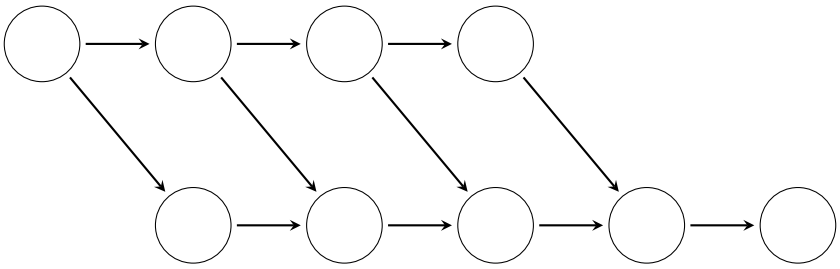
The Expressibility of Neural Networks for Regression and Classification Problems

Corbin Diaz

March 18, 2026



Contents

1 Introduction

Machine learning is an fast growing study that has recently caught the public's eye due to its growing power in the realm of artificial intelligence. Many modern day machine learning algorithms are built around neural networks, a model inspired by the way neurons interact with each other to form computations in the brain. Neural networks are widely used throughout machine learning, despite many not understanding how they may work and often treating them as a "black box". To be understand the capabilities of these models, one may wonder how well neural networks can even model these machine learning tasks.

Machine learning involves gathering data to fit a function of variables that predicts phenomenon in the real world. For instance, one may gather property statistics and financial data to help predict rent prices in their city, or one may scrape different attributes of email text to determine whether or not it is spam. To fit a neural network for these tasks, one has to consider many factors: How large does my network need to be to be within some prediction error? What activation functions should I use? Can my network approximate the function arbitrarily close? Importantly, these questions arise separately from training the model itself. That is, neural networks may be unable to approximate the desired phenomenon regardless of its parameters due to an inadequate "size" or architecture. In this case, training would get one nowhere. Therefore, it is desirable to study how well these networks can fit various phenomenon, that is, what their expressibility capabilities look like for various types of functions.

We will split our exploration into two parts. In the first part, we will explore fitting continuous variable outputs as in our rent prices example, a problem known as *regression*. In the second, we explore fitting binary variable outputs as in our spam detection example, a problem known as *classification*.

1.1 Basic Definitions

Before we explore the capabilities of these models, it is important to understand the underlying structure of neural networks. Each networks is comprised of a combination of affine linear functions and non-affine linear functions. These affine linear functions, defined below, allow us to model networks as "connecting neurons" together through weights, while the non-affine linear functions make the model complex enough to fit non-linear functions.

Definition 1.1 (Affine Linear Functions)**.** An *affine linear function* is a function $A : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ of the form $Ax = Wx + b$, where $W : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ is linear (usually given by a matrix) and $b \in \mathbb{R}^{d_2}$ is a constant function. In the context of neural networks, The values of all W_i are called the *weights* and the components of all b_i are called the *biases* of the neural network, and together these make up the *parameters* of the neural network.

Definition 1.2 (Feedforward Network)**.** A *feedforward network*, *neural network*, *network*, or *NN* is given by the following:

1. Positive integers $d_0, d_1, \dots d_L$.
2. Affine linear functions $A_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ for $i = 1, \dots, L$.
3. An *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ which is **not** affine linear.

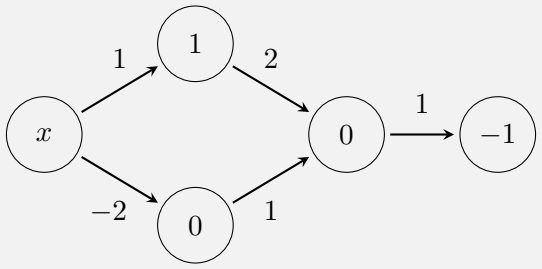
We define the neural network $F : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ as

$$F(x) = A_L \circ \sigma \circ A_{L-1} \circ \cdots \circ \sigma \circ A_2 \circ A_1(x)$$

where we define $\sigma : \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$ componetwise. That is, $\sigma(x_1, \dots x_{d_i}) = (\sigma(x_1), \dots, \sigma(x_{d_i}))$.

We can also write networks as a convenient diagram.

Example 1.1 (Network Diagrams)**.** Consider the diagram of the neural network F:



The values on each segment give the weights of the neural network, while the values in each circle (or *neuron*) gives the biases. There are no biases in the input layer, so we used these neurons to showcase our input values. Each column of neurons gives another layer, while the rows showecases the number of neurons in each layer. Note we apply the activation function σ except the output layer. In total, the network compuets as

$$F(x) = \sigma(2\sigma(x+1) + \sigma(-2x)) - 1$$

for some activation function σ.

Our next obvious question is what are some examples of activation functions. We define common examples of activation functions below. In particular, most of our results will be in terms of the popular ReLU activation function.

Example 1.2 (Activation Functions)**.** Types of activation functions from $\mathbb{R}$ to $\mathbb{R}$.

- The *Rectified Linear Unit*, denoted by ReLU, is defined as $\text{ReLU}(x) = \max(x, 0)$, or the identity function for positive x and zero everywhere else.
- The *Heaviside* function, denoted by H, is defined piecewise as

$$H(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

- The *Sigmoid* or logistic function is "smooth approximation" of the Heaviside, and is defined by $\sigma(x) = 1/(1 + e^{-x})$.
- The *Sigmoid Linear Unit* or *Swish* function, denoted SiLU, is a "smooth approximation" of ReLU, and is defined by $\sigma(x) = x/(1 + e^{-x})$.
- The *hyperbolic tangent*, denoted tanh, is defined as $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$, with a shape similar to sigmoid but with different growth properties.

Each of these functions has pros and cons, and their properties have been extensively studied in the realm of training for machine learning. It is true that better activation functions can more easily express certain types of functions. However, another important property, one that we will focus on for this paper, is the overall architecture of neural networks, which describes the "size" of the network. We define the relevant terminology below.

Definition 1.3. The positive integers $d_0, \ldots, d_L$ an activation function σ as given in Definition 1.2 determine the *architecture* of the neural network. Let $\mathcal{N}(\sigma; d_0, \ldots, d_L)$ denote the set of neural networks given by the architecture $(\sigma; d_0, \ldots, d_L)$.

Definition 1.4. Let $F \in \mathcal{N}(\sigma; d_0, \ldots, d_L)$ for some architecture $(\sigma; d_0, \ldots, d_L)$. The *depth* of F is given by L and the *width* of F is given by $w = \max(d_0, \ldots, d_L)$. Furthermore, we can rewrite this set of neural networks as

$$\mathcal{N}_{d_0}^{d_L}(\sigma; w, L) = \mathcal{N}(\sigma; d_0, \ldots, d_L)$$

We refer to any networks where $L = 1$ as *shallow* and any networks where $L > 1$ as *deep*.

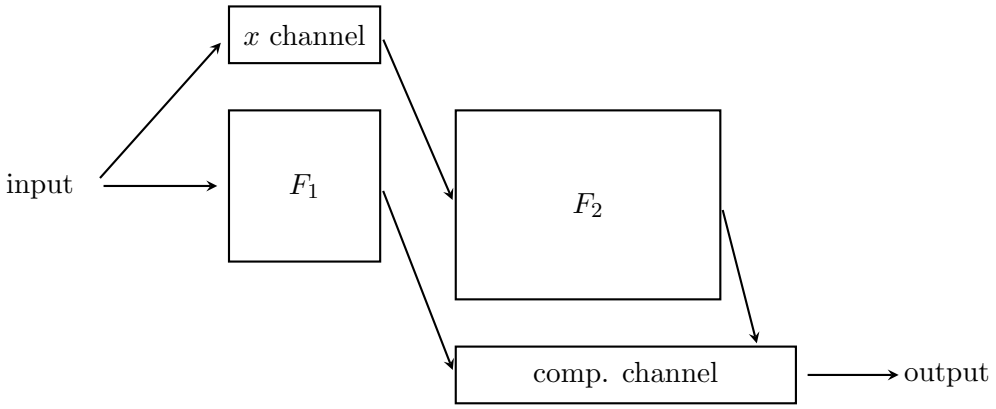
Finally, note that since each matrix W_i is at maximum $w \times w$, we have that the parameters of a neural network of width w and depth L grows quadratically with w and linearly with L. We can formalize this rate of growth using big-O notation.

Definition 1.5 (Big-O Notation)**.** $f(x) = O(g(x))$ if and only if $|f(x)| \leq M|g(x)|$ for some constant M for all x if the domain of f.

Therefore we can write that the number of parameters of $F \in \mathcal{N}_{d_0}^{d_L}(\sigma; w, L)$ is $O(w^2 L)$. Hence, it is more desirable to fix the width of the network and scale the depth, rather than scale the width, as this would result in quadratic growth of parameters to train. For this reason, we will focus on results with deep neural networks throughout this paper, or networks with constant width and arbitrary depth.

1.2 Computational Channels

One important technique for neural networks that we should mention before continuing is the computational channel. Frequently, to aid in our calculations we utilize different "channels" to hold our store values. For instance, consider a ReLU neural network of width 1 in which all weights are 1 and the biases are 0. If the inputs are positive, then such a network will hold the value of the input for all layers until it is outputted. We refer to such a channel as an x *channel*, as it holds the value of x. A *computational channel* in particular refers to using a channel to add values together. For instance, imagine we have two networks F_1 and F_2, and we wish to add their results. We can have two separate channels, one x channel that holds the value of x to input into both F_2 later, and one computational channel that is zero until we add the results of F_1 and F_2 together, and output the result. If F_1 and F_2 have width and depths w_1, w_2 and L_1, L_2, respectively, the resulting $F = F_1 + F_2$ network would have width $\max(w_1, w_2) + 2$ and depth $L_1 + L_2 + 1$. We will use this method to help aid in creating functions while keeping width mostly constant.



2 Regression

To use neural networks in the context of regression problems, one would be concerned with how well it can express continuous functions. That is, given that we want our model within some error ϵ, can we guarantee such a neural network exists that fits our functions within that error? How large does such a network need to be? The mathematical results behind these questions are known as the universal approximation theorems. Although there are many of these theorems, the one we will focus on in this paper concerns deep ReLU networks. Similar results, however, also hold for shallow networks and networks with more general activation functions.

Theorem 2.1 (Universal Approximation of Deep ReLU Networks)**.** *Let $f : [0,1] \to \mathbb{R}$ be a C^k function. Then for every $\epsilon > 0$ there exists $F \in \mathcal{N}_1^1(ReLU; w, L)$, where $w \leq C$ for some universal constant C which depends on k and $M = \sup_{x\in[0,1]} |f^{(k)}(x)|$ and L is $O(\epsilon^{-1/k} \log(1/\epsilon))$ such that*

$$\sup_{x\in[0,1]} |F(x) - f(x)| \leq \epsilon$$

To prove this result, we will show we can approximate more and more complex functions, starting with piecewise linear functions, then polynomials, and finally general continuous functions.

2.1 Piecewise Linear Functions

Before we prove the universal approximation theorem, its important to understand deep ReLU networks of small width. For example, deep ReLU networks of width 1 take the form

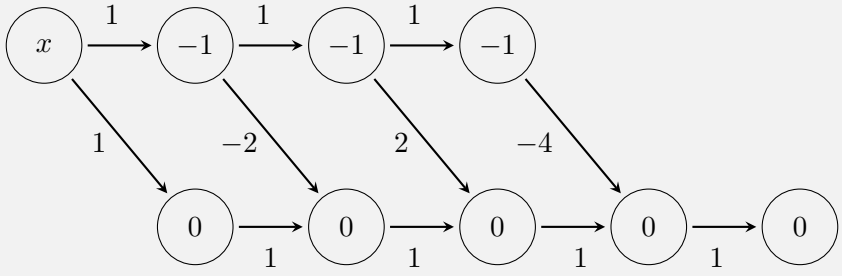
$$a_n \text{ReLU}(a_{n-1}\text{ReLU}(\dots a_1 \text{ReLU}(a_0 x + b_0) + b_1) \dots) + b_{n-1}) + b_n$$

for some real constants $a_0, \dots, a_n$, and $b_0, \dots, b_n$. Such a function is only non-zero when all inputs into the ReLU function are non-negative. Yet since ReLU is continuous, one can show that this forms an intersection of open sets, which itself must be open. Hence the result is always linear on an open interval in $\mathbb{R}$, and constant everywhere else.

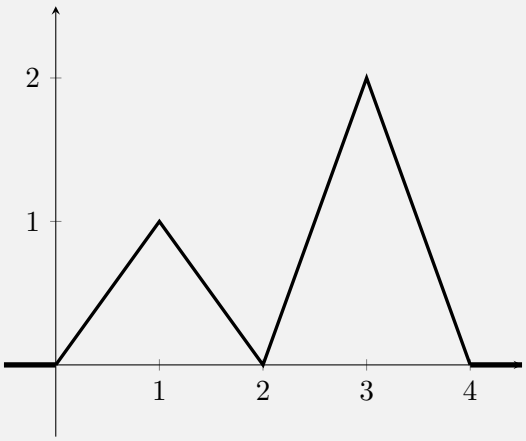
Example 2.1 (ReLU Network of Width 1)**.** The network given by $F(x) = \text{ReLU}(-\text{ReLU}(2x + 1) + 1)$ is equivalent to $-2x$ when both $2x + 1 > 0$ and $1 - \text{ReLU}(2x + 1) > 0$, which is when $-1/2 < x < 0$, and is constant everywhere else.

What if we had width 2? This greatly opens up the flexibility. For instance, by using a computational channel, we can create "tent functions", as shown below.

Example 2.2 (Tent Function)**.** Let $F(x)$ be the ReLU network of width 2:



This network, when worked out, creates the following tent like function:



Note the top channel acts as our x channel, shifting x to each of the breakpoints, while the bottom channel acts as our computational channel, adding out results weighted according to slope (noting that the weights accumulate)

This feature of ReLU networks immediately gives our first major result.

Proposition 2.1. *Let $f : [0,1] \to \mathbb{R}$ be a piecewise linear function with n pieces. Then there exists $F \in \mathcal{N}_1^1(ReLU; 3, L)$ such that $F(x) = f(x)$ for all $x \in [0,1]$, where L is $O(n)$.*

Sketch of Proof. We won't go into detail on how to build the entire network to avoid getting too technical, but here is a sketch. Let $f : [0,1] \to \mathbb{R}$ be a piecewise linear function with n pieces. Then it must have $n-1$ breakpoints $t_1, t_2, \dots t_{n-1} \in [0,1]$, or points of discontinuity. Now, build a tent function that realizes f on the interval $[0, t_1]$, and at each odd break point $t_3, t_5, \dots$. Call this network F_1. Since f is linear, we then have that $f - F_1$ is another tent function, which we can realize with another network F_2. Finally, we utilize a computational channel method to calculate $F_1 + F_2 = f$. Since F_1, F_2 are both width 2, and we already have a computational channel, adding another x channel (to allow us to input into F_2) gives us width 3. Since we need to build layers based off of each breakpoint, the depth scales with the number of pieces, giving that L is $O(n)$. ■

While Prop. 2.1 seems rather basic, this result becomes very powerful in helping us realize any continuous function. Indeed, continuous functions can be approximated by piecewise linear functions, so intuitively it may seem we are already done. However, we also want to more carefully construct our approximation to ensure we can build a network without exploding our depth. To do so, we will utilize approximations of polynomials, and use Prop. 2.1 to build on all our results.

2.2 Polynomials

Now that we've seen how well ReLU networks can realize piecewise linear functions, our next question becomes how well it can realize more complex functions, for example, polynomials. Approximating polynomials is useful as it gives us a well suited approximation of continuous functions through Taylor's theorem. We remind the reader, without proof, the result and terminology from Taylor's theorem below.

Theorem 2.2 (Taylor's Theorem)**.** *For $C^{(k)}$ function $f : \mathbb{R} \to \mathbb{R}$ (meaning the k^{th} derivative of f is*

continuous), we have that for any $a \in \mathbb{R}$

$$f(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \cdots + \frac{f^{(k-1)}(a)}{(k-1)!}(x-a)^{k-1} + R_k(x)$$

where

$$R_k(x) = \int_a^x \frac{f^{(k)}(t)}{(k+1)!}(x-t)^{k-1}\, dt$$

We call $P_k(x) = f(x) - R_k(x)$ the Taylor polynomial of f of degree $k-1$ centered at a.

In this section, we will avoid formulating full proofs, as writing these networks out exactly can get rather technical. However, we will give sketches for each result, which should give some intuition on how these networks can be constructed. We start with the most basic polynomial, the squared function, to help build our results.

Proposition 2.2. *Let $f : [0,1] \to [0,1]$ be given by $f(x) = x^2$. Then for every $\epsilon > 0$, there exists $F \in \mathcal{N}_1^1(ReLU; 3, L)$ such that*

$$\sup_{x \in [0,1]} |F(x) - f(x)| \leq \epsilon$$

where L is $O(\log(1/\epsilon))$.

Sketch of Proof. We can approximate x^2 using a linear interpolation between points $x = t_1, \ldots t_{n-1}$, creating a piecewise linear function of n pieces. By Prop. 2.1, we can represent this interpolation exactly by a neural network of width 3 and depth L, where L is $O(n)$. One can then show that the error ϵ decays exponentially with the number of intervals n. That is, ϵ is $O(2^{-n})$. Rearranging in terms of ϵ then gives n is $O(\log(1/\epsilon))$, and hence L is $O(\log(1/\epsilon))$. ■

It may seem like x^2 is a rather specific example. However, such a construction actually leads directly into computing *any* polynomial through the use of products. The next two lemmas will be key parts to proving Theorem 2.1.

Lemma 2.1. *For every $\epsilon > 0$ there exists $F \in \mathcal{N}_2^1(ReLU; w, L)$, where $w \leq C$ for some universal constant C and L is $O(\log(1/\epsilon))$ such that*

$$\sup_{x,y \in [0,1]} |F(x,y) - xy| \leq \epsilon$$

Sketch of Proof. We can rewrite products in terms of the square function using

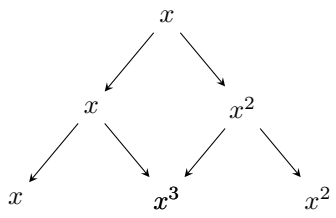
$$xy = \frac{1}{2}(x+y)^2 - \frac{1}{2}x^2 - \frac{1}{2}y^2$$

Then, it follows from Prop. 2.2 that we can approximate this equation through neural networks of constant width and depth that is $O(\log(1/\epsilon)$. ■

Lemma 2.2. *Let $P : [0,1] \to \mathbb{R}$ be a polynomial. Then for every $\epsilon > 0$ there exists $F \in \mathcal{N}_1^1(ReLU; w, L)$, where $w \leq C$ for some universal constant C which depends on P and L is $O(\log(1/\epsilon))$ such that*

$$\sup_{x \in [0,1]} |F(x) - P(x)| \leq \epsilon$$

Sketch of Proof. This proof is more technically complicated but follows directly from Prop. 2.2 and Lemma 2.1. We can successively create more powers of x utilizing the square function, linear function, and products by growing in a "tree" like fashion. Such a network then has a width that grows depending on the degree of P, which we call k. However, the depth grows like $O(k \log(1/\epsilon)) = O(\log(1/\epsilon)$. Linearly combining each power to match P then gives our result.



■

2.3 Universal Approximation Theorem

The trick to proving Theorem 2.1 is utilizing what is known as a partition of unity, which we define below.

Definition 2.1. A *partition of unity* on X is a set of continuous functions $\phi_i : X \to [0, 1]$ such that, for all $x \in \mathbb{R}$ there exists an open interval around x in X where all but a finite number of $\phi_i(x)$ are zero and

$$\sum_i \phi_i(x) = 1$$

By using a partition of unity, we can essentially "partition" the interval $X = [0, 1]$ into separate pieces so that we can approximate $f(x)$ using polynomials on each small interval, hence approximating $f(x)$ on the entire interval. This is useful, because if we can construct each ϕ in such a way to make it easily realized by a ReLU neural network, then together with 2.1 and 2.2 we can use a neural network to approximate $f(x)$.

To demonstrate the usefulness of the partition of unity, we prove the following lemma:

Lemma 2.3. *For $C^{(k)}$ function $f : [0, 1] \to \mathbb{R}$, there exists a partition of unity $\phi_0, \ldots, \phi_m$ on $[0, 1]$ and sequence of polynomials $P_0, \ldots, P_m$ for some m such that*

$$\sup_{x\in[0,1]} \left| \sum_{i=0}^{m} \phi_m(x) P_m(x) - f(x) \right| \leq \frac{C}{N^k}$$

where C is a universal constant which depends on k and $M = \sup_{x\in[0,1]} |f^{(k)}(x)|$ and N is $O(2m)$.

Proof. First, partition the interval $[0, 1]$ into N pieces, where N is an odd integer (N does not have to be odd, but we force it to be to simplify the proof). Make a partition of unity $\phi_0, \phi_1, \ldots, \phi_m : [0, 1] \to [0, 1]$, where $m = (N - 1)/2$ so that

$$\phi_0(x) = \begin{cases} 1, & 0 \leq x < \frac{1}{N} \\ -N(x - \frac{2}{N}), & \frac{1}{N} \leq x < \frac{2}{N} \\ 0 & \text{otherwise} \end{cases}, \qquad \phi_m(x) = \begin{cases} N(x - \frac{2m-1}{N}), & \frac{2m-1}{N} \leq x < \frac{2m}{N} \\ 1, & \frac{2m}{N} \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

and, for all $0 < i < m$,

$$\phi_i(x) = \begin{cases} N(x - \frac{2i-1}{N}), & \frac{2i-1}{N} \le x < \frac{2i}{N} \\ 1, & \frac{2i}{N} \le x < \frac{2i+1}{N} \\ -N(x - \frac{2(i+1)}{N}), & \frac{2i+1}{N} \le x < \frac{2(i+1)}{N} \\ 0 & \text{otherwise.} \end{cases}$$

Note that each ϕ_i is continuous on $[0,1]$ and we have that

$$\begin{aligned} \sum_{i=0}^{m} \phi_i(x) &= \phi_j(x) && (2.1) \\ &= 1 && \text{if } x \in \left[\frac{2j}{N}, \frac{2j+1}{N}\right] \text{ for some } 0 \le j \le m \end{aligned}$$

$$\begin{aligned} \sum_{i=0}^{m} \phi_i(x) &= \phi_{j-1}(x) + \phi_j(x) && (2.2) \\ &= -N\left(x - \frac{2j}{N}\right) + N\left(x - \frac{2j-1}{N}\right) \\ &= 1 && \text{if } x \in \left[\frac{2j-1}{N}, \frac{2j}{N}\right] \text{ for some } 0 < j \le m \end{aligned}$$

And hence this forms a partition of unity on $[0,1]$. Now, define $P_i(x)$ to be the Taylor polynomial of f of degree $k-1$ centered at $(2i + \frac{1}{2})/N$, and let

$$\hat{f}(x) = \sum_{i=0}^{m} \phi_i(x) P_i(x). \tag{2.3}$$

This gives

$$\begin{aligned} |\hat{f}(x) - f(x)| &= \left| \sum_{i=0}^{m} \phi_i(x) P_i(x) - \sum_{i=0}^{m} \phi_i(x) f(x) \right| \\ &\le \sum_{i=0}^{m} \phi_i(x) \left| P_i(x) - f(x) \right|. \end{aligned} \tag{2.4}$$

Let $M = \sup_{x \in [0,1]} |f^{(k)}(x)|$. We now split into two cases to utilize Taylor's Theorem:

1. *Case 1:* $x \in \left[\frac{2j}{N}, \frac{2j+1}{N}\right]$ for some $0 \le j \le m$, so by Eq (2.1) we have

$$\begin{aligned} |\hat{f}(x) - f(x)| &\le \phi_j(x) \left| P_j(x) - f(x) \right| \\ &= \left| \int_{\frac{2j+\frac{1}{2}}{N}}^{x} \frac{f^{(k)}(t)}{(k-1)!} \left(x - \left(\frac{2j + \frac{1}{2}}{N} \right) \right)^{k-1} dt \right| && \text{Thm. 2.2} \\ &\le \frac{1}{2N} \frac{M}{(k-1)!} \left(\frac{1}{2N} \right)^{k-1} \end{aligned}$$

2. *Case 2:* $x \in \left[\frac{2j-1}{N}, \frac{2j}{N}\right]$ for some $0 < j \leq m$, so by Eq (2.2) we have

$$\begin{aligned}
|\hat{f}(x) - f(x)| &\leq \phi_{j-1}(x)\,|P_{j-1}(x) - f(x)| + \phi_j(x)\,|P_j(x) - f(x)| \\
&= \phi_{j-1}(x)\left|\int_{\frac{2j-\frac{3}{2}}{N}}^{x} \frac{f^{(k)}(t)}{(k-1)!}\left(x - \left(\frac{2j-\frac{3}{2}}{N}\right)\right)^{k-1} dt\right| \\
&\qquad + \phi_j(x)\left|\int_{\frac{2j+\frac{1}{2}}{N}}^{x} \frac{f^{(k)}(t)}{(k-1)!}\left(x - \left(\frac{2j+\frac{1}{2}}{N}\right)\right)^{k-1} dt\right| \qquad \text{Thm. 2.2} \\
&\leq \phi_{j-1}(x)\frac{3}{2N}\frac{M}{(k-1)!}\left(\frac{3}{2N}\right)^{k-1} + \phi_j(x)\frac{3}{2N}\frac{M}{(k-1)!}\left(\frac{3}{2N}\right)^{k-1} \\
&= \frac{3}{2N}\frac{M}{(k-1)!}\left(\frac{3}{2N}\right)^{k-1}
\end{aligned}$$

Let

$$C = \frac{M}{(k-1)!}\left(\frac{3}{2}\right)^{k}$$

We conclude by the two cases above that

$$\sup_{x\in[0,1]} |\hat{f}(x) - f(x)| \leq \frac{C}{N^k}$$

■

Finally, we have the tools we need to prove Theorem 2.1.

Proof of Theorem 2.1. By Lemma 2.3, we have there exists $\hat{f}(x) = \hat{f}(x) = \sum_{i=0}^{m} \phi_i(x)P_i(x)$. for a partition of unity $\phi_0, \ldots, \phi_m$ and polynomials $P_0, \ldots, P_m$ such that $\sup_{x\in[0,1]} |\hat{f}(x) - f(x)| \leq C/N^k$ Note that each ϕ_i is piecewise linear with at most 5 pieces by definition, and hence by Proposition 2.1 can be made by a ReLU neural network of constant width and depth.

Let $\epsilon > 0$. Choose N large enough so that

$$C/N^k < \epsilon/2. \tag{2.5}$$

By Lemma 2.2, we can construct ReLU network F_i of width bounded by some universal constant and depth $O(\log(2N/\epsilon))$ such that

$$\sup_{x\in[0,1]} |F_i(x) - P_i(x)| \leq \frac{\epsilon}{2N}. \tag{2.6}$$

Now, by Lemma 2.1 we can further construct a ReLU network F_i' of width bounded by some universal constant and depth $O(\log(2N/\epsilon))$ such that

$$\sup_{x\in[0,1]} |F_i'(x) - \phi_i(x)F_i(x)| \leq \frac{\epsilon}{2N}. \tag{2.7}$$

Now, utilizing a computational channel, we can then realize a ReLU network $F(x) = \sum_{i=0}^{m} F_i'(x)$ of bounded

width and depth $O(m\log(2N/\epsilon))$. We then have that:

$$\begin{aligned}
|F(x)-f(x)| &\leq \left|\sum_{i=0}^{m} F_i'(x) - \sum_{i=0}^{m}\phi_i(x)F_i(x) + \sum_{i=0}^{m}\phi_i(x)F_i(x) - \hat{f}(x) + \hat{f}(x) - f(x)\right| \\
&\leq \left|\sum_{i=0}^{m} F_i'(x) - \sum_{i=0}^{m}\phi_i(x)F_i(x)\right| + \left|\sum_{i=0}^{m}\phi_i(x)F_i(x) - \hat{f}(x)\right| + \left|\hat{f}(x) - f(x)\right| \\
&\leq \sum_{i=0}^{m}\left|F_i'(x) - \phi_i(x)F_i(x)\right| + \sum_{i=0}^{m}\phi_i(x)\left|F_i(x) - P_i(x)\right| + \left|\hat{f}(x) - f(x)\right| \\
&\leq \sum_{i=0}^{m}\frac{\epsilon}{2N} + \sum_{i=0}^{m}\phi_i(x)\frac{\epsilon}{2N} + \frac{\epsilon}{2} && \text{By Eqs (2.5), (2.6), (2.7)} \\
&< \frac{N}{2}\frac{\epsilon}{2N} + \frac{N}{2}\frac{\epsilon}{2N} + \frac{\epsilon}{2} && \text{Since } N \text{ is } O(2m) \\
&= \epsilon
\end{aligned}$$

In our calculation, we always had universally bounded width. To calculate our depth, we note that N is $O(2m)$, and heave our depth is

$$\begin{aligned}
O(N\log(2N/\epsilon)) &= O(e^{-1/k}\log(e^{-1/k}/\epsilon)) && \text{By Eq 2.5} \\
&\leq O(e^{-1/k}\log(1/\epsilon))
\end{aligned}$$

■

We have now shown that neural networks can approximate any continuous function. This gives insight into the expressibility of our networks and how capable they are to handle regression problems. While we only shows this for ReLU networks, this can be extended to other activation functions. Indeed, we mainly used Prop. 2.1 to build all our results, and hence showing similar results would yield the same approximation. This theorem also gives some intuition behind how well these networks are at approximating networks, and how big of networks we need.

Remark 2.1. If our function is smooth, this can be used to show we can approximate functions up to an error ϵ using a depth that grows logarithmic with the inverse of our error. Say we are modeling a regression problem, and can achieve a precision of about $\epsilon = 10^{-1}$ with about 10 layers. If we wanted to increase our precision to $\epsilon = 10^{-3}$, we would need about 3 times as many layers.

Unfortunately, this theorem only proves the existence of such networks. While a higher precision would be possible with 30 layers, in practice training methods may not achieve such a network with so many parameters to handle. However, this theorem still proves that using these networks to model regression is possible, and gives a sense of how scalable they may be.

3 Classification

We have shown that neural networks, in particular deep ReLU neural networks, have the capability of approximating any $C^{(k)}$ function. This is particularly useful in areas of regression in which we are interested in modeling a continuous real variable given by some function f. However, another realm of machine learning involves classification, which models a discontinuous variable that gives different discrete classes.

For example, consider a function $f(x) : \mathbb{R} \to \{0, 1\}$ which measures body temperature to predict if one has a fever. Such a function is inherently discontinuous and hence the universal approximation theorems do not apply. In fact, since the ReLU function is continuous, one can show that no sequence of ReLU NNs converges uniformly to such a function. By contrast, this function can be given by shallow Heaviside network of width 1:

$$F(x) = H(x - T)$$

where T is the maximum healthy body temperature. This may motivate one to study classification using neural networks with Heaviside activation, a model known as a *perceptron*. Another method is to compose a network with H. For instance, we can consider when the output of a ReLU NN is positive or negative to determine whether it is 1 or 0, respectively.

How well can neural networks of this form approximate other binary functions? For example, we've show shallow Heaviside networks $F : \mathbb{R} \to \mathbb{R}$ of width 1 can differentiate between two points $x_1 < T$ and $x_2 > T$, for some T. By contrast, trying to differentiate the points $x_1 < x_2 < x_3$ so that $F(x_1) = F(x_3) = 0$ and $F(x_2) = 1$ is not possible and would require a width of at least 2. Thus, it is not trivial that we could fit any binary function this way. The mathematical term for how well function classes can differentiate points is known as its *VC dimension*. Our main result of this section will be showing that the VC dimension of ReLU networks scales at least quadratically with its number of layers, thus giving some intuition behind how well it can express functions for classification.

3.1 Shattering Points and VC Dimension

First, we formalize what we mean by "differentiating points" by defining the notion of *shattering*:

Definition 3.1 (Shattering)**.** Let $\mathcal{F}$ be a set of functions $f : X \to \mathbb{R}$ for some compact set X. We say that $\mathcal{F}$ *shatters* a set $S = \{x_1, \ldots, x_n\} \subseteq X$ if for all functions $g : S \to \{0, 1\}$ there exists $f \in \mathcal{F}$ such that $H \circ f = g$, where $\mathcal{H}$ is the Heaviside function.

In other words, for all subsets $A \subseteq S$, there exists $f \in \mathcal{F}$ such that

$$\begin{aligned} f(x_i) &\geq 0 \quad \text{if } x_i \in A \\ f(x_i) &< 0 \quad \text{if } x_i \notin A \end{aligned}$$

for all $x_i \in S$. In the realm of classification, we can think of A as the set of positives, or 1s.

Example 3.1. For the above scnario, the set of functions given by $H(x - T)$ for some $T \in \mathbb{R}$ shatters any set $\{x_1, x_2\}$ where $x_1 < x_2$, but cannot shatter any set $\{x_1, x_2, x_3\}$ where $x_1 < x_2 < x_3$.

This "maximum shattering set" is formalized through *VC Dimension*.

Definition 3.2 (VC Dimension)**.** Let $\mathcal{F}$ be a set of functions $f : X \to \mathbb{R}$ for some compact set X. The *VC Dimension* of $\mathcal{F}$, given by $\text{VCdim}(\mathcal{F})$, is the largest n such that there exists $S \subseteq X$ with $|S| = n$ such that $\mathcal{F}$ shatters S. If no such largest n exists, we say that $\text{VCdim}(\mathcal{F}) = \infty$.

Example 3.2 (Shallow Networks)**.** We have shown that, $\text{VCdim}(\mathcal{N}_1^1(H;1,1)) = 2$. Note shallow heaviside networks of this form have at most 2 pieces. Indeed, if we needed the shatter a set of 3 points, we would need at least 2 pieces, and hence need a shallow heaviside network of width 2. In general, we need a width of w to have $w+1$ continuous pieces and hence shatter a set of $w+1$ points. The same holds true for shallow ReLU networks: a width of w gives $w+1$ linear pieces and can shatter $w+1$ points. This gives

$$\text{VCdim}(\mathcal{N}_1^1(H;w,1)) = \text{VCdim}(\mathcal{N}_1^1(\text{ReLU};w,1)) = w+1$$

Note each of these networks require $3w$ parameters to define. One may think that, if $\mathcal{F}$ is given by ρ parameters, then $\text{VCdim}(\mathcal{F}) \leq \rho$. However, this is not the case. It turns out for the case of neural networks that the depth can dramatically increase the VC dimension. We are now ready to present our main result for this section:

Theorem 3.1 (Lower Bound On VC Dimension of Deep ReLU Networks)**.** *There exists a constant C such that*

$$VCdim(\mathcal{N}_1^1(ReLU;C,L)) = \Omega(L^2)$$

Note here that $h(x) = \Omega(g(x))$ implies that there exists some $c > 0$ and some y such that for all $|x| > |y|$, we have that $|h(x)| \geq c|g(x)|$. Hence, we have that the VC dimension of the ReLU network $F : \mathbb{R} \to \mathbb{R}$ of constant width and depth L grows at least as fast as L^2, despite its parameters only growing as fast as L!

3.2 Bit Extraction

To prove Theorem 3.1, we will first need to prove a lemma involving bit extraction, which we define below.

Definition 3.3. A *bit extraction* function is a function $B_N : X \times Y \to \{0,1\}$ for some integer n, where $X = \{0,1,\ldots,2^n-1\}$ and $Y = \{0,1,\ldots,n-1\}$ such that for all $x \in X, y \in Y$, we have that $B_n(x,y) = b_y$, where b_y is the y^{th} binary bit of x. That is, we have each b_y is given by

$$x = \sum_{i=0}^{n-1} 2^i b_i, \quad b_i \in \{0,1\}$$

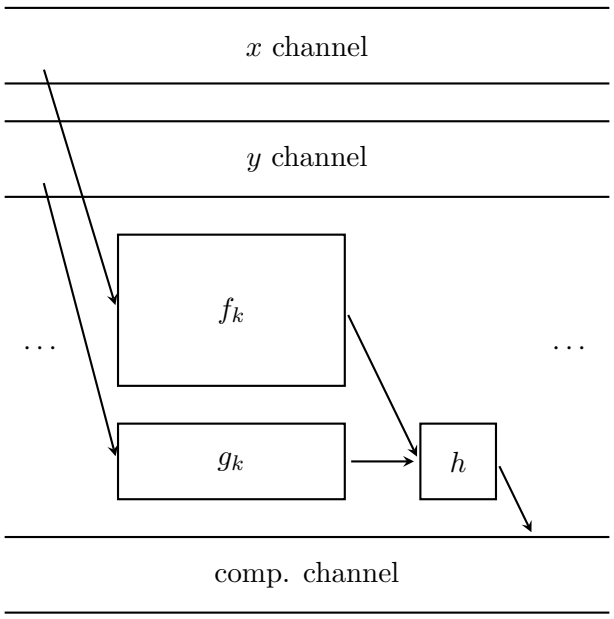
As you could imagine, bit extraction acts a bridge from neural networks to binary functions, although it is not immediately obvious. We will see that bit extraction allows us to "condense" information. That is, we can condense information about a quadratic amount of points into a bit extraction function that can be constructed in a linear about of layers. Our next step thereofre is showing this bit extraction construction is possible:

Lemma 3.1. *There exists $F \in \mathcal{N}_2^1(ReLU;6,L)$ such that $F(x,y) = B_n(x,y)$, where L is $O(n)$.*

Proof. Our construction will involved the gate functions $f_k(x)$ that check whether $x \geq 2^k$ and the tent functions $g_k(y)$ that check if $y = k$, both outputting 0 if no and 1 if yes on $\mathbb{Z}^+$, for all $0 \leq k \leq n-1$. By combining the results with an "AND" gate $h(a,b) = \text{ReLU}(2(a+b)-3)$, we can check for both the bit we are extracting and the value of that bit (Note that $h(0,0) = h(1,0) = h(0,1) = 0$, but $h(1,1) = 1$).

We can then string all the results together in a computational channel by adding the outputs of each h. If we are extracting the k^{th} bit, and its value is 1, then the sum will be one. This is because $h(f_k(x), g_k(y)) = 1$, but $h(f_i(x), g_i(y)) = 0$ for all $i \neq k$ since $g(y) = 0$. If the value is 0, then $h(f_i(x), g_i(y)) = 0$ for all i, and hence the sum will be 0.

The function f_k can be given by a piecewise linear function which is 0 for $x \leq 2^k - 1$, and 1 for $x \geq 2^k$, and linear in between. By our discussion in section 2.1, this can be given by a network of constant depth and width 1. The function g_k is a tent function with a value of 1 at $y = k$, which by our discussion in section 2.1 can be given by a network of constant depth and width 2. We can then have an x channel and y channel throughout the network to hold these values. We compute f_k and g_k, then combine both outputs into h. This gives maximum depth of 5 at all times. We then utilize a computational channel to add the results of each h output as described, bringing our total width to 6.



Since each f_k, g_k has constant depth, and increasing the value of n only increases the amount of f_k, g_k we need to string together horizontally, we have that the depth L grows linearly with n. That is, L is $O(n)$. ■

3.3 Binary Functions

We are now ready to complete our proof of Theorem 3.1.

Proof of Theorem 3.1. We want to show that given any set $X = \{0, 1, \ldots, n^2\}$ for some n, and function $f : X \to \{0, 1\}$ can be realized a network $F \in \mathcal{N}_1^1(\text{ReLU}; C, L)$, for some constant C, where L is $O(n)$. Define the integers $A_0, \ldots, A_{n-1}$ through their binary representation given by

$$A_0 = \sum_{i=1}^{n} 2^{i-1} f(i) \qquad \text{(so } f(i) \text{ is the } i^{th} \text{ bit of } A_0\text{)}$$

$$A_1 = \sum_{i=1}^{n} 2^{i-1} f(n+i) \qquad \text{(so } f(n+i) \text{ is the } i^{th} \text{ bit of } A_1\text{)}$$

$$\vdots$$

$$A_{n-1} = \sum_{i=1}^{n} 2^{i-1} f(n(n-1)+i) \qquad \text{(so } f(n(n-1)+i) \text{ is the } i^{th} \text{ bit of } A_{n-1}\text{)}$$

Notice how through this binary representation, $A_0, \ldots, A_{n-1}$ contain all the information about f on X. Now we define the functions

$$d(x) = \frac{\lceil x \rceil}{n} - 1, \tag{3.1}$$

$$r(x) = x - n \cdot d(x) \tag{3.2}$$

for $x \in X$. We can think of $d(x)$ as the "divider function", as it divides x by L and gives the result rounded to the nearest integer. Additionally, $r(x)$ is the "remainder function", as it gives the remainder of the resulting division. Note how $d(x)$ gives the index i of the constant A_i in which $f(x)$ appears in, In fact, we can define a new function $a(i) = W_i$, so that we have $a(d(x)) = A_i$. Note that all of these functions are piecewise continuous. Indeed, $d(x), r(x)$, and $a(i)$ all have at most n breakpoints on the domain $[0, n^2]$, and hence are piecewise linear with $n + 1$ pieces. By Prop. 2.1, we have that each can be realized by a ReLU network of width 3 and depth $O(n)$.

Additionally, $r(x)$ gives the bit of A_i that $f(x)$ is at. Hence, we can proceed to use bit extraction, which in total gives $B_n(a(d(x)), r(x)) = f(x)$. Lemma 3.1 gives that B_n can be given by a ReLU network of width 6 and depth $O(n)$. Hence, we can compose each of these functions together as the following: First, have the input enter a neural network that realizes $d(x)$ and $r(x)$. Adding them using an x channel and computational channel gives a maximum width of 6 with depth $O(n)$. Then, feed the value of $d(x)$ into a neural network realizing $a(i)$, while holding the value of $r(x)$. Finally, feed these outputs as inputs into a neural network realizing B_n, and output the result.

Notice how the maximum width is always 6, and the depth is always $O(n)$ for each network. Hence, the final network has width $C = 6$ with depth L that is $O(n)$. ■

Remark 3.1. This shows that if we had n^2 data points that we needed to differentiate, we would need around n layers. This calculation seems pessimistic, be keep in mind this is the worst case scenario, in which each data point is labeled as distributed as possible in space. This is unlikely, as points closer in space are usually labeled similarly in classification problems. Say we had a data set of 100 data points, and can achieve good model accuracy with around 5 layers. If we then trained on a dataset of 10,000 data points, we would need at most 10 times more layers, but likely need a lot fewer. Another important note is that this is the lower bound of the VC dimension. In fact, VC dimension may grow a lot higher, in which case we would need less layers.

As in the case for regression, while this is not particularly helpful in determining how well we can actually train to find these models, these results still give intuiton for how possible utilizing networks is for classification problems.

4 Conclusion

This paper gives expressibility results for ReLU neural networks in terms of continuous and discrete functions, giving one a sense of how well suited these models are for regression and classification machine learning problems, respectively. Such results can be extended to other activation functions as well. These results give a sense of the capabilities of neural networks for machine learning, making it no longer feel like a "black box" and helping understand how these models express various functions.

5 Acknowledgements

The author is very grateful to Ben Weinkove for teaching him this quarter and providing much of the content and proofs for this paper.