A Universal Uniform Approximation Theorem for Neural Networks

Olivier Bournez
Institut Polytechnique de Paris, Ecole Polytechnique, LIX, 91128 Palaiseau Cedex, France

Johanne Cohen
Équipe GALaC, LISN, CNRS, Université Paris-Saclay, 91190 Gif-sur-Yvette, France

Adrian Wurm
Computer Science Institute, BTU Cottbus-Senftenberg, Cottbus, Germany

Abstract

We show the existence of a **fixed** recurrent network capable of approximating any computable function with arbitrary precision, provided that an encoding of the function is given in the initial input. While uniform approximation over a compact domain is a well-known property of neural networks, we go further by proving that our network ensures effective uniform approximation – simultaneously ensuring:

- **Uniform approximation in the sup-norm sense**, guaranteeing precision across the compact domain $[0,1]^d$;
- **Uniformity in the sense of computability theory** (also referred to as *effectivity* or *universality*), meaning the same network works for all computable functions.

Our result is obtained constructively, using original arguments. Moreover, our construction bridges computation theory with neural network approximation, providing new insights into the fundamental connections between circuit complexity and function representation.

Furthermore, this connection extends beyond computability to complexity theory. The obtained network is efficient: if a function is computable or approximable in polynomial time in the Turing machine model, then the network requires only a polynomial number of recurrences or iterations to achieve the same level of approximation, and conversely. Moreover, the recurrent network can be assumed to be very narrow, strengthening the link our results and existing models of very deep learning, where uniform approximation properties have already been established.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Computability; Theory of computation → Complexity classes; Computing methodologies → Neural networks

Keywords and phrases Models of computation, Complexity theory, Formal neural networks

Digital Object Identifier 10.4230/LIPIcs.MFCS.2025.29

Funding O. Bournez and J. Cohen received support from the Programme Gaspard Monge (PGMO). O. Bournez received support from ANR Project OTC ANR-24-CE48-0335. A. Wurm received partial support from the BTU Graduate Research School (Conference Travel Grant).

1 Introduction

Deep learning models have revolutionized machine learning. One of the best-known fundamental results in neural network theory is that they are universal approximators: for any continuous function, any compact domain, and any precision ϵ, there exists a neural network within distance at most ϵ from the function over this compact domain. Mathematically, uniform approximation states that the class of functions realized by neural networks is dense in the set of continuous functions over compact domains. This holds under minimal assumptions (a single hidden layer and a non-polynomial activation function suffice [15, 29]).

We relate the questions of uniform approximation to questions from computability and complexity theory. Due to space constraints, we take the editorial choice of expecting our readers to be MFCS attendees and to have knowledge in these fields. For other potential readers, we do not claim that our results provide practical tools for training very deep networks or large language models, etc. Pursuing such directions would require a different focus and a different work.

However, we believe that our results clarify the intrinsic difficulty of the associated problems. In particular, we think that the gap between practical challenges studied in deep learning and their theoretical counterparts in computability and complexity theory remains largely unexplored. We agree that some -but not all, including those mentioned above- of the proofs of uniform approximation can be considered constructive. However, we also believe our results help clarify the limits of constructivity in this context. Furthermore, our results demonstrate that a fixed, universal, neural network is sufficient to get uniform approximation, a result that has not been established yet.

The natural measures of neural network complexity include their depth (number of layers), size (number of neurons), and width (number of neurons per hidden layer). Neural networks can be viewed as a special class of circuits and described using tools from circuit complexity. Unlike Boolean circuits, they operate over real numbers, and each neuron acts as a gate applying an activation function to an affine weighted combination of its inputs. Many classical results in circuit complexity can thus be revisited in this context – an idea that dates back at least to [43, 52]. Associated questions, such as training, are consequently at least as hard as their counterparts in the Boolean circuit setting. The intractability of various problems for neural network problems – especially training – remains a highly active area of research; see, for example, the recent works [5, 8, 10, 17, 18, 19, 20, 22, 42].

Uniform approximation by various classes of networks is also a highly active research area in neural networks literature. As expected, it connects to how continuous functions can be approximated over a compact domain by different function classes. The width of a neural network naturally reflects the "complexity" of the functions being approximated, which can be measured in various ways, such as through the coefficients of their Taylor or Fourier series, their polynomial approximation via Weierstrass's theorem, or their decomposition via the Kolmogorov–Arnold theorem for multivariate functions. Rather than surveying these developments, we refer the reader to [4] for a recent overview.

In contrast, our focus takes an orthogonal perspective: uniform approximation by a *fixed* network – that is, *universal* uniform approximation.

We can also add that, in the neural networks literature, uniform approximation typically falls into two fundamental scenarios. The most classical approach considers networks with arbitrary width but a limited number of hidden layers – referred to here as **shallow and wide** networks. In many statements, including the above-mentioned historical [15, 29] arbitrary width is essential to achieve function density. The dual perspective involves networks with arbitrary depth but limited width, known as **deep narrow** networks, or sometimes *very deep learning* in opposition to *deep learning*.

We focus on this second scenario – networks with limited width and arbitrary depth. We prove that uniform approximation holds with a fixed unique network, of small width.

On the deep narrow scenario. As mentioned, we do not attempt a full survey and we briefly highlight why deep narrow networks have only recently gained attention. Increasing depth to boost performance is intuitive, but only recently became practical after [27]. The main obstacle was the use of gradient-based training: as depth grows – especially beyond hundreds of layers – gradients vanish [21], leaving only the final layers effectively updated during backpropagation.

A key breakthrough was the recent introduction of skip connections, which allow layers to connect beyond their immediate predecessor. Skip connections were popularized about nine years ago by Residual Networks (ResNets) introduced in [27], and by Highway Nets introduced in [51]. It has been shown that this structure allows ResNets, or some considered architectures, to be interpreted as discrete approximations of continuous-time dynamical systems, particularly through the forward Euler method. As a result, such networks inherit desirable properties from numerical methods, including stability, convergence, and robustness. It was later mathematically proven that many efficient models are, in fact, reformulations of discretisation schemes for ODEs. For example, following [39], *PolyNet* [58] can be viewed as an approximation to the backward Euler scheme for solving ODE $u_t = f(u)$. *Fractalnet* [36] follows a well-known Runge-Kutta scheme. *ResNet* [23] corresponds to a forward Euler discretization of a simple dynamical system.

This perspective has also inspired models such as Neural Ordinary Differential Equations [14], in which the dynamics evolve continuously and are governed by the network parameters: $\mathbf{x}'_t = \mathbf{f}(\theta, \mathbf{x}_t)$. The parameters θ can be trained using methods such as the adjoint state method, which serves as a continuous-time analogue of backpropagation [14]: see e.g. [30] for a survey. All these models fall within the deep narrow framework considered here. Such approaches have contributed significantly to the widespread use of **very deep** networks today.

What is known about uniform approximation in the deep narrow scenario. It is known that a deep ReLU neural network [26] requires a minimum width of at least the input dimension plus one to ensure uniform approximation. The uniform norm requirement can also be relaxed: it is known that width of $d + 4$ is sufficient for approximation using the L^1-norm [40]. Deep ReLU networks approximate smooth functions more efficiently than shallow networks, as demonstrated by adaptive depth-6 architectures that outperform standard shallow designs [54]. Furthermore, recent work establishes connections between network depth and width [28], while also revealing that neural networks possess even greater approximation power than traditional nonlinear approximation methods [16]. ResNet, even with just one neuron per layer, has been proved to approximate any Lebesgue-integrable function arbitrarily well as the number of tunable weights goes to infinity [37]. Furthermore, [38] establishes that ResNets can be reformulated as feedforward neural networks, enabling the derivation of lower bounds on the number of tunable parameters required for ResNets to approximate different function classes. Universal approximation with general activation functions has been studied in [13, 16, 31, 32, 35, 55, 56, 57], focusing on which types of functions allow for approximation and, in some cases, achieving optimal bounds.

What is known about the computational capabilities of neural networks. In the mid-1990s, recurrent neural networks (RNNs) were widely used. It is known that if an ideal sigmoid function – often modeled as the *saturated linear function*, defined by $\sigma(x) = 0$ for $x \leq 0$, $\sigma(x) = x$ for $0 \leq x \leq 1$, and $\sigma(x) = 1$ for $x \geq 1$ is allowed, then such networks can simulate Turing machines using rational weights [47]. The computational power of these networks has also been extensively investigated in the case where real or even non-computable weights are allowed, leading to connections with non-uniform complexity classes such as $P/poly$ [48]. Refer to [46] for an overview. Since the 1990s, research has shifted toward alternative network architectures. Today, architectures using ReLU activation, defined as $\mathrm{ReLU}(x) = \max(x, 0)$ are widely adopted. It is easy to see that deep ReLU neural networks can simulate ideal sigmoid networks and conversely, as illustrated by the identity $\sigma(x) = \mathrm{ReLU}(x) - \mathrm{ReLU}(x-1)$.

Moreover, any continuous piecewise linear function can be represented by a neural network with ReLU activation [3]. Hence, clearly the previous statements about the possibility of simulating Turing machines hold for such "ideal" activation functions.

While it is known that neural networks can simulate Turing machines – implying universality in the computational sense (at least for ideal sigmoids) – this does not provide statements about their capabilities of universal approximations. The point is that uniform approximation is about "every inputs", while these statements about their computational capabilities are about "some inputs". Existing results only show that certain inputs can encode arbitrary Turing machine computations.

Our results. In this paper, we investigate neural networks $\mathbf{NN} : \mathbb{R}^d \to \mathbb{R}^d$ with matching input and output dimensions, focusing on their behaviour when repeatedly applied to the same input $x \in \mathbb{R}^d$, as seen in all these contexts of the deep narrow scenario. We prove that uniform approximation is also achievable in the sense of computation theory. Note that we are playing with two distinct notions of uniformity: one refers to the uniform norm, as in the classical uniform approximation theorem for neural networks; the other stems from computability theory, where uniformity means that the approximation is effective – that is, computable. We establish that both hold: we have *universality*, in the sense that a single, fixed neural network can serve as a universal approximator.

Neural networks are intrinsically computing over real numbers. This leads to various possibilities of considering computability, when stating results: either by restricting to networks with rational coefficients, and seeing them as functions over the rational numbers, or even more generally as functions from the reals to the reals, with possibly real coefficients. In the first case, we are in the framework of classical computability over finite words. In the second case, we are in the framework of computable analysis [11, 12, 53, 33]. Our results work for both, including aspects of time complexity.

2 Our contributions

We give precise definitions in Section 3, but at this point, it may help to realise that a Neural Network with activation function ReLU, denoted by $\mathbf{NN}[\mathrm{ReLU}]$, with n inputs and m outputs, defines a particular continuous piecewise affine map from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Classical approximation theorem. First, we recall more formally the uniform approximation theorem in the context of neural networks:

▶ **Theorem 1** (Classical Uniform Approximation Theorem [15, 29, 44]). *Let $\rho : \mathbb{R} \to \mathbb{R}$ be a continuous, non-polynomial function. For any continuous function $\mathbf{f} : [0,1]^d \to [0,1]^d$, for any $\epsilon > 0$, there exists a neural network with one hidden layer and activation function ρ, computing some function $\mathbf{NN}[\rho] : \mathbb{R}^d \to \mathbb{R}^d$ such that for all $\mathbf{x} \in [0,1]^d$, $\|\mathbf{NN}[\rho](\mathbf{x}) - \mathbf{f}(\mathbf{x})\| \leq \epsilon$.*

In this statement, the width of the hidden layer can be arbitrary large and depends on both the function being approximated and the precision parameter $\epsilon > 0$. This result holds in particular for ReLU-networks: a ReLU-*Neural-Net or* ReLU-*Net* is a neural network whose activation function ρ is defined as $\mathrm{ReLU}(x) = \max(x, 0)$. In this article, we mainly focus on such networks, and for conciseness, we write then $\mathbf{NN}(x)$ for $\mathbf{NN}[\mathrm{ReLU}](x)$. Note that Theorem 1 does not actually require the output and input dimensions to be equal. However, we have chosen this version to be more comparable to our setting (and both problems are mutually reducible). A *dyadic* is a number $d \in \mathbb{R}$ that has a finite binary expansion, meaning

it can be expressed as $d = m/2^p$ for some integers m and p. In this case, we denote the precision of d by $p = prec(d)$. The set of all dyadic rational numbers (respecitvely of precision p) is denoted by $\mathbb{D}$ (resp. $\mathbb{D}_p$). The *binary length* of a dyadic rational number is defined as $\text{size}(m/2^p) = \text{size}(m) + p$, in consistency with the standard convention for rational numbers.

Our contributions. Let $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ (respectively: $\mathcal{C}^{0,\mathbb{D}}_{PAF}$) denote the set of continuous piecewise affine functions with rational coefficients (resp. dyadic coefficients, preserving dyadics). We fix a representation $\langle \mathbf{f} \rangle$ for functions in $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ (resp. $\mathcal{C}^{0,\mathbb{D}}_{PAF}$): this is a surjective function from words over a finite alphabet to $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ (resp. $\mathcal{C}^{0,\mathbb{D}}_{PAF}$), so that, as expected, given a rational $\mathbf{x}$ and $\langle \mathbf{f} \rangle$, we can compute $\mathbf{f}(\mathbf{x})$ easily (say in polynomial time). We need only this: so, for the coming discussions when talking about circuit complexity, as an alternative to see $\langle \mathbf{f} \rangle$ as only a description of $\mathbf{f}$, this possibly helps to consider that $\langle \mathbf{f} \rangle$ can also be seen as a description of a circuit to compute $\mathbf{f}$.

▶ **Theorem 2** (Our Main theorem 1: Uniform Computability for Classical Complexity)**.** *For any fixed integer d, there exists a fixed* ReLU*-net* $\mathbf{NN} : [0,1]^{d+2} \to [0,1]^{d+2}$*, of width at most* 12*, such that for any $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ function $\mathbf{f}$ over $[0,1]^d$, for any name $\langle \mathbf{f} \rangle$ of function $\mathbf{f}$, for any $\epsilon = 2^{-p}$, there exists $T \in \mathbb{N}$, such that for all $\mathbf{x} \in [0,1]^d$,*

$$\|\pi_{[1,d]}(\mathbf{NN}^{[T]}(\mathbf{x}, \langle \mathbf{f} \rangle, \epsilon) - \mathbf{f}(\mathbf{x})\| \leq \epsilon. \tag{1}$$

Here, $\mathbf{NN}^T$ denotes the neural network obtained by composing $\mathbf{NN}$ with itself T times, and $\pi_{[1,d]}$ is the projection onto the first d coordinates. Furthermore, the involved T has a simple complexity interpretation. It is polynomially related to the time required to evaluate $\mathbf{f}$ on $\mathbf{x}$: when $\mathbf{f} \in \mathcal{C}^{0,\mathbb{D}}_{PAF}$, we can guarantee equality over dyadics: $\pi_{[1,d]}(\mathbf{NN}^{[T]}(\mathbf{x}, \langle \mathbf{f} \rangle, \epsilon) = \mathbf{f}(\mathbf{x})$ for $\mathbf{x} \in \mathbb{D}_p$, with T polynomial in p, and the size of $\langle \mathbf{f} \rangle$.

We emphasize here that while it is known that any function in $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ can be approximated by a neural network, we state here that a stronger statement holds: a single neural network architecture is sufficient to approximate all such functions. We have universal uniform approximation theorem.

Notice, that there is "no encoding involved in variable $\mathbf{x}$": we state that there is uniform approximation of $\mathbf{f}(\mathbf{x})$ by the function $\pi_{[1,d]}(\mathbf{NN}^{[T]}(\mathbf{x}, \langle \mathbf{f} \rangle, \epsilon)$: Assuming only this (with no a priori encoding involved) is precisely one of the main difficulties we solve.

Moreover, this result can be further strengthened within the framework of computable analysis: the class $\mathcal{C}^{0,\mathbb{D}}_{PAF}$ is dense in the space of continuous functions $\mathcal{C}^0([0,1]^d)$ with respect to the uniform norm. That is, any function $\mathbf{f} \in \mathcal{C}^0([0,1]^d)$ admits a sequence $\{\mathbf{f}_i\}_{i \geq 0}$ in $\mathcal{C}^{0,\mathbb{D}}_{PAF}$ such that $\|\mathbf{f} - \mathbf{f}_i\| \leq 2^{-i}$ for all $i \geq 0$. As a result, any function $\mathbf{f}$ in $\mathcal{C}^0([0,1]^d)$ can be encoded as the infinite word $\langle \mathbf{f} \rangle = \langle \mathbf{f}_0 \rangle \# \langle \mathbf{f}_1 \rangle \# \langle \mathbf{f}_2 \rangle \ldots$.This infinite word, when interpreted as the binary expansion of a real number in $[0,1]$, is *computable* as an infinite word if and only if the function $\mathbf{f}$ is *computable* in the sense of computable analysis [11, 12, 53, 33]. Hence, $\langle \mathbf{f} \rangle$ can be viewed as a (possibly irrational) real number in $[0,1]$ whose digits encode $\mathbf{f}$, and this number is computable if and only if $\mathbf{f}$ is computable.

▶ **Theorem 3** (Main theorem 2: Uniform computability for computable analysis)**.** *For any fixed integer d, there exists a fixed* ReLU*-net* $\mathbf{NN} : [0,1]^{d+2} \to [0,1]^{d+2}$ *such that, for any name $\langle \mathbf{f} \rangle$ of a continuous function $\mathbf{f} \in \mathcal{C}^0([0,1]^d)$, for any precision parameter $\epsilon = 2^{-p}$, there exists $T \in \mathbb{N}$, such that for all $\mathbf{x} \in [0,1]^d$, Equation* (1) *holds.*

In other words, this result establishes uniform approximation with uniformity in the sense of computability for computable analysis: given any description of a function – even if the function itself is not computable – the neural network $\mathbf{NN}$ will approximate it to

any prescribed precision, as determined by an input parameter. This goes beyond classical approximation: it guarantees that a single, fixed network architecture can approximate any described function with explicit, parameter-driven control over the precision parameter.

From a complexity-theoretic perspective, beyond mere computability, we prove that the parameter T admits a natural interpretation: for functions in $\mathcal{C}_{PAF}^{0,\mathbb{Q}}$, T corresponds – up to a polynomial factor – to the size of the family of Boolean circuits that compute $\mathbf{f}$ over the dyadic rationals. In this sense, T captures the inherent computational complexity of the function. In the next statement, we say that $\mathbf{f}$ is t-computable, if we can produce in time $t(p)$ some 2^{-p} approximation of $\mathbf{f}$ by some function in $\mathcal{C}_{PAF}^{0,\mathbb{Q}}$:

▶ **Theorem 4** (Main theorem 3: Computational Complexity Approximation Theorem)**.** *For any fixed d, there exists a fixed* ReLU $\mathbf{NN}$*: $[0,1]^{d+2} \to [0,1]^{d+2}$, such that for any name $\langle \mathbf{f} \rangle$ of a $\mathcal{C}_{PAF}^{0,\mathbb{Q}}$ function $\mathbf{f} : [0,1]^d \to [0,1]^d$ t-computable, for any precision parameter $\epsilon = 2^{-p}$, there exists $T \in \mathbb{N}$ such that for all $\mathbf{x} \in [0,1]^d$,*

$$\|\pi_{[1,d]}(N^{[T]}(\mathbf{x}, \langle \mathbf{f} \rangle, \epsilon)) - \mathbf{f}(\mathbf{x})\| \leq \epsilon, \tag{2}$$

where T is polynomial in t and p.

Unlike previous deep learning results, this is established by computability-theoretic arguments, specifically by simulating Circuit Machines. Furthermore, for more general functions, we provably relate T to the complexity of the function f to be approximated. Using recent results, this T can be connected to the Kolmogorov complexity of $\mathbf{f}$ [1, 2].

Main technical contribution with a high potential of new applications. Our results are obtained by programming with ReLU-nets. While the final conclusions may seem intuitive – essentially that ReLU-nets can be viewed as specific circuit models, and that we have universality – one might wonder why such statements were not well established. We now outline the exact issues we addressed and why they were not immediately apparent from existing work.

In short, we provide a method to map from reals to the world of Turing machines, and conversely, using only ReLU functions. The problem lies in the uniform approximation, which requires approximation over all inputs (i.e. real, rational and irrational). Restricting to a suitable subset would have been much easier, as existing results from the 1990s already address this case. However, achieving true uniform approximation demands mapping all inputs into a domain where Turing machines can operate effectively.

More precisely, one has to realise first that functions constructed by a ReLU-net must be specific $\mathcal{C}_{PAF}^{0,\mathbb{Q}}$ functions. In particular, even when iterated a bounded number of times, they always remain Lipschitz continuous. This forbids to obtain some simple functions. For example, the Heaviside function, being discontinuous, is impossible to compute with a ReLU-net in a finite number of iterations. This limitation extends beyond Heaviside, excluding many other functions as well:

▶ **Example 5.** Consider any $\mathbf{f} : [0,1] \times [0,1] \to [0,2]$ with values x when $y = 0$ and $2x$ when $y = 1$. This function cannot be computed within a bounded number of iterations. Indeed, it can be extended to a Lipschitz-continuous $\mathcal{C}_{PAF}^{0,\mathbb{Q}}$ function on $[0,1]^2$. However, this extension fails if we drop the restriction $x \in [0,1]$ and instead allow $x \in \mathbb{R}$, as it would no longer be Lipschitz-continuous in y. This occurs because $\frac{\partial}{\partial y}\mathbf{f}(x,y) \geq \mathbf{f}(x,1) - \mathbf{f}(x,0) = x$ at some point, making it unbounded. Consequently, while a network could compute the function, the number of iterations required cannot be bounded. As a result, it cannot be directly related to the number of steps required by the algorithm.

That stated, we indeed rely on the well-established result from the 1990s that Turing machines can be simulated by ReLU-nets. However, this leads us to discuss some coding issues, and the need to encode/decode inputs, if we want to come to a settings where we can use these results for uniform approximation.

This issues arises from the nature of the input data type: the input x of a network is a real number, while the Turing machine expects a discrete input (e.g. a word encoding x). A central first idea to bridge this gap is to use the *binary expansion* of $x = 0.x_1x_2x_3\dots x_n$. The Turing machine would then proceed the input bit by bit. However, extracting the first bit of a real number is generally impossible due to discontinuity. For example, consider $x = \frac{1}{2} + \varepsilon$, which has first bit $x_1 = 1$, while $x' = \frac{1}{2} - \varepsilon$ yields $x'_1 = 0$. For $\varepsilon \to 0$, both values converge to $1/2$, but their first bits remain distinct. Thus, a network $\mathbf{NN}$ satisfying $\mathbf{NN}(x) = x_1$ cannot exist as it would necessarily compute a discontinuous function – an operation fundamentally incompatible with the continuity and piecewise affine structure enforced by ReLU-nets. A trick, adapted from Turing machine simulations of the 1990s [47, 48], uses $1,3-$ *quadradics*. The discontinuity issue disappears if we encode $x_1, x_2, x_3 \dots, x_n$ as a $1,3-$quadric $\bar{x} = \sum_{i=1}^{n}(2x_i + 1)4^{-i}$. In this encoding, the first bit can be extracted by a function $extract(\bar{x}) = x_1$ that equals 0 on $[1/4, 1/2]$ and 1 on $[3/4, 1]$. Since *extract* is not constrained in $(1/2, 3/4)$, it can be extended arbitrarily on this domain as a continuous, semilinear function – thereby making it compatible with computation by a ReLU neural network.

Such encoding using $1,3$-quadradics is employed throughout the Turing machine simulations presented in [47, 48, 46], where all problems mentioned above were already present.

In other words, while it is well understood that, within the Turing machine framework (computability theory and computable analysis), we can achieve uniform approximation by mapping *the encoding of* x to an arbitrarily precise *encoding of* $f(x)$, this does not, however, immediately imply that we can compute $f(x)$ from x itself for all x from a neural network. The key difficulty lies in converting a real or dyadic number into a suitable $1,3$-quadric encoding that a Turing machine can process: this is exactly what we solve.

▶ Remark 6. We are not the first to have this issue of needing conversion, using continuous function. A point is that in many approaches, such as [45, 24, 25] work over unbounded domains, which makes somehow life really simpler, as they have functions such as $x - 0.2\sin(2\pi x)$ than can basically round to integers. Here, everything must be done on a bounded domain, and futhermore, with ReLU only. The closest to our approach is [6], where discrete ODEs and analytic functions are used. However, their framework differs fundamentally from ours: they do not work with neural networks, and their constructions rely on tools outside the scope of ReLU-nets. For example, they employ the function $\frac{1+\tanh(2^{m+2}x)}{2}$, which mimics a sigmoid as m grows. This construction requires a multiplication by 2^{m+2} (even if obtained recursively there). This seems impossible to obtain using finitely many iterations of a ReLU-net. In ReLU-nets, multiplications are not permitted, and all intermediate variables must remain bounded, whereas their approach relies on multiplications with potentially unbounded weights.

This transformation use only ReLU-nets. Our main technical result is stated below:

▶ **Proposition 7** (From the reals to the world of TMs). *There exists a* ReLU*-net* $\mathbf{NN}_{0,1\to 1,3}$ *such that, given* $1/2^p$ *as a numerical value and a dyadic* $d = \sum_{i=1}^{p} x_i 2^{-i} \in [0,1]$ *of precision* p*, the network returns:* $\mathbf{NN}^{[p^2]}_{0,1\to 1,3}(d, \frac{1}{2^p}, 0) = (0, \frac{1}{2^p}, \overline{d})$*, where* $\overline{d} := \sum_{i=1}^{p}(2x_i + 1)4^{-i}$ *is the corresponding* $1,3$*-quadric encoding of* d*.*

The proof of the following proposition is similar and simpler.

▶ **Proposition 8** (From the world of TMs to the reals)**.** *There exists* ReLU*-net* $\mathbf{NN}_{1,3\to 0,1}$ *such that, given* $1/4^p$ *as a numerical value, given a 1,3-quadradic* $\bar{d} = \sum_{i=1}^{p}(2x_i+1)4^{-i} \in [0,1]$ *of precision* $2p$*, then* $N^{[p]}_{1,3\to 0,1}(\bar{d}, \frac{1}{4^p}, 0) = (0, \frac{1}{4^p}, d)$*, where* $d = \sum_{i=1}^{p} x_i 2^{-i} \in [0,1]$ *is the corresponding dyadic.*

These constructions yield both decoders and encoders between dyadics and $1,3$-quadradics, with a major advantage: the latter encoding can be tailored to be robust against local errors. One important remaining issue is that they will work for most of the dyadic d, but not all.

▶ Remark 9. This limitation is unavoidable. Due to continuity constraints, it is impossible to transform d into $\bar{d}$ for all d, as no continuous function can map $[0,1]$ onto a discrete set. The functions constructed in the proof of Proposition 7, exhibit "jumps", where the output is necessarily "uncorrect".

We overcome this limitation by introducing an error indicator E, allowing two decoders to run in parallel and always ensure a correct result. Formally:

▶ **Proposition 10.** *The networks* $\mathbf{NN}_{0,1\to 1,3}$ *and* $\mathbf{NN}_{1,3\to 0,1}$ *can be designed to be robust against local errors. Specifically, for* $\mathbf{NN}_{0,1\to 1,3}$*, if the input* $\tilde{d}$ *is within distance* $\epsilon(p)$ *of a given* d*, then* $\mathbf{NN}^{[p^2]}_{0,1\to 1,3}$ *still correctly outputs* $\bar{d}$*, and* $\mathbf{NN}^{[p]}_{1,3\to 0,1}$ *still correctly returns* d*.*

Therefore, for any arbitrary real number $x \in [0,1]$*, the network* $\mathbf{NN}^{[p^2]}_{0,1\to 1,3}(x, 1/2^p, 0)$ *will output the base-4 encoding of a dyadic that is* $\frac{1}{2^p}$*-close to* x *with guarantee at least* $1-2^{-(2p+1)}$*. Furthermore, computing both* $\mathbf{NN}^{[p^2]}_{0,1\to 1,3}(x, 1/2^p, 0)$ *and* $\mathbf{NN}^{[p^2]}_{0,1\to 1,3}(x+1/2^{p+1}, 1/2^p, 0)$ *guarantees that at least one of them returns the base-4 encoding of a dyadic that is* $\frac{1}{2^p}$*-close to* x*. Additionally, we can include an error indicator in the second dimension to signal potential errors, such that* $\mathbf{NN}^{[p^2]}_{0,1\to 1,3}(x, 1/2^p, 0) = (E, 1/2^p, \bar{x})$*, where* $E = 0$ *only if the dyadic computation is correct. All of the above operations can be performed on dyadic (or base-4) vectors componentwise in parallel.*

Discussions. Notably, as a side effect, we also prove that universality can be achieved with width 3 (Theorem 23). To the best of our knowledge, this result is original. Furthermore, the ability to map neural computations onto the Turing machine framework gives rise to numerous variants of the previous statements. In essence, many of these variations become valid as soon as the corresponding operations are feasible on a Turing machine.

Interpretation of T as a measure of the complexity of the functions. Coming back to classical complexity, our results give a way to represent continuous functions using circuits, where the time T is polynomially related to the circuit size SIZE. In this sense, T measures the circuit complexity of the uniform approximation of a function; see [52] for a comprehensive reference on circuit complexity. More generally, several natural measures can be used to quantify the complexity of a function – among them, Kolmogorov complexity. It is commonly defined via the function $\mathrm{C}(x)$, which denotes the length of the shortest binary description $d \in \{0,1\}^*$ such that a fixed universal Turing machine U satisfies $U(d) = x$. If we adopt the slightly different definition of $\mathrm{C}(x)$ and $\mathrm{KT}(x)$ proposed in [2], which departs from the classical formulation, then, Kolmogorov complexity can be directly related to circuit complexity. In this sense, we establish that the time parameter T is polynomially related to both the size of the circuits computing the function (possibly relativized) and to its Kolmogorov complexity: It is known that $\mathrm{KT}(x) \approx \mathrm{SIZE}(x)$ and $C(x) \approx \mathrm{SIZE}^H(x)$, where H is the halting problem, and $\approx$ means "polynomially related" [2, 1].

3 Basic definitions and statements

The networks that we consider are either feedforward or built by iterating feedforward layers. In their most general form, they process real-valued inputs. We use only rational weights – making them both expressive and analytically tractable.

▶ **Definition 11** (Feedforward neural network). *A* feedforward neural network *(NN)* $\mathbf{NN}$ *is a layered graph that represents a function of the form* $\mathbb{R}^n \to \mathbb{R}^m$*, for some* $n, m \in \mathbb{N}$*. The first layer with label* $\ell = 0$ *is called the* input layer *and consists of* n *nodes called* input nodes*. Each node receives an input* $x_i \in \mathbb{R}$*, which is also taken as its output:* $y_{0i} := x_i$*. Layers indexed by* $1 \leq \ell \leq L-2$ *are* hidden layers*, each containing* $k(\ell)$ computation nodes*. The output of the* i*-th node in layer* ℓ *is given by* $y_{\ell i} = \sigma_{\ell i}(\sum_j c_{ji}^{(\ell-1)} y_{(\ell-1)j} + b_{\ell i})$*. Here,* $\sigma_{\ell i}$ *is (typically nonlinear)* activation *function and the sum runs over all nodes in the previous layer* $\ell - 1$*. The* $c_{ji}^{(\ell-1)}$ *are rational constants which are called* weights*, and* $b_{\ell i}$ *is a rational constant called* bias*. The outputs of layer* ℓ *are collectively denoted* $(y_{\ell 0}, \ldots, y_{\ell(k(\ell)-1)})$*. The final layer* $L-1$ *is the* output layer *and consists of* m output nodes*. The* i*-th node computes an output* $y_{(L-1)i}$ *in the same way as a node in a hidden layer. The vector* $(y_{(L-1)0}, \cdots, y_{(L-1)(m-1)})$ *is the output of the network when given input* x*, denoted* $\mathbf{NN}(x)$*.*

Let $\mathcal{NN}^{\rho}_{n,m,d,k}$ denote the class of functions $\mathbb{R}^n \to \mathbb{R}^m$ described by feedforward neural networks with the following structure: n input neurons, m output neurons, and d hidden layers, each containing at most k neurons. All hidden neurons use the activation function $\rho : \mathbb{R} \to \mathbb{R}$. We use the symbol $*$ to indicate that no constraint is imposed on some parameter. $\mathcal{N}^{\rho}_{d} := \mathcal{NN}^{\rho}_{d,1,1,*}$ represents for example the feedforward neural networks with activation function ρ, with d neurons in the input layer, one neuron in the output layer, and one hidden layer containing an arbitrary number of neurons.

▶ **Definition 12** (Composition of Neural Networks). *Let* $\mathbf{NN}_1 : \mathbb{R}^n \to \mathbb{R}^m$ *and* $\mathbf{NN}_2 : \mathbb{R}^m \to \mathbb{R}^k$ *be two neural networks. The notation* $\mathbf{NN}_2 \circ \mathbf{NN}_1 : \mathbb{R}^n \to \mathbb{R}^k$ *denotes both the resulting network formed by composing the two networks, i.e., by connecting the output nodes of* $\mathbf{NN}_1$ *to the corresponding input nodes of* $\mathbf{NN}_2$ *– and the function it computes. Moreover, in the case where* $n = m$*, we inductively define* $\mathbf{NN}_1^{[1]} := \mathbf{NN}_1$*, and* $\mathbf{NN}_1^{[t+1]} := \mathbf{NN}_1^{[t]} \circ \mathbf{NN}_1$ *for all* $t \in \mathbb{N}$*.*

As defined above, all considered networks exclusively use coefficients in $\mathbb{Q}$ and will employ ReLU activation function from now on. To avoid redundancy, we omit explicitly writing "$\mathbb{Q}$-ReLU-Net" in every instance. We assume a fixed notation for $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ functions by defining a surjective function $\langle\cdot\rangle$ from Σ^* to $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$. Thus, every function $\mathbf{f} \in \mathcal{C}^{0,\mathbb{Q}}_{PAF}$ has a *name* f, meaning there exists a finite word f such that $\langle f \rangle = \mathbf{f}$. Note that these objects represent functions from $\mathbb{R}^d$ to $\mathbb{R}^n$, even though their coefficients are restricted to rational numbers.

The following theorem is already well established:

▶ **Theorem 13** ([3]). *Every* $f \in \mathcal{C}^{0,\mathbb{Q}}_{PAF}$*,* $f : \mathbb{R}^d \to \mathbb{R}$ *can be represented by a* ReLU*-net* $\mathbf{NN}$*.*

For vectors $\mathbf{x}$ and $\mathbf{y}$ of the same dimension, we write $\mathbf{x} \leq \mathbf{y}$ (or $\mathbf{x} < \mathbf{y}$), if the inequality holds component-wise. The binary size of an object d is denoted by $\text{size}(d)$. For example, the binary length of an integer n is denoted by $\text{size}(n)$. A dyadic number of precision p can be represented in binary as $a_{-k} \ldots a_{-1} a_0 . a_1 a_2 \ldots a_p$ for some $k \in \mathbb{N}$, with each $a_i \in \{0, 1\}$, corresponding to $d = \sum_{i=-k}^{p} \frac{a_i}{2^i}$. To establish our results, we introduce the concept of $(1,3)$*-quadradics*, defined as the subset of real numbers whose base-4 expansion consists exclusively of the digits 1 and 3.

▶ **Definition 14** ($1,3$-quadradics). *We denote by $\mathbb{DD}_{1,3}$ the set of dyadic numbers whose base-4 expansion consist of finitely many either 1 or 3's, followed by an infinite sequence of zeros. We write $\overline{\mathbb{DD}_{1,3}}$ for the dyadic numbers whose base-4 expansion is infinite with only 1 and 3's. That is, $\mathbb{DD}_{1,3} = \{d | d = \sum_{i=-m}^{p} a_i/4^i,$ for some $a_i \in \{1,3\}, p \in \mathbb{N}\}$ and $\overline{\mathbb{DD}_{1,3}} = \{d | d = \sum_{i=-m}^{\infty} a_i/4^i,$ for some $a_i \in \{1,3\}, p \in \mathbb{N}\}$.*

▶ Remark 15 (Simple key observation). While the dyadics are dense in $\mathbb{R}$, the $1,3$-*quadradics* are not – they exhibit *gaps*, a crucial property we will use throughout this work. For example, no element of $\mathbb{DD}_{1,3}$ lies within interval $(\frac{1}{2}, \frac{3}{4})$. Moreover, the set $\overline{\mathbb{DD}_{1,3}}$ is homeomorphic to the Cantor set.

4 Programming with Neural Networks

To construct neural networks tailored to specific tasks – effectively programming with neural networks – it is useful to describe them using a simple programming language. We outline the structure of such programs, showing that these networks can simulate algorithms involving loops and conditional statements. We consider algorithms composed of *commands*, *while loops*, and *conditions*. A *command*, written as $x_i' \leftarrow \mathit{aff}(\mathbf{x})$ where $\mathit{aff} : \mathbb{R}^k \to \mathbb{R}$ is some function, corresponds, as expected to the function that maps $\mathbf{x} \in \mathbb{R}^k$ to $\mathbf{x}' \in \mathbb{R}^k$, where $x_j' = x_j$ for all $j \neq i$, and $x_j' = \mathit{aff}(\mathbf{x})$. A command is said to be *affine* if the function *aff* is affine.

Theorem 13, as stated above, asserts, that any function in $\mathcal{C}_{PAF}^{0,\mathbb{Q}}$ can be programmed/-computed by a feedforward network. However, careful attention is needed regarding the involved functions, and potential introduction of new variables or gap-related issues in the construction. This is why we need to come back to very concrete programs and functions. To address this, we present a simple yet useful observation on transforming affine-programmable functions into ReLU-net.

▶ **Lemma 16** (From a program to a Neural Net). *Let $g : [0,1]^k \to [0,1]^k$ be an affine-programmable function, such that every **if** and **while** condition of the form $x_i \geq \beta$ satisfies a* gap *condition: there exists $\alpha < \beta$ such that $x_i \notin (\alpha, \beta)$ is guaranteed, with $0 \leq \alpha < \beta \leq 1$. Additionally, assume that all intermediate values computed by the algorithm remain within the interval $[-1, 2]$. Then there exists a network $\mathbf{NN}_g$ computing g such that $\pi_{1,\ldots,k}(\mathbf{NN}_g^{[T]}(0, \mathbf{x})) = g(\mathbf{x})$ for the smallest T satisfying $\pi_0(\mathbf{NN}_g^{[T]}(1, \mathbf{x})) = 0$. Moreover, the number of iterations T required by the network is linear in the runtime complexity of the algorithm.*

Proof. Let all basic commands $C_1, \ldots, C_\ell : [0,1]^k \times [0,1]^k \to [0,1]^k$ be labeled with indices $1, \ldots, \ell$ according to the order in which they appear in the algorithm. We first define a *command-finding function* $c : \{1, \ldots, \ell\} \times [0,1]^k \to \{0, \ldots, \ell\}$ which, given the index of the current command and the current state, returns the index of the next command to execute.

Case 1: The **return**-command is followed by terminating the network, meaning $c(\ell, \mathbf{x}) = 0$ for all $\mathbf{x}$.

Case 2: Let i be the index of the last command before entering a **while**-loop with condition $x_i \geq \beta$, so $i+1$ is the index of the first command in that loop. Let j be the index of the first command after leaving the loop, making $j-1$ the last command in the loop. The command-finding function in this case is defined as:

$$c(i, \mathbf{x}) = c(j-1, \mathbf{x}) = \begin{cases} i+1 & if\ x_i \geq \beta \\ j & else \end{cases}$$

Case 3: **If**-conditions follow the same structure as in Case 2.

Case 4: All other commands are mapped to their immediate successor, that is, $c(i, \mathbf{x}) = i+1$. The function c defined above can be extended to a continuous piecewise affine map (CPAM) and is therefore computable by a ReLU-net by Theorem 13.

Second, we construct the network $\mathbf{NN}_g$. Given input $(i, x_1, \ldots, x_k)$, $\mathbf{NN}_g$ first computes $c(i, \mathbf{x})$ as well as the outputs of all commands $C_1, \ldots, C_\ell$. It then subtracts $1 - \delta_{i,j}$ from the result of each command C_j where $\delta_{i,j}$ denotes the Kronecker delta. In the first layer, this yields the vector $(c(i, \mathbf{x}), \mathrm{ReLU}(C_1(\mathbf{x}) - (1 - \delta_{i,1})), \mathrm{ReLU}(C_2(\mathbf{x}) - (1 - \delta_{i,2})), \ldots, \mathrm{ReLU}(C_\ell(\mathbf{x}) - (1 - \delta_{i,\ell})))$. Since $\mathrm{ReLU}(C_j(\mathbf{x}) - (1 - \delta_{i,j})) = 0$ for $i \neq j$ and $\mathrm{ReLU}(C_j(\mathbf{x}) - (1 - \delta_{i,j})) = C_j(\mathbf{x})$ for $i = j$, the first layer simplifies to $(c(i, \mathbf{x}), 0, 0, \ldots, 0, C_i(\mathbf{x}), 0, \ldots, 0)$. In the output layer, a component-wise sum over all $C_j(\mathbf{x}) - (1 - \delta_{i,j})$ ensures that the final output is $(c(i, \mathbf{x}), C_i(\mathbf{x}))$. ◀

The above statement and its proof may at first appear to be a trivial observation. However, the restriction to the domain $[0, 1]^k$ instead of $\mathbb{R}^k$ and the requirement for gaps in conditional queries, are essential for the construction to be valid. In this sense, the difficulty lies in ensuring that all computations are carried out using functions that can be explicitly implemented by ReLU-nets.

The above proof does not aim to optimize the width of the constructed ReLU-nets. We present it in this form as it facilitates the understanding of subsequent arguments. Later, we will apply several techniques to reduce the network width (e.g., in the proof of Theorem 23). Essentially, if we had applied the same optimizations in the above proof, we would obtain feasibility within a width that is at most the maximum width of the involved commands, as established by Lemma 21, by executing the commands sequentially.

To achieve width reduction, we will actually also leverage techniques similar to those used in the proof of the following statement, but sometimes even more optimized.

▶ **Proposition 17** (Width efficient computation of max-min string, [26, Proposition 2]). *A function $g : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ is* a max-min string *of length $L \geq 1$ on d_{in} input variables and d_{out} output variables if there exist affine functions $aff_1, \ldots, aff_L : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ such that $g = \sigma_{L-1}\left(aff_L, \sigma_{L-2}\left(aff_{L-1}, \ldots, \sigma_2\left(aff_3, \sigma_1\left(aff_1, aff_2\right)\right)\right)\right)$ where each σ_i is either a coordinate-wise max or a min. For every max-min string g of length L on d_{in} input variables and d_{out} ouput variables, and for every compact $K \subseteq \mathbb{R}^{d_{in}}$, there exists a* ReLU*-net with input dimension d_{in}, hidden layer width $d_{in} + d_{out}$, output dimension d_{out}, and depth L that computes $\mathbf{x} \mapsto g(\mathbf{x})$ for all $\mathbf{x} \in K$.*

5 From the world of real numbers to Turing machines and back

5.1 Turing machines

We assume our readers are familiar with Turing machines and their classical properties, we refer to [50] for a complete introduction. Following [50], we define a (one-tape) *Turing machine* as a 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject} \rangle$ where Q, Σ, Γ are finite non-empty sets, $B \in \Gamma$ is a blank symbol such that Q is the set of states; Σ is the set of input symbols, with $B \notin \Sigma$. Γ is the set of tape symbols, with $\Sigma \subseteq \Gamma$; $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function, with L the left shift and R the right shift. If δ is not defined on some $(q, \gamma) \in Q \times \Gamma$ then the machine halts; $q_0 \in Q$ is the initial state; $q_{accept} \in Q$ is the accept state; $q_{reject} \in Q$ is the reject state, with $q_{reject} \neq q_{accept}$. The machine M *halts* on input w if it eventually reaches either state q_{accept} or q_{reject}. A *configuration* (or *instantaneous description (ID)*) represents an current state of the machine: the state $q \in Q$, the tape content, and the head location.

The space of IDs of a Turing machine can be defined as $\mathcal{C}_M = \Gamma^\omega \times Q \times \Gamma^\omega$, where Γ^ω denotes infinite words over alphabet Γ. The *initial configuration* associated with input $w \in \Sigma^*$, denoted by $C[w]$, is given by $C[w] = (B^\omega, q_0, wB^\omega) \in \mathcal{C}_M$. A configuration $(\text{left} = a_{-1}a_{-2}\dots, q, \text{right} = a_0a_1a_2\dots) \in \mathcal{C}_M$ represents a machine state where the tape content is $\dots a_{-2}a_{-1}\underline{a_0}a_1a_2\dots$, with each $a_i \in \Gamma$, the head positioned over symbol a_0, and the internal state equal to q. We write $C \vdash C'$ if configuration C' is the immediate successor of configuration C under the transition function δ.

▶ Remark 18. Notice that we define $\mathcal{C}_M = \Gamma^\omega \times Q \times \Gamma^\omega$ instead of $\mathcal{C}_M = \Sigma^* \times Q \times \Sigma^*$ in order to possibly include infinite words, i.e the fact that their might be an infinite word in the tape. Notice also that, under this convention, the content to the right of the head (i.e.,$right$) is written in the standard order, from left to right, whereas the content to the left of the head (i.e., $left$ is written in reverse order, from right to left.

5.2 Simulating TMs with $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$s

It is well established that Turing machines can be simulated by PAMs [34, 41, 49]. This can be formalized as follows (see, for instance, [7]):

▶ **Lemma 19** ([7, Lemma 4.1]). *Let M be a Turing machine with $\Sigma = \{1, 3\}$, and $\Gamma = \{1, 3, B\}$, and let $\mathcal{C}_M = \Gamma^\omega \times Q \times \Gamma^\omega$ be its configuration space. There exists a piecewise affine function $g_M : \mathbb{R}^2 \to \mathbb{R}^2$ and an encoding function $\nu : \mathcal{C}_M \to ([0, 1] \cap \overline{\mathbb{DD}_{1,3}})^2$ such that the following diagram commutes:*

$$\begin{array}{ccc} \mathcal{C}_M & \xrightarrow{\vdash} & \mathcal{C}_M \\ {\scriptstyle\nu}\downarrow & & \downarrow{\scriptstyle\nu} \\ \mathbb{R}^2 & \xrightarrow{g_M} & \mathbb{R}^2 \end{array}$$

(i.e. $g_M(\nu(C)) = \nu(C')$ for all configurations C, $C' \in \mathcal{C}_M$ with $C \vdash C'$).

For an explicit proof, see [7]. The proof there essentially follows [34], itself building on the ideas from [41, 49].

Idea of the proof. It is helpful to briefly explain the idea behind the proof of Lemma 19. First, we can assume that the Turing machine's alphabet is $\Gamma = \{1, 3, B\}$, where the blank symbol is $B = 0$. In this setting, any finite word over alphabet Γ can be considered as $1, 3$-quadradics. Namely, any word $w = a_0a_1 \dots a_m \in \Gamma^*$ can be viewed as the base-4 number $0.a_0a_1 \dots a_m$, that is, the element of $\mathbb{DD}_{1,3}$ given by $\mu(w)$ with $\mu(w) = \sum_{i=-m}^{p} a_i/4^i$. This functions μ maps Γ^* into the subset of $[0, 1]$ corresponding to $[0, 1) \cap \mathbb{DD}_{1,3}$.

So, a configuration $(left, q, right) \in \mathcal{C}_M$ can be seen as element

$$\nu(left, q, right) = (\overline{left} = \mu(left), q, \overline{right} = \mu(right)) \in ([0,1] \cap \overline{\mathbb{DD}_{1,3}}) \times Q \times ([0,1] \cap \overline{\mathbb{DD}_{1,3}}).$$

A simple observation is that constructing a piecewise affine function g_M satisfying the desired commutative diagram becomes straightforward if we replace the role of $\mathbb{R}^2$ in this diagram with $\mathbb{R} \times Q \times \mathbb{R}$. Assuming $Q = \left\{\frac{1}{|Q|}, \frac{2}{|Q|}, \dots, 1\right\}$, then $\mathbb{R} \times Q \times \mathbb{R}$ lives in $\mathbb{R}^3$, and we get the above diagram where $\mathbb{R}^2$ is replace to $\mathbb{R}^3$. To indeed reduce it to $\mathbb{R}^2$, one can apply the encoding trick described in Lemma 21. ◀

▶ Remark 20. The main advantage of using $1, 3$-quadradics instead of binary representation is that we only need to specify the function g_M on the image of ν. As this image is a subset of $\mathbb{DD}_{1,3}$, it contains gaps (see Remark 15), meaning we can entirely ignore how g_M behaves

inside on it. Now, outside of gaps, operations such as moving the tape left or right, or replacing a symbol, is easily seen to correspond to some particular affine functions. Moreover, since the function g_M is only required on the structured subset $\mathbb{DD}_{1,3}$, and we do not need to specify its values outside of gaps: we can even suppose the function g_M to be continuous and, in some cases, even smooth. This aspect is discussed in references such as [7, 41, 34, 9, 46].

In general, we can encode a real, and some finite information into a unique real:

▶ **Lemma 21** (Encoding finite information into a real)**.** *Let $c \in \{1, 2, \ldots, \ell\}$. Then c can be represented as a $1,3$-quadradic of finite precision p: we can view c as $e(c)$, an element of $[0,1] \cap \mathbb{DD}_{1,3}$, whose base-4 expansion consists of exactly p digits, each in $\{1,3\}$. Using this, we can encode any pair $(c,x) \in \{1,2,\ldots,\ell\} \times [0,1]$ via the affine function with gaps: $encoding_\ell(c,x) = e(c) + \frac{x}{4^{p+1}}$. Then, recovering the pair (c,x) from $encoding_\ell(c,x)$ can be achieved using an affine function with gaps. We denote this inverse operation by $decoding_\ell$ for this reverse function, and and its restriction to the interval $[e(c), e(c)+1/4^{p+1}]$ by $decoding_{\ell,c}$.*

These are the classical techniques for simulating a Turing machine using piecewise affine functions over $\mathbb{R}^2$. Then, using Theorem 13, we can implement this simulation within ReLU-net. Actually, we improve this idea in order to consider low-width ReLU-nets.

▶ Remark 22. We could have used Proposition 17, but it would result in a width-4 construction and would still require proving that max-min strings can be implemented.

▶ **Theorem 23.** *The piecewise affine function g_M of Lemma 19 can be implemented by a* ReLU*-net of width* 3*.*

Proof. Indeed, the program of the Turing machine consists of instructions of the form $\delta(q,a) = (q', a', m)$ with $m \in \{L, R\}$. This corresponds to the high-level instruction of the form:

"line q: **If** one reads a **Then** write a'; shift the tape left or right depending on m,
Goto line q'"

For example, if $a = 1$, $a' = 3$, this translates into a line of the form:

"line q: **If** $\overline{r} \in [1/4, 1/2]$ **Then** $\overline{r} \leftarrow (\overline{r} - 1/4) + 3/4$; $move_q(m)$;

(here, $\overline{l}$ and $\overline{r}$ represent $\overline{left}$ and $\overline{right}$ of Lemma 19, encoding in $1,3$-quadradics the left and right part of the tape), where $move_q(R)$ is[1]: (we do not write $move_q(L)$, because it is symmetric)

- **If** $\overline{l} \in [1/4, 1/2]$ **Then** $\overline{l} \leftarrow 4(\overline{l} - 1/4)$; $\overline{r} \leftarrow \overline{r}/4 + 1/4$
- **If** $\overline{l} \in [3/4, 1]$ **Then** $\overline{l} \leftarrow 4(\overline{l} - 3/4)$; $\overline{r} \leftarrow \overline{r}/4 + 3/4$
- **If** $\overline{l} = 0$ **Then** $\overline{l} \leftarrow 4\overline{l}$; $\overline{r} \leftarrow \overline{r}/4$;
- **Goto** line q'

This involves lines all of the form, say, **If** $\overline{l} \in [1/4, 1/2]$ **Then** $\overline{l} \leftarrow 4\overline{l} - 1$. Because of the use of $1,3$-quadradics and the presence of gaps, this assignment still works if written as $\overline{x} \leftarrow A(\overline{x})$, for any function A that equals 0 when $\overline{x} \leq 1/8$, takes the value $4\overline{x} - 1$ on the interval $[1/4, 1/2]$, and returns 0 again when $\overline{x} \geq 5/8$. We could take $A(\overline{x}) = \mathrm{ReLU}(4\overline{x} - 1) - 3/2\,\mathrm{ReLU}(8\overline{x} - 4) + 8\,\mathrm{ReLU}(\overline{x} - 5/8)$.

[1] The last line is here to deal with the blank symbol B.

Then, this computation can be implemented using four layers of width 3 in a ReLU-network. The first two neurons are used to store the values of $\bar{l}$ and $\bar{r}$, while the third neuron serves as a *working neuron*, incrementally computing each term in the expression defining the function A. Concretely, the first layer maps (l, r, z) to $(\mathrm{ReLU}(l), \mathrm{ReLU}(r), \mathrm{ReLU}(4l - 1))$; the second layer maps (l, r, z) to $(\mathrm{ReLU}(l), \mathrm{ReLU}(r), \mathrm{ReLU}(z) - \frac{3}{2}\,\mathrm{ReLU}(8l - 4))$; the third layer computes $(\mathrm{ReLU}(l), \mathrm{ReLU}(r),\ \mathrm{ReLU}(z) + 8\,\mathrm{ReLU}(l - \frac{5}{8}))$; and finally, the fourth layer maps (l, r, z) to $(\mathrm{ReLU}(z), \mathrm{ReLU}(r), \mathrm{ReLU}(z))$, forwarding the result. ◀

6 Main Technical Statement 1: Proofs of Propositions 7 and 10

Proof. Observe that the function $f : \mathbb{R} \to \mathbb{R}$, defined by $f(x) = 2\,\mathrm{ReLU}\left(x - \frac{1}{4}\right) - 2\,\mathrm{ReLU}\left(x - \frac{3}{4}\right)$, can be computed by a ReLU-network. This function satisfies $f^{2p}(x) = 0$ for all $x \leq \frac{1}{2} - \frac{1}{2^{2p+1}}$, and $f^{2p}(x) = 1$ for all $x \geq \frac{1}{2} + \frac{1}{2^{2p+1}}$. Define the function $h(x) = \frac{1}{2} - \mathrm{ReLU}\left(\sigma(x) - \frac{1}{2}\right) - \mathrm{ReLU}\left(\frac{1}{2} - \sigma(x)\right)$. The function $h(x)$ satisfies the property that $h(x) = 0$ if and only if $x \notin (0, 1)$. Both $\sigma(x)$ and $h(x)$ are computable by neural networks of width 2.

We describe, at a precise level, how to transform a dyadic number in binary form into a $1, 3$-quadradic of the same precision.

Require: $a = 1/2^p, d \in [0, 1], E = 0$
  $\bar{d} \leftarrow 0$
  $d \leftarrow d + a/2$ ▷ move $d = 0.d_1 d_2 \ldots d_n 000 \ldots$ to $d = 0.d_1 d_2 \ldots d_n 1000 \ldots$
  while $a < 1$ **do** ▷ Outer loop: keep track on
    $b \leftarrow a$ ▷ how many of the p digits
    $a \leftarrow 2a$ ▷ we still need to POP
    $c \leftarrow d$
    while $b < 1$ **do** ▷ Inner loop: POP the most
      $b \leftarrow 2b$ ▷ significant bit $f^{2p}(x)$
      $c \leftarrow f(c)$
      $c \leftarrow f(c)$
    end while
    $\bar{d} \leftarrow \frac{\bar{d}}{4} + c$ ▷ Write most significant bit in output
    $d \leftarrow 2d - c$ ▷ Delete most significant bit from remaining input
    $E \leftarrow E + h(c)$ ▷ Update the error if failed to POP the bit
  end while
  return $(d, \bar{d}, E)$

The step of $d \leftarrow d + a/2$ ensures that the new value of dyadic d is in the middle of the interval of all real numbers, whose binary expansion starts with $0.d_1 d_2 \ldots d_n$.

The above algorithm returns $\bar{d}$, but with the digits in reversed order. This issue is easily corrected, as the POP-operation can be performed in base 4 in constant time. By Lemma 16, the code suffices to show that a recurrent network can implement this transformation.

Note that the above algorithm computes the first $2p$ digits, so it would have returned the same result as long as digits p to $2p$ do not all have the same value. This holds for any input in the interval $[d, d + 2^{-(p+1)} - 2^{-(2p+1)}]$. Therefore, if the algorithm fails for a non-dyadic d, it will still correctly a nearby value $d' = d + a/4$, because $2^{-2p} < |d - d'| < 2^{-p} - 2^{-2p}$.

This program has 7 variables (including the index of the current command). It can hence be implemented by a width 8 ReLU-net, by the techniques used in the proof of Theorem 23 (actually Lemma 21), and above remarks about the fact that $\sigma(x)$ and $h(x)$ can be computed with only one more variable. ◀

7 Proof of Theorems 2, 3 and 4

Proof. By lack of space, we assume that $x \in \mathbb{R}$ has dimension 1. Furthermore, we assume that f is Lipschitz-continuous with constant $L = 1$. If $L \neq 1$, we replace ε by a largest power of 2 smaller than ε/L. We also assume that $\langle f \rangle$ ends with a unique symbol # whose binary encoding is metrically isolated in the sense that $|\langle \# \rangle - \langle w \rangle| \geq 2^{-length(\langle \# \rangle)}$ for every other string $w \in \Sigma^*, w \neq \#$ that does not contain #. We now argue that the following pseudocode can be implemented within the constraints of Lemma 16 (we assume the decoding back into the dyadic numbers to be done implicitly in the return command; the same holds for the next pseudocode).

Require: $x \in [0,1], y = \langle f \rangle \in \mathbb{D} \cap [0,1], \epsilon = 2^{-n}$
  $(x', a, z, z', E_x, E'_x) \leftarrow (x + \frac{\epsilon}{2}, \epsilon, 0, 0, 0, 0)$
  while $a < 1$ **do** ▷ Transform x into base 4
    $(a, b) \leftarrow (2a, \epsilon)$
    while $b < 1$ **do**
      $b \leftarrow 2b$
      $(x, z, E_x) \leftarrow \mathbf{NN}_{0,1\to1,3}(x, z, E_x)$
      $(x', z', E'_x) \leftarrow \mathbf{NN}_{0,1\to1,3}(x', z', E'_x)$ ▷ If we fail on x, try a x' that is guaranteed
    end while ▷ to work by Proposition 10
  end while
  while $|(\langle \# \rangle) - y| \geq 2^{-length(\langle \# \rangle)}$ **do** ▷ Transform y into base 4
    $y \leftarrow \mathbf{NN}_{0,1\to1,3}(y)$
  end while
  $y' \leftarrow y$
  $(x, y) \leftarrow \nu(x, y)$ ▷ Map function and input into $\mathbb{R}^2$,
  $(x', y') \leftarrow \nu(x', y')$ ▷ as needed for Lemma 19
  while $1 > 0$ **do**
    $(E_x, E'_x) \leftarrow (2E_x, 2E'_x)$ ▷ Blow up the error, if $E > 0$, eventually $E > 1$
    $(x, y) \leftarrow g_{M_u}(x, y)$ ▷ Simulate the universal Turing machine M_u
    $(x', y') \leftarrow g_{M_u}(x', y')$
    return $\max(x - E_x, x' - E'_x)$
  end while

Note that at least one of x and x' is correctly encoded; without loss of generality, assume it is for x. This ensures that $E_x = 0$ at all times. Since the computation of $f(x)$ eventually halts, $x - E_x$ becomes static and is ε-close to $f(x)$. For x', two cases arise:

Case 1: x' is also correctly encoded in base 4. In this case, $E'_x = 0$ and by continuity of f, $x' - E'_x$ is also ε-close to $f(x)$. Consequently $max\{x - E_x, x' - E'_x\}$ is ε-close to $f(x)$ as well.

Case 2: x' is incorrectly transformed into base 4, which means that $E'_x > 0$. After n iterations of the last loop, $2^n E'_x > 1$, leading to $x' - E'_x < 0$. This ensures that $max\{x - E_x, x' - E'_x\} = x$, which is again ε-close to $f(x)$.

However, if an error remains nonzero, there is no bound on the runtime: for example, if E_x is very small, it may require an arbitrarily long time before successive multiplications by 2 make it exceed 1.

To establish a polynomial bound on the number of network iterations when x is a dyadic number, consider the following simplified version of the above algorithm:

Require: $x \in [0,1], y = \langle f \rangle \in \mathbb{D} \cap [0,1], \epsilon = 2^{-n}$
  $a \leftarrow \epsilon$
  $z \leftarrow 0$
  while $a < 1$ **do** ▷ Transform x into base 4
    $a \leftarrow 2a$
    $b \leftarrow \epsilon$
    while $b < 1$ **do**
      $b \leftarrow 2b$
      $(x, z, E_x) \leftarrow \mathbf{NN}_{0,1\to 1,3}(x, z, E_x)$
    end while
  end while
  while $|\langle \# \rangle - y| \geq 2^{-length(\langle \# \rangle)}$ **do** ▷ Transform y into base 4
    $y \leftarrow \mathbf{NN}_{0,1\to 1,3}(y)$
  end while
  $(x, y) \leftarrow \nu(x, y)$ ▷ Map function and input into $\mathbb{R}^2$,
  while (x, y) not an accepting configuration of M_u **do** ▷ as needed for Lemma 19
    $(x, y) \leftarrow g_{M_u}(x, y)$
  end while
  return x

To transform x into base 4, the outer loop runs p times, so the inner loop executes a total of p^2 iterations. The transformation of y into base 4 requires polynomially many steps in the precision of y that we need. The final loop depends polynomially on the complexity t of computing $f(x)$. Thus, Theorem 3 follows directly: instead of extracting the function immediately, we delete the first bits of $\langle f \rangle = \langle f_1 \rangle \# \langle f_1 \rangle \# \ldots \# \langle f_{p-1} \rangle \# \langle f_p \rangle \# \langle f_{p+1} 0 \rangle \# \ldots$ until reaching the n-th $\#$, and then extract f_p, which, due to of fast convergence is 2^{-p}-close to the desired function. We then proceed as in Theorem 2. This program as written above has 11 variables (including the index of the current command). It can hence be implemented by a width 12 ReLU-net by techniques similar to the proof of Theorem 23. ◀

8 Conclusion

We established a framework for approximating functions using recurrent networks and showed that universal approximation is achievable with the ReLU activation function. Beyond proving the existence of a universal network, we also proved that the number of network recurrences corresponds to the complexity of computing the approximated function. Our approach is guaranteed to be precise for functions in $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ and dyadic numbers, while efficiently approximating any continuous function on a bounded domain to arbitrary precision. Additionally, we examined the link between Turing machines and $\mathcal{C}^{0,\mathbb{Q}}_{PAF}$ functions. We extended it to a connection between Turing machines and neural networks and also provided a method for translating pseudocode into equivalent network computations.

Natural further open questions are:

1.) Do the above statements hold for other activation functions? First, note that we have restricted our analysis to bounded domains. In particular, when evaluating $\mathrm{ReLU}(x)$, we ensure that $x \in [-c, c]$ for c the largest weight added to the largest bias in the network, for every intermediate result was assumed to be in $[-1, 1]$. Any function f that coincides with ReLU in a neighborhood $(-\varepsilon, \varepsilon)$ around zero would therefore work as well, because such a function can be linearly stretched into a function $\frac{c}{\varepsilon} f(\frac{c}{\varepsilon} x)$ so that it coincides with ReLU on $[-c, c]$. For example, this implies to the linear sigmoid σ mentioned earlier, as well as to piecewise linear rescalings such as $\sigma'(x) = -1$ if $x \leq -1$, x if $-1 \leq x \leq 1$, 1 if $x \geq 1$. Clearly, since any non-affine function $f \in \mathcal{C}^{0,\mathbb{Q}}_{PAF}$ can be linearly transformed into a function that coincides with the ReLU function on an interval around zero, the same conclusion holds for any rational continuous piecewise affine activation function.

2.) Can the size of the overall network be bounded? What about its width and depth? Universality is known to hold with width 3 for Turing machine simulation, but whether it holds with width 2 remains open. After several attempts, we think this question is

related to the well-known open problem of whether universality can be achieved using one-dimensional systems, such as piecewise affine functions. In our construction, the encoding and decoding mechanisms, as described in this paper, require a ReLU network of width 12. It was not conceived to be width-minimal. As a result, the total width required by our universal construction is currently the maximum between 12 (for encoding/decoding) and 3 (for simulation), making the encoding/decoding part the main bottleneck. A natural question thus arises:what is the minimal width required for encoding and decoding?

References

1 Eric Allender. Circuit complexity, Kolmogorov complexity, and prospects for lower bounds. In *DCFS*, pages 7–13. Citeseer, 2008.

2 Eric Allender, Harry Buhrman, Michal Kouckỳ, Dieter Van Melkebeek, and Detlef Ronneburger. Power from random strings. *SIAM Journal on Computing*, 35(6):1467–1493, 2006. doi:10.1137/050628994.

3 Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491*, 2016. arXiv:1611.01491.

4 Midhun T Augustine. A survey on universal approximation theorems. *arXiv preprint arXiv:2407.12895*, 2024. doi:10.48550/arXiv.2407.12895.

5 Daniel Bertschinger, Christoph Hertrich, Paul Jungeblut, Tillmann Miltzow, and Simon Weber. Training Fully Connected Neural Networks is $\exists\mathbb{R}$-Complete. *Advances in Neural Information Processing Systems (NeurIPS 2023)*, 36:36222–36237, 2023.

6 Manon Blanc and Olivier Bournez. A characterisation of functions computable in polynomial time and space over the reals with discrete ordinary differential equations: Simulation of Turing machines with analytic discrete ODEs. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*, volume 272 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.MFCS.2023.21.

7 Vincent D. Blondel, Olivier Bournez, Pascal Koiran, and John Tsitsiklis. The stability of saturated linear dynamical systems is undecidable. *Journal of Computer and System Science*, 62(3):442–462, May 2001. doi:10.1006/jcss.2000.1737.

8 Digvijay Boob, Santanu S Dey, and Guanghui Lan. Complexity of training ReLU neural network. *Discrete Optimization*, 44:100620, 2022. doi:10.1016/J.DISOPT.2020.100620.

9 Olivier Bournez and Michel Cosnard. On the computational power of dynamical systems and hybrid systems. *Theoretical Computer Science*, 168(2):417–459, November 1996. doi:10.1016/S0304-3975(96)00086-2.

10 Cornelius Brand, Robert Ganian, and Mathis Rocton. New complexity-theoretic frontiers of tractability for neural network training. *Advances in Neural Information Processing Systems*, 36:56456–56468, 2023.

11 Vasco Brattka. Computability & complexity in analysis. Tutorial donné à *Computability in Europe (CIE'2005)*, 2005. doi:10.1016/j.jco.2006.10.001.

12 Vasco Brattka, Peter Hertling, and Klaus Weihrauch. A tutorial on computable analysis. In *New computational paradigms*, pages 425–491. Springer, 2008. doi:10.1007/978-0-387-68546-5_18.

13 Yongqiang Cai. Achieve the minimum width of neural networks for universal approximation. In *The Eleventh International Conference on Learning Representations*, 2023. URL: https://openreview.net/forum?id=hfUJ4ShyDEU.

14 Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Sys-*

tems, pages 6571–6583, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/69386f6bb1dfed68692a24c8686939b9-Abstract.html.

15 George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. doi:10.1007/BF02551274.

16 Ingrid Daubechies, Ronald DeVore, Simon Foucart, Boris Hanin, and Guergana Petrova. Nonlinear approximation and (deep) ReLU networks. *Constructive Approximation*, 55(1):127–172, 2022.

17 Vincent Froese and Christoph Hertrich. Training neural networks is np-hard in fixed dimension. *Advances in Neural Information Processing Systems*, 36:44039–44049, 2023.

18 Vincent Froese, Christoph Hertrich, and Rolf Niedermeier. The computational complexity of ReLU network training parameterized by data dimensionality. *Journal of Artificial Intelligence Research*, 74:1775–1790, 2022. doi:10.1613/JAIR.1.13547.

19 Vincent Froese, Christoph Hertrich, and Rolf Niedermeier. The computational complexity of ReLU network training parameterized by data dimensionality. *Journal of Artificial Intelligence Research*, 74:1775–1790, 2022. doi:10.1613/JAIR.1.13547.

20 Robert Ganian, Mathis Rocton, and Simon Wietheger. Training one-dimensional graph neural networks is NP-hard. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025.

21 Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010. URL: http://proceedings.mlr.press/v9/glorot10a.html.

22 Surbhi Goel, Adam Klivans, Pasin Manurangsi, and Daniel Reichman. Tight hardness results for training depth-2 ReLU networks. In *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*, volume 185 of *Leibniz International Proceedings in Informatics*, pages 22:1–22:14, 2021. doi:10.4230/LIPICS.ITCS.2021.22.

23 Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, volume 30, pages 2214–2224, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html.

24 Riccardo Gozzi. *Analog Characterization of Complexity Classes*. PhD thesis, Instituto Superior Técnico, Lisbon, Portugal and University of Algarve, Faro, Portugal, 2022.

25 Daniel S. Graça, Manuel L. Campagnolo, and J. Buescu. Robust simulations of Turing machines with analytic maps and flows. In S. B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE 2005: New Computational Paradigms*, LNCS 3526, pages 169–179. Springer, 2005. doi:10.1007/11494645_21.

26 Boris Hanin and Mark Sellke. Approximating continuous functions by ReLU nets of minimal width. *arXiv preprint arXiv:1710.11278*, 2017. arXiv:1710.11278.

27 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. doi:10.1109/cvpr.2016.90.

28 Ruiyang Hong and Anastasis Kratsios. Bridging the gap between approximation and learning via optimal approximation by ReLU MLPS of maximal regularity. *arXiv preprint arXiv:2409.12335*, 2024. doi:10.48550/arXiv.2409.12335.

29 Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991. doi:10.1016/0893-6080(91)90009-T.

30 Patrick Kidger. *On Neural Differential Equations*. PhD thesis, Mathematical Institute, University of Oxford, 2022. doi:10.48550/arXiv.2202.02435.

31 Patrick Kidger and Terry Lyons. Universal approximation with deep narrow networks. In Jacob Abernethy and Shivani Agarwal, editors, *Proceedings of Thirty Third Conference on Learning Theory*, volume 125 of *Proceedings of Machine Learning Research*, pages 2306–2327. PMLR, 2020. URL: `http://proceedings.mlr.press/v125/kidger20a.html`.

32 Namjun Kim, Chanho Min, and Sejun Park. Minimum width for universal approximation using ReLU networks on compact domain. In *The Twelfth International Conference on Learning Representations*, 2024. URL: `https://openreview.net/forum?id=dpDw5U04SU`.

33 Ker-I Ko. *Complexity Theory of Real Functions*. Progress in Theoretical Computer Science. Birkhaüser, Boston, 1991. `doi:10.1007/978-1-4684-6802-1`.

34 Pascal Koiran, Michel Cosnard, and Max Garzon. Computability with low-dimensional dynamical systems. *Theoretical Computer Science*, 132(1-2):113–128, September 1994. `doi:10.1016/0304-3975(94)90229-1`.

35 Anastasis Kratsios and Léonie Papon. Universal approximation theorems for differentiable geometric deep learning. *Journal of Machine Learning Research*, 23(196):1–73, 2022. URL: `https://jmlr.org/papers/v23/21-0716.html`.

36 Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *ICLR*, 2016.

37 Hongzhou Lin and Stefanie Jegelka. Resnet with one-neuron hidden layers is a universal approximator. *Advances in neural information processing systems*, 31, 2018.

38 Chenghao Liu, Enming Liang, and Minghua Chen. Characterizing resnet's universal approximation capability. In *Forty-first International Conference on Machine Learning*, 2024.

39 Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, pages 3276–3285. PMLR, OpenReview.net, 2018. URL: `https://openreview.net/forum?id=B1LatDVUM`.

40 Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *Advances in neural information processing systems*, 30, 2017.

41 Cristopher Moore. Generalized shifts: unpredictability and undecidability in dynamical systems. *Nonlinearity*, 4(3):199–230, 1991. `doi:10.1088/0951-7715/4/2/002`.

42 Anirbit Mukherjee, Amitabh Basu, Raman Arora, and Poorya Mianjy. Understanding deep neural networks with rectified linear units. In *International Conference on Learning Representations (ICLR 2018)*, 2018.

43 Ian Parberry. *Circuit complexity and neural networks*. MIT press, 1994. `doi:10.7551/mitpress/1836.001.0001`.

44 Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

45 Amaury Pouly. *Continuous models of computation: from computability to complexity*. PhD thesis, Ecole Polytechnique and Unidersidade Do Algarve, defended on july 6, 2015 2015. https://pastel.archives-ouvertes.fr/tel-01223284, Prix de Thèse de l'Ecole Polytechnique 2016, Ackermann Award 2017.

46 Hava T. Siegelmann. *Neural networks and analog computation - beyond the Turing limit*. Progress in theoretical computer science. Birkhäuser, 1999.

47 Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991. `doi:10.1016/0893-9659(91)90080-f`.

48 Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, September 1994. `doi:10.1016/0304-3975(94)90178-3`.

49 Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, February 1995. `doi:10.1006/jcss.1995.1013`.

50 Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

51 Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015. arXiv:1505.00387.

52 Heribert Vollmer. *Introduction to circuit complexity: a uniform approach*. Springer Science & Business Media, 1999.

53 Klaus Weihrauch. *Computable Analysis - An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000. doi:10.1007/978-3-642-56999-9.

54 Dmitry Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural networks*, 94:103–114, 2017. doi:10.1016/J.NEUNET.2017.07.002.

55 Dmitry Yarotsky. Elementary superexpressive activations. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11932–11940. PMLR, 18–24 July 2021. URL: https://proceedings.mlr.press/v139/yarotsky21a.html.

56 Dmitry Yarotsky. Structure of universal formulas. *Advances in Neural Information Processing Systems*, 36:54876–54902, 2023.

57 Shijun Zhang, Jianfeng Lu, and Hongkai Zhao. Deep network approximation: Beyond ReLU to diverse activation functions. *Journal of Machine Learning Research*, 25(35):1–39, 2024. URL: https://jmlr.org/papers/v25/23-0912.html.

58 Xingcheng Zhang, Zhizhong Li, Chen Change Loy, and Dahua Lin. Polynet: A pursuit of structural diversity in very deep networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 3900–3908. IEEE Computer Society, 2017. doi:10.1109/CVPR.2017.415.