.1007/978-3-031-07727-2_13`.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning Deterministic Weighted Automata with Queries and Counterexamples. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8558–8569, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/d3f93e7766e8e1b7ef66dfdd9a8be93b-Abstract.html`.

Takamasa Okudono, Masaki Waga, Taro Sekiyama, and Ichiro Hasuo. Weighted Automata Extraction from Recurrent Neural Networks via Regression on State Spaces. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5306–5314. AAAI Press, 2020. URL `https://aaai.org/ojs/index.php/AAAI/article/view/5977`.

Xiyue Zhang, Xiaoning Du, Xiaofei Xie, Lei Ma, Yang Liu, and Meng Sun. Decision-Guided Weighted Automata Extraction from Recurrent Neural Networks. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 11699–11707. AAAI Press, 2021. URL `https://ojs.aaai.org/index.php/AAAI/article/view/17391`.

Stéphane Ayache, Rémi Eyraud, and Noé Goudian. Explaining Black Boxes on Sequential Data Using Weighted Automata. In *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, pages 81–103. PMLR, 2018. URL `http://proceedings.mlr.press/v93/ayache19a.html`.

Rémi Eyraud and Stéphane Ayache. Distillation of Weighted Automata from Recurrent Neural Networks Using a Spectral Approach. *CoRR*, abs/2009.13101, 2020. URL `https://arxiv.org/abs/2009.13101`.

Clara Lacroce, Prakash Panangaden, and Guillaume Rabusseau. Extracting Weighted Automata for Approximate Minimization in Language Modelling. In Jane Chandlee, Rémi Eyraud, Jeff Heinz, Adam Jardine, and Menno van Zaanen, editors, *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 92–112. PMLR, 23–27 Aug 2021. URL https://proceedings.mlr.press/v153/lacroce21a.html.

Rémi Eyraud, Dakotah Lambert, Badr Tahri Joutei, Aidar Gaffarov, Mathias Cabanne, Jeffrey Heinz, and Chihiro Shibata. Taysir competition: Transformer+RNN: Algorithms to yield simple and interpretable representations. In François Coste, Faissal Ouardi, and Guillaume Rabusseau, editors, *Proceedings of 16th edition of the International Conference on Grammatical Inference*, volume 217 of *Proceedings of Machine Learning Research*, pages 275–290. PMLR, 10–13 Jul 2023. URL https://proceedings.mlr.press/v217/eyraud23a.html.

Tianyu Li, Doina Precup, and Guillaume Rabusseau. Connecting weighted automata, tensor networks and recurrent neural networks through spectral learning. *Machine Learning*, pages 1–35, 2022.

William Merrill, Ashish Sabharwal, and Noah A Smith. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022.

Yiding Hao, Dana Angluin, and Robert Frank. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*, 10: 800–810, 2022.

David Chiang, Peter Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer encoders. *arXiv preprint arXiv:2301.10743*, 2023.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking Like Transformers. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 11080–11090. PMLR, 2021. URL http://proceedings.mlr.press/v139/weiss21a.html.

Haoyu Zhao, Abhishek Panigrahi, Rong Ge, and Sanjeev Arora. Do transformers parse while predicting the masked word? *arXiv preprint arXiv:2303.08117*, 2023.

James K Baker. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65(S1):S132–S132, 1979.

Enrique Vidal, Franck Thollard, Colin De La Higuera, Francisco Casacuberta, and Rafael C Carrasco. Probabilistic finite-state machines-part i. *IEEE transactions on pattern analysis and machine intelligence*, 27(7):1013–1025, 2005.

Kai Fong Ernest Chong. A closer look at the approximation capabilities of neural networks. *arXiv preprint arXiv:2002.06505*, 2020.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. **Yes**
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. **Yes**
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. **Not Applicable**

2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. **Yes**
   (b) Complete proofs of all theoretical results. **Yes**
   (c) Clear explanations of any assumptions. **Yes**

3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). **No**, but we intend on making the code repository public once the AISTATS decision will be made.
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). **Yes**
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). **Yes**
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). **Yes**

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator if your work uses existing assets. **No**
   (b) The license information of the assets, if applicable. **Not Applicable**
   (c) New assets either in the supplemental material or as a URL, if applicable. **Not Applicable**
   (d) Information about consent from data providers/curators. **Not Applicable**
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. **Not Applicable**

5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. **Not Applicable**
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. **Not Applicable**
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. **Not Applicable**

Simulating Weighted Automata over Sequences and Trees with Transformers

Supplementary Material

A Background and Notation

A.1 The Recursive Parallel Scan Algorithm

In this section, we give an in-depth explanatation of the recursive parallel scan algorithm, which is crucial to the construction in the two first theorems in this paper. The recursive parallel scan algorithm or prefix sum algorithm is an algorithm which calculates the running sum of a sequence Blelloch (1990) Given a sequence of numbers $f x_0, x_1, x_2, \dots x_n g$, the method outputs the running sum

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 \quad x_1 \\ &\vdots \\ y_n &= x_0 \quad x_1 \quad \dots \quad x_n. \end{aligned}$$

Where   represents any associative binary operator such as addition, string concatenation or matrix multiplication, for example. In order to compute this running sum, the algorithm uses a divide-and-conquer scheme. The pseudocode for the algorithm is given below.

Algorithm 1: Prefix Sum (Scan) Algorithm

```
Data: Input sequence $f x_0, x_1, x_2, \dots x_n g$
for $i$    0 to floor($\log_2 n$) do
    for $j$    0 to $n$    1 do
        if $j < 2^i$ then
            $x_j^{i+1}$    $x_j^i$
        else
            $x_j^{i+1}$    $x_j^i +$    $x_{j \quad 2^i}^i$
```

Here, we take x_j^i to be the jth element in the sequence at timestep i.

The key idea of this algorithm is to recursively compute smaller partial sums at each step and combine them in the step after. The first for loop iterates through the powers of two which determine how far apart each precomputed sum is to the next (indexed by i). The second for loop recursively combines all partial sums that are powers of 2^i apart. By doing this a total of $\log n$ times, we are able to compute the running sum. Figure **??** gives an example of this procedure for a sequence of length 8.

Considering there are $O(n)$ sums to compute at every step, this leads to a total runtime of $O(n \log n)$. In terms of space complexity, such an algorithm can be executed in place, by storing the results of each intermediate computation in the input array, thus leading to a space complexity of $O(1)$.

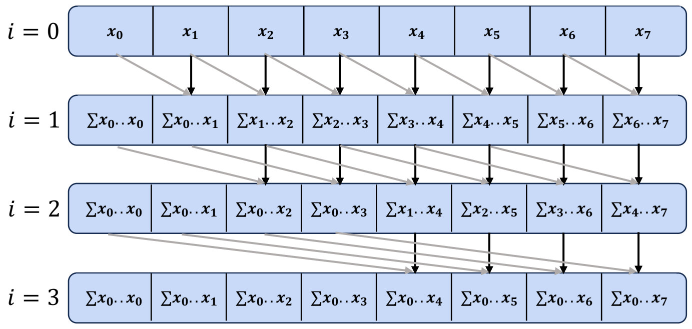

$i = 0$
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
$i = 1$
$\Sigma x_0..x_0$ $\Sigma x_0..x_1$ $\Sigma x_1..x_2$ $\Sigma x_2..x_3$ $\Sigma x_3..x_4$ $\Sigma x_4..x_5$ $\Sigma x_5..x_6$ $\Sigma x_6..x_7$
$i = 2$
$\Sigma x_0..x_0$ $\Sigma x_0..x_1$ $\Sigma x_0..x_2$ $\Sigma x_0..x_3$ $\Sigma x_1..x_4$ $\Sigma x_2..x_5$ $\Sigma x_3..x_6$ $\Sigma x_4..x_7$
$i = 3$
$\Sigma x_0..x_0$ $\Sigma x_0..x_1$ $\Sigma x_0..x_2$ $\Sigma x_0..x_3$ $\Sigma x_0..x_4$ $\Sigma x_0..x_5$ $\Sigma x_0..x_6$ $\Sigma x_0..x_7$

Figure 5: Illustration of the prefix sum algorithm

A.2 Weighted Finite Automata

In this section, we give a more thorough treatment of WFAs and detail the families/instances of automata considered in the experiments section. We start by recalling the definition of a WFA

Definition A.1. *A* weighted finite automaton *(WFA) of n states over Σ is a tuple $A = \langle \boldsymbol{\alpha}, \{\mathbf{A}^\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\beta}\rangle$, where $\boldsymbol{\alpha}$, $\boldsymbol{\beta} \in \mathbb{R}^n$ are the initial and final weight vectors, respectively, and $\mathbf{A}^\sigma \in \mathbb{R}^{n\times n}$ is the matrix containing the transition weights associated with each symbol $\sigma \in \Sigma$. Every WFA A with real weights realizes a function $f_A : \Sigma^* \to \mathbb{R}$,* i.e. *given a string $x = x_1 \cdots x_t \in \Sigma^*$, it returns $f_A(x) = \boldsymbol{\alpha}^\top \mathbf{A}^{x_1} \cdots \mathbf{A}^{x_t}\boldsymbol{\beta} = \boldsymbol{\alpha}^\top \mathbf{A}^{x}\boldsymbol{\beta}$.*

Hidden Markov models Hidden Markov models (HMMs) are statistical models that compute the probability of seeing a sequence of observable variables subject to some "hidden" variable which follows a Markov process. It is possible to show that HMMs are a special case of WFAs. Recall the definition of a HMM

Definition A.2. *Given a set of states $S = \{1, \ldots, n\}$, and a set of observations $\Sigma = \{1, \ldots, p\}$, a HMM is given by*

- *Transition probabilities $\mathbf{T} \in \mathbb{R}^{n\times n}$, where $\mathbf{T}_{ij} = \mathbb{P}(h_{t+1} = j \mid h_t = i)$;*
- *Observation probabilities $\mathbf{O} \in \mathbb{R}^{p\times n}$, where $\mathbf{O}_{ij} = \mathbb{P}(o_t = i \mid h_t = j)$;*
- *An initial distribution $\boldsymbol{\pi} \in \mathbb{R}^n$, where $\boldsymbol{\pi}_i = \mathbb{P}(h_1 = i)$.*

A HMM defines a probability distribution over all possible sequences of observations. The probability of a given sequence $x_1x_2x_3 \ldots x_k$ is given by

$$\mathbb{P}(x_1x_2x_3 \ldots x_k) = \sum_{i_1} \boldsymbol{\pi}_i \mathbf{O}_{x_1,i_1} \sum_{i_2} \mathbf{T}_{i_1,i_2}\mathbf{O}_{x_2,i_2} \ldots \sum_{i_k} \mathbf{T}_{i_{k-1},ik}\mathbf{O}_{x_k,i_k}.$$

Here, notice the similarities between this computation and that of a WFA. This lets us rewrite the HMM computation in terms of the definition of a WFA:

$$\begin{aligned}
\boldsymbol{\alpha} &= \boldsymbol{\pi} \\
\mathbf{A}^x &= \operatorname{diag}(\mathbf{O}_{x,1}, \ldots, \mathbf{O}_{x,n})\mathbf{T}, \ \forall x \in \Sigma \\
\boldsymbol{\beta} &= \mathbf{1} \in \mathbb{R}^n.
\end{aligned}$$

Where **1** is taken to be the vector full of ones. Thus, we have that

$$\mathbb{P}(x_1x_2x_3 \ldots x_k) = \boldsymbol{\alpha}^\top \mathbf{A}^{x_1}\mathbf{A}^{x_2}\mathbf{A}^{x_3} \ldots \mathbf{A}^{x_k}\boldsymbol{\beta}.$$

Probabilistic Finite Automata Probabilistic finite automata (PFA) Vidal et al. (2005) are another subcategory of WFAs which compute probabilities. Here, we do not assume the matrices are row-stochastic as is the case for HMMs.

Definition A.3. *A PFA is a tuple $A = \langle Q_A, \Sigma, \delta_A, I_A, F_A, P_A\rangle$, where:*

- *Q_A is a finite set of states;*
- *Σ is the alphabet;*
- *$\delta_A \quad Q \quad \Sigma \quad Q$;*
- *$I_A : Q_A \ ! \ \mathbb{R}^+$;*
- *$P_A : \delta_A \ ! \ \mathbb{R}^+$;*
- *$F_A : Q_A \ ! \ \mathbb{R}^+$.*

I_A, P_A and F_A are functions such that:

$$\sum_{q 2 Q} I_A(q) = 1$$

and

$$8q \ 2 \ Q_A, \ F_A(q) + \sum_{\sigma 2 \quad , q^0 2 Q_A} P_A(q, \sigma, q^0) = 1.$$

Assuming we have $Q_A = f0, \ldots, jQj \quad 1g$, we can relate PFAs to WFAs in the following way:

$$\begin{aligned} \boldsymbol{\alpha}_i &= I_A(i), \ 8i \ 2 \ Q_A \\ \mathbf{A}^{\sigma}_{ij} &= P_A(i, \sigma, j) \ 8i, j \ 2 \ Q_A \quad Q_A \\ \boldsymbol{\beta}_i &= F_A(i), \ 8i \ 2 \ Q_A. \end{aligned}$$

Counting WFA The WFA which counts the number of 0s considered in the Section 5 is defined with $n = 2$ and $\Sigma = f0, 1g$. The parameters of this automaton are

$$\boldsymbol{\alpha} = \begin{pmatrix} 0 & 1 \end{pmatrix}^{>}, \ \mathbf{A}^0 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \ \mathbf{A}^1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \boldsymbol{\beta} = \begin{pmatrix} 1 & 0 \end{pmatrix}^{>}.$$

Observe that the matrix $\mathbf{A}^0$ has the following property

$$\left(\mathbf{A}^0\right)^n = \begin{pmatrix} 1 & 0 \\ n & 1 \end{pmatrix},$$

The matrix $\mathbf{A}^1$ on the other hand is the identity and leaves the count alone. Thus for some word $w \ 2 \ \Sigma$, the state at the end of the computation would be

$$\boldsymbol{\alpha}^\top \mathbf{A}^w = \begin{pmatrix} jwj_0 & 1 \end{pmatrix}.$$

k-Counting WFA The k-counting WFA is a generalization of the automata presented in the previous paragraph. For the purpose of this work, we assume that we always count the k first characters of Σ and that $\Sigma = f0, 1, \ldots N \quad 1g$, where $N = j\Sigma j$. Such an automaton needs $n = k + 1$ states to execute such a computation. The parameters of this automaton are:

$$\begin{aligned} \boldsymbol{\alpha} &= \begin{pmatrix} 1 & 0 & \ldots & 0 \end{pmatrix} \ 2 \ \mathbb{R}^{k+1} \\ \mathbf{A}^i &= \mathbf{I}_{k+1} + \mathbf{e}_{k+1}\mathbf{e}_i^{>}, \ 8i < k \\ \mathbf{A}^i &= \mathbf{I}_{k+1}, \ 8i \quad k, \end{aligned}$$

with $i \ 2 \ \Sigma$. The value of $\boldsymbol{\beta}$ is not important here as we are solely interested in the state. One could choose a terminal vector with 1s at specific positions to count the sum of certain symbols or even use 1s and -1s to verify if two symbols appear the same number of times.

B Simulating Weighted Finite Automata

B.1 Proof of Theorem 1

First, we recall our first theorem.

Theorem 1. *Transformers using bilinear layers and hard attention can exactly simulate all WFAs A at length T, with depth $\log T$, embedding dimension $2n^2+2$, attention width $2n^2+2$ and MLP width $2n^2$, where n is the number of states of A.*

Before diving into the details of our construction, we provide a high-level intuition of the proof. Similarly to (Liu et al., 2022), the key idea of this proof is using the recursive parallel scan algorithm to compute the composition of all transition maps efficiently. To implement this algorithm using a transformer, we store two copies of each symbol's transition map in the embeddings and use the attention mechanism to "shift" one of them by a power of two.

Then we can use the MLP to calculate the matrix product between both transition maps to obtain the next set of values in the trajectory. By iterating this process for each of the $\log T$ layers, we are able to obtain every ordered combination of transition maps as it would be the case with the recursive parallel scan algorithm.

Proof. We prove our result by construction. After recalling the main assumptions, we define each element in the construction.

Important assumptions We list our main assumptions.

- For simplicity, we will only consider cases where the sequence length T is a power of 2. Using padding, we can extend the construction to arbitrary T.
- We pad the input sequence with an extra T tokens whose embedding are vectorized identities. This extra space will be used as a buffer to store the "shifted" version of the transition maps. The positions are indexed as $T+1, \dots, 0, 1, \dots, T$. This could equally be achieved using a more complicated MLP.
- Similarly to (Liu et al., 2022), our construction does not use any residual connections.
- This construction uses hard (or saturated) attention in the self attention block as well as a unique bilinear layer as the MLP block.
- We assume access to positional encodings at each layer. This could easily be implemented using either a third attention head (only two are used here) or using residual connections.

Please note that these assumptions are only for ease of exposition. It would be possible, but more complicated, to give a proof without them.

Embeddings For the embeddings, we will use an embedding dimension of $d = 2n^2+2$, where n is the number of states in the WFA. For a given symbol σ_t, t 2 $[T]$, we define the embedding vector as:

$$\mathbf{x}_t = \begin{pmatrix}\mathrm{vec}(\mathbf{A}^{\sigma_t}) & \mathrm{vec}(\mathbf{A}^{\sigma_t}) & P_1(t) & P_2(t)\end{pmatrix} = \begin{pmatrix}\mathbf{x}_L & \mathbf{x}_R & \mathbf{x}_{P_1} & \mathbf{x}_{P_2}\end{pmatrix},$$

with positional embeddings $P_1(t) = \cos\frac{\pi t}{T}$ and $P_2(t) = \sin\frac{\pi t}{T}$. For t 0 we have $\mathbf{x}_t = \begin{pmatrix}\mathrm{vec}(\mathbf{I}) & \mathrm{vec}(\mathbf{I}) & P_1(t) & P_2(t)\end{pmatrix}$. We refer to the dimensions associated to the first vectorized transition map as the left dimensions (indexed $\mathbf{x}_L$) and similarly the dimensions associated to the second vectorized transition map as the right dimensions (indexed $\mathbf{x}_R$). We also use indices $\mathbf{x}_{P_1}$ and $\mathbf{x}_{P_2}$ to refer to the positions associated with each positional encoding.

Note that the positional embeddings are defined for all values of t 2 f $T+1, \dots, 0, 1, \dots, T$g, meaning that they are also defined in the extra T identity-padded dimensions. This is crucial in order to implement the shifting mechanism explained in the following step.

Attention mechanism The attention mechanism used in this construction leverages the fact that transformers process tokens in parallel to implement the recursive parallel scan algorithm. We now detail its parameters and give some intuition as to their use in the construction.

Every self-attention block in this construction has a total of $h = 2$ heads. We index the heads using superscripts (L) and (R). This simplifies the notation as the left and right heads process the left and right parts of the embedding respectively. The construction uses $L = \log_2(T)$ layers, where a layer is taken to be the composition $f_{\mathsf{mlp}} \circ f_{\mathsf{attn}}$.

For all l layers with $1 \leq l \leq L$ let

$$\begin{aligned}\mathbf{W}_Q^{(L)} = \mathbf{W}_Q^{(R)} = \mathbf{W}_K^{(R)} &= \begin{pmatrix}\mathbf{0}_{n^2 \times 2} & \mathbf{0}_{n^2 \times 2} & \mathbf{I}_2\end{pmatrix}^\top \\ \mathbf{W}_K^{(L)} &= \begin{pmatrix}\mathbf{0}_{n^2 \times 2} & \mathbf{0}_{n^2 \times 2} & \boldsymbol{\rho}_\theta\end{pmatrix}^\top\end{aligned}$$

where $\boldsymbol{\rho}_\theta$ is the rotation matrix given by

$$\boldsymbol{\rho}_\theta = \begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix},$$

and θ is defined as

$$\theta = -\frac{\pi 2^{l-1}}{T}.$$

The three first matrices directly select the positional embeddings from the input, and the last matrix selects the positional embeddings and rotates them according to the value of l. In essence, this rotation is what creates the power of two shifting necessary for the prefix sum algorithm.

Finally, we set

$$\begin{aligned}\mathbf{W}_V^{(L)} &= \begin{pmatrix}\mathbf{I}_{n^2} & \mathbf{0}_{n^2} & \mathbf{0}_{n^2 \times 2} \\ \mathbf{0}_{n^2} & \mathbf{0}_{n^2} & \mathbf{0}_{n^2 \times 2} \\ \mathbf{0}_{n^2} & \mathbf{0}_{n^2} & \mathbf{0}_{n^2 \times 2}\end{pmatrix} \\ \mathbf{W}_V^{(R)} &= \begin{pmatrix}\mathbf{0}_{n^2} & \mathbf{0}_{n^2} & \mathbf{0}_{n^2 \times 2} \\ \mathbf{0}_{n^2} & \mathbf{I}_{n^2} & \mathbf{0}_{n^2 \times 2} \\ \mathbf{0}_{n^2} & \mathbf{0}_{n^2} & \mathbf{0}_{n^2 \times 2}\end{pmatrix}\end{aligned}$$

These can be thought of as selector matrices. They select the submatrices corresponding to the left and right embedding sequence blocks.

Feedforward network The feedforward network in this construction is the same for all L layers. It uses a bilinear layer to compute the vectorized matrix product between transition maps. Note that the bilinear layer is applied batch-wise. This means that the bilinear layer is applied independently to each row of the input matrix.

We define two linear transformations

$$\begin{aligned}\mathbf{W}_{\mathsf{sel}}^{(L)} &= \begin{pmatrix}\mathbf{I}_{n^2} & \mathbf{0}_{n^2} & \mathbf{0}_{n^2 \times 2}\end{pmatrix}^\top \\ \mathbf{W}_{\mathsf{sel}}^{(R)} &= \begin{pmatrix}\mathbf{0}_{n^2} & \mathbf{I}_{n^2} & \mathbf{0}_{n^2 \times 2}\end{pmatrix}^\top .\end{aligned}$$

These matrices select the left and right embeddings. Next, using Lemma 1, we define the weight tensor $\boldsymbol{\mathcal{T}}$. This tensor computes the matrix product between the left and right embeddings of the transition maps previously selected by the linear transformations

Finally, we need one last linear layer to copy the result of the compositions into both the left and the right embeddings. To do so, we define

$$\mathbf{W}_{\mathsf{out}} = \begin{pmatrix}\mathbf{I}_{n^2} & \mathbf{I}_{n^2} & \mathbf{0}_{n^2 \times 2}\end{pmatrix}^\top .$$

Using this construction, we are able to recover, at the final layer the transition maps $\mathrm{vec}(\mathbf{A}^{\sigma}_{1}), \mathrm{vec}(\mathbf{A}^{\sigma_1\sigma_2}), ..., \mathrm{vec}(\mathbf{A}^{\sigma_1\cdots\sigma_T})$ in the right embedding dimension of the output. In order to transform these transition maps into states, we need only apply a transformation $\mathbf{W}_{\text{readout}}$ that maps each transition map to its corresponding state such that $\text{vec}(\mathbf{A}^{\sigma_1 \cdots \sigma_t}) \mapsto \boldsymbol{\alpha}^\top \mathbf{A}^{\sigma_1 \cdots \sigma_t}$.

□

Lemma 1. *Let $\mathbf{A} \in \mathbb{R}^{n \times n}, \mathbf{B} \in \mathbb{R}^{n \times n}$ be two square matrices and let vec$(\cdot)$ denote their vectorization. Then there exists a tensor $\mathcal{T} \in \mathbb{R}^{n \times n \times n}$ that computes the vectorized matrix product such that*

$$\mathcal{T} \times_1 vec(\mathbf{A}) \times_2 vec(\mathbf{B}) = vec(\mathbf{AB})$$

Proof. Let $\mathbf{C} = \mathbf{AB}$. Using the component wise definition of matrix product, we have

$$\mathbf{C}_{ij} = \sum_{k=1}^{n} \mathbf{A}_{ik}\mathbf{B}_{kj}$$

Summing over all possible values in $\mathbf{A}$ and $\mathbf{B}$, we can write this as

$$= \sum_{i',j',i'',j''} \mathbb{1}\{i' = i, j' = i'', j'' = j\}\mathbf{A}_{i'j'}\mathbf{B}_{i''j''}$$

Which lets us define the tensor $\mathcal{T} \in \mathbb{R}^{n^2 \times n^2 \times n^2}$ componentwise as

$$\mathcal{T}(i', j', j'') = \mathbb{1}\{i' = i, j' = i'', j'' = j\}.$$

This operation can be thought of as taking all multiplicative interactions between the two transition matrices and summing together those that are related through the matrix product. □

B.2 Proof of Theorem 2

First, let us recall the theorem.

Theorem 2. *Transformers can approximately simulate all WFAs $\mathcal{A}$ at length T, up to arbitrary precision $\epsilon > 0$, with depth $\log T$, embedding dimension $2n^2 + 2$, attention width $2n^2 + 2$ and MLP width $2n^4 + 3n^2 + 1$, where n is the number of states of $\mathcal{A}$.*

The proof of this construction is very similar to that of Theorem 1, as we use the same key idea of the shifting mechanism. The main difference comes from the fact that we can no longer compute the attention filter exactly and that the MLP is no longer able to compute the composition of transition maps exactly. The most important aspect of this proof is understanding how the error propagates through the construction and choosing sufficient error tolerances. Given a certain ϵ, we want to make sure there exists a construction whose total error is no more than said ϵ. For the MLP, we leverage the result from (Chong, 2020) on approximating polynomial functions using neural networks. To state the theorem, we first give some related definitions.

Let $\mathcal{C}(\mathbb{R})$ denote the set of all real valued functions, $\mathcal{P}_{\leq d}(\mathbb{R})$ denote the set of all real polynomials of degree at most d and $\mathcal{P}_{\leq d}(\mathbb{R}^{m_1}, \mathbb{R}^{m_2})$ denote the set of all multivariate polynomials from $\mathbb{R}^{m_1}$ to $\mathbb{R}^{m_2}$.

Theorem 4. *(abridged version of Theorem 3.1 of (Chong, 2020)) Let $d \geq 2$ be an integer, let $f \in \mathcal{P}_{\leq d}(\mathbb{R}^{m_1}, \mathbb{R}^{m_2})$ and let ρ^σ be a two-layer MLP with activation function σ and parameters $\Theta = (\mathbf{W}_1, \mathbf{W}_2)$. If $\sigma \in \mathcal{C}(\mathbb{R}) \setminus \mathcal{P}_{\leq d-1}$, then for every $\epsilon > 0$, there exists some $\Theta \in \{(\mathbf{W}_1, \mathbf{W}_2) \mid \mathbf{W}_1 \in \mathbb{R}^{m_1 \times N}, \mathbf{W}_2 \in \mathbb{R}^{N \times m_2}\}$ with $N = \binom{m_1 + d}{d}$ such that $\|f - \rho^\sigma\|_\infty < \epsilon$.*

Note here that m, n and N do not depend on ϵ. This means that the size of the construction stays constant for all $\epsilon > 0$.

Proof. We prove our result by construction. Given the similarities to the proof of the exact case, we will only detail the sections where the construction differs. Let $\epsilon > 0$ be the error tolerance of the transformer.

Important assumptions The assumptions for this proof are the same as the previous one.

Embeddings We use exactly the same embedding scheme as in the previous proof.

Attention mechanism For the attention mechanism, we use a similar construction as in the previous proof with some key differences we highlight in this section.

For this attention mechanism, we still use $h = 2$ heads indexed with (L) and (R). We also use $L = \log_2(T)$ layers.

For all l layers, with $1 \leq l \leq L$, let:

$$\begin{aligned}
\mathbf{W}_Q^{(L)} = \mathbf{W}_Q^{(R)} = \mathbf{W}_K^{(R)} &= \sqrt{C}\begin{pmatrix}\mathbf{0}_{n^2 \times 2} & \mathbf{0}_{n^2 \times 2} & \mathbf{I}_2\end{pmatrix}^\top \\
\mathbf{W}_K^{(L)} &= \sqrt{C}\begin{pmatrix}\mathbf{0}_{n^2 \times 2} & \mathbf{0}_{n^2 \times 2} & \boldsymbol{\rho}_\theta\end{pmatrix}^\top
\end{aligned}$$

with

$$\begin{aligned}
\boldsymbol{\rho}_\theta &= \begin{pmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{pmatrix}, \\
\theta &= -\frac{\pi 2^{l-1}}{T}
\end{aligned}$$

and

$$C > 0,$$

where C is a saturating constant used to approximate hard attention. This lets us approximate the shifting mechanism to arbitrary precision. Notice that C is a constant and thus has no effect on the number of parameters of our construction.

Feedforward network We do not give an explicit construction for our MLP. Instead, we invoke Theorem 4 leveraging the fact that matrix multiplication is a multivariate polynomial. Let m_1, m_2 be the input and output dimensions respectively and N be the size of the hidden layer.

First, we set $d = 2$ as the multivariate polynomial corresponding to matrix multiplication considers second order interactions at most. Then, we set $m_1 = 2n^2 + 2$ and $m_2 = n^2$. The input dimension is the embedding dimension and the output dimension is the size of the resulting matrix multiplication.

Using Theorem 4, this gives us a hidden size of $N = \binom{n^2+2}{2} = 2n^4 + 3n^2 + 1 \in O(n^4)$. Note that this value does not depend on ϵ.

Finally, to this MLP, we append the following linear layer:

$$\mathbf{W}_{\mathsf{out}} = \begin{pmatrix}\mathbf{I}_{n^2} & \mathbf{I}_{n^2} & \mathbf{0}_{n^2 \times 2}\end{pmatrix}^\top,$$

which lets us copy the result of the composition into both the left and right embeddings.

Error analysis Using an MLP with approximation error ϵ_{mlp} and an attention layer with saturating constant C, we want to derive an expression which recursively bounds the error. Let $\mathbf{A}(\mathbf{X})$ be the attention filter. After the attention block of the first layer, we have

$$(\mathbf{A}(\mathbf{X}) + \mathbf{E}_{\mathsf{attn}})\,\mathbf{X}\mathbf{W}_V = \mathbf{A}(\mathbf{X})\mathbf{X}\mathbf{W}_V + \underbrace{\mathbf{E}_{\mathsf{attn}}\mathbf{X}\mathbf{W}_V}_{=\mathbf{E}'_{\text{attn}}}$$

where $\mathbf{E}_{\mathsf{attn}}$ is a matrix representing the error generated by the soft attention. Row-wise at the output of the MLP block, we have

$$\begin{aligned}
(\mathbf{A}^{\sigma_1} + \mathrm{mat}(\mathbf{e}'_1))(\mathbf{A}^{\sigma_2} + \mathrm{mat}(\mathbf{e}'_2)) + \mathrm{mat}(\mathbf{e}_{i,\mathsf{mlp}}) = \mathbf{A}^{\sigma_1}\mathbf{A}^{\sigma_2} + \\
\underbrace{\mathbf{A}^{\sigma_1}\mathrm{mat}(\mathbf{e}'_2) + \mathrm{mat}(\mathbf{e}'_1)\mathbf{A}^{\sigma_2} + \mathrm{mat}(\mathbf{e}'_1)\mathrm{mat}(\mathbf{e}'_2) + (\mathbf{e}_{i,\mathsf{mlp}})}_{\text{error after 1st layer}}
\end{aligned}$$

where $\mathbf{e}_i^0$ is the ith row vector of $\mathbf{E}_{\text{attn}}^0$, $\mathbf{e}_{i,\text{mlp}}$ is the error generated by the MLP and mat( ) represents matricization. Taking the norm, we get

$$\epsilon_{\text{tot}}^{(1)} = k\mathbf{A}^{\sigma_1}\text{mat}(\mathbf{e}_2^0) + \text{mat}(\mathbf{e}_1^0)\mathbf{A}^{\sigma_2} + \text{mat}(\mathbf{e}_1^0)\text{mat}(\mathbf{e}_2^0) + (\mathbf{e}_{i,\text{mlp}})k_2.$$

Using the triangle inequality and Cauchy-Schwarz, we obtain the following bound

$$\begin{aligned}\epsilon_{\text{tot}}^{(1)} \quad & \epsilon_{\text{attn}}\left(k\mathbf{A}^{\sigma_1}k_2 + k\mathbf{A}^{\sigma_2}k_2\right) + \epsilon_{\text{attn}}^2 + \epsilon_{\text{mlp}} \\ & \epsilon_{\text{attn}}\big(\underbrace{2\max_{\sigma 2} k\mathbf{A}^{\sigma}k_2}_{=M}\big) + \epsilon_{\text{attn}}^2 + \epsilon_{\text{mlp}}\end{aligned}$$

where ϵ_{attn} is the norm of the attention layers error and ϵ_{mlp} is the norm of the error incurred by the MLP. Thus at the second layer attention mechanism, we get

$$\left(\underbrace{\mathbf{A}(\mathbf{X} + \mathbf{E}_{\text{tot}}^{(1)})}_{=\mathbf{A}(\mathbf{X})} + \mathbf{E}_{\text{attn}}\right)\left(\mathbf{X} + \mathbf{E}_{\text{tot}}^{(1)}\right)\mathbf{W}_V = \mathbf{A}(\mathbf{X})\mathbf{X}\mathbf{W}_V + \underbrace{\mathbf{A}(\mathbf{X})\mathbf{E}_{\text{tot}}^{(1)}\mathbf{W}_V + \mathbf{E}_{\text{attn}}\mathbf{X}\mathbf{W}_V + \mathbf{E}_{\text{attn}}\mathbf{E}_{\text{tot}}^{(1)}\mathbf{W}_V}_{\text{error after attention layer 2}}.$$

Here, $\mathbf{E}_{\text{tot}}^{(1)}$ is a matrix such that the norm of each row is $\epsilon_{\text{tot}}^{(1)}$. To simplify the error analysis through the layer 2 MLP, we set

$$\mathbf{E}_{\text{tot}}^{(1)} := \mathbf{E}_{\text{tot}}^{(1)} + \mathbf{A}(\mathbf{X})\mathbf{E}_{\text{tot}}^{(1)}\mathbf{W}_V + \mathbf{E}_{\text{attn}}\mathbf{X}\mathbf{W}_V + \mathbf{E}_{\text{attn}}\mathbf{E}_{\text{tot}}^{(1)}\mathbf{W}_V.$$

Using a similar approach as for the first layer, we get

$$\epsilon_{\text{tot}}^{(2)} \quad \epsilon_{\text{tot}}^{(1)}M + \left(\epsilon_{\text{tot}}^{(1)}\right)^2 + \epsilon_{\text{mlp}}^{(2)}.$$

Thus, we can derive the following recursive expression for $\ell\ 2\ [L]$:

$$\epsilon_{\text{tot}}^{(\ell)} \quad \epsilon_{\text{tot}}^{(\ell\ \ 1)}M + \left(\epsilon_{\text{tot}}^{(\ell\ \ 1)}\right)^2 + \epsilon_{\text{mlp}}^{(\ell)}, \tag{1}$$

where $M = 2\max_{\sigma 2}\ k\mathbf{A}^{\sigma}k_2$, $\epsilon_{\text{mlp}}^{(\ell)}$ is the error incurred by the MLP at layer ℓ and $\epsilon_{\text{tot}}^{(\ell)}$ is the total error at layer ℓ. For any number of layers, the error remains bounded. This means that we can always choose a large enough C and a small enough $\epsilon^{(\ell)}$ such that $\epsilon_{\text{tot}}^{(\ell)} \quad \epsilon$.

Moreover, given that the size of the MLP N does not depend on the target accuracy epsilon to approximate matrix products, and that the saturating constant C does not affect the parameter count of the attention layer, we get this bound on $\epsilon_{\text{tot}}^{(\ell)}$ without ever increasing the number of parameters in our construction.

□

C Proof of Theorem 3

First, let us recall the theorem

Theorem 3. *Transformers can approximately simulate all WTAs A with n states at length T, up to arbitrary precision $\epsilon > 0$, with embedding dimension $O(n)$, attention width $O(n)$, MLP width $O(n^3)$ and $O(1)$ attention heads. Simulation over arbitrary trees can be done with depth $O(T)$ and simulation over balanced trees (trees whose depth is of order $\log(T)$) with depth $O(\log(T))$.*

Let $A = h\alpha\ 2\ \mathbb{R}^n, \mathcal{T}\ 2\ \mathbb{R}^{n\ \ n\ \ n}, f\mathbf{v}_\sigma\ 2\ \mathbb{R}^n g_{\sigma 2}\ \ i$ be a WTA with n states on T . We will construct a transformer f such that, for any tree $t\ 2\ T$, the output after $O(\text{depth}(t))$ layers is such that $f(\text{str}(t))_i = \mu(\tau(i))$ for all $i\ 2\ I_t$, where $T = j\text{str}(t)j$ and the subtree $\tau(i)$ is defined in Def. 4.1. This will show both parts of the theorem as the depth of any tree t is upper bounded by $j\text{str}(t)j = T$.

The construction has two parts. The first part complements the initial embeddings with relevant structural information (such as the depth of each node). This first part has a constant number of layers. The second part of the transformer computes the sub-tree embeddings $\mu(\tau_i)$ iteratively, starting from the deepest sub-trees up to the root. After depth(t) layers of the second part of the transformer, all the subtree embeddings have been computed.

Proof. We prove our result by construction. Each section details a specific part of the considered construction.

Initial embedding The initial embedding for the ith symbol σ_i in str(t) is given by

$$\mathbf{x}_i^{(0)} = (\mathbf{v}_{\sigma_i} \parallel \mathbf{p}_i \parallel m_i \parallel 1)$$

where

- $\parallel$ denotes vector concatenation
- $\mathbf{v}_{\sigma_i}$ is taken from the WTA A if $\sigma_i \in \Sigma$ and $\mathbf{v}_{\sigma_i} = \mathbf{0}$ if $\sigma_i \in \{\llbracket, \rrbracket\}$
- $\mathbf{p}_i$ is the positional encoding
- m_i is a marker to distinguish leaf symbols, opening and closing parenthesis. It is defined by $m_i = 0$ if $\sigma_i \in \Sigma$, $m_i = 1$ if $\sigma_i = \llbracket$ and $m_i = -1$ if $\sigma_i = \rrbracket$.
- the last entry equal to 1 is for convenience (to compactly integrate the bias terms in the attention computations).

Enriched embedding The purpose of the first layers of the transformer is to add structural information related to the tree structure to the initial embedding. We want to obtain the following representation for the ith symbol σ_i in str(t):

$$\mathbf{x}_i^{(1)} = (\mathbf{v}_{\sigma_i} \parallel \mathbf{p}_i \parallel m_i \parallel 1 \parallel d_i \parallel d_i^2)$$

where d_i is the depth of the root of τ_i in t.

Computing the depth at each position $i \in I_t$ can easily be done with two layers. The role of the first layer is simply to add a new component corresponding to shifting the markers m_i one position to the right (as shown in the construction for simulating WFAs, this can be done easily by the attention mechanism). After this first layer, we will have the intermediate embedding

$$\mathbf{x}_i^{(0.5)} = (\mathbf{v}_{\sigma_i} \parallel \mathbf{p}_i \parallel m_i \parallel 1 \parallel m_{i+1}).$$

Now one can easily check that, by construction, $\sum_{j \leq i} m_{j+1} = \text{depth}(\tau_i)$ for all $i \in I_t$ since the summation is equal to the difference between the number of open and closed parenthesis before position i. Hence, the attention mechanism of the second layer can compute the sum $d_i = \sum_{j \leq i} m_{j+1}$ by attending to all previous positions with equal weight. The component d_i^2 can then be approximated to arbitrary precision by an MLP layer with a constant number of neurons (see Theorem 4).

Tree parsing The next layers of the transformer are used to compute the final output of the WTA, $\mu(t)$. The two heads of each attention layer are built such that each position $i \in \text{pos}(\llbracket) := \{i \mid \sigma_i = \llbracket\}$ attends to the corresponding left and right subtrees, respectively. The MLP layers are used to compute the bilinear map $(\mu(\tau), \mu(\tau')) \mapsto \mathcal{T} \times_1 \mu(\tau) \times_2 \mu(\tau')$, which can be approximated to an arbitrary precision (from Theorem 4).

Left head First observe that, by construction, for each $i \in I_t$, the left child of τ_i is τ_{i+1}. We thus let the left head attention weight matrices $\mathbf{W}_Q^{(L)}, \mathbf{W}_K^{(L)} \in \mathbb{R}^{(n+6) \times 2}$ be defined by

$$A_{i,j}^{(L)} = \mathbf{x}_i^\top \mathbf{W}_Q^{(L)} \mathbf{W}_K^{(L)\top} \mathbf{x}_j = \mathbf{p}_i^\top \mathbf{R}^\top \mathbf{p}_j$$

where $\mathbf{R}$ denotes the matrix of a 2D rotation of angle $\frac{\pi}{T}$. One can easily check that $i + 1 = \arg\max_j A_{i,j}$, thus by multiplying the weight matrices by a large enough constant, the attention mechanism will have each position attend to the next one. We then use the value matrix $\mathbf{W}_V^{(L)}$ to select the first part of the corresponding embedding. The output of the left head is thus given by

$$\mathbf{H}^{(L)} = (\mathbf{v}_{\sigma_2}, \mathbf{v}_{\sigma_3}, \dots, \mathbf{v}_{\sigma_T}, \mathbf{v}_{\sigma_T})$$

and satisfies $\mathbf{H}^{(L)}_{:,i} = \mathbf{v}_{\mathsf{left}(i)}$ for all $i \in I_t$, where $\mathrm{left}(i)$ denotes the index of the left child of $\tau(i)$ in $\mathrm{str}(t)$.

Right head As mentioned above, for each position $i \in \mathrm{pos}(\llbracket)$, the left child of τ_i is τ_{i+1}, which is also the next tree in the sequence $\tau_i, \tau_{i+1}, \dots, \tau_T$ whose depth is $\mathrm{depth}(\tau_i) + 1$. Similarly, the right child of τ_i is the second tree in the sequence with depth equal to $\mathrm{depth}(\tau_i) + 1$. Thus, we use the attention mechanism to have position i attend to the closest position $j > i + 1$ satisfying $d_j = d_i + 1$. In order to do so, we let the right head attention weight matrices $\mathbf{W}_Q^{(R)}, \mathbf{W}_K^{(R)} \in \mathbb{R}^{(n+6)\times 7}$ be such that

$$A^{(R)}_{i,j} = \mathbf{x}_i^\top \mathbf{W}_Q^{(R)} \mathbf{W}_K^{(R)\top} \mathbf{x}_j = -\beta(1 - (d_j - d_i))^2 + \mathbf{p}_i^\top \mathbf{R}^\top \mathbf{p}_j + 2\mathbb{1}[j \geq i+2]$$

where $\mathbf{R}$ denotes the matrix of a 2D rotation of angle $\frac{2\pi}{T}$. The first term ensures that $j^* = \arg\max_j A_{i,j}$ is such that $d_{j^*} = d_i + 1$; we choose β to be a constant large enough such that the attention weights $A_{i,j}$ are very small for all j such that $d_j \neq d_i + 1$ (while they are unilaterally 0 for all positions such that $d_j = d_i + 1$). For all positions j such that $d_j = d_i + 1$, the second term enforces that the closest one to position $i + 2$ is chosen. Lastly, the last term enforces that $j \geq i + 2$. It is obtained by using the Fourier approximation of the Heaviside step function which can be constructed using a constant number of positional embeddings and feedforward layers:

$$\mathbb{1}[j \geq i+2] = \frac{1}{2}\left(1 + \sum_{l=0}^{k} \frac{1}{2l+1} \sin\left((2l+1)(i - j - \frac{1.25\pi}{T})\right)\right) \approx \begin{cases} 1 & \text{for } j = i+2, i+3, ..., T \\ 0 & \text{for } j = 1, \dots, i+1 \end{cases}$$

We thus have that, for all $i \in I_t$, the position with largest attention weight, $j^* = \arg\max_j A_{i,j}$, is equal to the second position after i satisfying $d_j = d_{i+1}$, which is the position of the right subtree of $\tau(i)$. By multiplying the weight matrices by a large enough constant, the attention mechanism will thus have each position in I_t attend to the corresponding right subtree. We then use the value matrix $\mathbf{W}_V^{(R)}$ to select the first part of the corresponding embedding. The output of the right head $\mathbf{H}^{(R)}$ thus satisfies $\mathbf{H}^{(R)}_{:,i} = \mathbf{v}_{\mathsf{right}(i)}$ for all $i \in I_t$, where $\mathrm{right}(i)$ denotes the index of the right child of $\tau(i)$ in $\mathrm{str}(t)$.

Computing embeddings of depth 1 subtrees We use a third attention layer to simply copy the input tokens. The MLP is thus fed the input vectors

$$\tilde{\mathbf{x}}_i = (\mathbf{x}_i^{(\mathsf{left})} \,\|\, \mathbf{x}_i^{(\mathsf{right})} \,\|\, \mathbf{v}_{\sigma_i} \,\|\, \mathbf{p}_i \,\|\, m_i \,\|\, 1 \,\|\, d_i \,\|\, d_i^2)$$

in a batch. These inputs are constructed such that, for all positions $i \in I_t$, $\mathbf{x}_i^{(\mathsf{left})} = \mathbf{v}_{\mathsf{left}(i)}$ and $\mathbf{x}_i^{(\mathsf{right})} = \mathbf{v}_{\mathsf{right}(i)}$. We thus choose the weight of the MLP layer such that it approximates the map

$$\tilde{\mathbf{x}}_i \mapsto \left(m_i^2 \cdot (\mathcal{T} \times_1 \mathbf{x}_i^{(\mathsf{left})} \times_2 \mathbf{x}_i^{(\mathsf{right})}) + (1 - m_i^2)\mathbf{v}_{\sigma_i} \,\|\, \mathbf{p}_i \,\|\, m_i \,\|\, 1 \,\|\, d_i \,\|\, d_i^2\right)$$

to an arbitrary precision. Since this map is a 4th order polynomial, this can be done with arbitrary precision with $O(n^4)$ neurons (from Theorem 4).

First, observe that we only care about the indices in I_t, which correspond to leaf symbols or opening parenthesis. We now make the following observations:

- For positions corresponding to leaf symbols (*i.e.* all sub-trees of depth 0), we have $m_i = 0$, thus this first attention layer copies only the corresponding input token without any modifications, which already contains the sub-tree embedding $\mu(\sigma_i)$ since the first embedding layer.
- Similarly, for all positions $i \in I_t$ such that $\mathrm{depth}(\tau(i)) > 1$, this first layer only copies the corresponding input. Indeed, for such positions we necessarily have that (i) $\sigma_i = \llbracket$, thus $m_i = 1$ and $\mathbf{v}_{\sigma_i} = \mathbf{0}$ and (ii) at least one of the children of $\tau(i)$ is not a leaf and has an initial embedding equal to zero, hence $\mathcal{T} \times_1 \mathbf{x}_i^{(\mathsf{left})} \times_2 \mathbf{x}_i^{(\mathsf{right})} = \mathbf{0}$.

- For all positions $i \in I_t$ such that $\mathrm{depth}(\tau(i)) = 1$, we have that both child embeddings $\mathbf{x}_i^{(\text{left})}$ and $\mathbf{x}_i^{(\text{right})}$ have been initialized to the corresponding leaf embeddings $\mu(\tau(\mathrm{left}(i)))$ and $\mu(\tau(\mathrm{right}(i)))$, respectively. Hence, for such positions, the corresponding output tokens are equal to

$$\begin{aligned}\tilde{\mathbf{x}}_i &= \left(\boldsymbol{\mathcal{T}} \times_1 \mu(\tau(\mathrm{left}(i))) \times_2 \mu(\tau(\mathrm{right}(i))) \,\|\, \mathbf{p}_i \,\|\, m_i \,\|\, 1 \,\|\, d_i \,\|\, d_i^2\right) \\ &= \left(\mu(\tau_i) \,\|\, \mathbf{p}_i \,\|\, m_i \,\|\, 1 \,\|\, d_i \,\|\, d_i^2\right).\end{aligned}$$

It follows that after this transformer layer, the output tokens $\mathbf{x}_1^{(2)}, \cdots, \mathbf{x}_T^{(2)}$ are such that, for any $i \in I_t$ such that $\mathrm{depth}(\tau(i)) \leq 1$, we have $x_i^{(2)} = \mu(\tau_i)$.

Computing embeddings of all subtrees One can then check that by constructing the following layers in a similar fashion, the output tokens $\mathbf{x}_1^{(\ell)}, \cdots, \mathbf{x}_T^{(\ell)}$ will be such that $x_i^{(\ell)} = \mu(\tau_i)$ for any $i \in I_t$ satisfying $\mathrm{depth}(\tau(i)) \leq \ell - 1$, which concludes the proof.

□

D Experiments

D.1 Experimental Details

In this section, we give an in-depth description of the training procedure used for the experiments in Section 5.

General considerations For all experiments, we use the PyTorch TransformerEncoder implementation and use a model with 2 attention heads. We train using the AdamW optimizer with a learning rate of 0.001 as well as MSE loss with mean reduction. We use a standard machine learning pipeline with an 80, 10, 10 train/validation/test split and retain the model with best validation MSE for evaluation on the test set. We evaluate our models on a sequence to sequence task, where, for a given input sequence, the transformer must produce as output the corresponding sequence of states. All experiments are conducted on synthetic data with number of examples $N = 10\,000$. For each task, we record the mean and minimum MSE over 10 runs. All experiments were run on the internal compute cluster of our institution.

Experiments with Pautomac For the experiments using the automata from the Pautomac Verwer et al. (2014) dataset, we consider only hidden Markov models (HMMs) and probabilistic finite automata (PFA), as deterministic probabilistic finite automata (DPFAs) are very close to DFAs and are more in the scope of the results of (Liu et al., 2022). We also consider only automata with a number of states inferior to 20 to keep the size of the required transformers small and use a hidden layer size/embedding size of 64 for all experiments. We use a linear layer followed by softmax at the output for readout, as the task at hand implies computing probability distributions. For all experiments, we use synthetic data sampled uniformly from the automata's support with sequence length $T = 64$.

Experiments with counting WFA For the experiments using the WFA which counts 0s, we use an embedding size and a hidden layer size of 16 and use a linear layer for readout at output. Note that here we **do not** append a softmax layer to the linear layer. We consider sequence lengths $T \in \{16, 32, 64\}$ and number of layers $L \in \{1, 2, \dots, 10\}$. The synthetic data is generated using the following procedure:

For each $t \in [T]$

- Sample a sequence x uniformly from $\Sigma = \{0, 1\}$.
- Compute the sequence of states for x and store in a $T \times n$ array.

An interesting remark concerning this experiment is that if we round the output to the nearest integer at test time, we obtain an MSE of 0.

Experiments with k-counting WFA For the experiments with the k-counting WFA, we fix the number of layers to 4 and evaluate the model on sequences of length $T = 32$. Here, we consider $k \in \{2, 4, 6, 8\}$ and choose the embedding size $d \in \{2, 4, 8, 16, 32, 64\}$. Note here that we use the same value for both embedding dimension and hidden size. As it is the case for the binary counting WFA, here we also use a linear layer for readout. The data is generated using the same procedure as described in the above section with the exception that here we sample from $\Sigma = \{0, 1, \ldots, k - 1\}$.

Feature visualization experiment

- fixed embedding size and hidden layer size of 16, nb heads of 2
- trained on sequences of length 16

D.2 Additional Experiments

In this section, we present an extended version of the results of the experiments presented in Section 5.

D.2.1 Can logarithmic solutions be found?

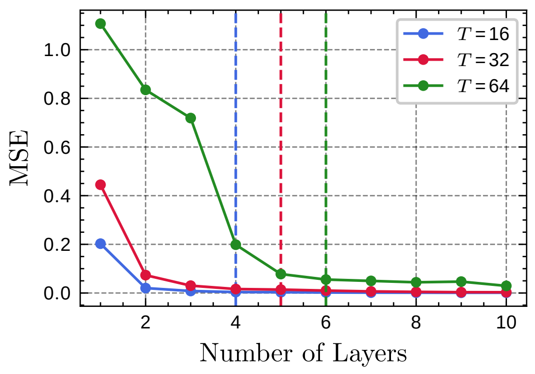
Here, we present the full table of results containing all MSE values for each considered automata. Here we report the minimum MSE over 10 runs and bold the best MSE for each automaton.

Table 2: MSE for all Pautomac automata

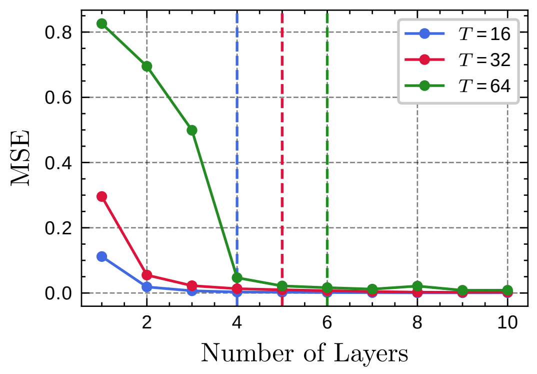
Nb/Nb layers	2	4	6	8	10
pautomac 12	0.005486	0.001660	0.000770	**0.000356**	0.000710
pautomac 14	0.000264	0.000130	**0.000109**	0.000158	0.006189
pautomac 20	0.007433	0.002939	0.000911	**0.000628**	0.000979
pautomac 30	0.029165	0.017486	0.013498	**0.012403**	0.068889
pautomac 31	0.007002	0.003804	0.001114	**0.000890**	0.000899
pautomac 33	0.003654	**0.001160**	0.008794	0.017104	0.016844
pautomac 38	0.001056	0.000466	0.000316	0.000216	**0.000213**
pautomac 39	0.014218	0.002677	**0.001310**	0.002736	0.002686
pautomac 45	0.020730	**0.018859**	0.021852	0.024375	0.023893

D.2.2 Do solutions scale as theory suggests?

Here we present the plots for both the mean and minimum MSE values for the synthetic experiments on number of layers and embedding size. We equally include tables containing the average MSE values with their respective standard deviation values. We include these values in a table instead of on the plots directly for readability.

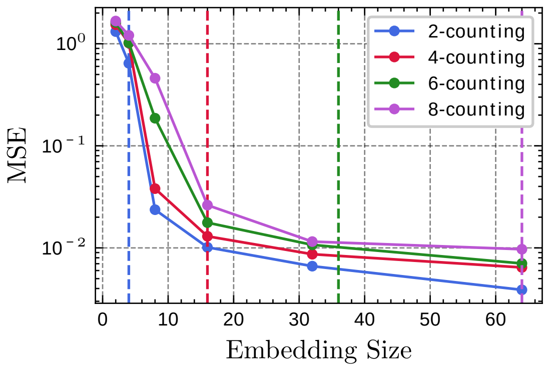


(a) Average MSE over 10 runs

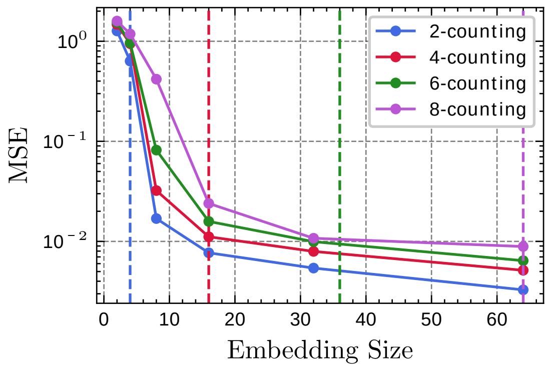


(b) Minimum MSE over 10 runs

Figure 6: MSE vs. number of layers: We notice that for both Figures 6a and 6b, the trend is very similar



(a) Average MSE over 10 runs



(b) Minimum MSE over 10 runs

Figure 7: MSE vs. embedding size: Here the trend between average and minimum values is also very similar

Table 3: MSE with standard deviation across layers

number of layers	$L = 16$		$L = 32$		$L = 64$	
1	0.202724	0.162529	0.445007	0.337551	1.107175	0.660082
2	0.020226	0.01240	0.073415	0.0454464	0.834934	0.253568
3	0.008247	0.002069	0.030223	0.0099422	0.718981	0.506068
4	0.003796	0.002531	0.016276	0.0187777	0.198496	0.369018
5	0.003137	0.001482	0.014042	0.0087545	0.077845	0.1208077
6	0.002000	0.001380	0.009957	0.008190	0.055315	0.060973
7	0.001449	0.002166	0.006567	0.004517	0.049691	0.09459
8	0.001223	0.000897	0.004663	0.006610	0.044012	0.059172
9	0.000945	0.000558	0.003357	0.002276	0.046805	0.146278
10	0.001156	0.000687	0.003093	0.004135	0.029291	0.059460

Table 4: MSE with standard deviation across embedding sizes

embedding size	$k = 2$		$k = 4$		$k = 6$		$k = 8$	
2	1.316259	0.0361747	1.537641	0.0553744	1.619518	0.0454326	1.669770	0.046430
4	0.643738	0.0401225	1.016194	0.704216	1.021940	0.091605	1.208901	0.092536
8	0.023678	0.011056	0.038078	0.011791	0.186437	0.769370	0.459510	0.164538
16	0.010134	0.0027882	0.01299	0.002661	0.01768	0.003942	0.026244	0.003718
32	0.006623	0.002187	0.008674	0.002234	0.010712	0.001917	0.011528	0.001126
64	0.003882	0.001105	0.006435	0.003151	0.007041	0.001648	0.009701	0.002908