How Neural Networks Work: Unraveling the Mystery of Randomized Neural Networks For Functions and Chaotic Dynamical Systems

Erik Bollt bolltem@clarkson.edu

Department of Electrical and Computer Engineering
the Clarkson Center for Complex Systems Science
Clarkson University
Potsdam, NY 13699, USA

July 28, 2023

Abstract

Artificial Neural Networks (ANN) have proven to be fantastic at a wide range of machine learning tasks, and they have certainly come into their own in all sorts of technologies that are widely consumed in society as a whole. A basic task of machine learning that neural networks are well suited to is supervised learning, including problems of function learning from samples of input and output data, and also when learning orbits from time samples of dynamical systems. The usual construct in ANN is to carefully and fully train all of the many, perhaps many millions, of parameters that define the network architecture and its use. Full training relies on intense computational resource such as a large multi-threading computers, and also excellent modern optimization algorithms. However, there are certain ANN algorithms that not only get away with random design, with only some training upon output layer, but even thrive. We have previously presented an explanation as to how a specific random ANN, the reservoir computing recurrent neural network architecture succeeds. Now, here we present discussion to explain how the random feedforward neural networks called random project network work, and why they works so well. In particular, we present examples for both general function learning and also for learning a flow from samples of orbits of chaotic dynamical systems. There is an interesting geometric explanation of the success, specifically in the case of the ReLu activation function, that relates to a classical mathematical question that is how configurations of random lines fall in a plane, or how planes or hyperplanes may fall in higher dimensional spaces. Furthermore, how these random configurations lead to a refinement of the domain toward the density of continuous functions by piecewise linear continuous functions.

1 Introduction

The successes of artificial neural networks (ANN) has been spectacular with numerous applications that have touched all of our lives. ANN's have overtaken other methods of machine learning in terms of popularity, even though the computational expense of training large and complex networks is immense. Applications have appeared in so many areas of our common experience, in economics Li and Ma (2010), social sciences Abdi et al. (1999); Garson (1998); Reier Forradellas et al. (2020); Strohmaier (2021), scientific Cuomo et al. (2022); Patel and Goyal (2007); Widrow et al. (1994); Zhang (2010) and engineering disciplines Willis et al. (1991); Rafiq et al. (2001); Himmelblau (2000); Awodele and Jegede (2009) in a wide variety of technology enhancements. While so useful and popular, ANN's typically require significant resources to train. However, certain computational advances have greatly alleviated the training time of back propagation used to train feed-forward neural networks, FFNN's, including algorithmic concepts including stochastic gradient descent Bottou et al. (1991); Amari (1993) or ADAM Jais et al. (2019); Zhang (2018); Kingma and Ba (2014) as well as hardware advances both in terms of chipsets with higher clock speeds and also especially cheap and abundant computational parallelism in the form of graphics processing units, GPU's Oh and Jung (2004); Strigl et al. (2010). These have greatly pushed the size and application of perhaps seemingly otherwise intractable utility of training an ANN with literally upwards of millions of parameters to fit. Nonetheless, understanding that there is a limit to what direct training methods might achieve when attempting to "train" (optimize) models with many parameters, perhaps millions of free parameters, entirely different approaches are welcome.

An innovative approach to the expense of the high-dimensional optimization that results from training all of the internal weights of an ANN is the so-called randomized neural network, RanNN Schmidt et al. (1992), but also more recently called an extreme learning machine, ELM following Huang et al., 2006b, 2015), and also closely related to the random vector functional link Pao et al. (1994). An RanNN skips entirely the cost of training most of a neural network's parameters. Instead, all internal weights and biases are simply chosen randomly. An RanNN is essentially a random neural network. Only the output weights are trained. Furthermore, choosing an identity function for the output weights allows a straight forward least squares solution for this step. Thus the RanNN essentially requires feed-forward only. The partial training that does occur can be computed by efficient linear algebraic matrix computational methods. Amazingly, despite entirely skipping the training of all the many internal parameters, the RanNN has proven to be a highly successful concept in terms of quality of fit. While the original RanNN was a single layer feed-forward ANN concept (SLFN) **?** and Huang et al., 2006b, 2015), multiple layer versions have been tested and these too have also proven to provide excellent and also enjoy the low cost fitting of just the output layer, Ding et al. (2015); Uzair et al. (2018).

Universal approximation theorems are central to the understanding that neural networks can fit functions to arbitrary accuracy as observed from data, Hornik et al. (1989); Cybenko (1989), and there are many varieties for different network architectures and details such as the activation functions, Kidger and Lyons (2020); Nishijima (2021); Hornik (1991); Leshno et al. (1993). It has even been shown that a SLFN including a randomly formed one in the RanNN concept enjoys a universal approximation theorem. While a universal approximation theorem is a general existence of representation statement, we may still wish to know how and why. It is our goal here to describe some of the geometric details as to how such an approximation works, even in the setting of random weights.

This work is in some ways a sequel to our previous work Bollt (2021); Gauthier et al. (2021), which was an effort to understand how reservoir computing, RC, works, despite that they are also a random variant of a neural network, but related to a random variant of recurrent neural networks. Just as with that work, here, we are more so focused on explaining mechanisms of the method than suggesting improvements. There are many good ways to model and forecast dynamical systems within a neural network machine learning framework, by deep learning Sangiorgio et al. (2021,?); Ramadevi and Bingi (2022), and other backpropagation networks such as recurrent neural networks

(RNN) and long-short term memory (LSTM) networks Cheng et al. (2016). However, RC which is itself a random network framework, but somewhat different from RanNN, has proven to be highly successful and popular both accuracy and computational simplicty reasons,

There have been so many recent works in the literature demonstrating the many ways in which neural networks are a powerful and useful entrant to data science in general, including specifically for data-driven science and engineering for dynamical systems Bollt et al. (2018); Li et al. (2017); Saqlain et al. (2022); Li et al. (2020); Lu et al. (2021); Kevrekidis et al. (2020); Karniadakis et al. (2021); Li et al. (2017); Chen et al. (2018); Brunton and Kutz (2022). The large fraction of these simply state a loss function, and then use a neural network as a black box. They rely implicitly on the universal approximation theorem, allowing popular general software packages, most notably the Google Brain Team produced Tensorflow Shukla and Fricklas (2018); Hope et al. (2017), to take the roll of optimizing the millions of parameters associated with a good fit. While these are certainly important contributions showing what a neural can do for us in applied dynamical systems, we take a different goal, which is to explore how these fantastic results are possible. There was a beautiful and early work by Lapedes and Farber in 1988, Lapedes and Farber (1987) that not only described how neural networks work, thus in the title, their ability to make simple bump functions from which obviously other general functions can be estimated, but they did so by demonstration of forecasting dynamical systems such as the Mackey-Glass differential delay equation. Here, we strive to understand how feedforward neural networks in general, and specifically the random neural network system called RanNN can not only succeed, but quite successfully so. While fully trained neural networks achieve excellent performance, but even a randomly selected neural network can also work quite well, and with skill. It is not so much our goal to present RanNN as a better method for forecasting of stochastic processes and dynamical systems, even though it does have certain good traits such as ease of training. Rather, we wish to explain the geometry of how RanNN, and even generally the FFNN architectures. To that end, there have been a number of recent works that have also developed a geometric understanding regarding the success of neural networks, and especially deep learning, including explaining expressivity, Raghu et al. (2016). Finally, we cite an important advancement in neural network training technology in a paper that we consider to be highly complementary to this one; in Bolager et al. (2023), the authors take an entirely new and creative approach to training which is partly based on random sampling but with geometric estimates of the local derivatives of the target functions to be trained, to design a random training methodology that generally specifies precision where it is likely most needed.

In this paper, we will study both single layer and deep learning RanNN, provide good fits to general continuous functions, and we specialize to the ReLu activation functions in the inner layer, and for single variate and also multivariate estimation. We are interested in not just general function estimation by the RanNN random neural network, but more so we are interested in the question of forecasting a dynamical system. Specifically, the flow $\varphi(t; z_0)$ resulting from a differential such as Eq. (53) is just yet another function that may be learned from data, in this case, by a sample orbit, or samples of orbits, and by a random neural network.

The explanation concept is similar for many other popular activation functions, but the ReLU is the simplest and most straight forward to explicitly analyze. For our convenience, we will declare a new notation, RanNN-q to refer to an RanNN with q inner layers. So the original single inner RanNN, meaning it is a SLFN, is stated as an RanNN-1, and we will also subsequently add further notation to declare the nature of those inner layers. In particular, both Montufar et al. (2014); Serra et al. (2018) have concerned with a similar issue as this current paper, which is characterizing the nature of a FFNN and a DFN with ReLu activation represents functions as a piecewise linear function and the number of such linear regions. In Telgarsky (2015) several special case scenarios of representing certain functions are described and contrasted to the general scenario. These all contrast from this current work in two major elements. The first major point is that since we are studying random neural networks, we connect the problem of regions to that of the classical problem of randomly fallen lines in the plane Hall (1981), and the number of domains that result, and the higher dimensional analogues of planes and hyperplanes Stanley et al. (2004). The second major point is that these random architectures can nonetheless be trained to not only work, but work very well, and the analysis here in describes how this works.

2 Review of Feed-forward Neural Networks

An RanNN is a feed-forward neural network, but one where most of the weights and biases are simply chosen randomly. The original RanNN is a SLFN, but one where only the output weights are trained, by a least squares ridge regression process. We will review and set notation for these.

First we specialize to SLFN. Assume N samples of "labelled" data samples, (X_i, Y_j) suitable for supervised learning, with each $X_i \in \mathbb{R}^{d_0}$, and $Y_i \in \mathbb{R}^{d_2}$, as vector valued features from d_0-dimensional real valued input space, and d_2-dimensional output space. For our purposes here, a general $Y_i \in \mathbb{R}^{d_2}$ problem allows convenient ridge regression methods at the heart of RanNN could be used to usefully perform data-driven function fitting tasks, and including even tasks of forecasting complex and chaotic dynamical systems as we will demonstrate in Secs. 5.

Consider a set of N data samples stated,

$$\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{N} \subset \mathbb{R}^{d_0} \times \mathbb{R}^{d_2} = D. \tag{1}$$

Notation $\mathcal{D}$ is the set of data, and D is the set which embedds it. A standard fitting of an ANN, when supervised learning, is a nonlinear regression of a function,

$$f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_2}, \tag{2}$$

to minimize an assumed loss function $\mathcal{L}(\mathcal{D}; \Theta) : D \times T \to \mathbb{R}^+$ with respect to whatever may be the parameters $\Theta \in T$ in the parameter space T describing all the weights and biases of the neural network architecture, and given a data sample in the data space $\mathcal{D} \subset D$.

Following notation summarized in Fig. 1, let

$$X = X^{(0)} = [x_1^{(0)}; x_2^{(0)}; ...; x_{d_0}^{(0)}], \tag{3}$$

describe the input variable. That is, a general data point $X \in \mathbb{R}^{d_0}$, which we do not yet index by a subindex i to denote the i^{i} point, we describe as input into the initial, or $^{(0)}$ layer, and this point in $\mathbb{R}^{d_0}$ has d_0 coordinate values which we each denote by lower case and the second subindex. Therefore, a specific vector data sample $X_i = X_i^{(0)} \in \mathbb{R}^{d_0}$ is denoted by the expanded notation into indices at the input (zero$^{\mathrm{th}}$) column vector,

$$X_i^{(0)} = [x_{1,i}^{(0)}; x_{2,i}^{(0)}; ...; x_{d_0,i}^{(0)}], \quad i = 1, ..., N. \tag{4}$$

Thus X_i as the i$^{\mathrm{th}}$ (vector) datum, and $X_i^{(0)}$ the datum once written into the input of the ANN, essentially synonymously.

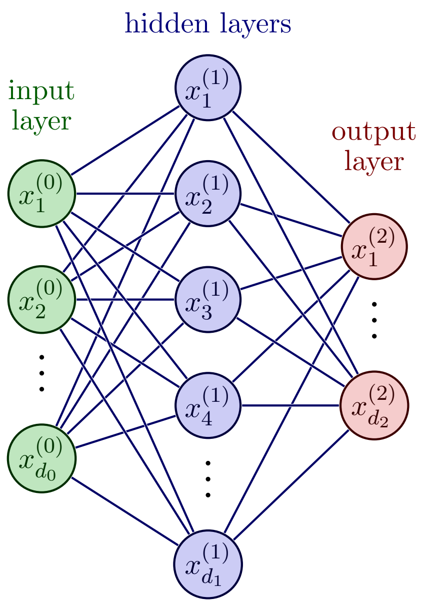


Figure 1: A single layer feed-forward neural network (SLFN) architecture assumes exactly one hidden layer. Input and output node dimensions must match the given data to be fitted, Eqs. (3)-(6). There are weights associated with the all the edges shown, as emphasized in Fig. 2. If the inner weights and biases are chosen randomly, but only output weights are trained, then this would be called an RanNN-1, or simply an RanNN because there is only $q = 1$ inner layer. Contrast to the DFN scenarios shown in Fig. 3.

The first hidden layer states, and the only hidden layer in the case of a SLFN, is the vector variable,

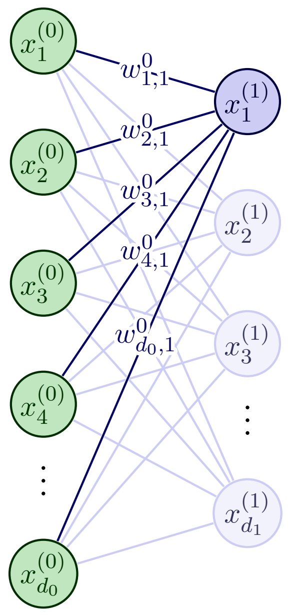
$$X^{(1)} = [x_1^{(1)}; x_2^{(1)}; ...; x_{d_1}^{(1)}] \in \mathbb{R}^{d_1}. \tag{5}$$

Finally, the output layer states are,

$$X^{(2)} = [x_1^{(2)}; x_2^{(2)}; ...; x_{d_2}^{(2)}] \in \mathbb{R}^{d_2}. \tag{6}$$

The notation relates the data to the last (output) layer, $Y_i = X_i^{(2)}$; we use these almost synonymously, indicating the i^{th} output data, and the superscript (2) as output reflects the SLFN assumption.

More generally, labelling $q+1$ layers allows for deep learning networks architectures, but $q = 1$ specializes the description to a SLFN, including the SLFN-RanNN. To this end, we use RanNN and RanNN-1 synonymously to indicate $q = 1$, that there is exactly one hidden layer. See Fig. 1. The general notation RanNN-q allows for multiple hidden layers when $q > 1$. Deep learning, or deep feed-forward networks (DFN) are indicated in Fig. 3, including those with mostly random weights and biases, which would then be called RanNN-q, with $q = 2$ as drawn.



$$= \sigma\left(w_{1,1}^0 x_0^{(0)} + w_{2,1}^0 x_1^{(0)} + \ldots + w_{d_0,1}^0 x_n^{(0)} + b_1^1\right)$$

$$= \sigma\left(\sum_{i=1}^{d_0} w_{1,i}^0 x_i^{(0)} + b_1^1\right)$$

$$\begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \\ \vdots \\ x_{d_1}^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,1}^0 & w_{2,1}^0 & \cdots & w_{d_0,1}^0 \\ w_{1,2}^0 & w_{2,2}^0 & \cdots & w_{d_0,2}^0 \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,d_1}^0 & w_{2,d_1}^0 & \cdots & w_{d_0,d_1}^0 \end{pmatrix}\begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \\ \vdots \\ x_{d_0}^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^1 \\ b_2^1 \\ \vdots \\ b_{d_1}^1 \end{pmatrix}\right]$$

$$X^{(1)} = \sigma\left(\mathbf{W}^0 X^{(0)} + \mathbf{b}^1\right)$$

Figure 2: Highlight of neural network forward propagation from layer 0 (input) to one of the nodes of Layer 1. Weights notation $w_{i,j}^r$ shown indicate transitions from the ith state in layer r to the jth state in layer $r+1$. Here $r=0$ is shown. These can all be collected in matrix notation as indicated, and also with biases so labelled, as summarized by Eqs. (13)-(12).

2.1 The Set of Fitting Parameters

To contrast the savings of a random feed-forward neural network, the RanNN, versus a fully trained network of the same topology, (network connectivity), we now count the training parameters. The savings are obvious. In subsequent sections we will discuss the suprise, which is that despite the mostly untrained, randomly selected parameters, nonetheless, RanNN performs quite well.

Scalar weights $w_{j,k}^r$ indicate interaction from a node, a scalar state x_j^r in an r^{th} layer to the x_k^{r+1} node state in the $r+1^{\text{th}}$ layer. Likewise, one bias b_k^{r+1} is used at each state x_k^{r+1}. Counting parameters, there are $d_0 d_1$ weights $w_{j,k}^{(0)}$ between the input layer and the (first) hidden layer. For a SLFN, there are $d_1 d_2$ weights $w_{j,k}^{(1)}$ between the hidden layer and the output layer. The role of the weights and biases and a threshold function σ in feed-forward transition between layers is written with the simplified matrix form in Fig. 3, and expanded upon in Eqs. (12)-(13).

Collecting the set of parameters, of q hidden layers,

$$\Theta = \cup_{r=0}^{q} \Theta_r, \tag{7}$$

where each of the $q+1$ layers consists of parameter sets Θ_r, associated with the forward propagation from layer r to layer $r+1$,

$$\Theta_r = \{\{w_{j,k}^r\}_{j=1,k=1}^{d_r,d_{r+1}}, \{b_k^{r+1}\}_{k=1}^{d_{r+1}}\}. \tag{8}$$

In the $q=1$ case of a general SLFN, there are in total,

$$|\Theta| = d_0 d_1 + d_1 d_2 + d_1 + d_2, \tag{9}$$

parameters. However, to simplify the parameter fitting process, when we build an RanNN, we a choose zero bias on the final layer, $\{b_k^{q+1} = 0\}_{k=1}^{d_{q+1}}$. Then,

$$|\Theta| = d_0 d_1 + d_1 d_2 + d_1, \tag{10}$$

for any such SLFN. Likewise, the general $q > 1$ DFN with zero bias on the output layer has,

$$|\Theta| = d_0 d_1 + d_1 d_2 + d_2 d_3 + ... + d_q d_{q+1} + d_q = d_q + \prod_{i=0}^{q} d_i d_{i+1}. \tag{11}$$

It is obvious that what makes an RanNN inexpensive is that most of these parameters are chosen randomly. "Common wisdom" might suggest that carefully designing all the weights is critical for any success at all. Nonetheless, surprisingly, even though chosen mostly randomly, training just the few parameters at the output layer yields a successful and powerful machine learning method. Much of the analysis herein discusses a relatively weak assumption that the distribution is absolutely continuous, and thus a broad range of basis functions will result if a large enough collection is chosen. While the fitting step at the last layer will correct poorly (randomly) selected slopes meaning weights w, the offsets together with the slopes design the spread of the basis functions, and thus the parcellation of the domain as we shall see. So a uniform distribution is one for example which tends to lead to a relatively fine parcellation. A recently submitted paper, by Dietrich *et. al.* **?**, as made a major advancement toward designing the random distribution so that results comparable to fully training can be achieved, and we believe this is closely related to the parcellation we discuss results in finer structure where the fitting function demands.

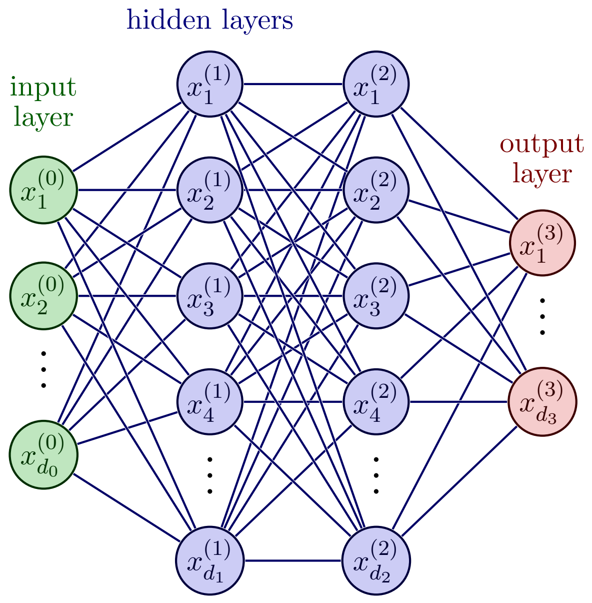


Figure 3: A deep learning architecture, the DFN. Contrast to the SLFN shown in Fig. 1. If the inner weights and biases are chosen randomly, but only output weights are trained, then this would be called an RanNN-2.

2.2 Training Loss and Feed-Forward

The fitting process of a standard ANN involves optimizing a loss function, across all the parameters with respect to the input data. Specifically, a typical loss function,

$$\mathcal{L}(\mathcal{D};\Theta) = \sum_{i=1}^{N} \|F(X_i,\Theta) - Y_i\|_2 + \lambda\mathcal{R}(\Theta). \tag{12}$$

includes a Tikhonov regularity term $\mathcal{R}$ that is usually included to prevent over fitting. Ridge regression, defined by the choice $\mathcal{R}(\Theta) = \|\Theta\|_2$, is a standard, including for RanNN.

To define the feed-forward step, including for a general DFN, $F(X_i,\Theta)$ is defined as a composition function of simple threshold elements, as follows. It is convenient to write the parameters Θ_r as matrices and vectors; again see Fig. 3. Let the $d_r \times d_{r+1}$ matrix W^r, and the $d_{r+1} \times 1$ vector B^{r+1} associated with layer r,

$$[W^r]_{j,k} = w^r_{j,k}, \text{ and, } [B^{r+1}]_k = b^{r+1}_k, j = 1,...,d_r, k = 1,..,d_{r+1}, \tag{13}$$

Forward propagation of data from layer r to layer $r+1$ is stated in matrix, vector form,

$$F_{r,\Theta_r}(X^r) = \sigma_r(W^r X^r + B^{r+1}). \tag{14}$$

The notation stated in Eqs. (5)-(6), describes that X^r is the vector $d_r \times 1$ state variable of values in layer r. Then by composition,

$$F(X,\Theta) = F_{q,\Theta_q} \circ F_{q-1,\Theta_{q-1}} \circ ... \circ F_{0,\Theta_0}(X), \tag{15}$$

defines forward propagation. This notation allows a deep learning, such as the DFN shown in Fig. 3 ($q = 2$) for the general scenario of $q > 1$-hidden layers network versus the SLFN shown in Fig. 1.

The activation functions σ_r we may use at each r^{th}-layer may include a wide variety of popular nonlinear choices including sigmoids, hyperbolic tangents, amongst many. In this work for sake of simplicity of our geometric analysis, we initially choose the popular rectifier (ReLu) function for all but the read-out layer. For the read-out layer activation function, here we always choose the identity function, for sake of how the RanNN is trained by linear regression. That is, let,

$$\sigma_r(s) = ReLu(s) = max(s,0), r < q, \sigma_{q+1}(s) = s. \tag{16}$$

We will allow for smooth activation functions near the end of this paper, as they as they are useful in the dynamical systems flow examples. Threshold functions are interpreted, as usual, to be component-wise evaluations when the argument of the functions is a vector.

2.3 (Not) Fully Training a Feed-forward Neural Network

A number of training strategies are used to minimize the loss function, Eq. (12), including variations on gradient descent, stochastic gradient descent Bottou et al. (1991); Amari (1993), Adam Jais et al. (2019); Zhang (2018); Kingma and Ba (2014), and others. We will leave this point with the obvious assertion that it is hard to optimize a function in a space of potentially millions of parameters, and even trying may have been considered surprising even just a few years ago. Yet, that underpins much of current ANN practice. Clearly this monumentally computationally intensive practice of optimizing extra-ordinarily high-dimensional loss functions, with potentially a massive number of local minimization, works well despite so many obstacles. Modern utility relies both on technical mathematically supported algorithmic advances as well as hardware advances with respect to cheap and widely available multi-threading in the form of GPU chipsets, Oh and Jung (2004); Strigl et al. (2010). Nonetheless, the promise of efficiency of training many fewer parameters presented by RanNN is welcome and potentially significant.

An RanNN **?**, Huang et al. (2006b) is a random neural network, meaning all weights and biases are chosen randomly and only the output layer is trained. By the set-up of choosing the identity function as activation on the final layer, this step is done cheaply with a standard linear regression. We will explore how the seemingly over-simplification of the difficult problem of fully training an SLFN or a DFN as an RanNN still yields not only a cheaper method, but also a successful method. We will consider geometric aspects of feed-forward neural networks in general, and in contrast to full training.

3 Not Fully Training a Random Neural Network: An RanNN-q Enjoys Considerable Savings

Any RanNN-q enjoys a considerable savings in terms of the number of parameters to train, when compared to training a standard ANN. Instead of the number of parameters $|\Theta|$ stated in Eq. (9), an RanNN only trains the output layer, simply leaving the other parameters as randomly selected.

Counting the fitting parameters of Eq. (10) for the RanNN-1, $q = 1$,

$$|\Theta_{out}| = |\Theta_1| = d_1 d_2, \tag{17}$$

are fitted by a least squares step is the major innovation of RanNN. Thus clearly,

$$|\Theta_{out}| = d_1 d_2 \ll |\Theta| = d_0 d_1 + d_1 d_2 + d_1 + d_2, \tag{18}$$

contrasts to Eq. (9). Likewise and more-so in general for an RanNN-q,

$$|\Theta_{out}| = d_q d_{q+1} \ll |\Theta| = d_{q+1} + \prod_{i=0}^{q} d_i d_{i+1}. \tag{19}$$

Instead, all of the other parameters counted in the above product are simply pre-chosen, randomly. Let each,

$$x^r_{j,k} \sim \mathcal{X}, \text{ and, } b^r_k \sim \mathcal{B}, 0 \leq r < q, \tag{20}$$

by two random variables, $\mathcal{X}, \mathcal{B}$. In practice we choose these to be identically independently distributed (i.i.d.). Either a normal distribution or a uniform distribution work well in practice.

The output layer parameters Θ_q remain to be trained. Whereas the usual optimization for fully training attempts to solve,

$$\Theta^*_{\text{ANN-}q} = \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\mathcal{D}; \Theta), \tag{21}$$

the RanNN problem uses the same cost function, but over a reduced parameters set,

$$\Theta^*_{\text{RanNN-}q} = \underset{\Theta_q}{\operatorname{argmin}}\, \mathcal{L}(\mathcal{D}; \Theta_0 \cup \Theta_1). \tag{22}$$

Notice the subtle, but important simplification from Eq. (21) to Eq. (22) in which optimization of $\mathcal{L}(\mathcal{D}; \Theta)$ is with respect to a greatly reduced set, $\Theta_q \subset \Theta$ as $|\Theta_q| \ll |\Theta|$.

Besides fewer parameters to optimize, the major simplicity of the RanNN framework is that solution of Eq. (22) which turns out to be a regularized least squares problem, by design of choosing the threshold function of the last layer to be the identity function. Specifically, solution of the ridge regression may be written in linear algebraic steps as follows. Consider Eqs. (14), (15),

$$X^{q+1} = F_{q,\Theta_q}(X^q) = W^q X^q, \tag{23}$$

and X^q follows the composition,

$$X^q = F_{q-1,\Theta_{q-1}} \circ ... \circ F_{0,\Theta_0}(X). \tag{24}$$

As reminder, see Fig. 2 and Eq. (14) which summarize the matrix notation of parameters, and also the composition notation of feed-forward through layers. Contrast this feed-forward to the last inner layer X^q, Eq. (24), to the full feed-forward $F(X,\Theta)$ in Eq. (15), since X_q explicitly omits the last forward propogation to be held out for the regression step.

Collecting all the data into columns, as data matrices, let

$$\mathbf{X} = [X_1^q | X_2^q | ... | X_N^q], \quad \mathbf{Y} = [Y_1 | Y_2 | ... | Y_N]. \tag{25}$$

These are each $d_q \times N$ and $d_{q+1} \times N$. Finally the inverse problem requires solution of the linear equation,

$$\mathbf{Y} = \beta \mathbf{X}. \tag{26}$$

We take the trained $d_q \times d_{q+1}$ matrix W^q to be the best least squares fitting of β.

In practice, to prevent overfitting, the ridge regression solution is as follows,

$$W^q := \underset{\beta}{\operatorname{argmin}} \|\mathbf{Y} - \beta \mathbf{X}\|_F + \lambda \|\beta\|_F, \tag{27}$$

from which follows,

$$\beta^* = \mathbf{Y}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda I)^{-1} := \mathbf{Y}\mathbf{X}_\lambda^\dagger. \tag{28}$$

The notation $\mathbf{X}_\lambda^\dagger$ of the regularized Penrose-pseudo inverse is described in detail in Appendix 9, Eqs. (69), (70), when formulating the "ridge" Tikhonov by regularized pseudo-inverse, for $\lambda > 0$, and $\lambda = 0$ is the ordinary least squares solution. It is worth considering the regularized pseudo-inverse for ridge regression by singular value decomposition which is the numerically stable way to interpret the inverse matrix in Eq. (28), which otherwise should never be formed directly but rather used simply as a symbol definitive of $\mathbf{X}_\lambda^\dagger$.

4 Random Shallow Learning: Span of The RanNN-1

In this section, first we present a simple 1-dimensional data domain example of how a small RanNN-1 corresponds to the span of just a few functions associated with a small RanNN-1. Then we will pursue the general result it suggests, in a multivariate domain. We will argue that a RanNN-1 essentially just develops a linear combination of piece-wise linear functions. Further, a larger hidden layer yields more basis functions. Corresponding to increasing hidden layer size, the random parameters of the RanNN-1 effectively partition the data domain with a refining grid. In the two-dimensional scenario, this story leads to a classical problem of randomly fallen lines in the plane, or randomly fallen planes or hyperplanes in three and more dimensions.

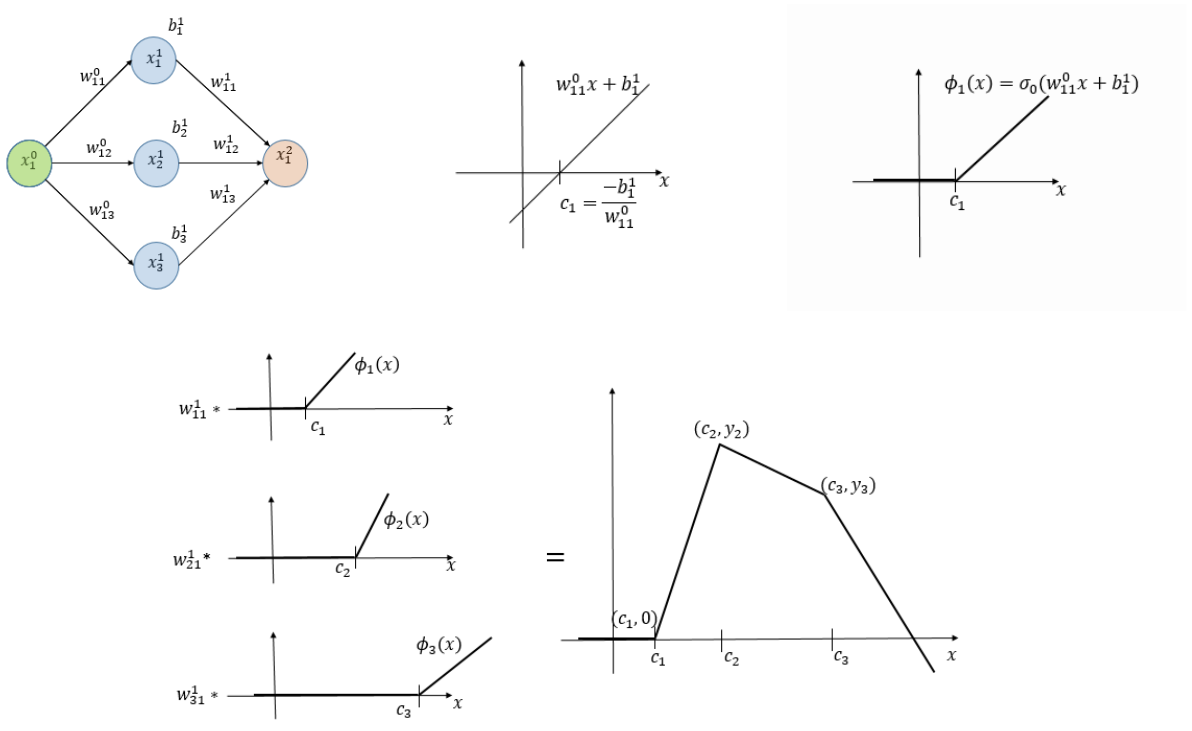


Figure 4: RanNN-1 caricature as a "small" "shallow" network. a) here, only 3 hidden nodes are illustrated for sake of a simple to draw example. With weights and hidden variable states as shown, it is easy to understand the span as linear combinations of the input basis, $\phi_1(x), \phi_2(x), \phi_3(x)$. b) The line corresponding to weights shown from input variable $x_1^{(0)}$ to hidden variable (top) $x_1^{(1)}$ associated with $x_1^{(1)} = w_{1,1}^{(0)} x_1^{(0)} + b_1^{(1)}$, which c) once the activation function $\sigma(s) = max(0, s)$ (ReLu) is formed yields $\phi_1(x) = w_{1,1}^{(0)} x + b_1^{(0)}$. Likewise for $\phi_2(x), \phi_3(x)$. d) The output weights $\{w_{1,1}^{(1)}, w_{2,1}^{(1)}, w_{3,1}^{(1)}\}$ serve to multiply the basis functions $\{\phi_1(x), \phi_2(x), \phi_3(x)\}$, which yield a piecewise linear function as shown, e). Without loss of generality, assume index labels so that intercepts order, $c_1 < c_2 < c_3$. Compare to fine covering in Fig. 6. Contrast to the "small" RanNN-2 in Fig. 17 as sequel to this Figure.

4.1 A Small Example of RanNN-1 in 1-Dimension for Motivation

Consider a scenario where there are exactly 3 nodes in the hidden layer shown in Fig. 4. Of course, normally for realistic problems, for a good fit, then a large network is better, especially if it is shallow. However, for sake of simplified illustration, we present this small example. With it, we demonstrate the interpretation that essentially 3 basis functions, $\phi_1(x), \phi_2(x), \phi_3(x)$ follows. The input layer serves to generate each of the 3 basis functions, the first of which $\phi_1(x)$ is illustrated in Fig. 4c, and similarly for the others. Only the slopes and the x-intercepts vary, as defined by the randomly chosen parameters, $\Theta_0 = \{\{w^0_{j,k}\}_{j=1,k=1}^{d_0=1,d_1=3}, \{b^1_k\}_{k=1}^{d_1=1}\}$. The main conclusion, as we will see, is that while optXlly choosing all the parameters may make for a best fit for the given network size and given dataset, even randomly selected parameters develop basis functions that could well make for a good fit, especially as the number of functions becomes large.

In Fig. 4, we illustrate a scenario where x-intercepts are ordered, $c_1 < c_2 < c_3$, which is without loss of generality since indices of the hidden nodes can be simply permuted. The x-intercept points are easily computed. The distribution from which $w^{(0)}_{1,i}$ and $b^{(1)}_i$ are selected results correspondingly in the distribution for the random variables c_i. For example, a standard derivation of functions of random variables Starnes et al. (2010), results in the "uniform ration distribution": if each parameter are drawn from uniform distributions,

$$w_{1,i} \sim U([0,1]), b^{(1)}_i \sim U([-1,0]), \tag{29}$$

then the x-intercept point,

$$c^{(0)}_i = \frac{-b^{(1)}_i}{w^{(0)}_{1,i}}, \tag{30}$$

is distributed as Marsaglia (1965),

$$c_i \sim \frac{(H(1-x) - H(x-1)/x^2}{2}, \text{ where } H(x) = \text{if}(0, x < 0, \frac{1}{2}, x = 0, 1, x > 0), \tag{31}$$

in terms of the Heaviside function, Berg (1936), as illustrated in Fig. 5. Other distributions for the random variables, $w^{(0)}_{1,i}, b^{(1)}_i$ yield similarly computable distributions for the random variable of the ratio random variables.

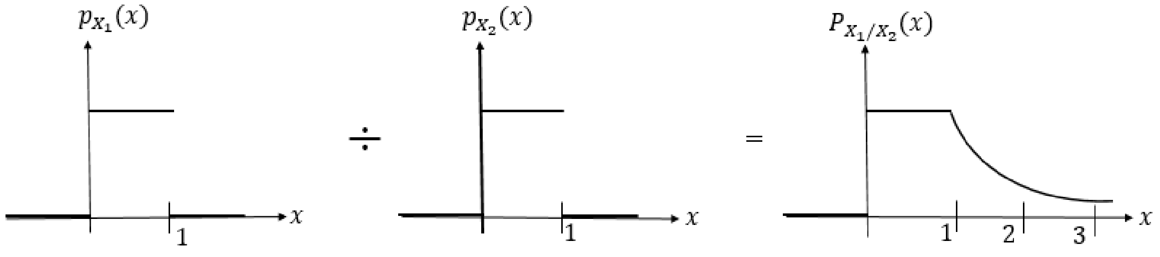


Figure 5: Ratio of uniformly distributed random variables, with distributions $p_{X_1}(x)$, and $p_{X_2}(x)$ shown, is distributed as $P_{X_1/X_2}(x) = (H(x-1)/x^2 - H(1-x))/2$, in terms of the Heaviside function, $H(x)$.

For any shallow neural network of three hidden variables, the resulting output is a linear combination of basis functions as follows,

$$\hat{f}(x) = w^{(1)}_{1,1}\phi_1(x) + w^{(1)}_{2,1}\phi_2(x) + w^{(1)}_{3,1}\phi_3(x), \tag{32}$$

where $x = x_1^{(0)}$ is the input variable, and $x_1^{(2)} = \hat{f}(x_1^{(0)})$ is the output variable; see Fig. 4. The 3 output weights, $\Theta_1 = \{w_{1,1}^{(1)}, w_{2,1}^{(1)}, w_{3,1}^{(1)}\}$, are the only parameters of this RanNN-1 that are trained. Contrast this to a fully trained SLFN of 3 hidden nodes where all 9 parameters are trained, $\Theta = \Theta_0 \cup \Theta_1 = \{w_{1,1}^{(0)}, w_{2,1}^{(0)}, w_{3,1}^{(0)}, b_1^{(1)}, b_2^{(1)}, b_3^{(1)}, w_{1,1}^{(1)}, w_{2,1}^{(1)}, w_{3,1}^{(1)}\}$. Furthermore, the later requires a more expensive nonlinear optimization process.

Considering the span of these functions,

$$\hat{f} \in \Delta_{d_1} = span(\{\phi_1(x), \phi_2(x), \phi_3(x)\}), \tag{33}$$

it is clear (see Fig. 4e) by Eq. (32) that $\hat{f}$ can exactly represent any data of the form, $y_1 = \hat{f}(x_1) = 0, y_2 = \hat{f}(x_2), y_3 = \hat{f}(x_3)$, if the x-intercepts happen to coincide with the data $c_1 = x_1, c_2 = x_2, c_3 = x_3$, and by the same prior assumption of linear ordering, $x_1 < x_2 < x_2$. Under these assumptions, a proof simply involves induction from left to right in the domain as suggested in Fig. 6, and generalizes to many points and many intercepts. The c_j are the randomly placed intercepts that follow from the random network parameters, by Eq. 30, and their positions relative to the data.

The example discussion of this section broadly suggests a construction that is expanded upon in Appendix 8 that tracks closely to the idea of neural networks as universal approximators. There do exist such theorems for fully trained SLFN but even for RanNN-1 Huang et al. (2006a). However, rather than worry if the neural network can approximate general functions, in Appendix 8 and Fig. 6 we describe in simple and more geometric terms how large RanNN-1 define a linear space of functions Δ_{d_1} that includes functions $\hat{f}$ that are close to a piecewise linear functions f that run exactly through the data. Comparably to the classical Stone-Weierstrass theorem Rudin et al. (1976); De Branges (1959); Burkill (1959) that asserts how a high enough degree polynomial can approximate a continuous function arbitrarily closely, so do piecewise linear functions Shekhtman (1982). But as for the danger of overfitting, where a high degree Lagrange polynomial can be designed that runs exactly through a finite data set, jef, so can a piecewise linear function as represented by a SLFN and closely by an RanNN-1. Therefore while increasing the size of the hidden layer relates to refining the representation by a piecewise linear function, this does leave open the significant problem issue of overfitting.

4.2 Example of a Large RanNN-1 with Regularization, in 1-Dimension

From the discussion of the previous subsection, and Appendix 8, we know that a large enough SLFN can exactly match given data in 1-dimension, and closely so even for a random one, an RanNN-1. However, the danger of over-fitting with an overly large random project network is extreme, fitting the noise rather than the process. So it is advisable not to try to fit the data exactly, but rather to do so approximately. While there are several mitigation strategies to prevent over fitting, here regularization is considered good practice. Tikhonov regularization with L2-data fidelity (as stated in Eq. (27)) as well as L1-regularization are shown in Fig. 7. We see shown an overfitted, RanNN-1 curve, and also "better" regularized versions of the same.

As seen in Fig. 8, increasing data sample size leads to reduced variance, and regularized models also improves fit, to a point but introduces bias. This is just a typical version of the classical issue in machine learning which is to balance bias versus variance, François-Lavet et al. (2019); Kohavi et al. (1996); Neal (2019). Fitting functions as observed through noisy measurements of data should be regularized, perhaps by choosing ridge regression (L2 regularization) for reduced variance, or alternatively by Lasso regression (L1 regularization) to emphasize sparsity. Each of

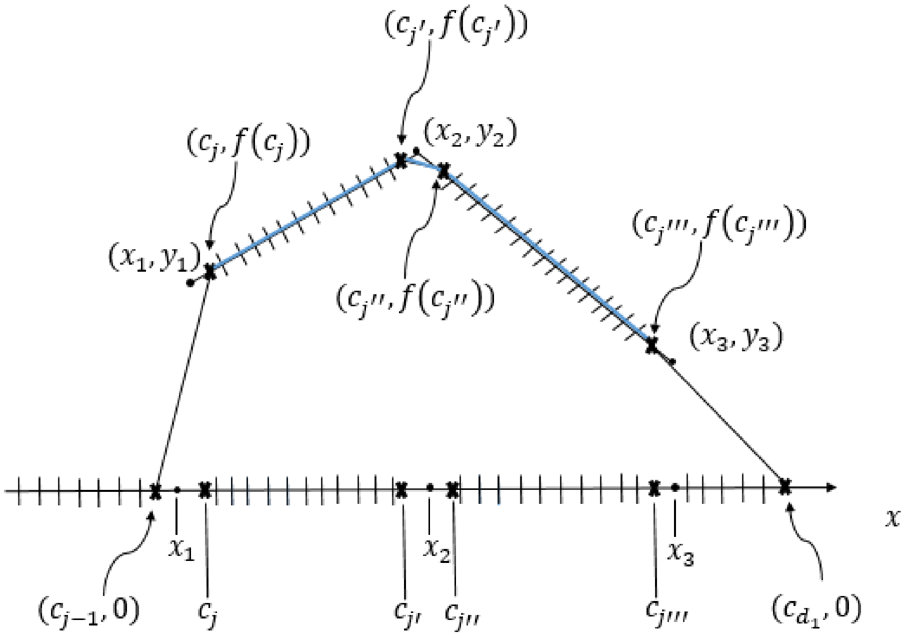


Figure 6: An RanNN-1 shown in one-dimension closely tracks to given data. Whereas universal approximation theorems describe how data generated by a function may be well estimated by a neural network, including there exists such theorems for RanNN, we describe mechanism and more weakly how there is a linear spline $f \in \Delta_{d_1}$ that is the span of ramp functions through a fine covering of (random) x-intercepts c_j that agree with a piecewise linear function f running through the data x_i on all but a small set, and deviates only slightly on that set.

these have their advantages depending on our prior assumptions regarding the problem. See Fig. 8 for a comparative example of these, for noisy data in 1-dimension. In Fig. 9, we show some of the basis functions, $\phi_i(x)$ which are included in the linear space Δ_{d_1}, $d_1 = 500$ from example in Fig. 7. In the next section, we consider RanNN-1 in more than 1-dimension.

4.3 Example of a Large RanNN-1 in 2-Dimensions

Now we transition to the utility of RanNN in multivariate examples. The ideas in Sec. 4.1 and Appendix 8 for 1-dimensional data generalize in a geometrically sensible manner, but with interesting new aspects in higher dimensions. Classical problems concerning partitions by random lines become relevant. We will explore these mathematical details in the next section.

First, in this section, we demonstrate how RanNN-1 works quite well for a simple 2-dimensional example. Random data is sampled uniformly from a function,

$$y = f(x_1, x_2) = 5 - x_1^2 - x_2^2. \tag{34}$$

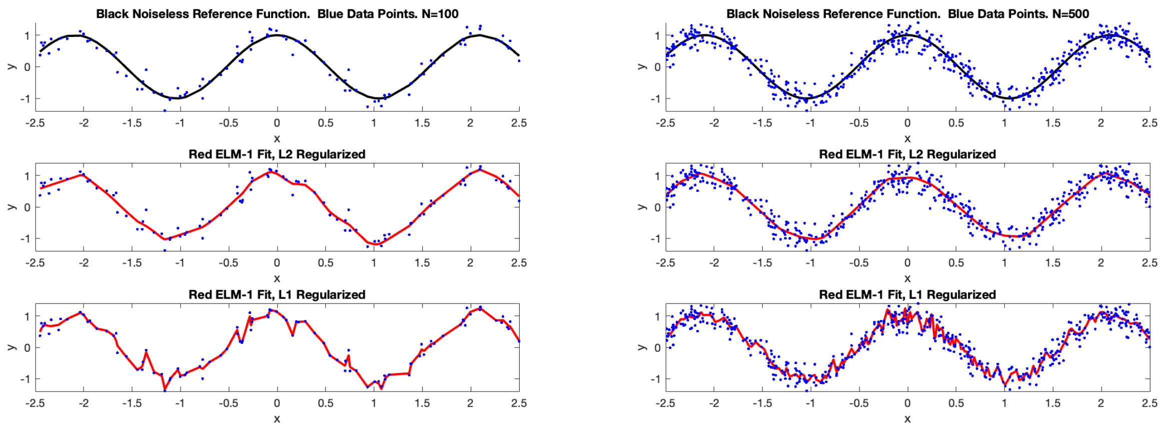


Figure 7: An RanNN-1 fitting of noisy data from an underlying 1-dimensional function, shown in black. (First row) Number of data points are $N = 100$ and $N = 500$, first and second column, respectively. (Second Row) Ridge regression, with L2 regularity parameter $\lambda = 1.0 \times 10^{-4}$. (Third Row) Lasso regression, with L1 regularity parameter $\lambda = 1.0 \times 10^{-4}$. See some of the fitting basis functions in Fig. 8.

Specifically, data $\mathcal{D}$ consists of N random samples of $x_i = (x_{1,i}, x_{2,i})$ and y_i according to,

$$x_{1,i}, x_{2,i} \sim \mathcal{U}([-2.5, 2.5]), y_i = f(x_{1,i}, x_{2,i}) + \xi, \text{ and } \xi \sim \mathcal{N}(0, \sigma), \tag{35}$$

for a normal noise amplitude $\sigma = 1.0 \times 10^{-4}$. Again we use ridge regression Eqs. (26)-(28) of the typical regularized loss function Eq. (12) fitting of just the output layer parameters, for feed-forward Eq. (15) of the SLFN that is typical of RanNN-1, with all but output parameters chosen randomly.

In Fig. 10 we illustrate some RanNN-1 model concepts,

- A piecewise-linear model, of increasing precision as d_1 increases.
- Boundaries of these linear segments are randomly fallen line segments.

Theory for both of these concepts is further developed in the next section. In Fig. 11, we show the success of increasing precision as d_1 increases.

4.4 Geometry of RanNN-1 in 2-Dimensions and Beyond Relates to Understanding the Classical Problem of Randomly Fallen Lines, Planes and Hyperplanes

There is a beautiful and fundamental relationship at the heart of the success of a random neural network to a very classical theory of randomly fallen lines, planes and hyperplanes. In Secs. 4.1-4.2, notably in Fig. 10, we observed that the domains of linear fit of the RanNN are bounded by randomly placed lines. We will now develop this idea as related to classical theory underlying this phenomenon.

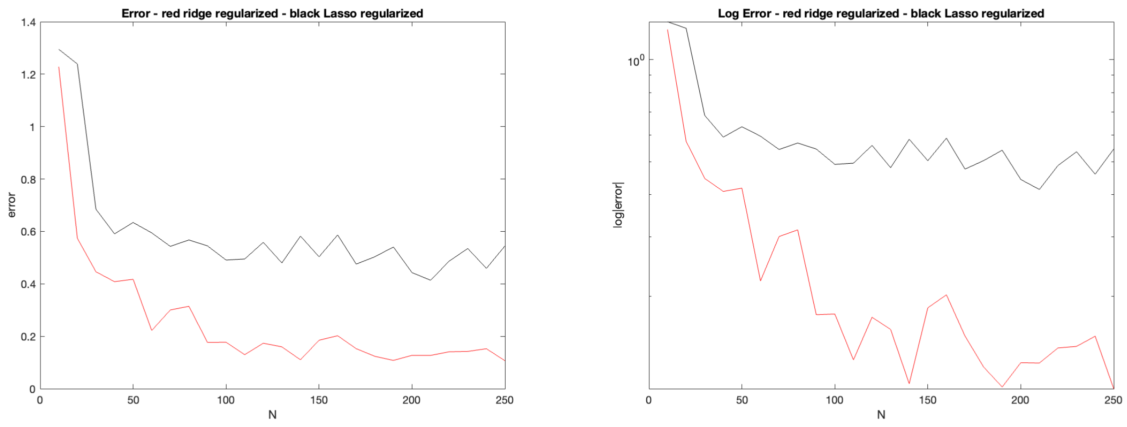


Figure 8: Error of an RanNN-1 regularized fitted to synthetic data Fig. 7. (Left) Sum of square errors shown with $\lambda = 0$, and increasing hidden layer size, $d_1 \uparrow$. (Right) Log of sum of square error. Increase hidden layer size size d_1 leads to reduced variance at least until saturation at roughly $d_1 = 140$.

.

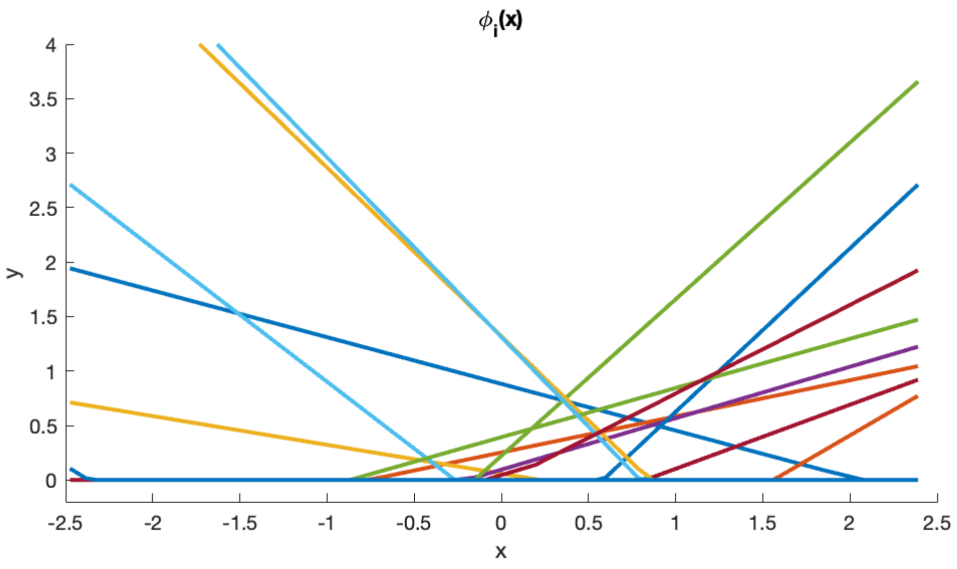


Figure 9: A dozen of the $d_1 = 500$ basis functions for the RanNN-1 used in Fig. 7 right column.

Reconsider an RanNN-1, but now for data domain in two or more dimensions, $d_0 > 1$. Specifically, we consider some geometric issues related to the ReLu thresholding that occur in an individual "neuron" in the central layer of an RanNN-1. While we choose to focus on ReLu mainly for the simplicity relative to our descriptions, and also its popularity, other thresholding functions have comparable geometric issues stemming how level sets intersect. Just as with the single variate

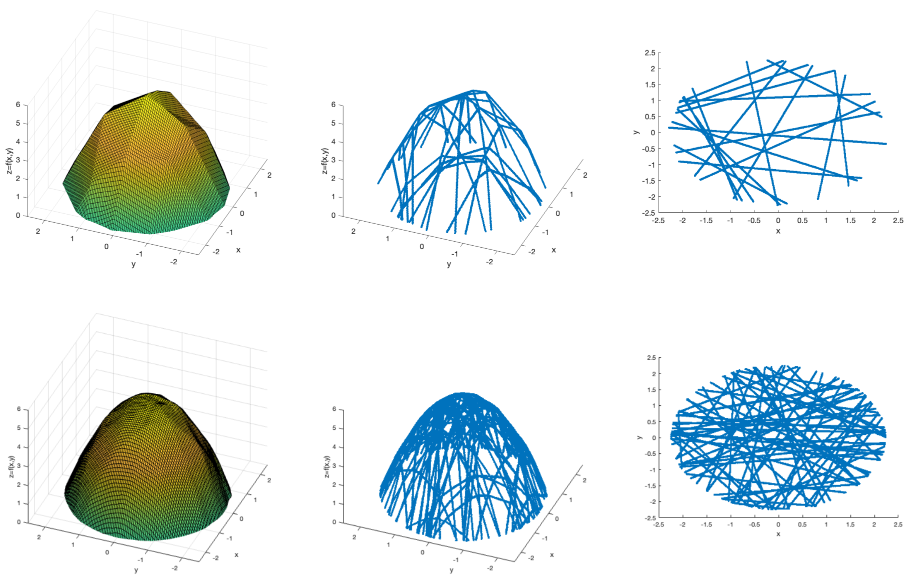


Figure 10: RanNN-1 model of Sec. 4.3 randomly sampled data according to Eq. (35), near an ellipsoid plus noise. (Top Row) $N = 1,000$ samples and a $d_1 = 30$ width randomly configured hidden layer of the SLFN of the RanNN-1. Correspondingly, since a ReLu activation function is used, $d_1 = 30$ basis functions of the form, $\phi_k(x)$ associate with Eq. (37) are used in the neural network last layer fitting. Since each $\phi_k(x)$ is continuous, the linear combination is continuous. We see the property of continuity in the piecewise linear fitting to the paraboloid that generated the data. (Second Row) $N = 10,000$ samples, and a larger $d_1 = 100$ RanNN-1, and so a finer fit of basis functions, $\phi_k(x)$. (First Column) Surface plot of the fitted RanNN-1 model reveals that as suggested by theory, a continuous piecewise linear function results that theory suggests converges to the true function $y = f(x_1, x_2)$ in the limit $N \to \infty$, and $d_1 \to \infty$. (Middle Column) The discontinuities (of the derivative of the function, that is the edges) between the planes of the piecewise linear model shown left, are the line segments shown that bound each linear-plane segment, as promised by the theory we develop here, Sec. 4.4. (Right Column) The line segments of the SLFN from an RanNN-1 projected into the (x_1, x_2) domain, actually correspond to randomly placed (fallen) lines, described in the theory sections "Arrangement of hyperplanes", Sec. 4.4.

scenario of a SLFN, discussed in Secs. 4.1-4.2, each neuron in the central layer serves as a basis function $\phi_i(\mathbf{x})$. Then regression fits the best weights to the output layer as linear combination of these basis functions in the corresponding subspace described by their span.

Consider the i^{th} neuron in the hidden layer, but before the threshold is applied,

$$z = W^0_{i,:} \cdot \mathbf{X}^{(0)} + b^{(1)}_i, \quad i = 1, 2, ..., d_1. \tag{36}$$

$W^0_{i,:}$ is the i^{th} row of matrix W^0 consisting of randomly assigned weights from the input layer to the hidden layer. Specialize Eq. (13) for $r = 0$ as the 0^{th} layer, and $X^{(0)} = [x^{(0)}_1; x^{(0)}_2; ...; x^{(0)}_{d_0}]$ is

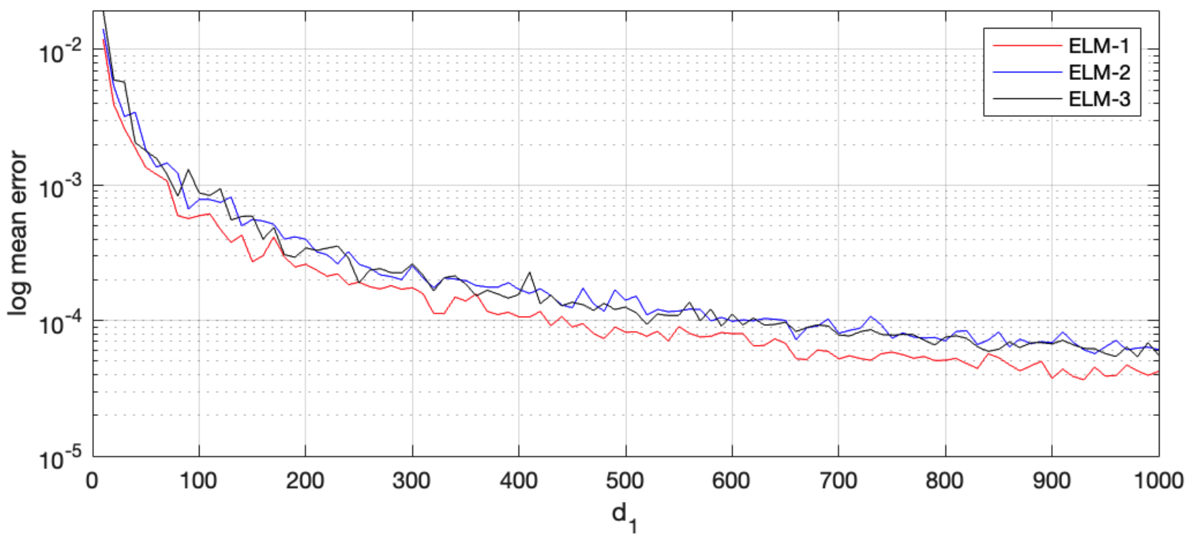


Figure 11: Error of RanNN-q, $q = 1, 2, 3$ of Sec. 4.3 on 2-dimensional domain example, shown in Fig. 10, by Eq. (35). Log total relative error. As the number of elements d_1 in the hidden layer of the SLFN of the RanNN-1 increases, we see fitting error decreases. However, we observe no benefit for using random deep feedforward network, RanNN-2 or RanNN-3.

the vector of states, defined Eq. (3). It is standard geometry that the set of points associated with Eq. (36) for any fixed value of z is a co-dimension-one linear surface, also called a hyperplane (or a line if $d_0 = 2$, or a plane if $d_0 = 3$). The set of points associated with the inequality,

$$z \geq W^0_{i,:} \cdot \mathbf{X}^{(0)} + b^{(1)}_i. \tag{37}$$

is a the half space,

$$\mathcal{S}_i = \{\mathbf{X}^{(0)} : ReLu(W^0_{i,:} \cdot \mathbf{X}^{(0)} + b^{(1)}_i) = W^0_{i,:} \cdot \mathbf{X}^{(0)} + b^{(1)}_i\}, \tag{38}$$

and the compliment is the level set,

$$\overline{\mathcal{S}_i} = \{\mathbf{X}^{(0)} : ReLu(W^0_{i,:} \cdot \mathbf{X}^{(0)} + b^{(1)}_i) = 0\}, \tag{39}$$

which is also a half space.

Analogous to Fig. 4, now consider Figs. 12. In the multivariate setting, again the output layer variables are a linear combinations basis functions, each of which we now write as,

$$\phi_i(\mathbf{X}^{(0)}) = ReLu(W^0_{i,:} \cdot \mathbf{X}^{(0)} + b^{(1)}_i). \tag{40}$$

Furthermore, each nonlinear function ϕ_i is simply linear in a subdomain that is bounded by a random configurations of lines. So it is interesting to consider arrangements of random lines, which is a classical problem of geometry.

4.5 Arrangements of Random Lines, Planes and Hyperplanes

A key feature suggested by the example shown in Fig. 10 describes arrangements of randomly fallen lines, or likewise of planes or hyperplanes in higher dimensions, Eq. (36). Since distributions of random lines has long been a classical problem, Hall (1981), in this context we now consider the number and size of resulting cells. If this equation is in a 2-dimensional domain space for the input x_i data, we have arrangements of lines in the plane, which is the case we will emphasize in this section. This classical geometric problem has strong bearing on estimation of a ReLU feed-forward ANN, especially RanNN. Specifically, we are interested in the number and size of cells, since in each one, the neural network reduces exactly to a linear function. jjjhi Since each hidden variable yields an inequality Eq. (37), and therefore a randomly placed linear cut of the plane, the associated interiors define linear regions of an otherwise complex continuous piecewise linear function. The geometry of finitely many random lines in the plane informs how the RanNN-1 samples a function. The placement of m-lines in the plane specifies the number of piecewise linear regions of the ReLu RanNN-1. An obvious upper bound on the number of such regions is $M_n < 2^m$ since each line cuts the space in half. This turns out to be significantly too large as an over-estimate. The low-dimensional topology of cuts in the plane leads to the issue of occlusion. A sharper estimate that considers occlusion of regions leads to a more nuanced idea of what to expect of the neural network.

Expanding on the idea that one line separates the plane into two regions, consider a symbolic notation,

$$s(x) = s_1(x), s_2(x), ..., s_m(x), \tag{41}$$

to denote how a point lies relative to an oriented plane:

$$s_i(x) = 1 \text{ if } x \in \mathcal{S}_i, \text{ and, } 0 \text{ else}, \tag{42}$$

with half plane set $\mathcal{S}_i$ defined in Eq. (38). Each region has a unique binary pattern regarding m-lines, however not every m-bit symbolic label is realized as a configuration. For example, in Fig. 12(a), x is the red point shown in the bottom panel illustrating the binary symbolized planar partition. Clearly $s_1(x) = 1$ since it lies relative to the line L1 labelled "1". Likewise, the same point x lies $s_2(x) = 1$, and $s_3(x) = 0$. In summary, $s(x) = 1, 1, 0$ by Eqs. (41)-(42). Likewise we get the same label for all the points in that partition element shown in the far right of Fig. 12. With each overlapping half plane, we gain labeled regions as shown. Counting regions in the upper half of Fig. 12a immediately follows that $M_1 = 2, M_2 = 4$, but $M_3 = 7$. The $0, 1, 0$ configuration is missing, due to occlusion.

The general statement of counting enclosed open regions due to n lines is Hall (1981),

$$M_n \leq \frac{n(n+1)}{2} + 1 = \# \text{ of cells in an arrangement of } n\text{-lines in 2-dimensional space.} \tag{43}$$

This inequality is as an equality if there is "a general configuration," which occurs when no lines are parallel and no intersection points are triple point or more intersections. Thus, not only $M_n < 2^n$, but exponentially so, $\frac{n(n+1)}{2} + 1 \ll 2^n$. This result follows Graham et al. (1994); Richards (1964) as solution of a recursion formula, as an initial value problem,

$$M_{m+1} \leq M_m + (m+1), M_1 = 2. \tag{44}$$

Again, this is an equality if the lines fall in general configuration. The proof of this fact follows by induction. Direct inspection yields the case, $M_1 = 2$. If we assume the result to be true for M_n,

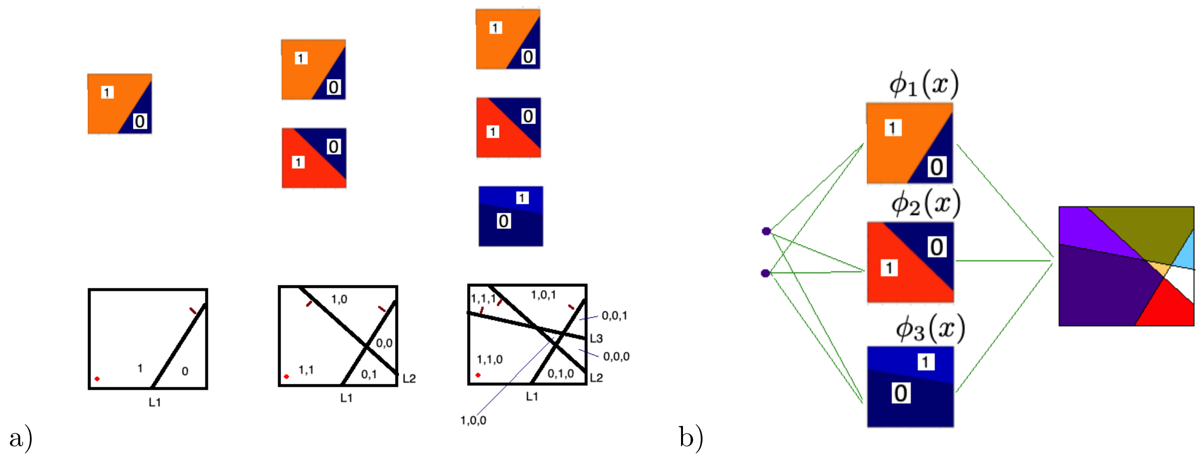


Figure 12: On random geometry of partitions by m-fallen (random) lines. (a) Column 1 shows one such oriented plane where $\mathcal{S}_1$ defined in Eq. (38) is shown colored as orange. Below the same is shown in labelled regions 0 or 1, and a red dot is shown for a reference point x. In the second column of a), with another oriented plane added, overlapping combinations result in all 4 possible combinations of 2 states. Symbols, $s(x) = 0, 0$, $0, 1$, $1, 0$, or $1, 1$ label these regions as shown. However, in the third column of a), with the introduction of a third oriented line, labels $s(x) = s_1(x)s_2(x)s_3(x)$ reveal the $M_3 = 7$ resulting labelled sets of the partition, by Eq. (44). Notice the symbol $0, 1, 1$ is missing, due to occlusion. In general the number of label sets is well below the maximal $M_n < 2^m$ which otherwise counts occluded configurations. b) Shown as an SLFN a two-dimensional input space results in $M_3 = 7$ distinct cells that are linear domains, by linear combination of the 3 piecewise linear functions $\phi_i(x)$.

then the formula for M_{n+1} follows a geometric assertion: a new line can in general intersect m lines at up to m distinct points. Since we wish the maximal number of region interiors, which occurs as the general configuration of the lines. In other words, the randomly placed line generally intersects $m + 1$ regions, so the new line divides the M_n regions, thus adding $m + 1$ subregions.

The geometric scenario of more than two input variables is comparable; if $d_0 > 2$, so beyond the two input variables as drawn Fig. 12, nonetheless each Eq. (36) associated with the hidden layer serves as a co-dimension-1 plane (if $d_0 = 3$), or hyperplane (if $d_0 > 3$) that nonetheless also splits the space, eventually into many cells in the d_0 embedding space.

Define $M(d_0, n)$ to be the maximum number of regions formed by n hyper-planes in d_0 dimensions, generalizing slightly the notation of Eqs. (43)-(44); thus $M(2, n) = M_n$ is the number of regions due to n lines written in two different notations, as both ways available for the 2-dimensional case. It can be shown that for $d_0 \geq 2$ Stanley et al. (2004),

$$M(d_0, n) \leq \sum_{i=0}^{d_0} \binom{n}{i} = \#\text{ of cells in an arrangement of } n\text{-planes (hyperplanes) in } d_0\text{-dimensional space.} \tag{45}$$

Similarly, this follows from a recursion relationship. This time,

$$M(d_0, n) \leq M(d_0, n-1) + M(d_0, n-1). \tag{46}$$

Equality occurs in both Eqs. (45)-(46) in the general configuration, meaning the zero-probability placement of any pairs of hyperplanes and their corresponding intersections is excluded. Finally, it is interesting to note the number of bounded regions, meaning those that are closed from all sides, which is given by Stanley et al. (2004); Trotter (1995),

$$B(d_0, n) \leq \left(\begin{array}{c} n-1 \\ d_0 \end{array} \right) = \#\text{ of bounded cells in an arrangement of } n\text{-hyperplanes in } d_0\text{-dimensional space}, \tag{47}$$

with equality in general configuration. Comparing Eq. (45) to Eq. (47), we see that,

$$B(d_0, n) < M(d_0, n). \tag{48}$$

In practice, a given fine data set must reside in a bounded domain. The result then is that in a bounded domain, we conclude,

- As the size of the hidden layer increases, $d_1 \uparrow$, then the number of linear domains increases. See example in Figs. 11, 10 and 13.
- As $d_1 \uparrow$, the size of these domains decreases. This is a random refinement. See Figs. 11, 13.
- As $d_1 \uparrow$, the number of basis elements in Δ_{d_1} increases, and the fit accuracy improves. See example Fig. 11.

All of this pertains to the expressivity of a wide shallow network, even with random weights. We have examples of fitting functions in the previous Subsection 4.2, and Figs. 7, 13, and flows in the next Sec. 5 and Figs. 14, 15, 16. Then in Sec. 6 we will consider why deep architectures are useful, for fully trained ANN, but less so for the partial training, mostly random scenario of RanNN-q, $q > 1$.

It is easy to see that more, and so smaller, domains of each of the linear elements of a piecewise linear fit can lead to improved precision when fitting a continuous function f. Now, if we assume a uniformly Lipschitz function f with Lipschitz constant $L > 0$ in a bounded domain D, then with an increasing number of cells, we can bound an error of fit between the true function $f(x)$ and the RanNN model function $\hat{f}(x)$. In any one of the subdomains, we call the piecewise linear function of fit, $f(x)$ as $\ell(x)$ to emphasize that it is linear (affine) in such a domain that contains x. Then let x_1 and x_2 be two points of maximal distance (diameter) on the boundary of such a cell that contains x. Thus,

$$\begin{aligned} |f(x) - \ell(x)| &\leq |f(x) - f(x_1)| + |f(x_1) - f(x_2)| + |\ell(x) - \ell(x_1)| + |\ell(x_2) - \ell(x_1)| + |f(x_2) - \ell(x_2)| \\ &\leq 2L|x - x_1| + 2L|x_2 - x_1| \leq 4L|x_2 - x_1|. \end{aligned} \tag{49}$$

In this sense, the quality of fit is bounded proportionally to the largest distance between two points in any given cell, or maximized over all cells, $r = max|x_2 - x_1$,

$$|f(x) - \ell(x)| \leq 4Lr. \tag{50}$$

And finally with a an absolutely continuous density from which the weights of the neural network are chosen, by discussions above, points "fill-in" leading to refinement in the sense that r decreases with increasing network size.

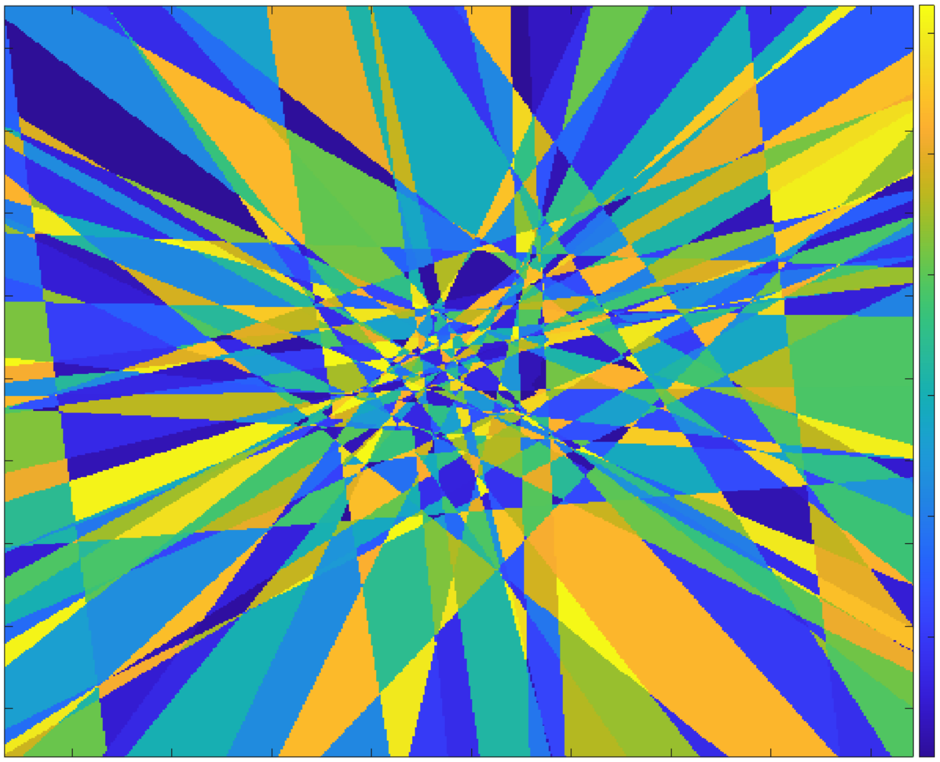


Figure 13: An RanNN of $d_1 = 250$ elements in the hidden layer, and input $d_0 = 2$, implying the data in the plane results in distinct linear regions as shown in colors, here, $B \cap M_n = 1248$ such regions in a domain $(x, y) \in B = [-5, 5] \times [-5, 5]$. Compare to Eq. (43) where the larger $M_{250} = \frac{250(250+1)}{2} + 1 = 31376$ is due to the chosen bounded box domain B. Nonetheless, as d_1 the suggestion is that there is a refinement. See Fig. 11 .

5 RanNN-1 Data-Driven Forecasting of Dynamical Systems

For our purposes of data-driven learning, a dynamical system, specifically a discrete time map autonomous,

$$z_{n+1} = F(z_n), \text{ for a given initial condition in its phase space } z_0 \in \mathcal{M} \subset \mathbb{R}^{d_0}, \tag{51}$$

is a matter of inferring the function F from many examples as data pairs (z_{k+1}, z_k) that may or may not come from a continguous sample of an individual (forward) orbit, $orbit(z_0) = \{z_0, F(z_0), F \circ F(z_0), F \circ F \circ F(z_0), ...\} = \{z_0, F(z_0), F^2(z_0), ...\} = \{z_0, z_1, z_2, ...\}$, noting standard notations for the same thing. Also while a mathematical definition of an orbit typically has samples infinitely into the future, a sampled data set is finite. However, whether from a single long trajectory sample, or several individual trajectories from several distinct initial conditions, or just many single time-step input-output pairs, we require that these must be collected, as a single total data set,

$$\mathcal{D} = \{(z_k, z_{k+1}\}_{k=1}^{N}. \tag{52}$$

Comparing this to the original and general way we stated data, Eq. (1), note that we simply have a special case of the supervised function learning problem problem, with $X_i = z_k$ and $Y_i = z_{k+1}$. Also, the dimensionality of input and output should match, $d_0 = d_2$, for the SLFN structure of the RanNN-1 learning of a dynamical system.

If instead a dynamical system is a flow (or a semiflow), Bollt and Santitissadeekorn (2013), thus continuous time, such as a solution from a differential equation,

$$\dot{z} = f(z), \text{ for a given initial condition in its phase space } z(t_0) = z_0 \in \mathcal{M} \subset \mathbb{R}^{d_0}, \tag{53}$$

then nonetheless the flow (semiflow) is simply a function,

$$\begin{aligned} \varphi : \mathbf{R}\mathcal{M}\& \rightarrow \quad & \mathcal{M} \\ (t, z_0) \quad & \mapsto \quad z(t) = \varphi(t; z_0). \end{aligned} \tag{54}$$

For some differential equations, this might be derived in closed form. For many, a solution is by numerical integration, or for others, this represents an experimental system where data comes directly from experiment. The data may be formally written,

$$z_k = z(t_k) = \varphi(t_k; z_0), \text{ with } t_{k+1} - t_k = \delta t \text{ evenly spaced by } \delta t \text{ for all samples}, \tag{55}$$

if a single orbit is sampled, but similarly multiple orbits can be useful. So, the flow map then relates to a discrete time map when evenly spaced in time, by

$$z_{k+1} = F(z_k) = \varphi(\delta t; z_k). \tag{56}$$

This exploits the standard group property associated with the definition of a flow, $\varphi(t_{k+1}; z_0) = \varphi(\delta t; \varphi(t_k; z_0))$, and assumed even time spacing. These samples of (z_k, z_{k+1}) pairs can come from a single orbit or snippets of many orbits, sampling the full phase space, or near the attractor. Thus we can form a data set $\mathcal{D}$ as Eq. (52) representing the dynamical system as pairs, (z_k, z_{k+1}) appropriately sampled.

Therefore, Eq. (26) specializes to the notation that,

$$\mathbf{Y} = [z_{k_1+1}|z_{k_2+1}|...|z_{k_N+1}], \tag{57}$$

and as before

$$\mathbf{X} = [X_1^1|X_2^1|...|X_N^1], \tag{58}$$

is the matrix whose columns are the N observations of the d_1-dimensional hidden states of the single hidden layer. Again, the hidden state is from the random feedforward, but now of each $z_{k_i}, i = 1, ..., N$, Eqs. (13),(15),

$$X_i^1 = F_{0,\Theta_0}(X_i^0), \text{ and, } X_i^0 = z_{k_i}, \quad i = 1, ..., N, \tag{59}$$

is solved, Eqs. (26),(28).

5.1 Forcasting Experiment: Lorenz Equations

The Lorenz-63 equations that have become somewhat of a standard in chaos theory presentations Lorenz (1963) and also they have become standard for demonstrating success of data-driven methods for chaotic dynamical systems, Gauthier et al. (2021); Bollt (2021); Vlachas et al. (2020). As such, consider,

$$\dot{x} = \sigma(y - x), \dot{y} = x(\rho - z) - y, \dot{z} = xy - \beta z, \tag{60}$$

and standard parameters,

$$\rho = 28, \sigma = 10, \beta = \frac{8}{3}. \tag{61}$$

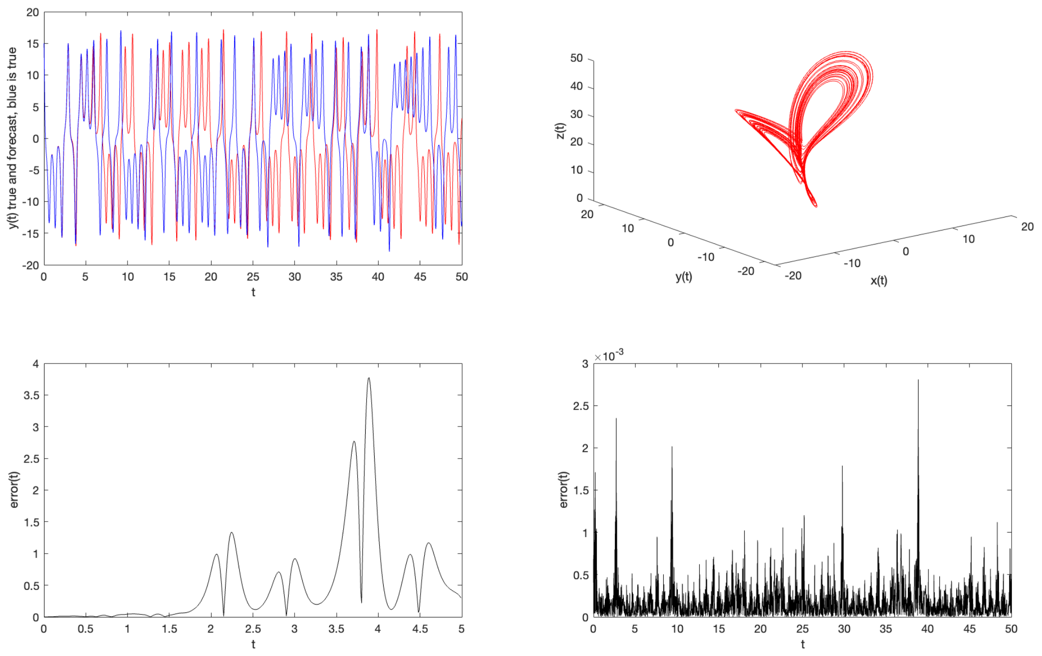


Figure 14: An RanNN-1 forecasting of Lorenz data with hidden layer sized $d_1 = 250$. Time step of data is $\delta t = 0.01$. Standard Lorenz Eq. (60) and parameters, Eq. (61). Time range $[0, 100]$ used with a randomly chosen initial condition. (Upper Left) After training, a true sample in blue, t versus x(t) time series, versus an RanNN-1 forecast from the same initial condition in red. (Upper Right) Forecast trajectory shown in phase space, $x(t), y(t), z(t)$. (Lower Left) Error of true versus forecast from upper left, shown in time range t$\in$ $[0, 5]$. (Lower Right) Error between trained RanNN-1 versus training data, Eqs. (26),(28) in training phase.

In Figs. 14, 15, we show results of an RanNN-1 used to forecast sample orbits from Lorenz-63, with data produced by high precision numerical integration with Runge-Kutta RK45 adaptive step to meet relative and absolute tolerances of 1.0×10^{-12}, and flow trajectory data $x(t), y(t), z(t)$ over

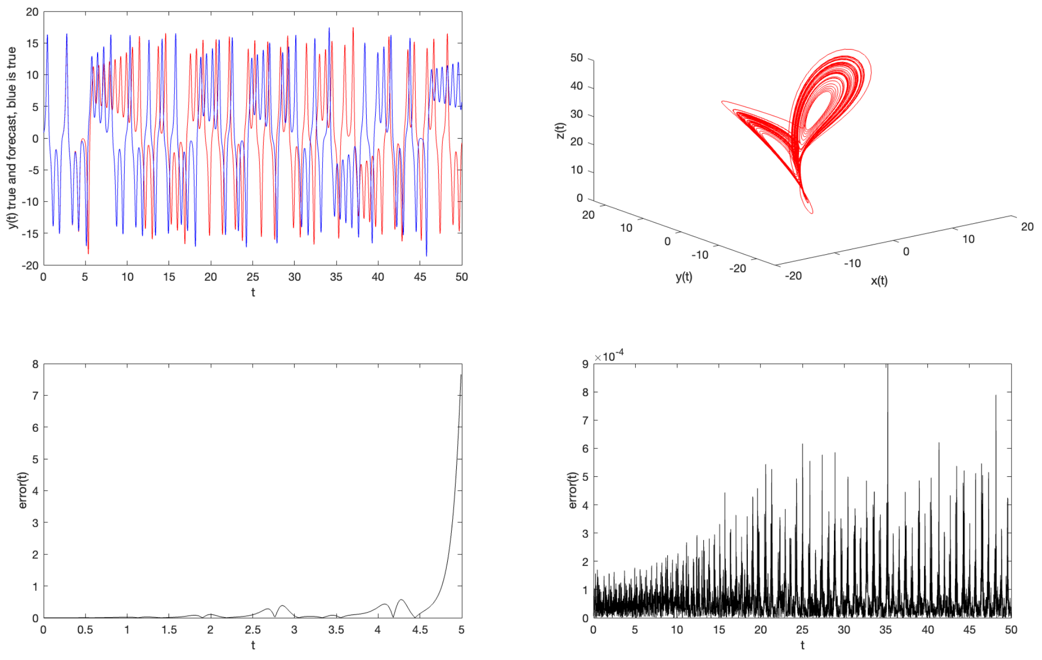


Figure 15: RanNN-1 forecasting of Lorenz data with hidden layer sized $d_1 = 1000$. Time step of data is $\delta t = 0.01$. Otherwise, layout and description is the same as in Fig. 14.

a time range, $t \in [0, 50]$ and time steps, $\delta t = 0.01$ for a total of $N = 5001$ training samples in the data set. The Figs. 14, 15 show $d_1 = 250$ and $d_1 = 1000$ respectively and progressively better accuracy in each.

The better accuracy is most clear in contrasting the forecast error quality horizon (Lower Left) which is roughly $t = 2$ and $t = 4.5$ when the error grows beyond small values, where-after the natural sensitive dependence to initial conditions clearly reveals itself with exponential growth of error; this is in *testing phase*. Also in the lower right images of the figures is the *training phase*, where it is clear that training error is significantly smaller with the larger d_1; notice the scale. By error in training phase, we mean the loss function, Eq. (12) when trained using the RanNN-1 chosen parameters meaning just at output layer, so Eq. (22).

It is a much lower standard of the ability of any machine learning algorithm, RanNN included, to simply describe how well the representation of the model can match the training data, as evidenced by both lower right figures where the loss function is extremely small. A better test of utility is to ask how the model performs out of sample data. To that end, by testing phase, we mean an initial condition $x(t_0), y(t_0), z(t_0)$ is input into the model, the RanNN-1, and the output is an estimated $x(t_0 + \delta t), y(t_0 + \delta t), z(t_0 + \delta t)$, which is then re-inserted into the model to produce an estimated $x(t_0 + 2\delta t), y(t_0 + 2\delta t), z(t_0 + 2\delta t)$, and so on, repeatedly. It is too much to expect that the model orbit might reproduce the true orbit, since after all, these are often chaotic dynamical systems, which display the classical property of sensitive dependence to initial conditions. Therefore, if with the true system, if there were the slightest difference, the error would grow quickly, exponentially by the Lyapunov exponent rate. However, what is a remarkable property of a "good model" is when it makes "good errors." By good errors, we mean the phrase "it reproduces the climate" as stated in Griffith et al. (2019) or said otherwise, it is statistically correct in the sense that we see almost the same attractor. This is exactly what we see in both Figs. 14, 15 upper right, where in testing phase, the RanNN-1 model of the flow produces an attractor that looks very much like the true Lorenz butterfly. Minor differences are too small to notice with the naked eye. Likewise, in both figures, we see the true flow (blue is true), and RanNN-1 model flow in testing phase (red) curve lie essentially on top of each other for times, approximately $t < 2.5$ and $t < 5$, respectively. Then after that the forecasts and true orbits deviate significantly, but at all times, the forecast looks very much like a true Lorenz orbit, but perhaps of a different initial condition. Thus the appearance of the butterfly attractor in the upper right of Figs. summarizes that idea.

We emphasize that these good results are by the random neural network framework, RanNN-1, which might be considered a worst case scenario versus the more careful, even if more expensive, standard fully trained FFNN. However, as we see, there is still enough expressivity in an RanNN-1 and with increasing hidden layer size, to make up for the randomness of placement, which the figures such as Fig. 13 suggests, also allow for a general refinement partitioning of the domain.

5.2 Forcasting Experiment: Mackey-Glass Differential Delay Equations

The Mackey–Glass differential delay equations Mackey and Glass (1977),

$$\dot{x} = \frac{ax(t-\tau)}{1+x(t-\tau)^d} - bx(t) \tag{62}$$

a standard example in time-series analysis Farmer (1982); Lichtenberg and Lieberman (2013) of a high (infinite) dimensional dynamical system with a low-dimensional attractor. This is a differential delay equation due to the explicit presence of a delay term τ in the equation. It was developed as a model of physiological control systems, and it is known to display a wide array of complex oscillatios and chaotic behaviors. We have chosen parameters,

$$a = 0.2, b = 0.1, d = 10, \tau = 17. \tag{63}$$

This system gives an embedding dimension of 4, but generally has an interesting property that the attractor dimensionality depends on τ and d. A projection of the attractor in delay coordinates with delay, $\tau = 6$, is shown in Fig. 16(Upper Right) showing the forecasted attractor. We see in Fig. 16(Upper Left) a segment of the true flow (blue) and forecast flow (red) which track closely at first and then diverge which again is no surprise for a chaotic system that therefore has sensitive dependence. What we see here again is the property that even once errors occur, they are plausible dynamics. This concept is described as reproducing the climate, or said mathematically, despite errors, the statistics of the attractor appear correctly. For this system, in a high dimensional space, note that we did use a very large RanNN-1, of $d_1 = 5000$ hidden random nodes, and again we make no claim of efficiency. Rather, our purpose is demonstration of the mechanisms and concepts of the RanNN-1 as a random SLFN. Compared to other random architectures, notably standard reservoir computing Vlachas et al. (2020); Lukoševičius and Jaeger (2009); Pathak et al. (2018); Griffith et al. (2019); Butcher et al. (2013); Tanaka et al. (2019); Schrauwen et al. (2007); Jaurigue and Lüdge (2022); Bollt (2021); Gauthier et al. (2021), the number of hidden nodes is comparable. $N = 10^5$ samples over a time range $t \in [0, 4000]$ were used in training, but $t \in [0, 50]$ is shown in testing phase, for useful illustration.

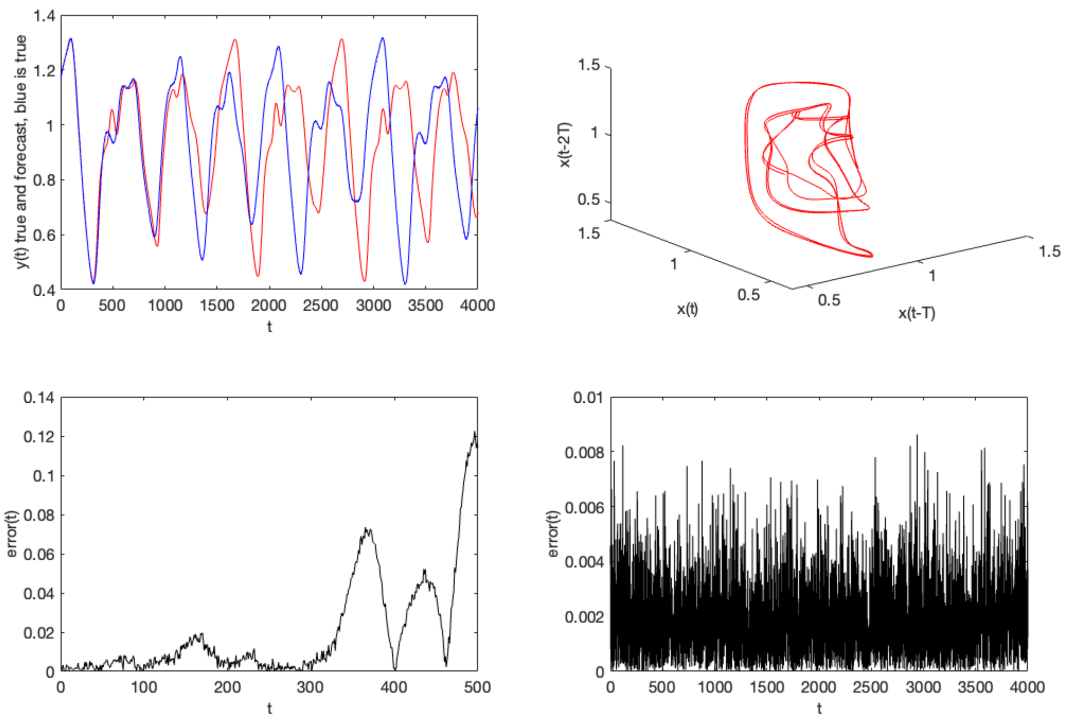
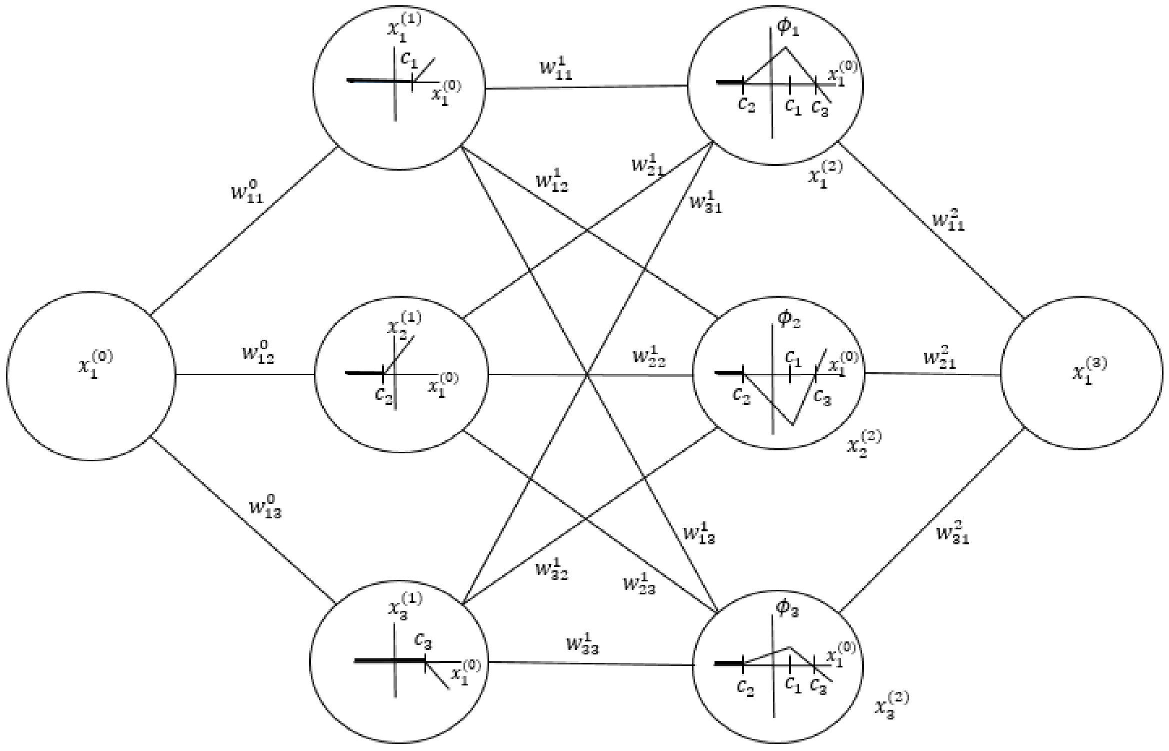


Figure 16: Mackey-Glass differential delay Eq. (62) with parameters, Eqs. (63) simulation data for training and testing an RanNN-1 with $d_1 = 5000$ hidden nodes and regularization parameter 10^{-5}. (Upper Left) a time series showing a true data in blue out of sample, and red testing forecast. Even after error grows large, the oscillations are comparable in type, called reproduces the climate. (Upper Right) Testing forecast demonstration of the attractor. (Lower Left) Errors between true and testing forecast from upper left. (Lower Right) Errors over full training data set during training phase, which is therefore much smaller in scale than errors in testing phase lower left.

6 Random Deep Learning - The Span of The Deep RanNN-q, $q > 1$

Deep learning neural network architectures have become extremely successful and popular. Architectures such as shown in Fig. 3 with $q \geq 2$ hidden layers are called deep, or DFN for deep feedforward network. While fully training, Eq. (21) for best parameter set $\Theta^*_{\text{ANN-}q}$ has been so successful, it is not as clear if the RanNN-q concept of random parameters except for the output layer, optimizing, $\Theta^*_{\text{RanNN-}q}$ by Eq. (22) will work. Even if it does work, it is important to know



$$x_1^{(3)} = w_{11}^2 x_1^{(2)} + w_{21}^2 x_2^{(2)} + w_{31}^2 x_3^{(2)}$$
$$= w_{11}^2 \phi_1(x_1^{(0)}) + w_{21}^2 \phi_2(x_1^{(0)}) + w_{31}^2 \phi_3(x_1^{(0)})$$

Figure 17: A "small" RanNN-2 in one-dimension. Contrast to the small RanNN-1 shown in Fig. 4. This is an RanNN-2 sequel to Fig. 4 which showed in an RanNN-1 how the output layer is a linear combination of the last hidden layer functions, in an RanNN-1 being simple ReLu-like randomly ramped and shifted functions, and here on the next layer, random piecewise linear splines. Compare also to the small RanNN-2 in two dimensions in Fig. 20.

if there are benefits for using any $q > 1$, since RanNN-1 already works. Therefore there would need to be some benefit of better training perhaps with less data by smaller networks, meaning fewer parameters to fit.

In Fig. 11 we show a study of error of fitting the function shown in Fig. 10, in a two-dimensional domain, across increasing hidden layer sizes, and by RanNN-1, 2, and 3. Clearly, there is no observed benefit of going deep when it comes to RanNN, even if there is for a fully trained system. We find similarly across many other examples, not shown here. We now explore how this underwhelming performance of RanNN-q, $q > 1$.

In Fig. 4, we showed a small RanNN-1 to illustrate how it can be understood as a linear combinations of basis functions $\phi_i(x)$ as presented Eqs. (32), (33) in Sec. 4.1 resulting in a piecewise linear spline as described in Appendix 8. Fig. 17 serves as a sequel of Fig. 4, but now in a small RanNN-2, the final layer as output is a linear combination of functions $\phi_i(x)$ which come from the second hidden layer. That second hidden layer is as if the output layer of an RanNN-1. Therefore, each node of the second hidden layer is a piecewise linear function, but now a random one since no training is done to the random parameters through all hidden layers. Then finally, the output layer x_1^3 shown is a linear combination of these random piecewise linear splines.

In Figs. 18-19, successive refinements of an RanNN-2 fitting to data derived from a true function $f(x) = e^{-50x^2}\sin(7(x-0.4))$. From $n = 100$ sample points, and a relatively small hidden layers, $n_1 = n_2 = 20$ in Fig. 18 to $n = 250$ sample points and larger $n_1 = n_2 = 25$ in Fig. 19, we see that fit improves. However, the nature of the errors that can occur with an RanNN-2 are distinct from those of am RanNN-1 and reveal the difficulty of such a random fitting. Notice in particular that even while over all error may decrease near input fitting data points, shown as the blue stars along the true curve, the RanNN-2 fitted curve can deviate significantly from the true curve in some subdomains, especially subdomains where there is a lack of sample data. This is especially true of the deep RanNN, the RanNN-q with $q \geq 2$, due to the compound piecewise linear nature of each individual effective basis function $\phi_i(x)$ that results from a deep RanNN. In particular, notice that each $\phi_i(x)$ shown in the bottom halves of Figs. 18-19 show exactly these compound "bends" and these become more pronounced with network depth, meaning increasing q.

Consider also the same issue from the perspective that adding more depth, so more parameters leads to more geometric complexity. This allows for expressivity which turns out to not be accessible with the limited number of fitting parameters used in fitting just the output layer, but leaving the rest random.

Finally consider the issue of the nature of the complex piecewise linear nature of each basis functions, of a deep RanNN, but now in more than one-dimensional input domain. Contrast Fig. 12 RanNN-1 which is effectively the two dimensional contrast to Fig. 20 a small RanNN-2 in two dimensional input domain analogue of the RanNN-1 suggested in Fig. 4 in one-dimension contrast to the EM-2 in Fig. 17 in one-dimension. The basic story is the same; the basis functions from an RanNN-1 one each have a single discontinuity and resulting fitting to data results in piecewise linear continuous functions which are otherwise basically simple, without any wild variations. However, and RanNN-2 and more so for $q > 2$, each basis function itself has many piecewise linear domains and so fitting with linear combinations can result in more compounding of the piecewise linear domains. With limited number of parameters in fitting an RanNN, meaning just the output layer, all of this flexibility is not accessible. It is only accessible when training the complete set of parameters, of a fully trained neural network. And indeed we see in Figs. 18-19, the fit of the fully trained network is better, for a given size number of layers, but it is more expensive, requiring a fully nonlinear optimization. Furthermore, the fact of more hidden parameters, means it would be more fair to compare to a large hidden layer, so matching the number of trained parameters of the RanNN for a fair comparison.

7 Acknowledgments

The author would like to acknowledge support from the NIH-CRCNS, DARPA RSDN, the ARO, the AFSOR and the ONR. We would also gratefully acknowledge Edmilson Roque dos Santos and also Felix Dietrich.

8 Appendix: How an RanNN-1 Yields a Piecewise Linear Spline in One-Dimension that Closely Tracks Data

Building on the discussion of Sec. 4.1 that describes how an RanNN-1 SLFN in one dimension generates basis functions that span certain piecewise linear functions, we now turn to describe how a large RanNN-1 can generate a continuous piecewise linear spline that almost agrees with given input data $\mathcal{D}$, in 1-dimension. This is similar, but not identical, to the idea of a universal approximation theorem that underpins neural networks theory in general **?**, and specifically such theorems exist for RanNN **?**. Rather than show that the RanNN can approximate a function g that generated the data, here we give just the description of how the RanNN can generate a function $\hat{f}$ that is close to the data points itself, or equivalently, it is close to a piecewise linear function f that runs through the data.

The general scenario is in three parts. We will assume as before that the data $\mathcal{D}$ is sorted by abscissa, $x_1 < x_2 < ... < x_N$ (sorted by re-indexing if necessary), as are the randomly selected x-intercepts. We also assume tat the c_i from Eq. (30) are distinct, which is almost always true.

1. The fitted function can exactly match the data if the abscissa match: If the data set size is the same as the number of hidden nodes, $d_1 = N$, and $y_1 = 0$, and the abscissa coincide $c_1 = x_1, c_2 = x_2, ..., c_{d_1} = x_{d_1}$, then the span of the set of ramp functions, $\Delta_{d_1} = span(\{\phi_i(x)\}_{i=1}^{d_1})$ is a linear spline that accommodates data $\mathcal{D} = \{(x_i, y_i)\}_{d_1}$. By accommodates, we mean there is an $\hat{f} \in \Delta$ such that $\hat{f}(c_i) = y_i$, for all $i = 1, 2, ..., d_1$. The proof of this fact is an expansion of what was shown by example in Sec. 4.1, especially Fig. 4, considering an induction from left to right.

2. If the set if abscissa x_i and c_j do not match, but the c_j finely cover the domain of x_i, then a fitted piecewise linear spline f exists that almost agrees a piecewise linear spline f through the data: If the randomly specified c_j x-intercept points by Eq. (30) do not coincide with all of the data points' abscissa x_i, or even some of the data points', then if there is a fine covering of c_j, then a given precision $\epsilon > 0$ can be met, meaning that for each x_i, there is a c_j such that 1. $|x_i - c_j| < \epsilon$ and $|y_i - \hat{f}(c_j)| < \epsilon$. A proof is as follows. Let $L = max_{x \in [x_1, x_N]} |f'(x)|$, essentially the Lipschitz constant of the f through the data. Since f is piecewise linear, $f(x_{i+1}) - f(x_i) \leq L|x_{i+1} - x_i|$, $i = 0, ..., N-1$. Let $\hat{f}(c_j) = f(c_j)$, and therefore both $|y_i - \hat{f}(c_j)| \leq L|x_i - c_j| < \epsilon$ and $|x_i - c_j| < \epsilon$ are true if we demand I. $|x_i - c_j| < max(\epsilon, \epsilon/L)$. This therefore explicitly states that for precision $\epsilon > 0$, a fine grid of c_j must allow condition I., and we call c_{j_i} such a point for a given x_i. With this precision, a piecewise linear $\hat{f}$ can be built through these ϵ precision points, $\{c_{j_i}\}_{i=1}^{N}$ by method similarly to the 3-point example of Sec. 4.1. See Fig. 6 and contrast with Fig. 4(e), and proceed left to right by induction.

3. Even random x-intercepts from a large SLFN random neural network, the RanNN-1, will generally finely cover the domain, thus allowing construction in step 2: If the support of the distribution of random parameters, results in a distribution of the random variables of ratios from which the x-intercepts c_j result, Eq. (30) has a distribution that is absolutely continuous in the interval ℓ that contains the data abscissa, all $x_i \in \ell$, then by the law of large numbers, for a large enough sample size $d_1 > 0$ (coinciding with the width of the shallow network), there is a c_j sufficiently close to each x_i, for any chosen precision. In other words, large shallow networks result in c_j filling in closely to each x_i.

With these points 1-3 enumerated, we conclude with,

- A large RanNN-1, can closely match data $\mathcal{D}$, with high probability: $\hat{f} \in \Delta_{d_1}$ for $d_1 \gg 1$ when trained to a continuous piecewise linear function f through data $\mathcal{D}$ deviates from f only slightly $|f(x) - \hat{f}(x)| < \epsilon$ and only on a small set.

However, while for large d_1, Δ_{d_1} includes functions $\hat{f}$ that almost run through all the data, we include some regularization in the fitting process, since otherwise such a fitting is highly prone to overfitting noisy data.

9 Appendix: Review of Regularized Pseudo-Inverse

We review how to numerically and stably compute the pseudo-inverse by the singular value decomposition, with regularized singular values (SVD). Reviewing the matrix theory of regularized pseudo-inverses for general matrices, if,

$$Xb = z, \tag{64}$$

$X_{n\times p}, b_{p\times 1}, z_{n\times 1}$, then if the SVD is $X = U\Sigma V$, with orthogonal matrices, $n \times n$, U satisfies, $UU^T = U^TU = I$ and $p \times p$, V satisfies $VV^T = V^TV = I$, and Σ is $n \times p$ "diagonal" matrix of singular values, $\sigma_1 \geq \sigma_2 \geq \sigma_r \geq 0 \geq \sigma_p \geq 0$,

$$\begin{aligned}
\Sigma \quad &= \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_p \end{bmatrix}, \text{ if } n = p, \\
\text{or} \quad &= \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \vdots \\ & & \sigma_p & \end{bmatrix}, \text{ if } n > p, \\
\text{or} \quad &= \begin{bmatrix} \sigma_1 & & & \cdots \\ & \ddots & & \cdots \\ & & \sigma_p & \cdots \end{bmatrix}, \text{ if } n < p.
\end{aligned} \tag{65}$$

Then,

$$X^\dagger := (X^TX)^{-1}X^T = V\Sigma^\dagger U^T, \text{ where, } \begin{bmatrix} \frac{1}{\sigma_1} & & \\ & \ddots & \\ & & \frac{1}{\sigma_p} \\ \vdots & \vdots & \vdots \end{bmatrix}, \text{ in the } n > p \text{ case }. \tag{66}$$

The least squares estimator of $Xb = z$ is,

$$b^* = (X^TX)^{-1}X^Tz := X^\dagger z, \tag{67}$$

and we write the ridge regression Tikhonov regularized solution,

$$b^*_\lambda = (X^TX + \lambda I)^{-1}X^tz = V(\Sigma^T\Sigma + \lambda I)^{-1}\Sigma^TU^Tz := X^\dagger_\lambda z. \tag{68}$$

The regularized pseudo-inverse $X_\lambda^\dagger$ is better stated in terms of the regularized singular values, by,

$$\Sigma_\lambda^\dagger := (\Sigma^T\Sigma + \lambda I)^{-1}\Sigma^T = \begin{bmatrix} \frac{\sigma_1}{\sigma_1^{(2)}+\lambda} & & \\ & \ddots & \\ & & \frac{\sigma_p}{\sigma_p^{(2)}+\lambda} \\ \vdots & \cdots & \end{bmatrix} \text{ in the } n > p \text{ case,} \tag{69}$$

and then,

$$b_\lambda^* = X_\lambda^\dagger z = V\Sigma_\lambda^\dagger U^T z. \tag{70}$$

Throughout, since we will always refer to regularized pseudo-inverses, we will not emphasize this by abusing notation allowing that b^* denotes b_λ^* even if only a very small $\lambda > 0$ is chosen, unless otherwise stated, $\lambda = 1.0 \times 10^{-8}$. This mitigates the tendency of overfitting or likewise stated in terms of zero or almost zero singular values that would otherwise appear in the denominators of $\Sigma^\dagger$. The theory is similar for $n = p$ and $n < p$, as well as the scenario where z is not just a vector but a matrix, and likewise as in Eq. (28) where we refer to the transpose scenario.

References

Weierstrass's theorem on approximation by polynomials" and" extension of weierstrass's approximation theory. §.

Hervé Abdi, Dominique Valentin, and Betty Edelman. *Neural networks.* Number 124. Sage, 1999.

Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5 (4-5):185–196, 1993.

Oludele Awodele and Olawale Jegede. Neural networks and its application in engineering. *Sci IT*, pages 83–95, 2009.

Ernest Julius Berg. *Heaviside's operational calculus.* McGraw-Hill Book Company New York, 1936.

Erik Bolager, Iryna Burak, Chinmay Datar, Qing Sun, and Felix Dietrich. Sampling weights of deep neural networks. *submitted*, 2023.

Erik Bollt. On explaining the surprising success of reservoir computing forecaster of chaos? the universal machine learning dynamical system with contrast to var and dmd. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(1):013108, 2021.

Erik M Bollt and Naratip Santitissadeekorn. *Applied and computational measurable dynamics.* SIAM, 2013.

Erik M Bollt, Qianxiao Li, Felix Dietrich, and Ioannis Kevrekidis. On matching, and even rectifying, dynamical systems through koopman operator eigenfunctions. *SIAM Journal on Applied Dynamical Systems*, 17(2):1925–1960, 2018.

Léon Bottou et al. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control.* Cambridge University Press, 2022.

John Charles Burkill. *Lectures on approximation by polynomials*, volume 16. Tata Institute of Fundamental Research, 1959.

John B Butcher, David Verstraeten, Benjamin Schrauwen, Charles R Day, and Peter W Haycock. Reservoir computing and extreme learning machines for non-linear time-series data analysis. *Neural networks*, 38:76–89, 2013.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics–informed neural networks: where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

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

Louis De Branges. The stone-weierstrass theorem. *Proceedings of the American Mathematical Society*, 10(5):822–824, 1959.

Shifei Ding, Nan Zhang, Xinzheng Xu, Lili Guo, and Jian Zhang. Deep extreme learning machine and its application in eeg classification. *Mathematical Problems in Engineering*, 2015, 2015.

J Doyne Farmer. Chaotic attractors of an infinite-dimensional dynamical system. *Physica D: Nonlinear Phenomena*, 4(3):366–393, 1982.

Vincent François-Lavet, Guillaume Rabusseau, Joelle Pineau, Damien Ernst, and Raphael Fonteneau. On overfitting and asymptotic bias in batch reinforcement learning with partial observability. *Journal of Artificial Intelligence Research*, 65:1–30, 2019.

G David Garson. *Neural networks: An introductory guide for social scientists.* Sage, 1998.

Daniel J Gauthier, Erik Bollt, Aaron Griffith, and Wendson AS Barbosa. Next generation reservoir computing. *Nature communications*, 12(1):5564, 2021.

RL Graham, DE Knuth, and O Patashnik. Answer to problem 9.60 in concrete mathematics: A foundation for computer science. *ed: Reading, MA: Addison-Wesley*, 1994.

Aaron Griffith, Andrew Pomerance, and Daniel J Gauthier. Forecasting chaotic systems with very low connectivity reservoir computers. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 29(12):123108, 2019.

Peter Hall. On the distribution of random lines. *Journal of Applied Probability*, 18(3):606–616, 1981.

David M Himmelblau. Applications of artificial neural networks in chemical engineering. *Korean journal of chemical engineering*, 17:373–392, 2000.

Tom Hope, Yehezkel S Resheff, and Itay Lieder. *Learning tensorflow: A guide to building deep learning systems.* " O'Reilly Media, Inc.", 2017.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4 (2):251–257, 1991.

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

Gao Huang, Guang-Bin Huang, Shiji Song, and Keyou You. Trends in extreme learning machines: A review. *Neural Networks*, 61:32–48, 2015.

Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: a new learning scheme of feedforward neural networks. *2004 IEEE international joint conference on neural networks (IEEE Cat. No. 04CH37541, volume=2, pages=985–990, year=2004.*

Guang-Bin Huang, Lei Chen, Chee Kheong Siew, et al. Universal approximation using incremental constructive feedforward networks with random hidden nodes. *IEEE Trans. Neural Networks*, 17(4):879–892, 2006a.

Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006b.

Imran Khan Mohd Jais, Amelia Ritahani Ismail, and Syed Qamrun Nisa. Adam optimization algorithm for wide and deep neural network. *Knowledge Engineering and Data Science*, 2(1): 41–46, 2019.

Lina Jaurigue and Kathy Lüdge. Connecting reservoir computing with statistical forecasting and deep neural networks. *Nature Communications*, 13(1):227, 2022.

George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

Panayotis G Kevrekidis, Jesús Cuevas-Maraver, and Avadh Saxena. *Emerging Frontiers in Nonlinear Science.* Springer, 2020.

Patrick Kidger and Terry Lyons. Universal approximation with deep narrow networks. In *Conference on learning theory*, pages 2306–2327. PMLR, 2020.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Ron Kohavi, David H Wolpert, et al. Bias plus variance decomposition for zero-one loss functions. In *ICML*, volume 96, pages 275–83, 1996.

Alan Lapedes and Robert Farber. How neural nets work. In *Neural information processing systems*, 1987.

Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

Qianxiao Li, Felix Dietrich, Erik M Bollt, and Ioannis G Kevrekidis. Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the koopman operator. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(10):103111, 2017.

Yuhong Li and Weihua Ma. Applications of artificial neural networks in financial economics: a survey. In *2010 International symposium on computational intelligence and design*, volume 1, pages 211–214. IEEE, 2010.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Allan J Lichtenberg and Michael A Lieberman. *Regular and chaotic dynamics*, volume 38. Springer Science & Business Media, 2013.

Edward N Lorenz. Deterministic nonperiodic flow. *Journal of atmospheric sciences*, 20(2):130–141, 1963.

Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM review*, 63(1):208–228, 2021.

Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer science review*, 3(3):127–149, 2009.

Michael C Mackey and Leon Glass. Oscillation and chaos in physiological control systems. *Science*, 197(4300):287–289, 1977.

George Marsaglia. Ratios of normal variables and ratios of sums of uniform variables. *Journal of the American Statistical Association*, 60(309):193–204, 1965.

Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

Brady Neal. On the bias-variance tradeoff: Textbooks need an update. *arXiv preprint arXiv:1912.08286*, 2019.

Takato Nishijima. Universal approximation theorem for neural networks. *arXiv preprint arXiv:2102.10993*, 2021.

Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

Yoh-Han Pao, Gwang-Hoon Park, and Dejan J Sobajic. Learning and generalization characteristics of the random vector functional-link net. *Neurocomputing*, 6(2):163–180, 1994.

Jigneshkumar L Patel and Ramesh K Goyal. Applications of artificial neural networks in medical science. *Current clinical pharmacology*, 2(3):217–226, 2007.

Jaideep Pathak, Brian Hunt, Michelle Girvan, Zhixin Lu, and Edward Ott. Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach. *Physical review letters*, 120(2):024102, 2018.

MY Rafiq, G Bugmann, and DJ Easterbrook. Neural network design for engineering applications. *Computers & Structures*, 79(17):1541–1552, 2001.

Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. Survey of expressivity in deep neural networks. *arXiv preprint arXiv:1611.08083*, 2016.

Bhukya Ramadevi and Kishore Bingi. Chaotic time series forecasting approaches using machine learning techniques: A review. *Symmetry*, 14(5):955, 2022.

Ricardo Francisco Reier Forradellas, Sergio Luis Náñez Alonso, Javier Jorge-Vazquez, and Marcela Laura Rodriguez. Applied machine learning in social sciences: neural networks and crime prediction. *Social Sciences*, 10(1):4, 2020.

Paul I Richards. Averages for polygons formed by random lines. *Proceedings of the National Academy of Sciences*, 52(5):1160–1164, 1964.

Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-hill New York, 1976.

Matteo Sangiorgio, Fabio Dercole, and Giorgio Guariso. Forecasting of noisy chaotic systems with deep neural networks. *Chaos, Solitons & Fractals*, 153:111570, 2021.

Sheikh Saqlain, Wei Zhu, Efstathios G Charalampidis, and Panayotis G Kevrekidis. Discovering governing equations in discrete systems using pinns. *arXiv preprint arXiv:2212.00971*, 2022.

Wouter F Schmidt, Martin A Kraaijveld, Robert PW Duin, et al. Feed forward neural networks with random weights. In *International conference on pattern recognition*, pages 1–1. IEEE Computer Society Press, 1992.

Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th european symposium on artificial neural networks. p. 471-482 2007*, pages 471–482, 2007.

Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pages 4558–4566. PMLR, 2018.

Boris Shekhtman. Why piecewise linear functions are dense in c [0, 1]. *Journal of Approximation Theory*, 36(3):265–267, 1982.

Nishant Shukla and Kenneth Fricklas. *Machine learning with TensorFlow*. Manning Greenwich, 2018.

Richard P Stanley et al. An introduction to hyperplane arrangements. *Geometric combinatorics*, 13(389-496):24, 2004.

Daren S Starnes, Dan Yates, and David S Moore. *The practice of statistics*. Macmillan, 2010.

Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *2010 18th Euromicro conference on parallel, distributed and network-based processing*, pages 317–324. IEEE, 2010.

David Strohmaier. Ontology, neural networks, and the social sciences. *Synthese*, 199(1-2):4775–4794, 2021.

Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019.

Matus Telgarsky. Representation benefits of deep feedforward networks. *arXiv preprint arXiv:1509.08101*, 2015.

William T Trotter. Partially ordered sets. *Handbook of combinatorics*, 1:433–480, 1995.

Muhammad Uzair, Faisal Shafait, Bernard Ghanem, and Ajmal Mian. Representation learning with deep extreme learning machines for efficient image set classification. *Neural Computing and Applications*, 30:1211–1223, 2018.

Pantelis R Vlachas, Jaideep Pathak, Brian R Hunt, Themistoklis P Sapsis, Michelle Girvan, Edward Ott, and Petros Koumoutsakos. Backpropagation algorithms and reservoir computing in recurrent neural networks for the forecasting of complex spatiotemporal dynamics. *Neural Networks*, 126:191–217, 2020.

Bernard Widrow, David E Rumelhart, and Michael A Lehr. Neural networks: applications in industry, business and science. *Communications of the ACM*, 37(3):93–106, 1994.

MJ Willis, C Di Massimo, GA Montague, MT Tham, and AJ Morris. Artificial neural networks in process engineering. In *IEE proceedings D (control theory and applications)*, volume 138, pages 256–266. IET, 1991.

Wenjun Zhang. *Computational ecology: artificial neural networks and their applications*. World Scientific, 2010.

Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*, pages 1–2. Ieee, 2018.

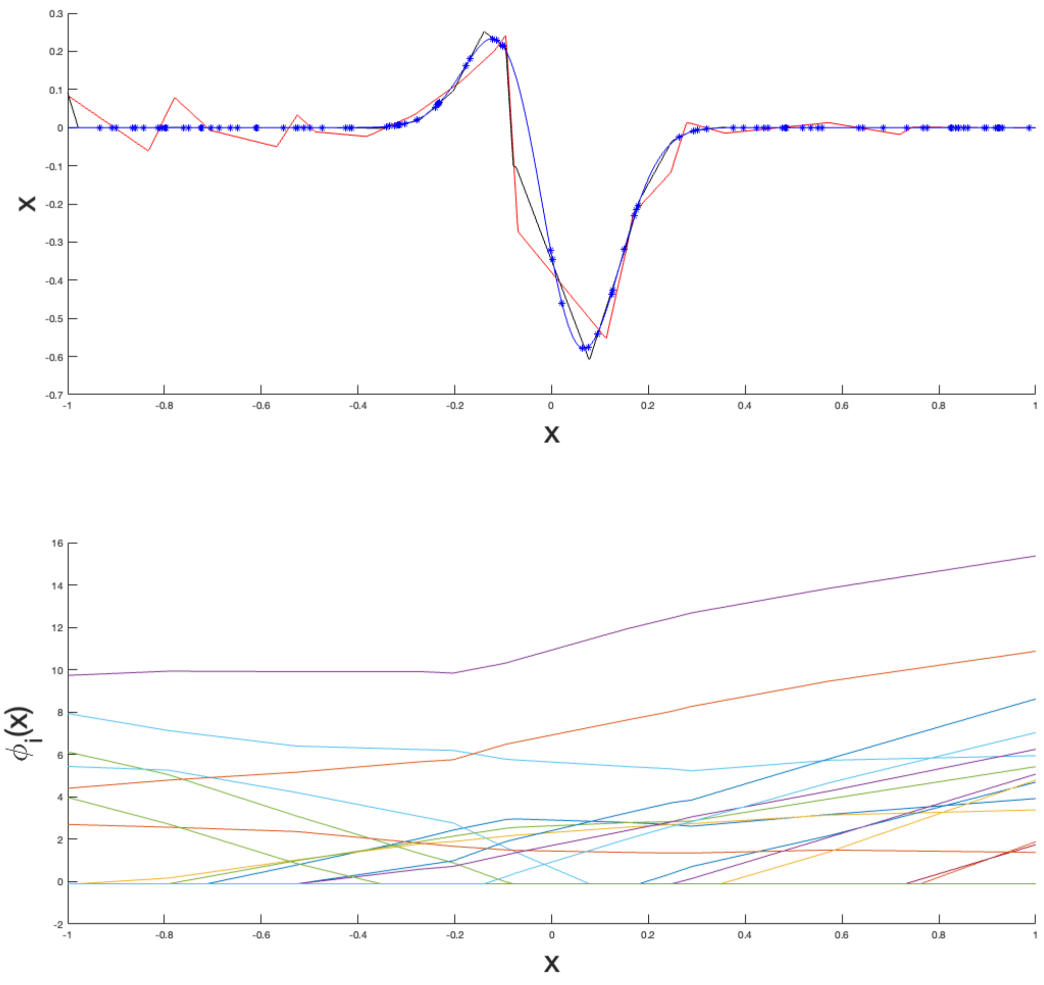
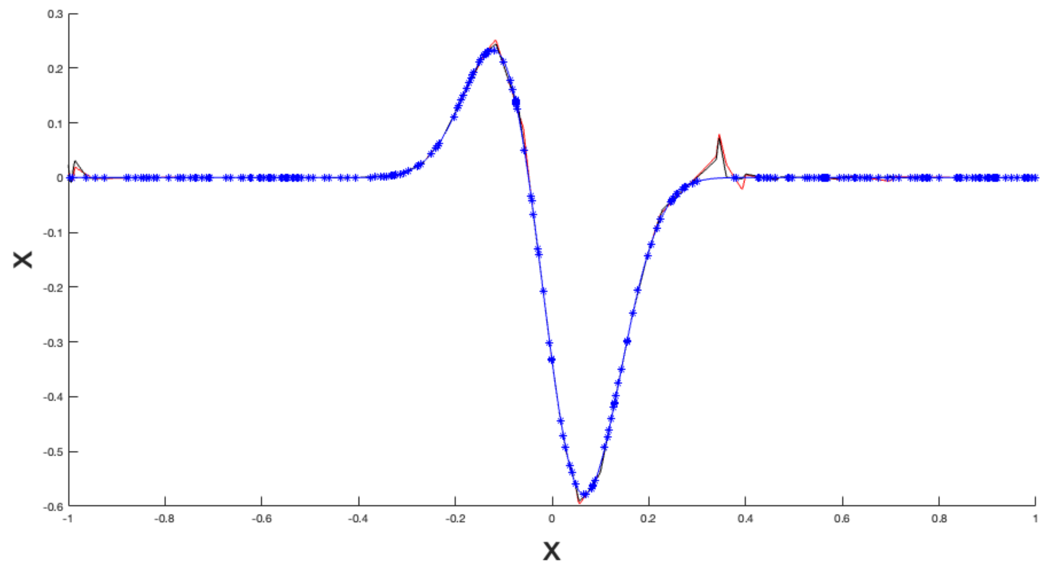


Figure 18: Random deep learning versus deep learning. (Top) Synthetic data of $n = 50$ data points (x_i, y_i) from $y = f(x) = e^{-50x^2}\sin(7(x - 0.4))$ are shown as blue stars, along with the function sampled on a fine grid for illustration. An RanNN-2 model with hidden layers of size, $n_1 = n_2 = 20$ nodes where used to generate the red curve shown, sampled on a the same fine grid as the smooth function curve. A fully trained FFNN of the same hidden two layer structure is also shown, but again on the same finer grid than the training process for illustration sake. Compare to Fig. 19 which uses more training data and a larger hidden layers. (Bottom) The $n_2 = 20$ resulting functions $\phi_i(x)$ associated with the output layer. Compare to Figs. 19 and 20.



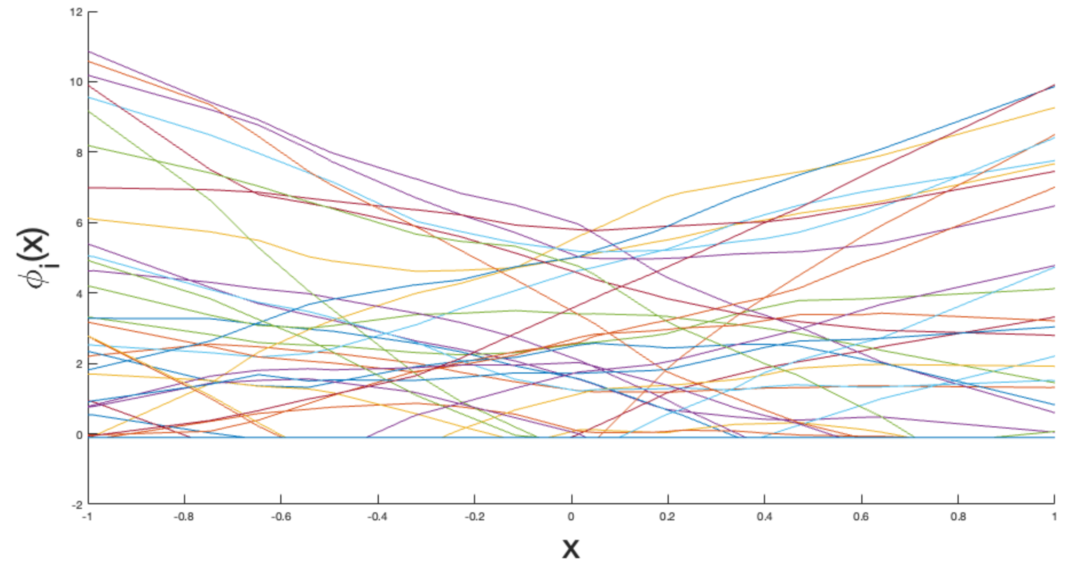


Figure 19: Random deep learning versus deep learning. Synthetic data of $n = 250$ data points (x_i, y_i) from $y = f(x) = e^{-50x^2}\sin(7(x - 0.4))$ are shown as blue stars, along with the function sampled on a fine grid for illustration. An RanNN-2 model with hidden layers of size, $n_1 = n_2 = 25$ nodes where used to generate the red curve shown, sampled on a the same fine grid as the smooth function curve. A fully trained FFNN of the same hidden two layer structure is also shown, but again on the same finer grid than the training process for illustration sake. Compare to Fig. 18 which uses fewer training data and a smaller hidden layers. (Bottom) The $n_2 = 25$ resulting functions $\phi_i(x)$ associated with the output layer. Compare to Figs. 19 and 20.

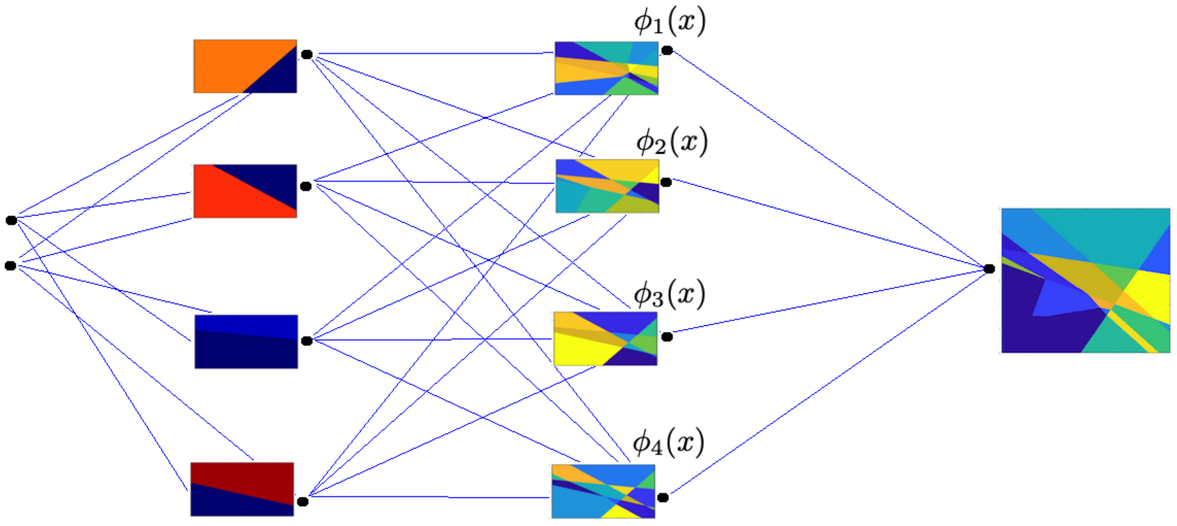


Figure 20: A small RanNN-2 in two-dimensions, with $n_1 = n_2 = 4$ nodes in each hidden layer. The corresponding basis functions $\phi_i(x)$ are piece-wise linear, and linear in the domain shown, in subdomains counting, $17, 12, 10, 14$ each. Compare to the RanNN-2 in one-dimension in Fig. 17.