Recurrent Convolutional Neural Networks Learn Succinct Learning Algorithms

Surbhi Goel
Microsoft Research &
University of Pennsylvania
surbhig@cis.upenn.edu

Sham Kakade
Harvard University
sham@seas.harvard.edu

Adam Tauman Kalai
Microsoft Research
adam@kal.ai

Cyril Zhang
Microsoft Research
cyrilzhang@microsoft.com

Abstract

Neural Networks (NNs) struggle to efficiently learn certain problems, such as parity problems, even when there are simple learning algorithms for those problems. Can NNs discover learning algorithms on their own? We exhibit a NN architecture that, in polynomial time, learns as well as any efficient learning algorithm describable by a constant-sized learning algorithm. For example, on parity problems, the NN learns as well as row reduction, an efficient algorithm that can be succinctly described. Our architecture combines both recurrent weight-sharing between layers and convolutional weight-sharing to reduce the number of *parameters* down to a constant, even though the network itself may have trillions of nodes. While in practice the constants in our analysis are too large to be directly meaningful, our work suggests that the synergy of Recurrent and Convolutional NNs (RCNNs) may be more powerful than either alone.

1 Introduction

Neural Networks (NNs) can seem magical in what they can learn. Yet humans have designed simple learning algorithms, even for binary classification, which they cannot match. A well-known example is the class of parity functions over the d-dimensional hypercube, i.e., d-bit strings. In that problem, there is an unknown subset S of the d bits, and the label of each example x is 1 if x has an odd number of 1's in S. While gradient-based learning struggles to learn parity functions (Kearns and Valiant, 1993) even over uniformly random x, row reduction (i.e. Gaussian elimination) can be used to find S using only $O(d)$ examples and $O(d^2)$ runtime.

A tantalizing question is whether a NN *can discover an efficient learning algorithm itself*, thereby learning classes such as parities. We refer to this as *Turing-optimality*, since algorithms can be described by Turing machines. More specifically, we will give an example of a simple NN architecture that achieves Turing-optimality. In particular, this is the first NN architecture that provably discovers a efficient parity learning algorithm in polynomial time. The parity learning algorithm is efficient, like row reduction, requiring $O(d)$ examples and $O(d^2)$ runtime. Our learning architecture would be quite simple to describe with a modern library such as pytorch. However, we do not expect our specific architecture to be especially good in practice, as the constants in our analysis are much too large to be practical. Nonetheless, it does suggest that the ingredients used in the architecture, especially the combination of recurrent weight-sharing across layers and convolutional weight-sharing within layers, may be useful in designing practical architectures for NNs to learn algorithms.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

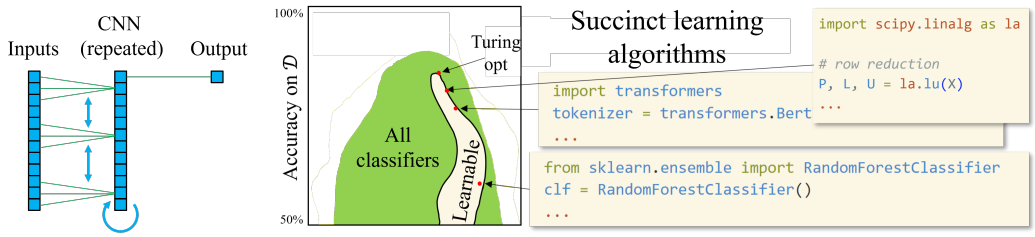

Figure 1: *(Left)* A simple RCNN by which we mean one small set of parameters is repeated within each layer and across layers. *(Right)* A Turing-optimal algorithm must output a classifier that is as accurate, on future examples from $\mathcal{D}$, as that which is output by any other succinct efficient learning algorithm. Such classifiers are only a tiny subset of the set of all classifiers, some of which may be more accurate but cannot even be stored in a computer. Moreover, even competing with the best poly-sized NN is intractable, assuming the existence of one-way functions (Kearns and Valiant, 1994); it is not possible for any learning algorithm to efficiently compete with the region depicted in green.

Figure 1 illustrates the difference between *classifiers*, such as NNs, and the learning *algorithms* that learn their parameters, such as Stochastic Gradient Descent (SGD) with a given architecture (we use the term *architecture* broadly to include other algorithmic features including learning random initialization, learning rate schedule, restarts, and hyperparameter search). As *classifiers*, two-layer NNs can compute any Boolean function on d binary inputs, including parity functions. However, it is unclear whether these architectures can learn such functions efficiently using gradient-based approaches without any priors encoded in the architectures.

More formally, a Turing-optimal learning algorithm is one which learns as well as any bounded learning algorithm, specifically a constant-sized Turing machine that outputs a binary classifier in polynomial time, such as row reduction for parity learning. Our key contribution is a simple recurrent convolutional (RCNN) architecture which combines recurrent weight-sharing across layers and convolutional weight-sharing within each layer. The number of weights in the convolutional filter can be very few, even a constant, but these weights can determine the activations of a very wide and deep network. We show that any algorithm A represented by a constant-sized TM has a corresponding constant-sized convolutional filter for which the RCNN computes the same function as A. Because the convolutional filter is constant-sized, with constant probability random initialization will find it (or something even better, assuming our reduction is not optimal). Thus, using a validation set and random restarts, the RCNN will find a filter which performs as well as A, with high probability.

Unfortunately, the above argument would apply to an RCNN architecture that strangely takes the *entire training set* as input at once, and outputs a classifier. Fortunately, Abbe and Sandon (2020) show how to use a few additional simple NN components and SGD updates to memorize relevant information in the weights of these components. Similarly, we add a few extra non-convolutional layers to our architecture so that it can be learned "normally" with SGD rather than requiring the entire dataset at once. Fortunately, the implementation of this functionality is compatible with the RCNN with only a constant overhead in terms of size.

Probably Algorithmically Optimal (PAO) learning. To define Turing-optimality, it is convenient to formalize a weaker requirement than PAC learning which we call PAO learning, that in some sense turns PAC learning on its head. Rather than requiring optimality among the space of all classifiers $c \in \mathcal{C}$, it requires only optimality compared to classifiers output by learning algorithms $A \in \mathcal{A}$. In many cases $|\mathcal{C}| \gg |\mathcal{A}|$, as $\log |\mathcal{C}|$ and $\log |\mathcal{A}|$ are the number of bits required to encode the parameters (modern models approaching terabytes) and the learning program source code (e.g., kilobytes), respectively (see Arora and Zhang (2021) for a more setailed discussion). In these cases, matching the performance of the classifier output by the best learning algorithm (within some family) may be more reasonable matching the performance of the best overall classifier. Turing-optimality is the special case of $\mathcal{A}$ consisting of the set of succinct programs, specifically constant-sized time-bounded Turing machines. In Section 5, we also discuss how this approach can be used across multiple problems to discover a learning algorithm that can be reused on future problems, so the search need not be repeated for each learning problem.

Turing-optimality as a lens. Like any asymptotic notion, Turing-optimality does not guarantee efficient learning. Just as a polynomial-time algorithm is not guaranteed to be faster than an exponential-time algorithm on inputs of interest, such notions can still provide a useful lens to understand algorithms. If learning algorithm L_1 is Turing-optimal and L_2 is not, then L_1 can nearly match (or exceed) the performance of L_2 on any distribution, with polynomial overhead. Data distributions where L_2 requires super-polynomially resources to match L_1's performance would need to be examined to see if they are important. We show that a Recurrent Convolutional Neural Network (RCNN) architecture, with random initialization, is Turing-optimal. The contribution of this work is showing that a definition of Turing-optimality is achievable by a simple NN architecture. In future work, it would be interesting to better understand which other combinations of architectures, initializations, and learning rates are Turing-optimal.

1.1 Related work

We review some prior lines of work which establish or use other notions of computational universality. We note that most of the notions defined in these works apply to representations rather than algorithms.

Universal function approximation. The first line of work relevant to our results is the basic theory of universal function approximation, which quantifies the ability to fit any sufficiently well-behaved function for neural networks (Hornik et al., 1989; Cybenko, 1989; Funahashi, 1989), nearest neighbors (Devroye et al., 1994) and SVMs with RBF kernel (Wang et al., 2004). However, they lack statistical insight, e.g., lookup tables are universal over $\mathcal{X} = \{-1, 1\}^d$ but offer little statistical power. Further refinements (Barron, 1993, 1994; Lee et al., 2017) consider Fourier-analytic criteria for functions to be representable by smaller neural networks. The goal of subsequent lines of work described in this section, as well as the present work, is to investigate the *computationally efficient* approximation of functions—in other words, the ability of neural networks to emulate efficient learning algorithms.

Turing-completeness of neural architectures. Siegelmann and Sontag (1995) establish that recurrent neural networks are Turing-complete, using a trick to store the entire TM tape in a single rational number, therefore requiring an extreme amount of bit precision. More recently, Graves et al. (2014) construct a differentiable TM-inspired architecture. A number of recent works establish Turing-completeness (a classical and weaker notion) for variants of the Transformer architecture (Dehghani et al., 2018; Yun et al., 2019; Bhattamishra et al., 2020a,b), motivated by empirical advances in discrete reasoning tasks found in natural language processing, theorem proving, and program synthesis. Recently, Wei et al. (2021) propose a notion of statistically meaningful (SM) approximation which requires the approximation to be statistically learnable as well. They show that Transformer architectures can "SM-approximate" time bounded TMs with sample complexity logarithmic in the time. Unlike our notion, Turing-completeness does not take computational efficiency into account.

Enumerative program search. A folklore argument, similar to Levin's classic universal search (Levin, 1973), states that one can achieve Turing-optimality by enumerating all Turing machines of a fixed size, run them all on a training set, and choose the one which performs best on a validation set. The algorithm, however, is also completely infeasible in normal programming languages because the probability of even generating a single program that compiles is minuscule.

Efficient universality of deep learning. Most closely related to our work is that of Abbe and Sandon (2020), which shows how, given any circuit C, e.g., encoding a learning algorithm for parity, one can initialize the weights of a NN so that it emulates C when the NN is trained by SGD. This emulator requires C to be given as input. Now, row reduction, like any polynomial-time algorithm, can be converted to a circuit C. However, the size of this circuit is polynomial in the runtime of the algorithm. This is why C is required as input, e.g., one has no hope of discovering the Gaussian elimination algorithm by random initialization as its probability would be exponentially small in the dataset size. Thus, their algorithm does not "discover" the learning algorithm itself–it is hard-coded into the network. As they discuss, they could encode in the circuit C an enumerative program search, but this is also a parity learning algorithm that needs to be encoded into the network (and is in fact significantly more involved to encode as a circuit). Their work was recently extended to mini-batch SGD by Abbe et al. (2021), and it would be interesting to see if our result could be similarly extended.

2 Preliminaries

For simplicity, we focus on binary classification with $\mathcal{Y} := \{-1, 1\}$. For domain $\mathcal{X}$, a (binary) *classifier* is a function $c : \mathcal{X} \to \mathcal{Y}$. For any distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, the *error* of c is, $\mathrm{err}_{\mathcal{D}}(c) := \Pr_{(x,y)\sim\mathcal{D}}[c(x) \neq y] \in [0, 1]$. A learning algorithm takes as input $m \geq 1$ labeled training examples in $(\mathcal{X} \times \mathcal{Y})^+ = \bigcup_{m\geq 1}(\mathcal{X} \times \mathcal{Y})^m$ and outputs a classifier. For further simplicity, we focus on data on the hypercube $\mathcal{X}_d := \{-1, 1\}^d$. The powers of 2 less than 1 are denoted by $2^{-\mathbb{N}} = \{2^{-i} \mid i \in \mathbb{N}\}$. We say an algorithm is *polytime* if it runs in time polynomial in its input length, which is $\mathrm{poly}(dm)$ for a learning algorithm when run on m examples in d dimensions.

2.1 Turing machines, circuits, and efficient computability

Since our main results require the simulation of an arbitrary efficient learning algorithm, we will need to establish formal notation for relevant concepts from the theory of computation. Various notions of computational efficiency may be used. To be concrete, we may use a 2-tape Turing Machine (TM) where the input is on the first tape and the 2nd tape is used for computation (e.g. see Hopcroft et al. (2001) for a standard reference on Turing machines).

One issue that complicates runtime analysis of learning algorithms is that a classifier may be very slow to evaluate, even if the learning algorithm is fast.[1] There are two solutions to this issue, which are equivalent up to polynomial time. The first is learning algorithms that output classifiers, which we represent as Boolean circuits. Circuits circumvent this technicality because they can be evaluated in time nearly linear in the time it takes to output them. Thus time spent on classification is folded into training time. Moreover, any binary classifier on $\mathcal{X}_d$ can be represented as a circuit, and it is straightforward to convert a NN to a circuit with linear blowup. It is also well-known that other universal representations such as (time-bounded) TMs can be converted to Boolean circuits in polynomial time using unrolling. Other succinct representations could be used, but this choice simplifies runtime analysis.

Formally, we assume that each classifier output by a learning algorithm $c : \mathcal{X}_d \to \mathcal{Y}$ is represented as Boolean circuit, with False representing -1 and True representing 1. If the output of the learner is not a valid circuit classifier, then by default we assume it classifies everything as 1. We also consider learners that can be simulated by a TM with size $\leq s$ in time t using only m labeled examples.

Definition 1 ((s, m, t)-bounded learner)**.** *A learner A is a s-bounded learner which outputs a classifier circuit in at most t steps on any dataset consisting of at most m labeled examples.*

2.2 Components of deep learning

In this section, we establish some notation for the building blocks of common deep learning pipelines.

Feedforward layers. A fully-connected feedforward layer $\mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$, with activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is parameterized by a matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and bias $b \in \mathbb{R}^{d_{\text{out}}}$, specifying the map $x \mapsto \sigma(Wx + b)$, where $\sigma(\cdot)$ is applied entrywise. A feedforward network is the iterative composition of feedforward layers, possibly omitting an application of σ at the final layer.

Convolutional layers. Our main construction will apply the same constant depth feedforward network repeatedly to each 3×3 patch of a 2-dimensional "image". This can be viewed as applying multi-channel convolutional layers followed by non-linear activation consecutively. Due to weight sharing across patches, the number of parameters do not depend on the input dimension but rather on the patch dimension and the number of channels. Often in practice, to ensure same output dimension as input, it is common to add a constant padding (say p) around the boundaries. This is crucial for our construction. More formally, a convolutional layer specifies a $k \times k$ patch-wise linear maps from $\mathbb{R}^{k\times k\times C_{\text{in}}}$ to $\mathbb{R}^{C_{\text{out}}}$; in particular, when $k = 1$, a convolutional layer specifies a *pixel-wise* linear map from $\mathbb{R}^{C_{\text{in}}}$ to $\mathbb{R}^{C_{\text{out}}}$. We let $\mathsf{Conv2D}$ be the application of the linear maps extended to the entire input. We overload $\mathsf{Conv2D}$ to also allow for patch-wise fully-connected feedforward layers.

[1] Natural examples where inference is more computationally expensive than learning arise in nonparametric models such as nearest-neighbors or Gaussian processes.

Recurrent layers. Finally, our construction will use recurrent weight sharing: for a function $f : \mathcal{Z} \times \Theta \to \mathcal{Z}$ and a positive integer L, we use $f^{(L)} : \mathcal{Z} \times \Theta \to \mathcal{Z}$ to denote the L-times iterated composition of f, sharing the parameters $\theta \in \Theta$ between iterations; for example,

$$f^{(3)}(X; \theta) := f(f(f(X; \theta); \theta); \theta).$$

The training pipeline: SGD with random initialization. Finally, we establish some notation for stochastic gradient descent, whose variants form the predominant class of methods for training neural networks. Given a continuously differentiable[2] loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ and continuously differentiable function $f : \mathcal{X} \times \Theta \to \mathcal{Y}$ where $\Theta = \mathbb{R}^d$, a step of stochastic gradient descent (SGD) on a single example $(x, y) \in \mathcal{X} \times \mathcal{Y}$, with learning rate $\eta \in \mathbb{R}$, maps the current iterate θ to

$$\theta' := \theta - \eta \nabla_\theta \ell(f(x, \theta), y).$$

SGD on a sequence of examples $\{(x_t, y_t)\}_{t=1}^T$ is defined by applying this recurrence iteratively from an initialization θ_0 (usually selected randomly from a specified distribution), giving a sequence of iterates $\{\theta_t\}_{t=1}^T$. It is routine to specify a subset $S \subseteq [d]$ of the parameters to be optimized; in this case, the parameters in S are updated according to the above equation, while the rest are unchanged.

3 Algorithm learning and Turing-optimality

In this section, we adopt a model of learning which turns PAC learning upside down. A *learning algorithm* is a function A that, for any $d, m \geq 1$, outputs a classifier $A(Z) : \mathcal{X}_d \to \mathcal{Y}$ for any dataset $Z = \{(x^{(i)}, y^{(i)})\}_{i=1}^m \in (\mathcal{X}_d \times \mathcal{Y})^m$ of $m \geq 1$ labeled d-dimensional examples. Recall that $\mathcal{X}_d = \{-1, 1\}^d$ and $\mathcal{Y} = \{-1, 1\}$.

The following definition captures efficient learnability of a class of learning algorithms A. The run-time of the algorithm is required to be polynomial in its input size $\mathrm{poly}(dm)$. An important feature of this definition is that it requires the number of examples to be polynomial in the dimension d, avoiding the curse of dimensionality. Since we will soon consider ϵ, δ as inputs, we consider only powers of 2 to avoid having to represent arbitrary real numbers.

Definition 2 (PAO-learner)**.** *Polytime learning algorithm A is a Probably Algorithmically Optimal (PAO) learner for family $\mathcal{A}$ if there is a polynomial p such that for any $\epsilon, \delta \in 2^{-\mathbb{N}}$, for any $d \geq 1$, any distribution $\mathcal{D}$ over $\mathcal{X}_d \times \mathcal{Y}$, and any dataset sizes $m \geq 1, M \geq p(d, m, 1/\epsilon, 1/\delta)$,*

$$\Pr_{Z \sim \mathcal{D}^m, Z' \sim \mathcal{D}^M} \left[\mathrm{err}_{\mathcal{D}}(A(Z; Z')) \leq \min_{B \in \mathcal{A}} \mathrm{err}_{\mathcal{D}}(B(Z)) + \epsilon \right] \geq 1 - \delta,$$

where $Z; Z'$ is the concatenation of the two datasets Z, Z'. We further assume that d, m and M can be determined from the PAO-learner's input.

We now observe that one can equivalently design a learning algorithm that has $\epsilon, \delta > 0$ as inputs.

Observation 1 (ϵ, δ-PAO-learner reduction)**.** *Let $A_{\epsilon,\delta}$ be an "ϵ, δ-PAO learner" for $\mathcal{A}$ meaning that it is a polytime learning algorithm that takes additional inputs ϵ, δ, and there exists some constant k such that: for any $\epsilon, \delta \in 2^{-\mathbb{N}}$, any dataset sizes $m \geq 1, M \geq (dm/\epsilon\delta)^k$,*

$$\Pr_{Z \sim \mathcal{D}^m, Z' \sim \mathcal{D}^M} \left[\mathrm{err}_{\mathcal{D}}(A_{\epsilon,\delta}(Z; Z')) \leq \min_{B \in \mathcal{A}} \mathrm{err}_{\mathcal{D}}(B(Z)) + \epsilon \right] \geq 1 - \delta.$$

Then, for $r = 2^{\lfloor -\frac{1}{3k} \log M \rfloor}$, $A_{r,r}$ is a PAO-learner for $\mathcal{A}$.

The proof is straightforward and can be found in Appendix A. The $M \geq (dm/\epsilon\delta)^k$ requirement is a convenient equivalent to a polynomial bound $M \geq p(d, m, 1/\epsilon, 1/\delta)$.

Although we only analyze PAO learning for the family $\mathcal{A}$ of bounded Turing machines, it can be analyzed even for continuous classes A. For instance, it would be straightforward to show that grid search can PAO learn a constant number of bounded hyperparameters of a given algorithm if the

[2] It is routine to extend these definitions to continuous functions which are piecewise continuously differentiable, such as neural networks with ReLU activations. We omit the details in this paper, as our constructions will never evaluate a gradient of f at a discontinuity.

algorithm's error is Lipschitz continuous in those hyperparameters, using a separate validation set to choose the best hyperparameters. PAO learning, as defined, does not specify how classifiers are represented, and could apply to any classifier representation. Recall that we represent classifiers by Boolean circuits as discussed in Section 2.

We next define Turing-optimal learners, which are PAO-learners for the class of bounded TMs (constant size, run in polynomial time, and output a circuit classifier).

Definition 3 (Turing-optimal)**.** *Fix constants $s, k \in \mathbb{N}$. Let the set $\mathcal{B}_{s,k}$ be the set of Turing machines which have $\leq s$ states and, run in time $\leq (2dm)^k$ on a dataset $Z \in (\mathcal{X}_d \times \mathcal{Y})^m$ and output a circuit. Learning algorithm A is (s,k)-Turing-optimal if A PAO-learns (or equivalently ϵ, δ-PAO learns) $\mathcal{B}_{s,k}$. Learning algorithm A is Turing-optimal if A is (s,k)-Turing-optimal for all constants $s, k \in \mathbb{N}$.*

Note that a Turing-optimal learner A must run in polytime, but the number of examples required to learn each $\mathcal{B}_{s,k}$ can be different, i.e., it will learn $\mathcal{B}_{s,k}$ using $M \geq (dm/\epsilon\delta)^{e_{sk}}$ additional examples, for a different constant e_{sk} for each s and k. Similar to Observation 1, a Turing-optimal learner can be constructed from an (s,k)-Turing optimal learner. The claim below, together with Observation 1, imply that a (ϵ, δ, s, k)-Turing-optimal learner can be converted to a Turing optimal learner. Algorithm 2 and its proof are presented in Appendix A.

Claim 1 ((s,k)-Turing-optimal reduction)**.** *Let $A_{s,k}$ be an algorithm that takes inputs s, k and is (s,k)-Turing optimal for each pair of constants $s, k \in \mathbb{N}$. Then, Algorithm 2$(A_{s,k})$ is Turing-optimal.*

Finally, it is not difficult to see that a Turing-optimal learner also PAC-learns any concept class $\mathcal{C}$ that is PAC-learnable. Following standard conventions, the PAC learning algorithm is given target accuracy ϵ and failure probability δ as inputs. Also, say a distribution $\mathcal{D}$ is said to be *consistent* with set $\mathcal{C}$ of classifiers if there is some $c \in \mathcal{C}$ with $\mathrm{err}_{\mathcal{D}}(c) = 0$.

Definition 4 (PAC-learning)**.** *Let $\mathcal{C} = \bigcup_{d \geq 1} C_d$, where $c : \mathcal{X}_d \to \mathcal{Y}$ for each $c \in \mathcal{C}_d$. Polytime*[3] *learning algorithm $A_{\epsilon,\delta}$ PAC-learns $\mathcal{C}$ if $A_{\epsilon,\delta}$ and there is a polynomial p such that, for any $\epsilon, \delta \in 2^{-\mathbb{N}}$, $d \in \mathbb{N}$, $m \geq p(d, 1/\epsilon, 1/\delta)$ and distribution $\mathcal{D}$ consistent with $\mathcal{C}_d$:*

$$\Pr_{Z \sim \mathcal{D}^m} [\mathrm{err}_{\mathcal{D}}(A_{\epsilon,\delta}(Z)) \leq \epsilon] \geq 1 - \delta.$$

The computational polynomial-time efficiency requirement on $A_{\epsilon,\delta}$ means that its runtime is polynomial in its input size, $\mathrm{poly}(dm + \log 1/\epsilon\delta)$, because it takes $O(\log 1/\gamma)$ bits to describe $\gamma \in 2^{-\mathbb{N}}$.

Claim 2. *Suppose there is some learning algorithm that PAC-learns $\mathcal{C}$ and suppose that A is a polynomial-time Turing-optimal learner. Then A PAC-learns $\mathcal{C}$ as well.*

We defer the proof to the Appendix A.

4 Turing-optimality of SGD on randomly initialized RCNNs

In this section, we will design a Turing-optimal leaner in the form of a NN and a corresponding training pipeline. Our NN will be of the form of a RCNN (see Figure 2) with very few trainable parameters and our training pipeline (see Algorithm 1) will use random initialization, and random restarts to find good parameters. Let us present our main result.

Theorem 1. *There exists constants $c_1, c_2, c_3 > 0$ such that the following holds. For any $d, s, m, t \in \mathbb{N}$, there exists learning rate $\eta \in \mathbb{R}$, $L = c_1(t + m + d)$, and $\mathcal{U}_s \subseteq \mathbb{R}^4$ where: for any probability measure $\mathcal{D}$ on $\mathcal{X}_d \times \{-1, 1\}$, (s, m, t)-computable learner A, and training set $\mathcal{S} \in (\mathcal{X}_d \times \{-1, 1\})^m$ drawn i.i.d. from $\mathcal{D}$, Algorithm 1 returns a function f such that with probability at least $s^{-c_2 s^2}$,*

$$\mathrm{err}_{\mathcal{D}}(f) \leq \mathrm{err}_{\mathcal{D}}(A, \mathcal{S}).$$

The bit precision required by Algorithm 1 is $\lceil \log(s) \rceil + c_3$.

[3]The standard PAC learning definition requires the learner to run in time also $q(d, 1/\epsilon, 1/\delta)$ for some polynomial q, which would admit an algorithm that is not polytime, e.g., if it uses $m = 1$ examples but runs in time $q(d, 1/\epsilon, 1/\delta)$. However, such an algorithm can trivially be converted to a polytime algorithm by padding its input with an additional $q(d, 1/\epsilon, 1/\delta)$ examples.

[4]This set can be constructed with knowledge of only s.

Algorithm 1 SGD on randomly initialized RCNN

Input: training set $\mathcal{S} := \{(x^{(i)}, y^{(i)})\}_{i=1}^m$, size s, initialization set $\mathcal{U}_s$, depth L, learning rate η
Create dummy sample $(x^{(m+1)}, y^{(m+1)}) = (\mathbb{1}_d, 1)$
Initialize $f \in \mathcal{F}_{\mathsf{RCNN}}^{d+1,m+1,100s,100,L}$ (see Definition 5) with parameters $\Theta_{\mathsf{mem}}^{(1)}, \Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}}$ such that $W^{(1)} = 0$, and all entries of $V_1, V_2, V_3, V_4, V_5, U_1, U_2$ are sampled uniformly from $\mathcal{U}_s$
for $i = 1$ **to** $m + 1$ **do**
Update parameters in the memory layer:

$$W^{(i+1)} = W^{(i)} - \eta \, \nabla_W \ell \left(f \left(\left[{x^{(i)}}^\top \; 1 \right]^\top ; W, \Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}} \right), y^{(i)} \right) \Big]_{W=W^{(i)}}$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is the squared loss, that is, $\ell(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2$
Output: function $f(\cdot; W^{(m+2)}, \Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}})$

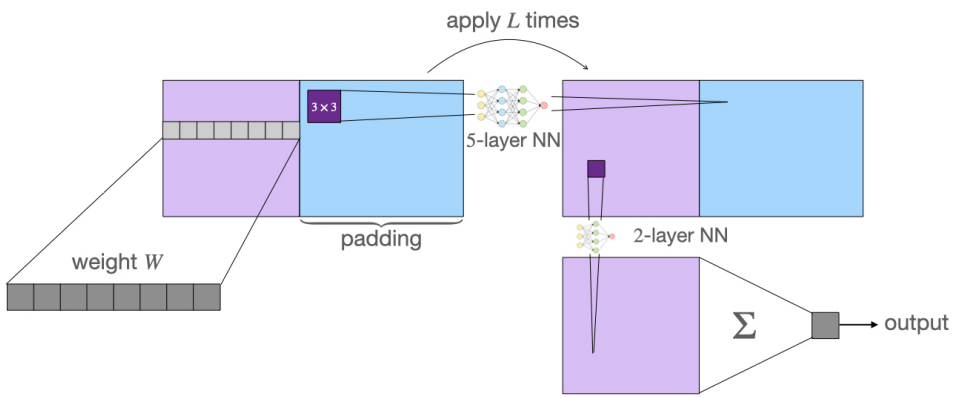

Figure 2: Recurrent convolutional neural network from our construction. A dense weight matrix W is applied to the input to convert it into 2D followed by adding padding of 1s to the right. This is followed by L applications of a 3×3 convolutional layer where each layer applies a 5-layer NN to each patch. This is followed by a pixel-wise convolutional layer consisting of a 2-layer NN. Lastly, the output corresponding to the main grid is summed.

Remark. Our Algorithm 1 sets the learning rate of 0 for the shared weights in the RCNN part of the NN and updates only the dense memory layer. Since our result is constructive, it is entirely possible that a search for an optimal-learning rate may perform better in practice.

Finally, we will use Theorem 1 to create a (s, k)-Turing optimal learner with the help of random restarts and an additional validation set,

Corollary 2. *For fixed constants $c_1, c_2 > 0$, Algorithm 1 can be converted to an (ϵ, δ, s, k)-Turing-optimal learner for any fixed s, k, time bound $t = (2dm)^k$ and $\epsilon, \delta \in (0, 1)$ by running it $s^{c_1 s} \log(1/\delta)$ times with random restarts, and selecting the classifier that performs best on a validation set of size $c_2 s \log s \log(1/\delta)/\epsilon^2$.*

Claim 1 converts this to a Turing-optimal learner. Because our formal definition of Turing-optimality applies only to deterministic circuit classifiers, one must also convert the NN to a circuit and derandomize the algorithm (which can be done using random bits extracted from additional iid random labeled examples (Kearns and Vazirani, 1994)).

Next we give a detailed description of the architecture and a proof overview of Theorem 1. The proof of Corollary 2 and the complete proof of Theorem 1 can be found in Appendix B.

4.1 Network architecture: RCNN with a memory layer

Let us first describe the neural network architecture that Algorithm 1 uses in more detail. The architecture comprises of a dense layer of size linear in m (the number of samples) and d (the dimension

of the input) as the first layer. The output of this layer is padded with 1s on the right, and then fed into a RCNN (recurrent weight-sharing across depth and convolutional weight sharing across width). The RCNN has only $\mathsf{poly}(s)$ shared parameters, due to its recurrent and convolutional nature. These shared parameters are in the form of a 5-layer NN applied as a convolutional layer to 3×3 patches of the input.[5] This convolutional layer is applied recurrently $L = p(t, d, m)$ times for some polynomial p (with the same parameters) where t is the bound on the runtime of the TM. The final outputs from the RCNN are then passed through a pixel-wise 2-layer NN and finally summed to give a scalar prediction.

Definition 5 (RCNN with a memory layer)**.** *For $d_{\mathsf{in}}, d_{\mathsf{mem}}, d_{\mathsf{rc}}, d_{\mathsf{head}}, l > 0$, define the function class $\mathcal{F}_{\mathsf{RCNN}}^{d_{\mathsf{in}}, d_{\mathsf{mem}}, d_{\mathsf{rc}}, d_{\mathsf{head}}, l}$ of RCNNs where each $f \in \mathcal{F}_{\mathsf{RCNN}}^{d_{\mathsf{in}}, d_{\mathsf{mem}}, d_{\mathsf{rc}}, d_{\mathsf{head}}, l}$ is parameterized by memory layer parameters $\Theta_{\mathsf{mem}} = \{W\}$ with $W \in \mathbb{R}^{d_{\mathsf{mem}} \times d_{\mathsf{in}}}$, RC layer parameters $\Theta_{\mathsf{rc}} := \{V_1, V_2, V_3, V_4, V_5\}$[6] with $V_1 \in \mathbb{R}^{d_{\mathsf{rc}} \times 9}$, $V_2, V_3, V_4 \in \mathbb{R}^{d_{\mathsf{rc}} \times d_{\mathsf{rc}}}$, $V_5 \in \mathbb{R}^{1 \times d_{\mathsf{rc}}}$, and head layer parameters $\Theta_{\mathsf{head}} = \{U_1, U_2\}$ with $U_1 \in \mathbb{R}^{d_{\mathsf{head}} \times 1}, U_2 \in \mathbb{R}^{1 \times d_{\mathsf{head}})}$. The local convolutional operation is denoted by $f_{\mathsf{Conv2D}} : \mathbb{R}^{3 \times 3} \to \mathbb{R}$ and is applied on the 3×3 grid centered at each coordinate of the $d_{\mathsf{mem}} \times (d_{\mathsf{in}} + l)$ input (with padding p for the edge coordinates). We overload the notation f_{Conv2D} to denote the $d_{\mathsf{mem}} \times (d_{\mathsf{in}} + l)$ output post the local application on the input.*

$$
\begin{aligned}
f(z; \Theta_{\mathsf{mem}}, \Theta_{\mathsf{RCNN}}; \Theta_{\mathsf{head}}) &= \mathsf{sum}\left(f_{\mathsf{head}}\left(f_{\mathsf{Conv2D}}^{(l)}\left(f_{\mathsf{mem}}(z; \Theta_{\mathsf{mem}}); \Theta_{\mathsf{RCNN}}\right)_{:,:d_{\mathsf{in}}}\right); \Theta_{\mathsf{head}}\right) \\
\textit{where}\ \ f_{\mathsf{mem}}(z; \Theta_{\mathsf{mem}} = \{W\}) &= [\ \ W\mathrm{diag}(z) \quad \mathbb{1}_{d_{\mathsf{mem}} \times l}\ \ ] \\
f_{\mathsf{head}}(z : \theta = \{U_1, U_2\}) &= U_1 \sigma(U_2 z) \\
f_{\mathsf{Conv2D}}(z; \Theta = \{V_1, V_2, V_3, V_4, V_5\}) &= V_5 \sigma(V_4 V_3 \sigma(V_2 V_1 \mathsf{vec}(z)))).
\end{aligned}
$$

with σ being the ReLU activation.

Remark. If our architecture did not have both recurrent and convolutional weight sharing, then the number of parameters would have dependence on $d_{\mathsf{in}}, d_{\mathsf{mem}}$ and l, which depend on m, d, and T.

4.2 Proof sketch for Theorem 1

Here we present a proof sketch for Theorem 1. Our proof follows by construction, that is, we show that for each TM A of size s, there exists parameters Θ_{rc} and Θ_{head} that ensure that (1) for the first $m + 1$ steps, when Θ_{mem} is trained with SGD, the gradients assist with memorizing the training set in the values of W_{mem}, and (2) given the memorized training set, the RCNN computes the roll-out of A with the input tape having the training set and the test example giving the prediction of A on the test example as the output. We finally show that the parameters Θ_{rc} and Θ_{head} in our construction for all TM A of fixed size and runtime belong to a fixed finite set of size $O(s)$ that can be constructed with knowledge of only s. The following lemma summarizes the aforementioned properties:

Lemma 1. *For any $d, s, m, t \in \mathbb{N}$, $\delta \in (0, 1)$, and any (s, m, t)-computable learner A, there exists $\Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}}$ with each parameter belonging to a fixed set $\mathcal{U}_s$ of size $O(s)$ (that can be constructed with the knowledge of only s) such that Algorithm 1 with $L =$ satisfies:*

1. *Memorization: For $1 \le i \le m + 1$, $W_{ab}^{(i)} = \begin{cases} \frac{1}{3} y^{(a)} x_b^{(a)} & \text{if } a \le i, b \le d \\ \frac{1}{3} y^{(a)} & \text{if } a \le i, b = d + 1 \\ 0 & \text{otherwise.} \end{cases}$*
2. *Computation: For all $x \in \{\pm 1\}^d$, $f\left(\begin{bmatrix} x \\ 1 \end{bmatrix}; W^{(m+2)}, \Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}}\right) = A(\mathcal{S})[x]$.*

Theorem 1 follows from the above the computation property of Lemma 1, since it implies that the error of Algorithm 1 will be exactly equal to the error of A.

What remains is to prove the existence of parameters that satisfy Lemma 1. Let us now briefly describe the key functionality we require the RCNN to implement for this:

Computation. Each roll-out step of the TM is a local update around the head of the TM. To implement one step of the TM, we need to compute the transition function of A at the location of the

[5] Note that we did not optimize our construction. It is quite possible to improve this to a shallower network.

[6] For ease of presentation, we hide the biases from the parameters. We can assume that the input is padded with 1 to account for biases.

head, update the new head and state, and copy the inputs of the rest of the tape. Our first observation is that this local update can be implemented using a convolutional layer if we interpret the input as the tape of A (input and working concatenated) with the head and state information stored along with the tape value. Composing these layers t times (with the same parameters) allows us to simulate t steps of the TM. More importantly, the convolutional layer requires only $O(s)$ parameters since A only has s states. In order to decode the tape content, head position, and state from the values fed to the layers of the RCNN, we interpret the input in base 3 and use different positions to encode the desired information (see Appendix B.1 for more details).

Memorization. Given that we can simulate the TM, we need to ensure that the input to the RCNN has the training set and the test example encoded onto it. Similar to Abbe and Sandon (2020), we can use SGD to memorize the training examples into the weights of the memory layer, with each row storing one example. We do this by ensuring that the gradient at iteration i through the RCNN is 1 for exactly the ith row and 0 for every other row. We also ensure that the output is 0 through the memorization phase. By using chain rule, this gives us a gradient of $[y^{(i)}x^{(i)}, y^{(i)}]$ for the ith row of W_{mem} and 0 otherwise. We can this in a local manner using the RCNN. Note that the memory layer has $O(dm)$ parameters, however we can learn these parameters from 0 initialization.

Communication. Lastly, we need to ensure that the network can differentiate between memorization phase (passing meaningful gradients) and computation phase (implementing the roll-out of the TM) using the local operations in the RCNN layers. We do this by implementing a local communication protocol: we broadcast a message based on certain conditions, where each RCNN layer implements a step of the broadcast. In order to broadcast to the entire input, we require $\approx d + m$ overhead in terms of the depth of the network.

Finally, we show that the above mentioned functionality can be achieved by a 5-layer NN with $O(s^2)$ parameters. To do so, we first describe the exact function we require the network to compute and its Jacobian on all inputs that our bit precision allows (see Appendix B.2 for the exact function). Given the function and its Jacobian on a finite set, we prove a general representational theorem (see Appendix B.4.1) that constructs a 5-layer NN with weights from a fixed set that can be constructed with knowledge of only s. We refer the reader to the Appendix B for the complete proof.

5 Discussion

In this section, we discuss potential practical implications of Turing-optimality, and broadly discuss corroborations and tensions with empirical trends.

Discovering reusable algorithms. Our analysis is wasteful in that, if one has multiple learning problems, e.g., multiple parity problems, one has to relearn the learning algorithm for each one. In fact, arguably the NN may not have learned an algorithm for parity problems in general, but rather a specialized algorithm that works on just one. To find a *reusable* algorithm, one needs multiple problems, say drawn from a meta-distribution μ over learning problems. The idea is simple: viewing the constant number of weights of the RCNN filter as hyperparameters, one tries multiple such hyperparameters on $\log 1/\delta$ learning problems, and finally selecting the hyperparameters that perform best on average. With a constant-sized random sample of hyperparameters, with high probability, one of them will perform nearly as well as the best constant-sized TM not only on these few training problems, but also on future problems drawn from μ. This is the setting considered by the literature on meta-learning (Hospedales et al., 2021) and data-driven algorithm design (Balcan, 2020). Recently, Garg et al. (2022) show empirically that Transformer architectures can meta-learn and execute simple learning algorithms in-context. We leave this and numerous other interesting directions for future work.

Concise architectures. Many Turing-complete architectures have been proposed and used in practice. The lens of Turing-optimality may help us understand what architectures are minimally adequate from a theoretical perspective. In particular, it has been popular to report ever-growing parameter counts for state-of-the-art models in domains such as natural language processing (Brown et al., 2020; Fedus et al., 2021; Lin et al., 2021). Although the other benefits of over-parameterization are at play, this work suggests that very parameter-efficient architectures are sufficient to simulate any computationally efficient learning algorithm. In light of the above, one concrete direction for further investigation is to develop practical variants of our RCNN construction, in domains dominated by other architectures. Although our analysis is too pessimistic to be of immediate practical use, it

highlights the computational power of an architecture that has occasionally appeared in applications-focused research (Pinheiro and Collobert, 2014; Liang and Hu, 2015; Spoerer et al., 2017; Alom et al., 2021). Significantly closer to our work, Schwarzschild et al. (2021) conduct an empirical study on the ability of RCNNs to extrapolate from easier to harder tasks (thus "learn an algorithm"); our work shows that it is possible for these architectures to learn any computationally efficient algorithm. Similarly, RCNNs have been investigated for planning in RL (Tamar et al., 2016); other empirical works which take a "computation time" view of depth include (Graves et al., 2014; Banino et al., 2021; Kaiser and Sutskever, 2015). RCNNs have not seen widespread adoption in state-of-the-art deep learning compared to their non-recurrent and/or non-convolutional counterparts.[7]

Beyond local search for recurrent models. The proliferation of non-recurrent attention-based models in domains previously dominated by recurrent networks, along with the under-representation of RCNNs, is perhaps due to instabilities in training recurrent networks with SGD (Pascanu et al., 2013). Indeed, Kasai et al. (2021) demonstrate that a carefully designed training procedure can convert a trained Transformer into a more parameter-efficient RNN. There may be undiscovered practical training algorithms which can bridge the gap in favor of recurrent models. Although our Turing-complete algorithm uses SGD, it uses the gradients in a way that is far from making local greedy progress on an objective; the as-efficient-as-possible search for the correct TM is implemented by random initialization.

Exhaustive search. Our Turing-optimal training pipeline relies upon exhaustively comparing classifiers trained from different random initializations to choose the best classifier within the class of concise Turing machines. This runs counter to the classical viewpoint from continuous optimization, where gradient descent is seen as a *local search* method. In problem settings of a combinatorial or algorithmic nature, we posit that exhaustive search may be unavoidable; indeed, state-of-the-art pipelines already include forms of exhaustive search such as beam search and its variants (Reddy et al., 1977; Leblond et al., 2021), as well as chain-of-thought generation (Wang et al., 2022).

Memory modules. There have been many attempts to build practical memory modules into neural networks (Graves et al., 2014; Sukhbaatar et al., 2015; Grave et al., 2016; Dai et al., 2019). Our construction proposes an integrated memory mechanism: use SGD to store samples in the first layer's trainable parameters, ahead of the deep (RCNN) computation layers, by carefully ensuring that the gradient signal back-propagates through the RCNN layers correctly.

Limitations and broader impact. The primary limitation of this work is that the constant factors in our analysis are much too large to be meaningful in practice. Nonetheless, we hope that idea of combining recurrent and convolutional weight sharing will have impact. Also, the algorithms that our NNs learn would be in a format that is largely uninterpretable. Using such an algorithm carries risks, especially if the algorithm is not doing what one expects it to do.

6 Conclusion

In this paper, we present a simple NN architecture, combining recurrent and convolutional weight sharing, that achieves Turing-optimality. Among other things, it learns the well-studied class of parity functions in polynomial time, whereas prior NN analyses of parity require time exponential in the size of the parity function (or require a parity learning algorithm to be initialized into the networks weights). Our proposed architecture has connections to the deep learning literature and observed empirical trends, discussed in Section 5. Immediate improvements to make the architecture more concise and natural include: (1) reducing the size of the dense parameters to depend on the algorithm's memory usage instead of the training sample size, and (2) using SGD beyond memorization. In future work, it would be interesting to understand which other architectures are Turing-optimal, answering questions such as: are 2d convolutions necessary, and are there natural Transformer-based architectures that are Turing-optimal?

Acknowledgments. We thank Santosh Vempala for useful discussions. Sham Kakade acknowledges funding from the Office of Naval Research under award N00014-22-1-2377 and the National Science Foundation Grant under award #CCF-1703574.

[7]Note that the popular R-CNN of (Girshick et al., 2014) is not recurrent; the "R" stands for *region*.

References

Emmanuel Abbe, Pritish Kamath, Eran Malach, Colin Sandon, and Nathan Srebro. 2021. On the Power of Differentiable Learning versus PAC and SQ Learning. *Advances in Neural Information Processing Systems* 34 (2021).

Emmanuel Abbe and Colin Sandon. 2020. On the universality of deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 20061–20072.

Md Zahangir Alom, Mahmudul Hasan, Chris Yakopcic, Tarek M Taha, and Vijayan K Asari. 2021. Inception recurrent convolutional neural network for object recognition. *Machine Vision and Applications* 32, 1 (2021), 1–14.

Sanjeev Arora and Yi Zhang. 2021. Rip van Winkle's Razor: A Simple Estimate of Overfit to Test Data. *arXiv preprint arXiv:2102.13189* (2021).

Maria-Florina Balcan. 2020. Data-driven algorithm design. *arXiv preprint arXiv:2011.07177* (2020).

Andrea Banino, Jan Balaguer, and Charles Blundell. 2021. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407* (2021).

Andrew R Barron. 1993. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory* 39, 3 (1993), 930–945.

Andrew R Barron. 1994. Approximation and estimation bounds for artificial neural networks. *Machine learning* 14, 1 (1994), 115–133.

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

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. 2020b. On the computational power of transformers and its implications in sequence modeling. *arXiv preprint arXiv:2006.09286* (2020).

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 2, 4 (1989), 303–314.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. 2018. Universal transformers. *arXiv preprint arXiv:1807.03819* (2018).

Luc Devroye, Laszlo Gyorfi, Adam Krzyzak, and Gábor Lugosi. 1994. On the strong universal consistency of nearest neighbor regression function estimates. *The Annals of Statistics* (1994), 1371–1385.

Benjamin L Edelman, Surbhi Goel, Sham Kakade, and Cyril Zhang. 2021. Inductive Biases and Variable Creation in Self-Attention Mechanisms. *arXiv preprint arXiv:2110.10090* (2021).

William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961* (2021).

Ken-Ichi Funahashi. 1989. On the approximate realization of continuous mappings by neural networks. *Neural networks* 2, 3 (1989), 183–192.

Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. 2022. What Can Transformers Learn In-Context? A Case Study of Simple Function Classes. *arXiv preprint arXiv:2208.01066* (2022).

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Edouard Grave, Armand Joulin, and Nicolas Usunier. 2016. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426* (2016).

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401* (2014).

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

Timothy M Hospedales, Antreas Antoniou, Paul Micaelli, and Amos J Storkey. 2021. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* (2021).

Łukasz Kaiser and Ilya Sutskever. 2015. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228* (2015).

Jungo Kasai, Hao Peng, Yizhe Zhang, Dani Yogatama, Gabriel Ilharco, Nikolaos Pappas, Yi Mao, Weizhu Chen, and Noah A Smith. 2021. Finetuning Pretrained Transformers into RNNs. *arXiv preprint arXiv:2103.13076* (2021).

Michael Kearns and Leslie Valiant. 1994. Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. *J. ACM* 41, 1 (jan 1994), 67–95. `https://doi.org/10.1145/174644.174647`

Michael J Kearns and Leslie G Valiant. 1993. Cryptographic limitations on learning boolean formulae and finite automata. In *Machine Learning: From Theory to Applications*. Springer, 29–49.

M. J. Kearns and U. V. Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA.

Rémi Leblond, Jean-Baptiste Alayrac, Laurent Sifre, Miruna Pislar, Jean-Baptiste Lespiau, Ioannis Antonoglou, Karen Simonyan, and Oriol Vinyals. 2021. Machine translation decoding beyond beam search. *arXiv preprint arXiv:2104.05336* (2021).

Holden Lee, Rong Ge, Tengyu Ma, Andrej Risteski, and Sanjeev Arora. 2017. On the ability of neural nets to express distributions. In *Conference on Learning Theory*. PMLR, 1271–1296.

Leonid A. Levin. 1973. Universal Sequential Search Problems. *Problems of Information Transmission* 9, 3 (1973).

Ming Liang and Xiaolin Hu. 2015. Recurrent convolutional neural network for object recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3367–3375.

Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, et al. 2021. M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining. *arXiv preprint arXiv:2110.03888* (2021).

Michael A Nielsen. 2015. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International conference on machine learning*. PMLR, 1310–1318.

Pedro Pinheiro and Ronan Collobert. 2014. Recurrent convolutional neural networks for scene labeling. In *International conference on machine learning*. PMLR, 82–90.

D Raj Reddy et al. 1977. Speech understanding systems: A summary of results of the five-year research effort. *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA* 17 (1977), 138.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. 2021. Can You Learn an Algorithm? Generalizing from Easy to Hard Problems with Recurrent Networks. *arXiv preprint arXiv:2106.04537* (2021).

Hava T Siegelmann and Eduardo D Sontag. 1995. On the computational power of neural nets. *Journal of computer and system sciences* 50, 1 (1995), 132–150.

Courtney J Spoerer, Patrick McClure, and Nikolaus Kriegeskorte. 2017. Recurrent convolutional neural networks: a better model of biological object recognition. *Frontiers in psychology* 8 (2017), 1551.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. 2015. End-to-end memory networks. *arXiv preprint arXiv:1503.08895* (2015).

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. 2016. Value iteration networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 2154–2162.

Junping Wang, Quanshi Chen, and Yong Chen. 2004. RBF kernel based support vector machine with universal approximation and its application. In *International symposium on neural networks*. Springer, 512–517.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).

Colin Wei, Yining Chen, and Tengyu Ma. 2021. Statistically meaningful approximation: a case study on approximating turing machines with transformers. *arXiv preprint arXiv:2107.13163* (2021).

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. 2019. Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint arXiv:1912.10077* (2019).

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Last paragraph of Section 5
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] Last paragraph of Section 5
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] Many proof are in the appendix
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [N/A]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [N/A]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Algorithm 2 Reduce (s, k)-Turing-optimal learning to Turing-optimal learning.

Input: learner $A_{s,k}$ (that takes inputs s, k), $d, m, M \in \mathbb{N}$, $Z \in (\mathcal{X}_d \times \mathcal{Y})^m$, $Z' \in (\mathcal{X}_d \times \mathcal{Y})^M$
Let $Z'[: t]$ denote the first t examples of Z' and $Z'[-r :]$ denote the last r examples
$r = M^{1/10}$
for $s = 1$ **to** r **do**
  for $k = 1$ **to** r **do**
    for $t = 1$ **to** r **do**
      Let $c_{s,k,t} = A_{s,k}(Z; Z'[: t])$, running $A_{s,k}$ with a maximum time limit of M.
      Let $\hat{e}_{s,k,t}$ be the empirical error of $c_{s,k,t}$ over $Z'[-r :]$
Output: $c_{s,k,t}$ for s, k, t that minimize $\hat{e}_{s,k,t}$.

A Turing-optimality: Proofs of Observation 1 and Claims 1 and 2

We first restate and prove Observation 1, that an ϵ, δ-PAO learner $A_{\epsilon,\delta}$ that requires $M \geq (dm/\epsilon\delta)^k$ examples gives rise to a PAO-learner $A_{r,r}$ for $r = 2^{\lfloor -\frac{1}{3k} \log M \rfloor}$.

Observation 1 (ϵ, δ-PAO-learner reduction)**.** *Let $A_{\epsilon,\delta}$ be an "ϵ, δ-PAO learner" for $\mathcal{A}$ meaning that it is a polytime learning algorithm that takes additional inputs ϵ, δ, and there exists some constant k such that: for any $\epsilon, \delta \in 2^{-\mathbb{N}}$, any dataset sizes $m \geq 1, M \geq (dm/\epsilon\delta)^k$,*

$$\Pr_{Z \sim \mathcal{D}^m, Z' \sim \mathcal{D}^M} \left[\mathrm{err}_{\mathcal{D}}(A_{\epsilon,\delta}(Z; Z')) \leq \min_{B \in \mathcal{A}} \mathrm{err}_{\mathcal{D}}(B(Z)) + \epsilon \right] \geq 1 - \delta.$$

Then, for $r = 2^{\lfloor -\frac{1}{3k} \log M \rfloor}$, $A_{r,r}$ is a PAO-learner for $\mathcal{A}$.

Note that r is defined so as to be a power of 2 in $[M^{-1/3k}/2, M^{-1/3k}]$.

Proof. Let $p(d, m, \epsilon, \delta) := (\epsilon\delta)^{-3k} + (4dm)^{3k}$. For any $M \geq p(d, m, \epsilon, \delta)$, note first that, by definition of r, $M^{-1/3k}/2 \leq r \leq M^{-1/3k} \leq \epsilon\delta \leq \max(\epsilon, \delta)$. We also claim that $M \geq (dm/r^2)^k$. To see this, note that,

$$(dm/r^2)^k \leq (4dmM^{2/3k})^k = (4dm)^k M^{2/3} \leq M,$$

for our polynomial p. Thus $A_{r,r}$ satisfies the requirements of a PAO-learner, since $r \leq \epsilon$ and $r \leq \delta$. □

The reduction used in Claim 1 to show the equivalence of Turing-optimal and (s, k)-Turing-optimal, is shown in Algorithm 2. Note that the algorithm was chosen for its simplicity rather than optimizing parameters.

Claim 1 ((s, k)-Turing-optimal reduction)**.** *Let $A_{s,k}$ be an algorithm that takes inputs s, k and is (s, k)-Turing optimal for each pair of constants $s, k \in \mathbb{N}$. Then, Algorithm 2($A_{s,k}$) is Turing-optimal.*

Proof. Since $A_{s,k}$ is a PAO learner for $\mathcal{B}_{s,k}$, there is some polynomial $p(d, 1/\epsilon, 1/\delta)$ such that, for any $\epsilon, \delta \in 2^{-\mathbb{N}}$, given $M \geq p(d, m, 1/\epsilon, 1/\delta)$ additional examples, with probability $\geq 1 - \delta/2$, it outputs a classifier with error within $\epsilon/2$ of the best in $\mathcal{B}_{s,k}$. Also, $A_{s,k}$ runs in time in $q(d, m + M)$ for some polynomial q. We must show that Algorithm 2 is also a PAO learner for $\mathcal{B}_{s,k}$, even though it does not take s, k as inputs. Let $M' = p(d, m, 2/\epsilon, 2/\delta)$. As long as $r \geq \max(s, k, M')$, and as long as $M \geq q(d, m + M')$, we will consider $c_{s,k,t}$ as one of the candidate classifiers. Both will be the case for,

$$M \geq \big(s + k + p(d, m, 2/\epsilon, 2/\delta)\big)^{10} + q\big(d, m + p(d, m, 2/\epsilon, 2/\delta)\big) = \mathrm{poly}(d, m, 1/\epsilon, 1/\delta).$$

Finally, clearly Algorithm 2 runs in polynomial time, i.e., time $\mathrm{poly}(d, m + M)$ due to the timeout and number of iterations. □

We now move to the poof of Claim 2. Note that the Turing-optimal learner A is a PAC learner though A does not even require ϵ, δ as inputs.

Claim 2. *Suppose there is some learning algorithm that PAC-learns $\mathcal{C}$ and suppose that A is a polynomial-time Turing-optimal learner. Then A PAC-learns $\mathcal{C}$ as well.*

Proof. Call the PAC learner $P_{\epsilon,\delta}$. Let constant k be such that both $P_{\epsilon,\delta}$ runs in time $\leq (dm + \log(1/\epsilon\delta))^k$ and uses $p(d, 1/\epsilon, 1/\delta) \leq (d/\epsilon\delta)^k$ examples. In particular, for $r = m^{-1/3k}$, $P_{r,r}$ is a learning algorithm that, when it is given $m \geq (d/r^2) = d^k m^{2/3k}$ (equivalently $m \geq d^{3k}$) examples, with probability $\geq 1 - r$ outputs a classifier with error at most r. And $P_{r,r}$ runs in time at most $(dm + \log(m))^k \leq (2dm)^k$. Thus as long as $m \geq (8/\min(\epsilon, \delta)^3)^k$, with probability $\geq 1 - \delta/2$ it outputs a classifier with error at most $\epsilon/2$. Since A is a Turing-optimal learner, it outputs a classifier whose error is within $\epsilon/2$ of $P_{r,r}$, with probability $\geq 1 - \delta/2$, using additional samples $M = \mathrm{poly}(d, m)$. By the union bound, this means that with probability $\geq 1-\delta$, it outputs a classifier with error at most ϵ as required by PAC learning. □

B Proof of Theorem 1

In this section we will give a complete proof of 1. We will keep this self contained and repeat necessary content from the main paper.

As discussed before, our proof follows by construction, that is, we show that for each TM A of size s, there exists parameters Θ_{rc} and Θ_{head}, such that, when Algorithm 1 is run with these parameters:

- *Memorization*: for the first $m + 1$ steps, when Θ_{mem} is trained with SGD, the gradients assist with memorizing the training set in the values of W
- *Computation*: given the memorized training set, the RCNN computes the roll-out of A with the input tape having the training set and the test example giving the prediction of A on the test example as the output.

We then show that we can choose the parameters Θ_{rc} and Θ_{head} in our construction such that for all TM A of size s belong to a fixed finite set of size $O(s)$ that can be constructed with knowledge of only s.

Let us restate Lemma 1 (more formally):

Lemma 2 (Restatement of Lemma 1)**.** *There exists constants c_1', c_2', c_3' such that: for any $d, s, m, t \in \mathbb{N}$, there exists a fixed initialization set $\mathcal{U}_s$ of size $c_1's$ where: for any (s, m, t)-computable learner A, there exists $\Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}}$ such that for all $\mathcal{S} \in (\mathcal{X}_d, \mathcal{Y})^m$, Algorithm 1 run on $\mathcal{S}$ with $l = c_2'(t + m + d)$, $\eta = 1/54$, and initialization set $\mathcal{U}_s$, satisfies:*

1. *Memorization: For $1 \leq i \leq m + 1$, $W_{ab}^{(i)} = \begin{cases} 3^{-1} y^{(a)} x_b^{(a)} & \text{if } a < i, b \leq d \\ 3^{-1} y^{(a)} & \text{if } a < i, b = d + 1. \\ 0 & \text{otherwise.} \end{cases}$*
2. *Computation: For all $x \in \{\pm 1\}^d$, $f\left(\begin{bmatrix} x \\ 1 \end{bmatrix}; W^{(m+2)}, \Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}}\right) = A(\mathcal{S})[x]$.*

Using the above lemma (part 2), we know that there exists parameters $\Theta_{\mathsf{rc}}, \Theta_{\mathsf{head}}$ that allow for our training pipeline to output exactly the function (say f) that A learns on training set $\mathcal{S}$. Note that our construction implements the underlying learning algorithm A and is able to provide this guarantee for all training sets $\mathcal{S}$ simultaneously. Thus, this implies that $\mathrm{err}_{\mathcal{D}}(f) = \mathrm{err}_{\mathcal{D}}(A, \mathcal{S})$ for any distribution $\mathcal{D}$. The chance that random initialization in Algorithm 1 will find these parameters is at least s^{-cs^2} for fixed some $c > 0$ since there are $\approx 10^4 s^2$ parameters and each parameter has $c_1's$ potential values. This proves Theorem 1.

Organization . In the remaining section we will prove Lemma 2. We will first describe how our construction will interpret the data values in $O(\log(s))$ precision (B.1), and the type of TMs we will consider. We will then describe the exact function of the RCNN layer and its corresponding Jacobian on the inputs (B.2). We will then show how SGD uses this functionality to memorize the training

examples (B.3). Finally, we show that the desired functionality of the RCNN layer can be achieved by a 5-layer NN with $O(s^2)$ parameters (B.4). [8]

B.1 Data representation and TM modifications

B.1.1 Data

In order to implement the required functionality of the RCNN layers, we will need $O(\log s)$ bit precision for each input to store relevant information. We work in base 3 to allow for storing $\{-1, 0, 1\}$ uniquely in each bit. More formally, the representation of each input/output in the network will have bit precision $\log(s) + 4$. Our construction will ensure that the inputs/outputs at each point are exactly representable with this bit precision in the following form: each input/output $a = \sum_{i=0}^{\log_2(s)+4} a(i)3^{-i}$ where $a(i) \in \{-1, 0, 1\}$ for $i \in \{0, 1, \ldots, \log_2(s) + 4\}$. The indices will encode different functionalities as follows:

- *Index 0* indicates whether the input is part of the padding (1) or not (0).
- *Index 1* indicates the content on the tape: blank symbol (0) or ± 1.
- *Index 2* indicates whether the current phase in the training is memorization (-1), computation (1), or unknown (0).
- *Index 3* indicates the presence of the head: 1 implies head is present, and 0 indicates no head.
- *Index* $4, \ldots, \log_2(s) + 4$ indicate the state of the TM $\{0, 1\}^{\log(s)}$, and are only non-zero when the head is set, that is, bit $3 \neq 0$.

For ease of presentation, let us define $\mathsf{val} : \mathbb{R} \times \mathbb{Z}_{\geq 0} \to \{-1, 0, 1\}$ as the function that given input a and index i, extracts bit i, that is, $a(i)$ if $a = \sum_{j=0}^{\log(s)+4} a(j)3^{-j}$ and is undefined otherwise. We also define $\mathsf{state} : \mathbb{R} \to \mathbb{R}$ which given input a and index i, extracts the value corresponding to the state, that is $\sum_{j=4}^{\log(s)+4} a(j)3^{-j+4}$. Let us define the set of possible values satisfying the above by $\mathcal{V}_s := \left\{\sum_{i=1}^{\log(s)+4} a(i)3^{-i} : a(i) \in \{-1, 0, 1\}\right\}$.

B.1.2 Turning Machines (TMs)

Let us formally define one-tape TMs,

Definition 6 (Turing machine)**.** *A Turing machine is defined as a tuple $\langle Q, \Gamma, \delta, F\rangle$ where Γ is a finite non-empty set of tape alphabet symbols, Q is a finite set of states, $F \subseteq Q$ is the final state indicating accept or reject, and $\delta : Q \times \Gamma \to Q \times \Gamma \times \{0, 1\}$ is the transition function where $0, 1$ are left and right shifts. Given input $x \in \Gamma^*$ put in the start of the tape, let $\mathcal{TM}_{\langle Q,\Gamma,\delta,F\rangle}(x) \in \{0, 1\}$ denote the output the TM produces if it halts on x. We denote the size of the TM s by the number of states, that is, $s = |Q|$.*

For technical ease, our algorithm will be competitive with all TMs of size s with the following modifications:

- *States*: We do a 1-1 mapping of state space Q of size s to $\{\sum_{i=4}^{\log(s)+4} a_i 3^{-i} | a_i \in \{-1, 1\}\}$ with 0 being the start state.
- *Single tape*: Instead of assuming 2 tapes with the training set on one tape and the test example on another, we will assume that they are concatenated onto one tape, followed by the working tape.
- *2D tape instead of 1D tape*: TM will run on a 2D tape with functionality for up, down, left, right, and no move. This implies that the transition function will have the following form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \downarrow, \leftarrow, \rightarrow, \times\}$. The 2D tape allows us to have markers for the number of samples and data dimension without requiring that in the state space.
- *First step*: We assume that the machine's first step is to not move and not change the tape symbol. This is useful for starting the communication protocol.

[8] In the main submission, we erroneously wrote s instead of s^2 parameters. We have corrected this in the current version.

- *XOR input*: In order to compute $A(\mathcal{S})[x]$ for $\mathcal{S} = \{(x^{(i)}, y^{(i)})_{i=1}^{m}\}$, we will have a 2D matrix of size $(m+1) \times (d+1)$ with the $m+1$th row containing $[x^\top \ 1]$, and row i ($\leq m$) containing $y^{(i)} \left[(x \circ x^{(i)})^\top \ y^{(i)}\right]$ where $\circ$ is the coordinate-wise dot product.
- *Halting position*: TM halts with output on the left-top corner of the tape and the rest of the input tape set to blank. Working tape can have any value.

Standard reductions show that two tape TMs can be implemented using 1 tape TMs. It is not hard to see that the TM with the 2D tape can implement any TM on the 1D tape. Not moving on the first step can be made possible by adding a single additional state with a path to the starting state. As for the XOR input tape, this can be converted to the original tape by adding extra states to multiply each coordinate of x down the column and multiplying $y^{(i)}$ across the row. Lastly, we can use two additional states to make the output 0 on the entire input tape except the left-top corner with the output. All these conversion causes a blow up of a constant number in states and polynomial in m, d extra runtime. We skip the details here as these follow from standard reductions of TMs, and assume from now on that our TM has the above form.

B.2 RCNN functionality

Here we will describe the exact function the RCNN layer (f_{Conv2D}) implements and its Jacobian. We will describe how to convert this into a 5-layer NN in the subsequent sections.

Our RCNN layers take in 3×3 grid around each input coordinate with the coordinate as the center. Our construction ensures that each input/output coordinate has representation as above. For the corner coordinates, we consider padding $p = 1/6$ so it is outside our bit representation. This allows it to be identified distinctly from any value of the input/output.

Now, we would like for all $X = \begin{pmatrix} x_{-1,-1} & x_{-1,0} & x_{-1,1} \\ x_{0,-1} & x_{0,0} & x_{0,1} \\ x_{1,-1} & x_{1,0} & x_{1,1} \end{pmatrix}$ such that each entry is in the set $\mathcal{V}_s$, the function computed by $\bar{f}_{\mathsf{Conv2D}}$ has three modes depending on the value of index 2 corresponding to the phase: *message passing and memorization* ($\bar{f}_{\mathsf{msg}}$), *computation* ($\bar{f}_{\mathrm{TM}}$), and *identity pass through*.

$$\bar{f}_{\mathsf{Conv2D}}(X) = \begin{cases} \bar{f}_{\mathsf{msg}}(X) & \text{if } \underbrace{\mathsf{val}(x_{0,0}, 2) = 0}_{\text{is phase unknown?}}, \\ \bar{f}_{\mathrm{TM}}(X) & \text{if } \underbrace{\mathsf{val}(x_{0,0}, 2) = 1}_{\text{is compute phase?}}, \\ x_{0,0} & \text{otherwise.} \end{cases}$$

Here, $\bar{f}_{\mathsf{msg}}$ runs the message passing protocol which identifies the correct phase and then broadcasts it to all inputs. It also assists with memorization, by identifying the location to memorize and ensuring that the gradients are zero for all non-memorizing coordinates. Let us formally describe the *message passing and memorization* functionality:

$$\bar{f}_{\mathsf{msg}}(X) = \begin{cases} \underbrace{-3^{-2}}_{\text{memory phase}} & \text{if } \underbrace{x_{0,0} = 0}_{\text{not memorized?}} \text{ and } \underbrace{(x_{-1,0} = p \text{ or } \mathsf{val}(x_{-1,0}, 1) \neq 0)}_{\text{is first row or is above row memorized?}} \\ \underbrace{-3^{-2}}_{\text{memory phase}} & \text{if } \underbrace{\mathsf{val}(x_{-1,0}, 2) = -1 \text{ or } \mathsf{val}(x_{1,0}, 2) = -1}_{\text{are below or above coordinates in memory phase?}} \\ x_{0,0} + \underbrace{3^{-2}}_{\text{compute phase}} + \underbrace{3^{-3}}_{\text{head}} & \text{if } \underbrace{\mathsf{val}(x_{0,0}, 1) \neq 0 \text{ and } x_{0,1} = 1 \text{ and } x_{1,0} = p}_{\text{is row memorized and is coordinate bottom-left coordinate?}} \\ x_{0,0} + \underbrace{3^{-2}}_{\text{compute phase}} & \text{if } \underbrace{\mathsf{val}(x_{-1,0}, 2) = 1 \text{ or } \mathsf{val}(x_{1,0}, 2) = 1 \text{ or } \mathsf{val}(x_{0,-1}, 2) = 1 \text{ or } \mathsf{val}(x_{0,1}, 2) = 1}_{\text{are any of the neighbouring coordinates in compute phase?}} \\ x_{0,0} & \text{otherwise.} \end{cases}$$

The invariant that is maintained during training is that samples are memorized row wise in the weights of W. Since the weights are 0 for rows that are yet to be memorized, we can identify the row to memorize (first if condition). Similarly, once all rows are memorized, it can be identified at the last row and computation phase can begin with the head being assigned (third if condition). See Figure

3for a visual explanation of this. Post identification, our message passing protocol essentially lets the other entries set the phase themselves (second and fourth if conditions). In the memorization phase, this is supplemented with zeroing the output (and the gradient) of any irrelevant coordinate.

Now we are ready to describe the *TM Roll-out Functionality*. Once the compute phase is established and the head is assigned. Observe that our TM does not move in the first step in order to ensure that the compute phase message is broadcast one step ahead of the movement of the TM.

$$\bar{f}_{\text{TM}}(X) = \begin{cases} \underbrace{\gamma(x_{0,0})[1]/3}_{\text{new tape value}} + \underbrace{3^{-2}}_{\text{compute phase}} + \underbrace{\mathbb{1}[\gamma(x_{0,0})[2] = \times](3^{-3} + \gamma(x_{0,0})[0]3^{-4})}_{\text{update head and state if it is not moving}} & \text{if } \underbrace{\mathsf{val}(x_{0,0}, 3) = 1}_{\text{is head?}} \\ x_{0,0} + \underbrace{\mathbb{1}[\gamma(x_{1,0})[2] = \uparrow](3^{-3} + \gamma(x_{1,0})[0]3^{-4})}_{\text{update head and state if it is moving to the current coordinate}} & \text{if } \underbrace{\mathsf{val}(x_{1,0}, 3) = 1}_{\text{is head below?}} \\ x_{0,0} + \underbrace{\mathbb{1}[\gamma(x_{-1,0})[2] = \downarrow](3^{-3} + \gamma(x_{-1,0})[0]3^{-4})}_{\text{update head and state if it is moving to the current coordinate}} & \text{if } \underbrace{\mathsf{val}(x_{-1,0}, 3) = 1}_{\text{is head above?}} \\ x_{0,0} + \underbrace{\mathbb{1}[\gamma(x_{0,1})[2] = \leftarrow](3^{-3} + \gamma(x_{-1,0})[0]3^{-4})}_{\text{update head and state if it is moving to the current coordinate}} & \text{if } \underbrace{\mathsf{val}(x_{0,1}, 3) = 1}_{\text{is head on the right?}} \\ x_{0,0} + \underbrace{\mathbb{1}[\gamma(x_{0,-1})[2] = \rightarrow](3^{-3} + \gamma(x_{0,-1})[0]3^{-4})}_{\text{update head and state if it is moving to the current coordinate}} & \text{if } \underbrace{\mathsf{val}(x_{0,-1}, 3) = 1}_{\text{is head on the left?}} \\ x_{0,0} & \text{otherwise.} \end{cases}$$

for $\gamma(x) = \delta(\mathsf{state}(x), \mathsf{val}(x, 1))$ is the output of the transition matrix for the current state, head, and tape contents. Here index 0 is the new state, index 1 is the new tape contents, and index 2 is the direction of head movement. In the above functionality, the checks identify where the head is and update the TM based on the current state and tape contents. For the coordinates not adjacent to the head, the function just passes through the tape value. This implements one step of the TM.

Finally, in order to ensure that gradients are passed through during memorization, we require the gradients to satisfy,

$$\nabla_{x_{i,j}} \bar{f}_{\mathsf{Conv2D}}(X) = \begin{cases} 1 & \text{if } i = j = 0 \text{ and } \underbrace{\mathsf{val}(x_{0,0}, 2) = -1}_{\text{is memory phase?}} \\ 1 & \text{if } i = j = 0 \text{ and } \underbrace{\mathsf{val}(x_{0,0}, 2) = 0}_{\text{is phase unknown?}} \text{ and } \underbrace{x_{0,0} = 0}_{\text{not memorized?}} \text{ and } \underbrace{(x_{-1,0} = p \text{ or } \mathsf{val}(x_{-1,0}, 1) \neq 0)}_{\text{is first row or is above row memorized?}} \\ 0 & \text{otherwise.} \end{cases}$$

Observe that, the above formulation act as a filter, to allow gradients to pass through only when the convolution starts the memorization, and as long as the memorization continues. This allows us to pass gradients down to only the row of W that is memorizing the example. We will explain this in more detail in the next section.

Finally, we want $\bar{f}_{\mathsf{head}} : \mathbb{R} \to \mathbb{R}$ to implement the following function

$$\bar{f}_{\mathsf{head}}(x) = \begin{cases} -1 & \text{if } x \leq -1/6 \\ 18(x + 3^{-2}) & \text{if } x \in (-1/6, 0) \\ -18(x - 3^{-2}) & \text{if } x \in (0, 3^{-2}) \\ 18(x - 3^{-2}) & \text{if } x \in (3^{-2}, 1/6) \\ 1 & \text{otherwise.} \end{cases}$$

The head operation works as a truncation to remove all the irrelevant bits in our output.

B.3 Training via SGD

Let us now show how our construction performs memorization via SGD. In order to compute the gradient update, it will be helpful to define some new notation. For a matrix $X \in \mathbb{R}^{d_1 \times d_2}$, define a grid extractor function $\mathsf{grid} : X \times [d_1] \times [d_2] \to \mathbb{R}^{3\times 3}$ such that $\mathsf{grid}(X, i, j)$ outputs the 3×3 sub-matrix of X centered at i, j with padding $p = 1/6$ at the edges. We will also define $\bar{f}_{\mathsf{ConvLayer}}$ as the application of $\bar{f}_{\mathsf{Conv2D}}$ to the entire input, that is, for input X, $\bar{f}_{\mathsf{ConvLayer}}(X)_{i,j} = \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(X, i, j))$ for all i, j.

Figure 3: *(Left, Center)* $\bar{f}_{\mathsf{Conv2D}}$ can identify the row to memorize in two ways: (left) At iteration 1, check if the current coordinate is 0 and the coordinate above is padding, and (center) at iteration $i \leq m+1$, check if current coordinate is 0 and coordinate above has bit 1 set, that is, has value $\pm 1/3$. Subsequently, $\bar{f}_{\mathsf{Conv2D}}$ can update the coordinate value by adding -3^{-2} to signal memorization phase to the neighbors. *(Right)* $\bar{f}_{\mathsf{Conv2D}}$ can identify when memorization is done and set the head appropriately to start the TM roll-out: The left-bottom corner coordinate of the input (ignoring the padding) can identify its global position using the pattern around it, and check if it has memorized (by checking bit 1). Subsequently, $\bar{f}_{\mathsf{Conv2D}}$ can update the coordinate value by adding $3^{-2} + 3^{-3}$ to signal computation phase to the neighbors and assign head location.

Lemma 3 (Forward pass and backward gradient calculation)**.** *For inputs $X \in \mathbb{R}^{(m+1)\times(d+1)}$, if there exists $1 \leq \tau \leq m+1$, such that $X_{i,j} \in \{\pm 3^{-1}\}$ for all $i < \tau, j \in [d+1]$ and $X_{i,j} = 0$ for all $i \geq \tau, j \in [d+1]$, then*

- $\bar{f}^{(l)}_{\mathsf{ConvLayer}}(Z)_{a,b} = -3^{-2}$ *for all* $a \in [m+1], b \in [d+1]$
- *For all* $i \in [m+1], j \in [d+1]$, $\nabla_{X_{i,j}} \bar{f}^{(l)}_{\mathsf{ConvLayer}}(Z)_{a,b} = \begin{cases} 1 & \text{if } a = i = \tau, b = j \\ 0 & \text{otherwise} \end{cases}$

where $Z = [X \ \mathbb{1}_{(m+1)\times l}]$.

Proof. Suppose the condition is satisfied for τ, then we will prove by induction, the following claim: for $t > 0$, for all $b \in [d+1]$

$$\bar{f}^{(t)}_{\mathsf{ConvLayer}}(Z)_{a,b} = \begin{cases} 3^{-2} & \text{if } |a-\tau| < t \\ Z_{a,b} & \text{otherwise.} \end{cases}$$

Note that this holds for $t = 1$:

$$\begin{aligned} \bar{f}_{\mathsf{ConvLayer}}(Z)_{\tau,b} &= \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(Z,\tau,b)) \\ &= \bar{f}_{\mathsf{msg}}(\mathsf{grid}(Z,\tau,b)) && \textit{(since } X_{\tau,b} = 0\textit{)} \\ &= -3^{-2} && \textit{(first if condition of } \bar{f}_{\mathsf{msg}}\textit{)} \end{aligned}$$

For all $a \neq \tau$,

$$\begin{aligned} \bar{f}_{\mathsf{ConvLayer}}(Z)_{a,b} &= \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(Z,a,b)) \\ &= \bar{f}_{\mathsf{msg}}(\mathsf{grid}(Z,a,b)) && \textit{(since } \mathsf{val}(X_{a,b},2) = 0\textit{)} \\ &= Z_{a,b} && \textit{(no condition (1-4) is true of } \bar{f}_{\mathsf{msg}}\textit{)}. \end{aligned}$$

Let us assume it holds for t, and prove for $t+1$. For a such that $|a-\tau| < t$, since memory phase is set, $\bar{f}_{\mathsf{Conv2D}}$ behaves like an identity match. For a such that $|a-\tau| = t$, we have

$$\begin{aligned} \bar{f}^{(t+1)}_{\mathsf{ConvLayer}}(Z)_{a,b} &= \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(\bar{f}^{(t)}_{\mathsf{ConvLayer}}(Z),a,b)) \\ &= \bar{f}_{\mathsf{msg}}(\mathsf{grid}(\bar{f}^{(t)}_{\mathsf{ConvLayer}}(Z),a,b)) && \textit{(since } \mathsf{val}(X_{a,b},2) = 0\textit{)} \\ &= 3^{-2} && \textit{(second if condition of } \bar{f}_{\mathsf{msg}}\textit{)}. \end{aligned}$$

Similar to the base case argument, for all a outside this band, none of the if conditions are satisfied and it acts as a pass through. Since $l > m+1$, we get the first part of the above lemma.

For the second part, we will again prove by induction on depth of RCNN. For $l' = 1$, we have

$$\begin{aligned}
&\nabla_{X_{i,j}} \bar{f}_{\mathsf{ConvLayer}}(Z)_{a,b} \\
&= \nabla_{X_{i,j}} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(Z,a,b)) \\
&= \sum_{a',b' \in \{-1,0,1\}} \nabla_{a',b'} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(Z,a,b)) \cdot \nabla_{X_{i,j}} Z_{a+a',b+b'} && \textit{(by chain rule)} \\
&= \nabla_{0,0} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(Z,a,b)) \cdot \nabla_{X_{i,j}} Z_{a,b} && \textit{(by gradient construction of } \hat{f}_{\mathsf{Conv2D}}\textit{)} \\
&= \nabla_{0,0} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(Z,a,b)) \cdot \mathbb{1}[a = i \wedge b = j].
\end{aligned}$$

Observe that, by if condition 2 of gradient of $\bar{f}_{\mathsf{Conv2D}}$, for $a = \tau$, the gradient is 1. For all other coordinates, none of the if conditions are satisfied and hence the gradients are 0.

Let us assume for $l' < l$ and prove for $l'+1$.

$$\begin{aligned}
\nabla_{X_{i,j}} \bar{f}^{(l'+1)}_{\mathsf{Conv2D}}(Z)_{a,b} &= \nabla_{0,0} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(f^{(l')}_{\mathsf{Conv2D}}(Z),a,b)) \cdot \nabla_{X_{i,j}} \bar{f}^{(l')}_{\mathsf{Conv2D}}(Z)_{a,b} \\
&= \nabla_{0,0} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(f^{(l')}_{\mathsf{Conv2D}}(Z),a,b)) \cdot \mathbb{1}[a = i = \tau \wedge b = j].
\end{aligned}$$

From the induction before, we have that $\bar{f}^{(l')}_{\mathsf{ConvLayer}}(Z)_{\tau,b} = 3^{-2}$ for all $l' > 0$, therefore by the gradient condition, we have $\nabla_{0,0} \bar{f}_{\mathsf{Conv2D}}(\mathsf{grid}(f^{(l')}_{\mathsf{Conv2D}}(Z),\tau,b)) = 1$ giving us the desired result. □

Let us now compute the back-propagated gradient for each input coordinate through the RCNN layers. Now we are ready to prove part 1 of Lemma 2 with $\bar{f}_{\mathsf{Conv2D}}, \bar{f}_{\mathsf{head}}$. We restate it as,

Lemma 4. *For $\tau < m+2$ of Algorithm 1 with RCNN layers computed by $\bar{f}_{\mathsf{Conv2D}}$, head computed using $\bar{f}_{\mathsf{head}}$, and $\eta = 1/54$, we have:*

$$W^{(\tau+1)}_{ab} = \begin{cases} 3^{-1} y^{(a)} x^{(a)}_b & \textit{if } a < \tau, b \le d \\ 3^{-1} y^{(a)} & \textit{if } a < \tau, b = d+1 \\ 0 & \textit{otherwise.} \end{cases}.$$

Proof. We will prove by induction. Observe that, this trivially holds for $\tau = 0$ since $W^{(1)} = {}_{(m+1)\times(d+1)}$. Now let us assume it hold for $m+1 > t > 0$, we will show that it holds for $t+1$.

Since $x^{(t+1)} \in \{\pm 1\}^d$, we have $X^{(t+1)} = W^{(t+1)} \mathrm{diag}\left(\left[{x^{(t+1)}}^\top \; 1\right]^\top\right)$ is such that its first t rows have entries in $\{\pm 3^{-1}\}$ and rest of the rows have entries 0. This satisfies the condition of Lemma 3 with $\tau = t+1$, giving us the following:

- $\bar{f}^{(l)}_{\mathsf{ConvLayer}}(Z^{(t+1)})_{a,b} = -3^{-2}$ for all $a \in [m+1], b \in [d+1]$
- For all $i \in [m+1], j \in [d+1]$, $\nabla_{X_{i,j}} \bar{f}^{(l)}_{\mathsf{ConvLayer}}(Z^{(t+1)})_{a,b} = \begin{cases} 1 & \text{if } a = i = t+1, b = j \\ 0 & \text{otherwise} \end{cases}$

where $Z^{(t+1)} = [X^{(t+1)} \; \mathbb{1}_{(m+1)\times l}]$.

Now let us compute the update step. Observe that $\bar{f}_{\mathsf{head}}(-3^{-2}) = 0$, therefore $f(x^{(t+1)}; W^{(t+1)}, \bar{f}_{\mathsf{Conv2D}}, \bar{f}_{\mathsf{head}}) = 0$. Now, using chain rule and the above observations, we can compute the full desired gradient,

$$\begin{aligned}
&\nabla_{W^{(t+1)}_{a,b}} \ell\left(f(x^{(t+1)}; W^{(t+1)}, \bar{f}_{\mathsf{Conv2D}}, \bar{f}_{\mathsf{head}}), y^{(t+1)}\right) \\
&\quad = -(y^{(t+1)} - f(x^{(t+1)}; W^{(t+1)}, \bar{f}_{\mathsf{Conv2D}}, \bar{f}_{\mathsf{head}})) \nabla_{W^{(t+1)}_{a,b}} f(x^{(t+1)}; W^{(t+1)}, \bar{f}_{\mathsf{Conv2D}}, \bar{f}_{\mathsf{head}}) \\
&\quad = -18 y^{(t+1)} \sum_{i,j} \nabla_{Z^{(t+1)}_{a,b}} \bar{f}^{(l)}_{\mathsf{rc}}(Z^{(t+1)}) \cdot \nabla_{W^{(t+1)}_{a,b}} \bar{f}_{\mathsf{mem}}(x^{(t+1)}; W^{(t+1)})_{i,j} \\
&\quad = -18 y^{(t+1)} \mathbb{1}[a = i] z^{(t+1)}_b.
\end{aligned}$$

where $z^{(t+1)} = [x^{(t+1)^\top}\ 1]^\top$. Note that the above follows from observing that the gradient of $\bar{f}_{\mathsf{head}}$ at -3^{-2} is 18 and the properties of the gradient of $\bar{f}_{\mathsf{Conv2D}}$ from above.

With $\eta = 1/54$, we get that, $W^{(t+2)}$ satisfies the induction argument. This completes the proof. $\square$

Now once the memorization phase is over, the input to $\bar{f}_{\mathsf{Conv2D}}$ will have on the tape, the training samples (in the XOR form with the current input) along with a clean input at the end (since $W_{m+1}^{(m+2)} = \mathbb{1}_{d+1}$ due to our dummy sample). At this stage, our $\bar{f}_{\mathsf{Conv2D}}$ will implement $\bar{f}_{\mathsf{msg}}$ to set the head (at $(m+1, d+1)$ position) using the third if condition. Once this is set, it will broadcast (using if condition 4) while the computation starts on those positions with computation flag set. Since our first step does not involve any movement of the head, the compute phase message passing protocol is always ahead of the head movement. Once the compute phase is set, $\bar{f}_{\mathsf{Conv2D}}$ starts implementing $\bar{f}_{\mathrm{TM}}$ which computes the roll-out of the TM. Finally, once the RCNN layers have been applied $\geq l + m + 1 + d$ times, the TM will terminate. The final layer output prior to $\bar{f}_{\mathsf{head}}$ will be such that the top-left corner will have output set to either $\pm 3^{-1} + 3^{-2}$ (bit 1 will be set to ± 1, bit 2 will be set to 3^{-2}, and the rest of the bits being set to the final state), and the rest will be set at 3^{-2}. $\bar{f}_{\mathsf{head}}$ will ensure that this is truncated to ± 1 accordingly. The rest of the coordinates in the final sum will be 0 since they will have value exactly 3^{-2} and will be truncated to 0 by $\bar{f}_{\mathsf{head}}$. This proves the second part of Lemma 2.

Last we need to show that $\bar{f}_{\mathsf{Conv2D}}$ and $\bar{f}_{\mathsf{head}}$ can be constructed using NNs. It is not hard to see that $\bar{f}_{\mathsf{head}}$ can be constructed using a one layer ReLU network with at most 6 ReLUs. To construct $\bar{f}_{\mathsf{Conv2D}}$ requires not only function value matching on $\mathcal{V}_s$, but also gradient matching. The subsequent section proves a general representation representation result that allows us to do the same.

B.4 Constructing $\bar{f}_{\mathsf{Conv2D}}$ as a NN

Here show how the $\bar{f}_{\mathsf{Conv2D}}$ can be implemented by 5-layer neural network in a way that keeps the parameters and their choices bounded by $\mathsf{poly}(s)$. We show a general result that takes any function and gradient specification on discrete domains and converts it into a 5-layer network. Corollary 3 can be directly applied to our construction of $\bar{f}_{\mathsf{Conv2D}}$ with $\mathcal{X} = \mathcal{Y}$ being all rational numbers with precision $O(\log(s))$ in base 3. This lemma gives us a construction of a 5-layer net where each parameter lies in a set of size $O(s)$. This set can be computed based on knowing the input domain, which is known a priori.

B.4.1 Representing discrete functions with gradient pass-throughs

To implement the discrete functions used in the main construction, we make use of Lemma 5, which we state and prove in this section. It constructs a 5-layer fully-connected neural network whose values match those of an arbitrary multivariate real function on a finite domain, while allowing gradients to pass through for an arbitrary choice of inputs. The basic idea for the construction (build a basis of indicator functions, and enumerate over all possible input-output pairs) is not new, but we could not find an explicit theorem satisfying our additional requirements.[9] We hope this *function approximation lemma with custom gradients* will be useful beyond the scope of this paper. Specifically, beyond typical universal function approximation results, we need:

- Simultaneous control over the function values (all d_{out} coordinates at all $|\mathcal{X}|$) and Jacobians (all $d_{\mathrm{in}} \times d_{\mathrm{out}}$ partial derivatives at all $|\mathcal{X}|$ points), so that the gradient signal from SGD can propagate through the recurrent computations to the memory layer in a controlled way.
- A bound on the number of distinct values the weights can take, so that we can analyze the probability that an i.i.d. random initialization scheme finds the desired weights.

To simplify notation, we will use a single matrix parameter $W^{d_{\mathrm{out}} \times (d_{\mathrm{in}}+1)}$ to parameterize an affine map $x \to W[x^\top\ 1]^\top$. Furthermore, we will overload notation and use the notation $W : \mathbb{R}^{d_{\mathrm{in}}} \to \mathbb{R}^{d_{\mathrm{out}}}$ to represent the same affine map.

[9] A proof sketch for the indicator construction can be found in (Nielsen, 2015). A quantitative version, without the additional considerations in this work, appears as Lemma B.8 in (Edelman et al., 2021), which requires a function representation lemma with robustness and weight norm bounds, for the purpose of obtaining non-vacuous generalization guarantees arising from sparsified inputs.

Lemma 5. *Let $\mathcal{X} \subset \mathbb{R}^{d_{\text{in}}}, \mathcal{Y} \subset \mathbb{R}^{d_{\text{out}}}, \mathcal{G} \subset \mathbb{R}$ be such that $|\mathcal{X}|, |\mathcal{Y}|, |\mathcal{G}| < \infty$. Let $f : \mathcal{X} \to \mathcal{Y}$, and let $g : \mathcal{X} \times [d_{\text{out}}] \times [d_{\text{in}}] \to \mathcal{G}$. Then, there exists a 3-layer ReLU network (letting σ denote the entrywise ReLU function), with fully-connected layers specifying affine functions $W_1 : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_1}, W_2 : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}, W_3 : \mathbb{R}^{d_2} \to \mathbb{R}^{d_{\text{out}}}$, such that:*

$$f_{\text{net}}(x) := (W_3 \circ \sigma \circ W_2 \circ \sigma \circ W_1)(x) = f(x), \quad \forall x \in \mathcal{X}; \tag{1}$$

$$\frac{\partial}{\partial x_j}(W_3 \circ \sigma \circ W_2 \circ \sigma \circ W_1)(x)_i = g(x, i, j), \quad \forall x \in \mathcal{X}, i \in [d_{\text{out}}], j \in [d_{\text{in}}]. \tag{2}$$

The intermediate dimensions satisfy

$$d_1 = 10 d_{\text{in}} |\mathcal{X}|, \quad d_2 = 6 d_{\text{in}} d_{\text{out}} |\mathcal{X}|. \tag{3}$$

Proof. The construction consists of 3 steps:

(i) Build a univariate indicator (a bump function $\psi_{x_0} : \mathbb{R} \to \mathbb{R}$) for each input coordinate's domain $\mathcal{X}_i$, whose gradient is always 1 in the "bump" region, using a linear combination of 5 ReLU activations.

(ii) For any $x \in \mathcal{X}$ and $y, \gamma \in \mathbb{R}$, build multivariate indicators $\psi^{(\gamma)}_{x_0,y} : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}$, using a linear combination of 3 univariate indicators. We ensure that $\psi^{(\gamma)}_{x_0,y}$ has gradient γe_i.

(iii) Assemble the function f piece-by-piece: sum over indicators for each $(x, f(x), i \in [d_{\text{out}}])$, using the appropriate indicators to match every desired partial derivative $g(x, i, j)$.

We will implement all of the indicators from part (i) using linear combinations of ReLU activations $W_1 \circ \sigma \circ W_1'$, then the sum the indicators from part (ii) using $W_2' \circ \sigma \circ W_3$. The final network will simply compress the two intermediate affine functions into one: $W_2 = W_1' \circ W_2'$.

Part (i). Let $\Delta \leq \frac{1}{10} \min\left(1, \min_{x,x' \in \mathcal{X}, i \in [d_{\text{in}}]} |x_i - x_i'|\right)$. For a given $x_0 \in \mathbb{R}$ and desired gradient $\gamma \in \{0, 1\}$, let $\psi^{(\gamma)}_{x_0}$ denote the unique continuous piecewise linear function $\psi : \mathbb{R} \to \mathbb{R}$ such that:

- $\psi(x) = 0$ for $x \in (-\infty, x_0 - 2\Delta]$ and $x \in [x_0 + 2\Delta, \infty)$.
- $\psi(x_0) = 1$, and $\psi'(x) = \gamma$ for $x \in (x_0 - \Delta, x_0 + \Delta)$.
- $\psi'(x)$ is constant on $[x_0 - 2\Delta, x_0 - \Delta]$ and $[x_0 + \Delta, x_0 + 2\Delta]$.

By our choice of Δ, for each $x_0, x \in \cup_{i \in [d_{\text{in}}]} \mathcal{X}_i$, $\psi^{(\gamma)}_{x_0}(x) = \mathbb{1}[x = x_0]$. In other words, ψ is an indicator for a unique real number that appears in any coordinate of $\mathcal{X}$. Furthermore, $\psi^{(\gamma)\prime}_{x_0}(x) = \gamma \times \mathbb{1}[x = x_0]$. When $\gamma = 1$, ψ lets the gradient pass through in the active region of the indicator; when $\gamma = 0$, ψ blocks the gradient at every input in the domain.

Furthermore, ψ consists of 5 linear regions, so it can be written as an affine function (i.e. linear combination, plus constant bias term) of 5 ReLU activations $W \circ \sigma \circ W'$, where $W : \mathbb{R} \to \mathbb{R}^5, W' : \mathbb{R}^5 \to \mathbb{R}$.[10] We construct W_1, W_1' by concatenating the indicators $\psi^{(\gamma)}_{x_0}(e_i^\top x)$ for each $i \in [d_{\text{in}}], \gamma \in \{0, 1\}, x_0 \in \mathcal{X}_i$, so that $W_1' : \mathbb{R}^{d_1} \to \mathbb{R}^{d_1'}$, where $d_1' = 2 d_{\text{in}} |\mathcal{X}|$, the number of indicators we have constructed.

Part (ii). We will build a multivariate indicator Ψ by summing per-coordinate univariate indicators, and checking that they sum to d_{in}. For a desired output y and partial derivative $\gamma \in \mathcal{G}$, let $\tau^{(\gamma)}_y$ denote the unique piecewise linear function $\tau : \mathbb{R} \to \mathbb{R}$ such that:

- $\tau(x) = 0$ for $x \in (-\infty, d_{\text{in}} - 2\Delta]$.

[10] A sketch of this construction: for each discontinuity ζ of the desired piecewise linear function ψ, place a ReLU activation $\sigma(x - \zeta)$; also, for a value ζ_0 less than all ζ, place one more ReLU activation $\sigma(x - \zeta_0)$. Solve a linear system in the coefficients to get each linear region to agree with ψ.

- $\tau(d_{\text{in}}) = y$, and $\tau'(x) = \gamma$ for $x \in [d_{\text{in}} - \Delta, \infty)$.
- $\tau'(x)$ is constant on $(d_{\text{in}} - 2\Delta, d_{\text{in}} - \Delta)$.

By our choice of Δ, the output of this function is $f(x_0)$ only when each summand is 1, which only happens when $x = x_0$. Otherwise, the input to τ is in the flat region, where $\tau(\cdot) = \tau'(\cdot) = 0$. In summary, τ is an indicator like ψ. It is slightly simpler to construct, since it only needs to implement a threshold function, and only needs to recognize that its input is d_{in} (rather than an arbitrary x_0). Since each τ has 3 linear regions, it can be built with an affine function of 3 ReLU activations.

Our network will construct $d_2' = 2d_{\text{in}}d_{\text{out}}|\mathcal{X}|$ of these indicators:

$$\tau_y^{g(x_0,i,j)} \quad \forall x_0 \in \mathcal{X}, i \in [d_{\text{in}}], j \in [d_{\text{out}}], y \in \{0, f(x_0)_j\}.$$

We set W_2' to be the concatenation of all of these indicators, so that $W_2' : \mathbb{R}^{d_2} \to \mathbb{R}^{d_2'}$.

Part (iii). Then, for each $x_0 \in \mathcal{X}$ and $j \in [d_{\text{out}}]$, with corresponding desired function values $f(x_0)_i$ and partial derivatives $g(x_0, i, j)$, we construct the indicator for a single output coordinate:

$$\Psi_{f,g,x_0,j}(x) := \underbrace{\sum_{i'=1}^{d_{\text{in}}} \tau_{f(x_0)_j \cdot \mathbb{1}[i'=1]}^{(g(x_0,i',j))} \left(\sum_{i=1}^{d_{\text{in}}} \underbrace{\psi_{(x_0)_i}^{(\mathbb{1}[i=i'])}(e_i^\top x)}_{\text{computed by } W_1 \circ \sigma \circ W_1'} \right)}_{\text{computed by } W_1 \circ \sigma \circ W_2 \circ \sigma \circ W_2'} \tag{4}$$

The intuition behind Equation 4 is the following: if we only cared about matching the function value f at all points in $\mathcal{X}$, it would suffice to use one indicator per $x_0 \in \mathcal{X}, j \in [d_{\text{out}}]$. However, we need to get every coordinate of the gradient correct. To implement this, we create d_{in} redundant indicators for each x_0, and sum over all of them, ensuring that the gradient is counted once per input coordinate, and the function value is counted once in total. Finally, W_3 is constructed by summing over all of the indicators from the previous part:

$$f_{\text{net}}(x)_j = \sum_{x \in \mathcal{X}} \Psi_{f,g,x_0,j}(x), \quad \forall j \in [d_{\text{out}}].$$

Lines 1 and 2 in the statement follow from the inline discussions above of properties of the indicator modules. □

Next, we will bound the size of the support of i.i.d. random initialization weights needed to construct the network in Lemma 5. To do this with fewer distinct values, we will make the following changes to the architecture:

- Split the layer W_2 into the composition of two affine layers $W_1' \circ W_1^+$, with intermediate dimension d_1', according to the above analysis.
- Similarly, split the layer W_3 into $W_2' \circ W_2^+$, with intermediate dimension d_2'.

Overall, this expands the 3-layer 2-ReLU architecture to a 5-layer 2-ReLU equivalent, changing the parameterization but not the class of representable functions. With this modified construction, we show that the unique nonzero matrix weights for the linear layers $W \in \{W_1, W_1', W_1^+, W_2', W_2^+\}$ lie in a bounded-size domain $\mathcal{U}(W)$, which only depends on $\mathcal{X}, \mathcal{Y}, \mathcal{G}$, not f, g. These $\mathcal{U}$ allow us to define the support of the random initialization distribution, and determine the probability of success. We state and prove the bounds on $|\mathcal{U}(\cdot)|$ below.

Lemma 6. *Let $W_1(f, g), W_1'(f, g), \ldots$ be the matrices arising from the modified construction from Lemma 5. Then, there exist finite sets $\mathcal{U}(W) \subset \mathbb{R}$ depending only on $\mathcal{X}, \mathcal{Y}, \mathcal{G}$, such that $\mathcal{U}(W) \cup \{0\}$ contains all elements $W(f, g)$, and:*

- $|\mathcal{U}(W_1)| \leq 4|\mathcal{X}_i| + 2$.
- $|\mathcal{U}(W_1')| \leq 12|\mathcal{X}_i|$.
- $|\mathcal{U}(W_1^+)| \leq 4$.

- $|\mathcal{U}(W_2')| \leq 4|\mathcal{G}|\,|\mathcal{Y}_i|$,
- $|\mathcal{U}(W_2^+)| = 1$.

where $\mathcal{X}_i := \{x_i : x \in \mathcal{X}, i \in d_{\text{in}}\}$ and $\mathcal{Y}_i := \{x_i : x \in \mathcal{Y}, i \in d_{\text{out}}\}$ denote the sets of possible input and output values.

Proof. First, notice that with a known, finite $\mathcal{X}$, there is a deterministic way to choose a sufficiently small Δ, and a sufficiently large ζ_0 (the lowest bias in the ReLU-to-piecewise linear constructions). We analyze the construction of each layer, and enumerate the possible nonzero weights and biases:

- It is clear from the construction that the linear coefficients in W_1 are in $\{0, 1\}$. Furthermore, for each x_0 occurring in any coordinate of an element in the domain $\mathcal{X}$, there are 5 bias terms: $-\zeta_0, -x_0 \pm \Delta, -x_0 \pm 2\Delta$.
- W_1' maps groups of 5 ReLU activations to the corresponding indicators ψ. The coefficients are each functions of a single x_0 and Δ; there are 6 coefficients (including one bias) per indicator, and 2 indicators ($\gamma \in \{0, 1\}$) per x_0.
- W_1^+ combines the indicators ψ to form the inputs to the ReLUs, which W_2' will use to form the indicators τ. These weights are again in $\{0, 1\}$, and biases are from the ReLU discontinuity locations: $-\zeta_0, -(d_{\text{in}} - 2\Delta), -(d_{\text{in}} - \Delta)$.
- W_2' forms the d_2' indicators τ, with 4 coefficients per indicator, depending on $\mathcal{G}, \mathcal{Y}, \Delta, d_{\text{in}}$.
- W_2^+ simply takes a summation over the indicators τ, so its coefficients are in $\{0, 1\}$.

□

We summarize the results in Lemma 6 with a looser corollary:

Corollary 3. *Let $\mathcal{X}, \mathcal{Y}, \mathcal{G}$ be known and finite, and let $\mathcal{X}_i = \mathcal{Y}_i$, $\mathcal{G} = \{0, 1\}$. Using the construction of $\mathcal{U}(\cdot)$ in Lemma 6, we can define a single set*

$$\mathcal{U} := \{0\} \bigcup_{W \in \{W_1, W_1', W_1^+, W_2', W_2^+\}} \mathcal{U}(W),$$

which contains all possible weights and biases in all layers of the ReLU network. The cardinality of $\mathcal{U}$ satisfies

$$|\mathcal{U}| \leq 24|\mathcal{X}_i| + 5.$$

Given this corollary, for our setting, $\mathcal{U}_s$ has size $24 \cdot 16 \cdot s + 5$, and is described above, can be constructed with knowledge of only s.

B.5 Proof of Corollary 2

Let M be the number of random restarts and N be the size of the validations set, Using standard concentration bounds, we know that with probability $1 - \delta/2$, the expected error $\text{err}_{\mathcal{D}}$ of classifiers for all $f_1, \ldots, f_M$ over the distributions is within $\sqrt{\frac{\log(2M/\delta)}{2N}}$ of the empirical error $\hat{\text{err}}_{\mathcal{D}}$. This implies that the error of the classifier $\hat{f}$ selected by our validation set satisfies

$$\begin{aligned}
\text{err}_{\mathcal{D}}(\hat{f}) &\leq \hat{\text{err}}_{\mathcal{D}}(\hat{f}) + \sqrt{\frac{\log(2M/\delta)}{2N}} \\
&\leq \min_{i \in [M]} \hat{\text{err}}_{\mathcal{D}}(f_i) + \sqrt{\frac{\log(2M/\delta)}{2N}} \\
&\leq \min_{i \in [M]} \text{err}_{\mathcal{D}}(f_i) + 2\sqrt{\frac{\log(2M/\delta)}{2N}}
\end{aligned}$$

By Theorem 1, we know that with probability $1 - Ms^{c_2 s^2}$, at least one (say f_1) of the $f_1, \ldots, f_M$ will satisfy the $\mathrm{err}_{\mathcal{D}}(f_1) \leq \mathrm{err}_{\mathcal{D}}(A, \mathcal{S})$. This gives us, with probability $1 - \delta/2 - Ms^{c_2 s^2}$, we have,

$$\mathrm{err}_{\mathcal{D}}(\hat{f}) \leq \mathrm{err}_{\mathcal{D}}(A, \mathcal{S}) + 2\sqrt{\frac{\log(2M/\delta)}{2N}}.$$

Setting M, N such that $2\sqrt{\frac{\log(2M/\delta)}{2N}} = \epsilon$ and $Ms^{-c_2 s^2} = \delta/2$, we get the desired result. $M = \delta s^{c_2 s^2}/2$ and $N = \frac{c_2 s^2 \log(s/2)}{8\epsilon^2}$.