Deriving Multi-layer Scaffolding of Compositional Neural Networks from Existing, Monolithic Networks

Zachary Rowland

April 2021

Abstract

Neural networks are notoriously difficult to understand which prompts many researchers to indiscriminately increase the size of neural networks in hopes of achieving higher accuracy in classification or prediction tasks. However, lower-complexity models run faster, are more predictable, and offer more insight into the type of problem being solved. This paper offers a discussion of the difficulties and possibilities of finding low-complexity mathematical representations of tasks typically assigned to neural networks. We conclude by proposing an evolutionary equation-learning algorithm that finds polynomial approximations of a network. We apply this algorithm to learn a concise equation for image classification on the MNIST data set.

1 Introduction

Learning systems have always been an important part of data analysis. The simplest method of learning a function from data is the linear regression, but today's deep neural networks have proven remarkably capable of learning a variety of non-linear functions. However, deep neural networks are still not well understood. While a simple linear model between two real variables is described with two parameters, deep neural networks can have hundreds of thousands of parameters that specify its precise behavior. And while the mathematical theory behind the fitting of a linear regression is well studied, the training of a neural network via gradient descent is messy, inexact, and sub-optimal.

Nevertheless, it is common among researchers and data scientists to use neural networks with ever-increasing complexity, driven by a desire for models that exhibit higher accuracy on prediction and classification tasks. However, accuracy is not the only metric by which to judge a model, and pursuing accuracy alone can have consequences.

First, and most obvious, a lower complexity model computes its result using less computation. Lin et. al. [4] hypothesizes that once trained, neural networks

are almost certainly performing far more computations than the task requires in principle. So extra effort spent reducing the size of the model will be paid off with a speed-increase after deployment.

Second, lower complexity models avoid the problem of *over-fitting*, the inability of models to extrapolate predictions beyond the domain on which it was trained. The dimensionality of data is increasing as image and audio processing become more and more useful, but it is becoming harder to predict how even small alterations to the input will affect the output of a neural network.

Finally, having a low-complexity representation of a data set is in some way what it means to *understand* the relationships among the data. Scientists and mathematicians define common functions like $\sin(x)$ or e^x because they describe incredibly common relationships within systems involving circles, waves, and rates of change. Low-complexity mathematical models can be compared with models for other systems and data sets.

In this paper, we explore the possibility of understanding neural network behavior by employing polynomial regression to find low-complexity approximations.

2 The Difficulties of Understanding Neural Network Approximation

Neural networks are hard to understand because their mathematical representation is dissimilar to any other mathematical models we usually come across. To illustrate, consider the following function that represents a trained neural network with three hidden nodes using the standard sigmoid activation function $\sigma(x) = (1 + e^{-x})^{-1}$:

$$\begin{aligned}\hat{f}(x) &= 1.98\sigma(2.32x - 0.27) \\ &+ 2.18\sigma(-2.20x + 6.91) \\ &+ 1.81\sigma(2.37x - 14.45) - 3.07\end{aligned} \tag{1}$$

This function turns out to be a very good approximation of $\sin(x)$ for $x \in [0, 2\pi]$ as demonstrated by Figure 1. We can measure the quality of this approximation using the Euclidean-norm of the difference vector:

$$||f|| = \sqrt{\int_0^{2\pi} (f(x))^2 \, dx} \tag{2}$$

$$\mathsf{error}(\hat{f}, f) = ||\hat{f} - f|| \tag{3}$$

The error between $\sin(x)$ and the function from equation (1) is only 0.081. The quality of this approximation can be better understood by calculating the %-error, the error as a proportion of the Euclidean norm of our target function:

$$\%\mathsf{error}(\hat{f}, f) = \frac{||\hat{f} - f||}{||f||} \times 100\% \tag{4}$$

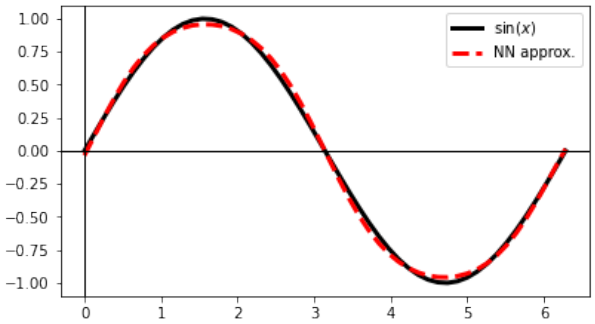

Figure 1: The function $\sin(x)$ plotted with the neural network approximation from equation 1 over the interval $[0, 2\pi]$. The area between the two curves is only 0.17 and their Euclidean distance separation is only 0.081.

The norm of $\sin(x)$ is $\sqrt{\pi} = 1.77$ which makes the percent error only 4.57%. Unfortunately, while it is easy to calculate the accuracy of a neural network approximation given a target function, there is no known procedure to find possible target functions given some error constraint aside from randomly guessing and checking. Such a procedure would require knowledge on some level of *how* the neural network arrives at a result. Is there some analytical connection between equation 1 and the function $\sin(x)$? A good question is whether or not there is some theory that formalizes, and makes intuitive, the connection between the parameters and structure of f and the more commonly used sine function.

To illustrate the challenge, we will contrast neural networks with polynomials which, as we will discover, have two useful properties that make them amenable to approximation. Given data $(x_0, y_0)...(x_n, y_n)$, there is a unique n-degree polynomial $p(x; \boldsymbol{\theta}) = \sum_{i=0}^{n} \theta_i x^i$ that precisely models this data. This polynomial can be found by solving a simple linear system (we will assume that the x_i values in the data are all distinct):

$$\mathbf{0} = X\boldsymbol{\theta} - \boldsymbol{y} \tag{5}$$

$$\Leftrightarrow \begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots \\ 1 & x_1 & x_1^2 & \\ & \vdots & & \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \end{bmatrix} - \begin{bmatrix} y_0 \\ y_1 \\ \vdots \end{bmatrix} \tag{6}$$

Lower-degree, best-fit polynomials can be found through a similar method called regression. For degrees less than n, the X matrix becomes non-square and the system from equation (5) will probably be unsolvable. So instead, the goal is to minimize the distance between $X\boldsymbol{\theta} - \boldsymbol{y}$ and the zero vector:

$$\boldsymbol{\theta}_{min} = \min_{\boldsymbol{\theta}} ||X\boldsymbol{\theta} - \boldsymbol{y}||^2 \tag{7}$$

This problem can be generalized past polynomials to learning systems in general.

We will define a learning system to be a function of the form $\hat{f}(x;\boldsymbol{\theta})$ where the vector $\boldsymbol{\theta}$ is called the model parameters and $x \in \mathbb{R}$ is the function's main input. We will use $\hat{f}(\boldsymbol{x};\boldsymbol{\theta})$ to denote the vector obtained by applying $\hat{f}(;\boldsymbol{\theta})$ element-wise to the values in $\boldsymbol{x}$. Learning can be defined as discovering parameter values $\boldsymbol{\theta}_{min}$ that minimizes the square-distances between the mapped inputs $\hat{f}(\boldsymbol{x};\boldsymbol{\theta}_{min})$ and the target outputs $\boldsymbol{y}$:

$$\boldsymbol{\theta}_{min} = \min_{\boldsymbol{\theta}} ||\hat{f}(\boldsymbol{x};\boldsymbol{\theta}) - \boldsymbol{y}||^2 \tag{8}$$

For neural networks, finding $\boldsymbol{\theta}_{min}$ is difficult. We are forced to rely on iterative methods like gradient descent which are not even guaranteed to converge to a global minimum, only a local minimum. But for polynomials, $\hat{f}(\boldsymbol{x};\boldsymbol{\theta}) = X\boldsymbol{\theta}$ is linear with respect to $\boldsymbol{\theta}$ which means $\boldsymbol{\theta}_{min}$ is unique and calculable from a simple formula: $\boldsymbol{\theta}_{min} = (X^tX)^{-1}X^t\boldsymbol{y}$.

Conceptually, equation 8 can be understood in terms of vectors. $\hat{f}(\boldsymbol{x};\boldsymbol{\theta})$ and $\boldsymbol{y}$ are both vector representations of some function f in an $(n+1)$-dimensional space, $\boldsymbol{\theta}$ is a vector representation of f in some lower dimensional space, and $\hat{f}(\boldsymbol{x};)$ is a transformation from the lower to the higher dimensional space. Equation 8 says to find the $\boldsymbol{\theta}$ whose image under $\hat{f}(\boldsymbol{x};)$ is closest in a Euclidean sense to our target function $\boldsymbol{y}$.

The fact that optimal polynomial coefficients can be directly calculated from the data reveals two important properties of polynomials as learning systems. Let $\mathcal{F} = \{\hat{f}(\boldsymbol{x};\boldsymbol{\theta}) : \boldsymbol{\theta} \in \mathbb{R}^d\}$ be the set of possible approximations given particular inputs $\boldsymbol{x}$. Then for polynomials:

1. (Approximation Uniqueness) There exists a unique function $\hat{f}(\boldsymbol{x};\boldsymbol{\theta}_{min}) \in \mathcal{F}$ such that $||\hat{f}(\boldsymbol{x};\boldsymbol{\theta}_{min}) - \boldsymbol{y}||^2$ is locally minimal (and therefore also globally minimal across $\mathcal{F}$).

2. (Parameter Uniqueness) There is a one-to-one mapping between functions in $\mathcal{F}$ and parameter vectors $\boldsymbol{\theta}$. In other words, every function in $\mathcal{F}$ is characterized by a single parameter vector. For polynomials, this is a consequence of the standard basis functions $\{1, x, x^2, \dots\}$ being linearly independent.

Neural networks on the other hand violate both of these properties:

1. There may exist may neural network configurations that are locally optimal which makes finding close approximations a process of trial and error.

2. The function implemented by a neural network can be characterized by many sets of parameters θ. For example, $\sigma(x) = 1 - \sigma(-x)$. This complicates any attempt to gain information about the network's input-output behavior by examining the network's parameters and structure.

Part of the reason neural networks don't have nice properties like this is because the feed-forward layers perform function composition. Complex functions built as a sum of terms are generally much easier to understand than those built via function composition.

3 Related Work

The interplay between expressivity and complexity of neural networks has been studied. Safran et. al. [5] concluded that deep networks learned more accurately than shallow networks of similar complexity when trained to learn L_2-indicator functions and L_1 radial functions. Chui et. al. [3] found that deep networks are also better at localized approximation, i.e. that when a target function is modified only on a small local area of the domain, only a few neurons in a trained network need to be retrained to retain an optimal approximation. Achieving localized approximation is an important step toward developing human-understandable learning models.

Model compression attempts to reduce a neural network's complexity by pruning off pieces that have little effect on the network's behavior. Cheng et. al. [2] categorizes model compression techniques into four basic strategies:

1. Parameter pruning and quantization
2. Low-rank factorization
3. Transferred/compact convolutional filters
4. Knowledge distillation

Alternatively, some researchers are ditching neural networks in favor of new learning systems whose behavior is more intuitive to humans. The *fuzzy inference system* is one such learning system.

4 Searching for Analytical Models Embedded in Neural Networks

One way we could start is by defining a mechanized scheme for how a neural network *could* approximate a given function, but not necessarily how the network *will* approximate that function after training. For example, Lin [4] points out that the operation of multiplying two real numbers $\mathbb{R}^2 \to \mathbb{R}$ can be approximated to arbitrary accuracy with a single hidden layer with four nodes:

$$\text{mult}(x_1, x_2) = \lim_{h \to 0} \frac{\sigma(ha) + \sigma(-ha) - \sigma(hb) - \sigma(-hb)}{4h^2\sigma''(0)}$$

$$\text{where} \quad a = x_1 + x_2 \qquad b = x_1 - x_2$$

This approximation is independent of the activation function σ used, only requiring $\sigma''(0) \neq 0$. This multiplication operation can be used to construct neural networks that approximate any polynomial. The only additional component needed is a way to propogate values unchanged from one layer to the next, in other words, an approximation of the identity function:

$$\mathrm{id}(x) = x = \lim_{h \to 0} \frac{\sigma(hx) - a(0)}{h\sigma'(0)}$$

These two components, along with the native addition operation, can be used to define neural networks that approximate any multivariate polynomial. Furthermore, Taylor's theorem provides a way of constructing polynomials that approximate any differentiable curve on some domain. Consequently, given a function g, we can generate sequences of neural networks that approach g by changing h or by approximating more terms in the Taylor expansion of g.

Theoretically, a tool could be built that searches for these Taylor constructions in trained neural networks and identifies the original function being approximated. However, building such a tool might not be practical for the reasons described above. There are many ways of combining mult and id that yield the same function. For example, different compositions of multiplication often yield the same power of x e.g. $\mathrm{mult}(\mathrm{mult}(x,x),\mathrm{mult}(x,x)) = \mathrm{mult}(x,\mathrm{mult}(x,\mathrm{mult}(x,x)))$. This means that the set of network configurations we are searching for is large and hard to describe.

Second, requiring h to approach zero leads to neural networks with very large and small parameters, especially as powers of h become involved. In practice, neural networks are initialized randomly with numbers roughly in the single digits, and the training rate will probably be too small to push these parameters high enough while simultaneously being too large to make subtle changes to parameters close to zero. In essence, real networks will almost certainly never learn parameters like this.

5 Simplification via Regression Fitting

As evidenced by the discussion in the previous section, analyzing the input-output behavior of a neural network may be computationally easier than examining the network parameters. Brunton et. al. [1] describes an algorithm called SINDy for finding sparse mathematical representations of time data. SINDy finds a best fit function of the form $f(x) = w_1 f_1(x) + w_2 f_2(x) + \cdots + w_n f_n(x) + b$ for data (x_t, y_t) given functions $f_1 \ldots f_n$. More precisely, it finds the coefficients w_i and b that optimize the following approximation:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \end{bmatrix} \approx \begin{bmatrix} f_0(x_0) & f_1(x_0) & \\ f_0(x_1) & f_1(x_1) & \cdots \\ \vdots & \vdots & \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \end{bmatrix} + b \begin{bmatrix} 1 \\ 1 \\ \vdots \end{bmatrix}$$

The approximation above is specifically for functions $f : \mathbb{R} \to \mathbb{R}$, but it generalizes to data of arbitrary dimensions. SINDy is traditionally used to find sparse mathematical representations of dynamical systems given a set of time data.

To apply this regression-based approach to approximate neural networks, we need data generated by the neural network (x_t, y_t) and a set of "base functions" $\{f_1, \ldots, f_n\}$ that will become the framework of our model. We can generate data

from the neural network by feeding through uniformly-distributed or random inputs in the neural network's domain of interest. For simple networks (such as $\mathbb{R} \to \mathbb{R}$ networks over a finite interval $[a, b]$), it is possible to generate data that evenly covers the entire possible input space. However, uniform coverage for larger input spaces may not be possible (such as over a set of possible images). In this case, we can trust that the training and testing data is representative of all inputs of interest.

The library functions should be chosen carefully to have a few desired properties. First, they should be linearly independent, and ideally orthogonal, so that the regression procedure will yield a single set of weights that describe the best-fit model. Second, linear combinations of the functions should be able to approximate other functions of interest. Polynomials and sinusoids are two great candidates for library functions.

Fitting a regression to input-output data for a neural network would find a model that is fundamentally a sum of simple terms, but it may be useful to find other mathematical representations instead. Since neural networks are compositions of layers, it may make sense to break the network into sections and find a mathematical representation of each section separately via regression. We will call this a "heuristic approach" because the network parameters may serve as a gentle guide for discovering good approximations, but searching for concise equations will ultimately come down to trial-and-error.

Neural networks can be split up in a few ways. First, pretty much all neural networks are compositions of layers. A single layer is just a linear transformation followed by applying an activation function which is already a sufficiently concise mathematical representation, so a neural network chunk could be a few sequential layers together. Additionally, there are benefits (fewer parameters, faster training) to using networks that are not fully connected such as convolutional neural networks. This means that finding equations for intermediate nodes would be more feasible since intermediate nodes would not be functions of all the inputs, but only some of the inputs.

6 Fitting Polynomials & Sinusoids

Suppose we know that a neural network $\hat{f}$ was trained on some unknown common analytical function f in a predefined set. The set could include functions such as x^2, $\sin(\pi x)$, $\log(x+1)$. How can we discover which function it was trained on? Well, neural networks are trained to minimize an error function (usually a square-difference error function), so if we calculate the error between the network approximation $\hat{f}$ and each function in our collection, the function with the lowest error will most likely be the correct function.

This may take some time though, so is there a smarter way? Basically any method of comparing a neural network approximation with an analytical function is going to involve computing some distance metric between vector representations of $\hat{f}$ and f. So what if we intentionally chose low-dimensional vector representations. For example, let's represent $\hat{f}$ and f with three num-

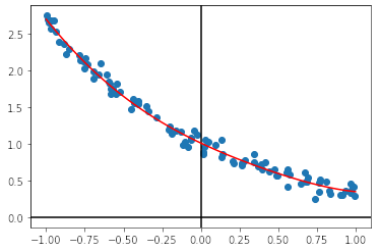

Figure 2: $f(x) = e^{-x}$

bers: the values of the functions at $x = -1, 0, 1$. We can easily split any function represented this way into eight categories depending on the signs of the three numbers. As another example, suppose we represented functions with two numbers: the first and second derivatives at $x = 0$ of the best-fit quadratic function. We can split functions into nine categories depending on which derivatives are near $1, 0$, or -1:

	$f''(0) = -1$	$f''(0) = 0$	$f''(0) = 1$
$f'(0) = -1$	$-e^x$	$-\sin(x)$	e^{-x}
$f'(0) = 0$	$\cos(x)$		$-\cos(x)$
$f'(0) = 1$	$-e^{-x}$	$\sin(x)$	e^x

These vectors have an added benefit: all functions that differ by only a constant have the same vector representation.

7 Image Classification

Moving from simple $\mathbb{R} \to \mathbb{R}$ functions to a more practical scenario comes with challenges. We will discuss these challenges and some possible solutions by learning equations for image classification using the MNIST dataset. A function trained on the MNIST dataset will have the form $\mathbb{R}^{784} \to \mathbb{R}^{10}$ mapping 28-by-28-pixel images into an activation value for each of the ten possible digits 0 through 9. Throughout this section, we will mostly focus on the output for the digit 0 and ignore the other outputs treating the function as $\mathbb{R}^{784} \to \mathbb{R}$.

We will learn this function using a 3-layer fully-connected network with output dimensions 183, 57, and 10. These numbers were chosen to have approximately the same ratio between consecutive layers ($784/183 \approx 183/57 \approx 57/10$). We will also only consider one of the ten outputs at a time.

Data will be gathered by feeding the 10,000 MNIST test images through the trained network and collecting the output values corresponding to a single digit. This was considered preferable to feeding through randomly generated input data.

The value assumed by the pixel in row $r \in [0, 27]$ and column $c \in [0, 27]$ will be referenced using either two subscripts $x_{r,c}$ or just one x_{rn+c} (where n is the number of columns).

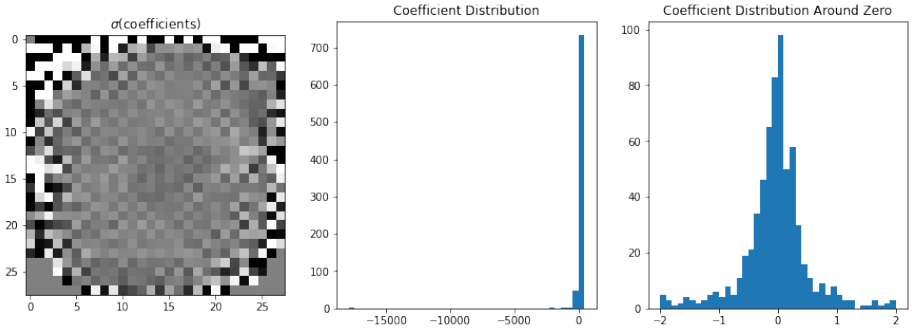

Figure 3: *Left*: A graphical representation of the learned coefficients for the zero output node. White pixels represent positive weights while black pixels represent negative weights. *Middle*: Distribution of all coefficients. *Right*: Distribution of coefficients near zero.

7.1 Linear Model

As a start, we will fit a simple linear regression from the 784 input pixels to the activation for `0`. The output of the regression is 784 coefficients, one for each pixel, depicted in Figure 3. As the coefficient distribution shows, most of the coefficients cluster around zero, but there are several outliers that are extremely negatively correlated. For this reason, Figure 3 displays the pixels normalized using the sigmoid function $\sigma(x) = (1+e^{-x})^{-1}$. The R^2 for this linear regression is 0.82.

Already from a linear regression, the function of the neural network is apparent. The ring of light-grey pixels indicates the pixels that are more likely to be activated in an image of a `0`, and the cluster of darker pixels in the middle show pixels that are usually dark. The coefficients around the edges are mostly random noise because most of these pixels are never activated anyway.

7.2 High-Variance Pruning

A model with 784 terms could hardly be considered tractable. It is pretty clear that some inputs are more important for determining the output than others, so we need metrics of sorting the important pixels from the unimportant ones.

First, we can abuse the fact that the inputs in which we are interested, and on which the network was trained, are not evenly distributed across the 784-dimensional input space. In particular, as noted before, the pixels on the edges of the image are never activated, so including them in the equation would effectively just add a bunch of zero constants. In line with this observation, we will hypothesize that pixels with higher variance are more likely to have a greater impact on accuracy.

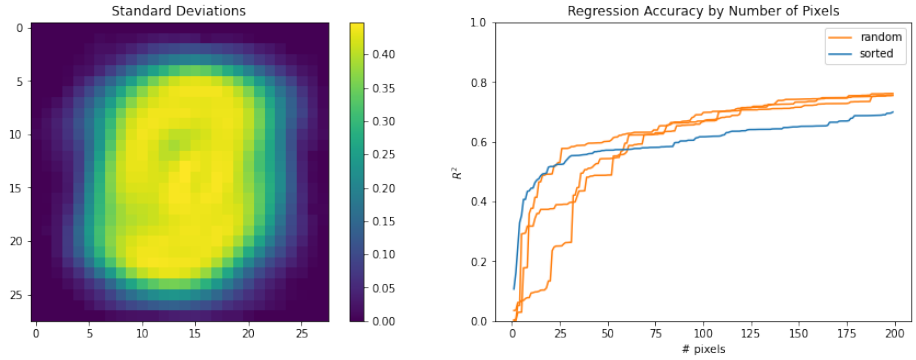

Figure 4: *Left*: Standard deviation of each pixel across the MNIST testing data set. *Right*: Tracks the change of the goodness of fit of a linear model as more pixels become represented. For small models, R^2 increases more quickly if high-variance pixels are added first (blue graph) instead of random pixels (orange graphs).

7.3 VIC metric

Given a mathematical model, it is important to figure out what changes to the model are most likely to increase the accuracy. In hopes of speeding up learning, we introduce a metric for terms called VIC (Variance times Inverse Coefficient) that gives an indication of which mutations are most likely be valuable improvements. The VIC metric for a term is intended to balance two intuitions. First, a term with high variance is more likely to be significant in the calculation than a term with low or zero variance. Second, terms with coefficients close to zero contribute less in the calculation of the final result. Therefore, a term with a low coefficient and high variance does not contribute much towards improving accuracy, but could potentially contribute more if combined with other terms.

$$\text{VIC}(t) = \frac{\text{variance}(t)}{|\text{coefficient}(t)|} \tag{9}$$

7.4 Evolutionary Equation Learning

Lets define some things:

Definition 1 (variable set)**.** *A variable set V is a set of all possible inputs. For image classification, we will consider*
$V = \{x_0, x_1, \ldots, x_{783}\}$.

Definition 2 (term)**.** *A term $t = x_{i_1} x_{i_2} \ldots x_{i_n}$ is a string of elements from V representing a product.*

Definition 3 (model)**.** *A model $(m)_{i=1}^{n}$ is a sequence of terms representing a sum.*

Definition 4 (fitting)**.** *Given a model* m *and labeled data* $d = (x_{1_i}, x_{2_i}, \ldots, x_{n_i}, y_i)$, $\textsc{fit}(m, d)$ *is a sequence containing the best-fit coefficients on the terms in* m*.*

Definition 5 (prune-by-coefficient)**.**

After a linear model, we can define an evolutionary algorithm to find equations with progressively higher R^2 values. The algorithm maintains a list of candidate equations which undergo a cycle of mutation and selection. During mutation, new equations are generated and added to the list by applying random changes to current equations. During selection, the equations with the lowest R^2 values are deleted so that the list returns to its original size. Each equation will be the sum of a series of terms. A *term* is the product of one or more inputs (pixels).

Algorithm 1: Equation Mutation

```
function Mutate(m):
    Data: m: a mathematical model
    Result: The mutated model
    Fit(m) ;
    SortByDescendingVIC(m);
    i ← RandGeo();
    for t ∈ UniformSample(m) do
        m.Append(t × m_i);
    m.Append(nextLinearTerm());
    m.PruneByCoef();
    return m;
function Learn():
    Data: initial linear model m_init, number of equations to keep in the
          bank n, number of iterations i.
    Result: The final best-fit model.
    B ← list of length n, elements initialized to m_init;
    repeat i times
        B′ ← empty list;
        for m ∈ B do
            B′.Append(m);
            B′.Append(Mutate(m));
        sort B′ by descending R² value;
        B′.Truncate(n);
        B ← B′;
    return B[0];
```

The evolutionary learning algorithm was run to learn a polynomial equation for the digit `0` activation. The algorithm maintained a list of ten equations with a maximum degree of five. The initial ten equations were identical: linear equations in the ten highest-varying pixels. The resulting polynomial with $R^2 = 0.75$ is displayed in equation 10. A pictorial representation of the pixels present

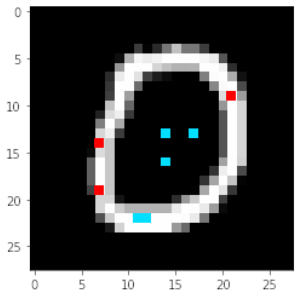

Figure 5: The eight pixels present in the learned polynomial from equation 10. The blue pixels are present in the first eight terms while the red pixels are present in the last two.

in equation 10 is shown in Figure 5 laid over top of an example input.

$$
\begin{array}{rrcccccccc}
\hat{f}(\vec{x}) = & +9.5 & & & & & & & x_{22,11} & x_{22,12} \\
& -10.0 & & x_{13,14} & & & & & x_{22,11} & x_{22,12} \\
& -9.4 & & & x_{13,17} & & & & x_{22,11} & x_{22,12} \\
& -10.9 & & & & & x_{16,14} & & x_{22,11} & x_{22,12} \\
& +9.6 & & x_{13,14} & x_{13,17} & & & & x_{22,11} & x_{22,12} \\
& +10.5 & & & x_{13,17} & & x_{16,14} & & x_{22,11} & x_{22,12} \\
& +11.2 & & x_{13,14} & & & x_{16,14} & & x_{22,11} & x_{22,12} \\
& -11.0 & & x_{13,14} & x_{13,17} & & x_{16,14} & & x_{22,11} & x_{22,12} \\
& +5.6 & x_{9,21} & & & x_{14,7} & & x_{19,7} & & \\
& +3.6 & & & & x_{14,7} & & x_{19,7} & &
\end{array}
\tag{10}
$$

A close analysis reveals a striking resemblance between 10 and boolean operations on the values of the pixels. The first eight terms all contain the factors $x_{22,11}$ and $x_{22,12}$ multiplied by one of the eight subsets of $\{x_{13,14}, x_{13,17}, x_{16,14}\}$. Also, notice how the magnitude of the coefficients are all ≈ 10 while the sign of the coefficients depends on the degree of the term (negative for odd terms and positive for even terms). If we consider each pixel to be a binary input from $\{0, 1\}$, the first eight terms of equation 10 seem to be a numeric representation of the following formula:

$$x_{22,11} \wedge x_{22,12} \wedge \neg(x_{13,14} \vee x_{13,17} \vee x_{16,14})$$

Looking at Figure 5, this interpretation makes sense. If any of the three blue pixels in the center of the image are activated, there is a high chance the image does not represent a zero. Conversely, activation of pixels $x_{22,11}$ and $x_{22,12}$ are highly indicative of zero.

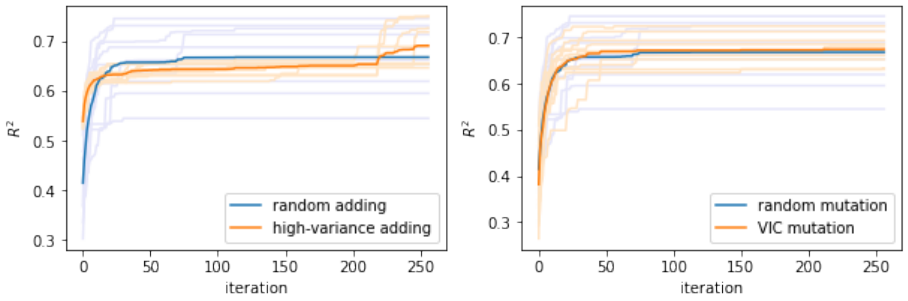

Figure 6: Progress of the evolutionary algorithm over time.

8 Evaluation

While the VIC-metric attempted to improve mutation, in practice it had little benefit over random mutation.

High-variance-adding did produce better fitting equations in the end, although not by much.

9 Conclusion

The promising direction of applying regression methods to neural networks prompts more research into learning systems that learn more traditional mathematical models.

References

[1] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. *Sparse Identification of Nonlinear Dynamics with Control (SINDYc)*. 2016. arXiv: `1605.06682 [math.DS]`.

[2] Yu Cheng et al. "A survey of model compression and acceleration for deep neural networks". In: *arXiv preprint arXiv:1710.09282* (2017).

[3] CK Chui, Xin Li, and Hrushikesh Narhar Mhaskar. "Neural networks for localized approximation". In: *Mathematics of Computation* 63.208 (1994), pp. 607–623.

[4] Henry W. Lin, Max Tegmark, and David Rolnick. "Why Does Deep and Cheap Learning Work So Well?" In: *Journal of Statistical Physics* 168.6 (2017), p. 1223. ISSN: 0022-4715.

[5] Itay Safran and Ohad Shamir. "Depth-width tradeoffs in approximating natural functions with neural networks". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2979–2987.