Proceedings of Machine Learning Research vol 107:1–26, 2020

SAND2019-14904C

Robust Training and Initialization of Deep Neural Networks: An Adaptive Basis Viewpoint

Eric C. Cyr ECCYR@SANDIA.GOV
Center for Computing Research, Sandia National Laboratories

Mamikon A. Gulian MGULIAN@SANDIA.GOV
Center for Computing Research, Sandia National Laboratories

Ravi G. Patel RGPATEL@SANDIA.GOV
Center for Computing Research, Sandia National Laboratories

Mauro Perego MPEREGO@SANDIA.GOV
Center for Computing Research, Sandia National Laboratories

Nathaniel A. Trask NATRASK@SANDIA.GOV
Center for Computing Research, Sandia National Laboratories

Abstract

Motivated by the gap between theoretical optimal approximation rates of deep neural networks (DNNs) and the accuracy realized in practice, we seek to improve the training of DNNs. The adoption of an adaptive basis viewpoint of DNNs leads to novel initializations and a hybrid least squares/gradient descent optimizer. We provide analysis of these techniques and illustrate via numerical examples dramatic increases in accuracy and convergence rate for benchmarks characterizing scientific applications where DNNs are currently used, including regression problems and physics-informed neural networks for the solution of partial differential equations.

1. Introduction

Universal approximation properties of neural networks are often touted as an explanation of the success of deep neural networks (DNNs) in applications. Despite their importance, such theorems offer no explanation for the advantages of neural networks, let alone *deep* neural networks, over classical approximation methods, since universal approximation properties are enjoyed by polynomials (Cheney and Light, 2009) as well as single layer neural networks (Cybenko, 1989). To address this, a recent thread has emerged in the literature concerning optimal approximation with deep ReLU networks, where the error in an optimal choice of weights and biases is bounded from above using the width and depth of the neural network.

For example, using the "sawtooth" function of Telgarsky (2015), Yarotsky (2017) constructed an exponentially accurate (in the number of layers) ReLU network emulator for multiplication $(x, y) \mapsto xy$. This construction is used to obtain upper bounds on optimal approximation based upon DNN emulation of polynomial approximation. Building on these ideas, Opschoor et al. (2019) proved that optimal approximation with deep ReLU networks can emulate adaptive hp-finite element approximation, with greater depth allowing p-refinement to obtain exponential convergence rates. An additional contribution by He et al. (2018) reinterpreted single hidden layer ReLU networks as r-adaptive piecewise linear finite element spaces.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Despite this, it remains a challenge to realize these theorized convergence rates for DNNs using practical initialization and training methods. The need is particularly acute in scientific machine learning (SciML) applications which demand greater accuracy and robustness from DNNs (Raissi et al., 2019; Baker et al., 2019). In practice, optimization and initialization challenges preclude the realization of theoretical convergence rates. Optimizers are susceptible to finding suboptimal local minima of loss functionals, and as a result DNN regression typically stagnates after achieving only a few digits of accuracy. For example, using the aforementioned architecture of Yarotsky (2017) for the deep ReLU emulator of $x \mapsto x^2$, but with random initial weights, Fokina and Oseledets (2019) showed that training with stochastic gradient descent to approximate $x \mapsto x^2$ fails to demonstrate a significant improvement in error with depth, let alone exponential convergence with the number of layers. Lu et al. (2018, 2019) demonstrate consistent failure of deep ReLU networks to approximate the function $|x|$ on $[-1, 1]$ due to gradient death at initialization. These results illustrate the need for robust training and initialization algorithms for regression and approximation in scientific problems. We aim to bridge the gap between theoretical optimal error estimates and the error one can consistently achieve with training.

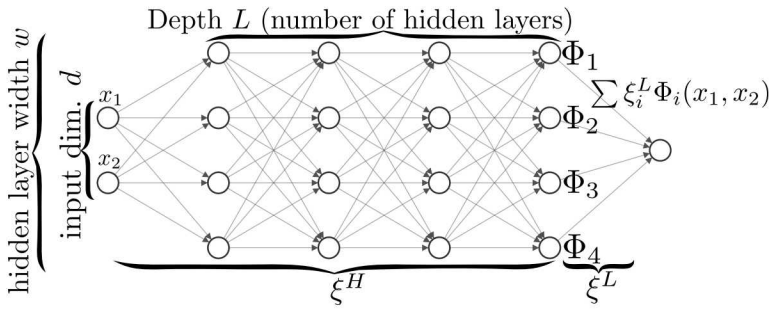

Figure 1: Adaptive basis view of a DNN with linear output layer, with notation used in this article.

In the current work we adapt the perspective that DNNs provide a meshfree technique to construct an adaptive basis. This viewpoint suggests an optimizer that alternates between least squares (LS) and gradient descent (GD) steps. This process amounts to adapting the basis functions with respect to to the data using GD while ensuring with LS that basis functions optimally fit the data. This training strategy is applicable to networks with arbitrary activation function in the hidden layers, a final linear activation layer, and a mean-square loss functional.

From the adaptive basis viewpoint, we also propose a new initialization for deep ReLU networks that we refer to as the "box initialization", designed to provide an expressive initial guess for the basis. We show that this initialization outperforms the Glorot (Glorot and Bengio, 2010) and He initializations (He et al., 2015) for one-dimensional approximation using plain networks for a mild number of layers. Via a novel analysis of DNNs in terms of this adaptive basis perspective, we extend the box initialization to residual ReLU networks and show improvements upon the He and Glorot initializations through 256 layers.

Combining our initialization for residual neural networks (ResNets) with the hybrid LSGD training algorithm, we demonstrate convergence of the approximation error for very deep ReLU neural networks with increasing depth. While the variance in errors remains high and the "convergence rates" are lower than suggested theoretically, the improvement in reliability across a range of regression-like applications is substantial. Further, the architectures used are standard. In con-

trast, previous works in the literature have focused on how to prevent collapse or else considered specialized architectures.

2. Problem statement

We consider in this work the following class of ℓ_2 regression problems:

$$\underset{\boldsymbol{\xi}}{\operatorname{argmin}} \sum_{k=1}^{K} \epsilon_k \left\| \mathcal{L}_k[u] - \mathcal{L}_k\left[\mathcal{NN}_{\boldsymbol{\xi}}\right] \right\|^2_{\ell_2(\mathcal{X}_k)} \tag{1}$$

where for each $k = 1, 2, ..., K$, $\mathcal{X}_k = \{x_i^{(k)}\}_{i=1}^{N_k}$ denotes a finite collection of data points, $\mathcal{NN}_{\boldsymbol{\xi}}$ a neural network with parameters $\boldsymbol{\xi}$, and $\mathcal{L}_k$ a linear operator. In the case where $k = 1$ and $\mathcal{L}$ is the identity, we obtain the standard regression problem

$$\underset{\boldsymbol{\xi}}{\operatorname{argmin}} \left\| u - \mathcal{NN}_{\boldsymbol{\xi}} \right\|^2_{\ell_2(\mathcal{X})}. \tag{2}$$

In general (1) represents a broader class of multi-term loss functions, including those used in physics-informed neural networks (Raissi et al. (2019)) for solving linear PDEs (see Section 5.3). Moreover, while we restrict our study to a single scalar "target" function u in most of the paper, in Section 5.2 we apply our framework to regress multiple functions simultaneously.

We consider the family of neural networks $\mathcal{NN}_{\boldsymbol{\xi}} : \mathbb{R}^d \to \mathbb{R}$ consisting of L hidden layers of width w composed with a final linear layer (see Fig. 1), admitting the representation

$$\mathcal{NN}_{\boldsymbol{\xi}}(\boldsymbol{x}) = \sum_{i=1}^{w} \xi_i^{\mathrm{L}} \Phi_i(\boldsymbol{x}; \boldsymbol{\xi}^{\mathrm{H}}) \tag{3}$$

where $\boldsymbol{\xi}^{\mathrm{L}}$ and $\boldsymbol{\xi}^{\mathrm{H}}$ are the parameters corresponding to the final linear layer and the hidden layers respectively, and we interpret $\boldsymbol{\xi}$ as the concatenation of $\boldsymbol{\xi}^{\mathrm{L}}$ and $\boldsymbol{\xi}^{\mathrm{H}}$. Working with this form allows us to highlight the interpretation of neural networks as an adaptive basis.

A broad range of architectures admit this interpretation. In this work we consider both plain neural networks (also referred to as multilayer perceptrons) and residual neural networks (ResNets). Defining the affine transformation, $\boldsymbol{T}_l(\boldsymbol{x}, \boldsymbol{\xi}) = \boldsymbol{W}_l^{\boldsymbol{\xi}} \cdot \boldsymbol{x} + \boldsymbol{b}_l^{\boldsymbol{\xi}}$, and given an activation function σ, plain neural networks correspond to the choice

$$\boldsymbol{\Phi}^{\text{plain}}(\boldsymbol{x}, \boldsymbol{\xi}) = \boldsymbol{\sigma} \circ \boldsymbol{T}_L \circ \cdots \circ \boldsymbol{\sigma} \circ \boldsymbol{T}_1, \tag{4}$$

while residual networks (see He et al. (2016a,b)) correspond to

$$\boldsymbol{\Phi}^{\text{res}}(\boldsymbol{x}, \boldsymbol{\xi}) = (\boldsymbol{I} + \boldsymbol{\sigma} \circ \boldsymbol{T}_L) \circ \cdots \circ (\boldsymbol{I} + \boldsymbol{\sigma} \circ \boldsymbol{T}_2) \circ (\boldsymbol{\sigma} \circ \boldsymbol{T}_1), \tag{5}$$

where $\boldsymbol{\Phi}$ is the vector of the w functions Φ_i, $\boldsymbol{\sigma}$ the vector of the w activation functions σ and $\boldsymbol{I}$ denotes the identity. In both cases $\boldsymbol{\xi}^{\mathrm{H}}$ corresponds to the weights and biases $\boldsymbol{W}$ and $\boldsymbol{b}$.

In the case of a single hidden layer plain network with ReLU activation, one obtains a piecewise linear C^0 finite element space. This case has been considered by He et al. (2018), who show that training amounts to adapting a piecewise linear finite element space to data. In the broader context considered here, an adaptive basis tailored to the choice of activation function is obtained. For example, selecting a radial basis function (RBF) as activation for a single layer network corresponds to a RBF space with centers and shape parameters adapted to data. Many other architectures admit the proposed interpretation, such as e.g. convolutional networks.

3. Hybrid least squares/GD training approach

Using the Neural Network representation in (3), equation (1) reads

$$\underset{\boldsymbol{\xi}^{\mathrm{L}},\boldsymbol{\xi}^{\mathrm{H}}}{\operatorname{argmin}} \sum_{k=1}^{K} \epsilon_k \left\| \mathcal{L}_k[u] - \sum_i \xi_i^{\mathrm{L}} \mathcal{L}_k\left[\Phi_i(\boldsymbol{x},\boldsymbol{\xi}^{\mathrm{H}})\right] \right\|^2_{\ell_2(\mathcal{X}_k)} . \tag{6}$$

A typical approach to solving Equation 6 is to apply gradient descent with backpropagation jointly in $(\boldsymbol{\xi}^{\mathrm{L}},\boldsymbol{\xi}^{\mathrm{H}})$. Given the adaptive basis viewpoint, an alternative is to hold the hidden weights $\boldsymbol{\xi}^{\mathrm{H}}$ constant and minimize w.r.t. to $\boldsymbol{\xi}^{\mathrm{L}}$, yielding the LS problem (for simplicity focusing on $K = 1$):

$$\underset{\boldsymbol{\xi}^{\mathrm{L}}}{\operatorname{argmin}} \left\| A\boldsymbol{\xi}^{\mathrm{L}} - \boldsymbol{b} \right\|^2_{\ell_2(\mathcal{X})} \tag{7}$$

Here we have $\boldsymbol{b}_i = \mathcal{L}[u](\boldsymbol{x}_i)$ and $A_{ij} = \mathcal{L}\left[\Phi_j(\boldsymbol{x}_i,\boldsymbol{\xi}^{\mathrm{H}})\right]$ for $\boldsymbol{x}_i \in \mathcal{X},\ i = 1,\dots,N, j = 1,\dots,w$. Problem 7 is well posed if $N \geq w$ and A is a full-rank matrix; otherwise the problem is underdetermined and admits multiple solutions. This occurs if the basis functions Φ_j are linearly dependent over $\ell_2(\mathcal{X})$, as can occur for many weights initializations (see Section 4). In that case, the Moore-Penrose pseudo-inverse A^+ can be used to compute the minimum-norm solution $\boldsymbol{\xi}^{\mathrm{L}} = A^+\boldsymbol{b}$. In this work, we use the TensorFlow (Abadi et al., 2015) implementation provided by the function `lstsq` to compute the minimun-norm solution $\boldsymbol{\xi}^{\mathrm{L}}$.

Exposing the LS problem in this way prompts a natural modification of gradient descent. The optimization algorithm proceeds by alternating between: a LS solve to update $\boldsymbol{\xi}^{\mathrm{L}}$ by a global minimum for given $\boldsymbol{\xi}^{H}$; and a GD step to update $\boldsymbol{\xi}^{\mathrm{H}}$ (Algorithm 1).

Algorithm 1 Hybrid least squares/gradient descent

function LSGD($\boldsymbol{\xi}_0^H$)
  $\boldsymbol{\xi}^H = \boldsymbol{\xi}_0^H$ ▷ Input initialized hidden parameters
  $\boldsymbol{\xi}^L = LS(\boldsymbol{\xi}^H)$ ▷ Solve LS problem for $\boldsymbol{\xi}^L$
  for $i = 1\dots$ **do**
    $\boldsymbol{\xi}^H = GD(\boldsymbol{\xi})$ ▷ Solve GD problem
    $\boldsymbol{\xi}^L = LS(\boldsymbol{\xi}^H)$
  end for
end function

Problem 6 is referred to in the inverse-problems literature as a *separable nonlinear least square* problem. It is often solved with the variable projection method (Golub and Pereyra, 1973, 2003) in which $\boldsymbol{\xi}^{\mathrm{L}}$ is computed by solving (7) as a function of $\boldsymbol{\xi}^{\mathrm{H}}$ and is substituted into (6), leading to a minimization problem over the the hidden parameters $\boldsymbol{\xi}^{\mathrm{H}}$ only, which can then be solved with a suitable optimization method. The variable projection method has been used for shallow (one hidden layer) neural networks in Pereyra et al. (2006). A LS approach was also used in a greedy algorithm to generate adaptive basis elements by Fokina and Oseledets (2019).

In the approach presented here, instead of eliminating $\boldsymbol{\xi}^{\mathrm{L}}$ through a LS solve, we alternate between the minimization of the two sets of parameters, $\boldsymbol{\xi}^{\mathrm{L}}$ and $\boldsymbol{\xi}^{\mathrm{H}}$, which is simpler to implement. In fact, with libraries such as Tensorflow (Abadi et al., 2015) and PyTorch (Paszke et al., 2017), one may automate extraction of the least squares problem (Equation 7) directly from the graphical

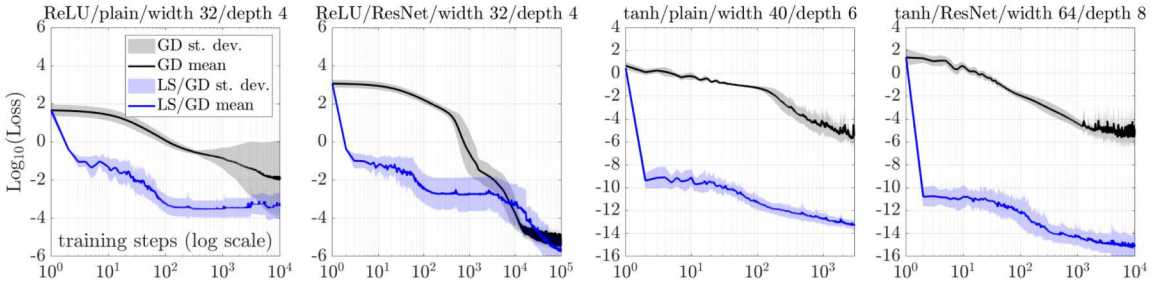

Figure 2: Mean of $\log_{10}(\text{Loss})$ over 16 training runs $\pm$ one standard deviation of the same quantity. Training rate 0.0005 for GD and 0.005 for LSGD for plain network *(left)* and ResNet *(right)*.

representation of a neural network. Hence, algorithm 1 may be easily implemented as a "black-box" layer on top of any architecture described by Equation 3.

We illustrate the advantages of LSGD training for approximating $\sin(2\pi x)$ on $[0, 1]$ using DNNs with ReLU and $\tanh$ activation in plain and ResNet architectures in Fig. 2. We use uniform He initialization and the Adam optimizer (Kingma and Ba, 2014) for the gradient descent steps; learning rates are tuned by hand to give stable training. We found that the LSGD optimizer performs best with a higher learning rate than that of GD – roughly 10 times higher for ReLU networks, and 100 times higher for $\tanh$ networks. The results show that the loss in the LSGD method is typically several orders of magnitude lower than the loss in the GD algorithm after the same number of iterations. This is particularlly apparent for the $\tanh$ networks. However, we also included in Fig. 2 a rare case in which the LSGD loss is momentarily overtaken by the pure GD loss to show that LSGD training and GD training do not admit a simple "global" comparison; for a further discussion of this as well as computational cost of LSGD, see Appendix A.

4. The Box Initialization for deep ReLU networks

The first step in Algorithm 1 is to initialize the hidden layer parameters. An initialization resulting in a well-conditioned, basis that is linearly independent in $\ell_2(\mathcal{X})$ will provide a richer approximation space for the least squares problem and give the gradient descent optimizer several "active" basis functions to tune. In contrast, an initialization leading to poorly-conditioned, linearly dependent basis functions – such as a basis functions with support disjoint from the data – will yield a less expressive basis in which a local variation of the hidden parameters may not improve the loss.

4.1. Plain Neural Networks

Analyses of the representation power of ReLU networks have shed light on the role played by the biases for representing continuous piecewise linear (CPWL) functions (Arora et al., 2016; Hanin, 2017; Hanin and Sellke, 2017; He et al., 2018). For example, for CPWL functions of one variable, He et al. (2018) identified their single layer ReLU network representations $\sum \lambda_i \text{ReLU}(x - \beta_i)$ with nodal finite element representations, with the nodes given by β_i. In higher dimensions, the cut planes (See Figure 3) defined by the bias vectors of single layer ReLU networks correspond to the facets of a CPWL finite element mesh. This implies that to obtain a "feature-rich" initial basis, assuming the data input is normalized to $[0, 1]^d$, one should scatter the cut planes of the ReLU functions over $[0, 1]^d$ randomly.

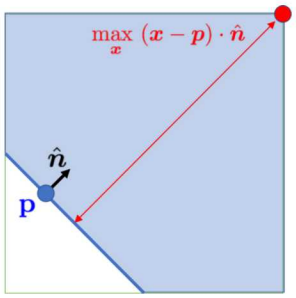

Figure 3: Notation used in the "box initialization" of each node. A random point $\boldsymbol{p}$ with random orientation $\hat{\boldsymbol{n}}$ is used to define a ReLU function of form $\sigma(k(\boldsymbol{x}-\boldsymbol{p})\cdot\hat{\boldsymbol{n}})$. Using Lemma 1, one may choose the slope of the ReLU α to impose an upper bound on the output of each layer. We refer to the hyperplane normal to $\hat{\boldsymbol{n}}$, where the ReLU "switches on", as the *cut plane*.

Loosely speaking, if the above initialization results in hidden layer with "feature-rich" output, it is reasonable to speculate that composing two such layers has a good chance to also result in a "feature-rich" output, provided the first layer maps *into* the domain of the second layer and is as close as possible to being *onto*. The idea behind the "box initialization" for plain networks is to normalize the *output* of each layer to $[0,1]^d$. The goal is to apply the above initialization inductively for each hidden layer and prevent "blow-up" of the initial basis for deeper networks. In the remainder of this section, we consider neural network architectures in which the width of the hidden layers is a constant w throughout the network. This simplifies the analysis, although the algorithm can be considered for networks with variable hidden layer width w_l, $l = 1, ..., L$; see Appendix B.

Referring to Fig. 3, the procedure is for each output row $(1 \dots i \dots w)$ of the layer:

1. Select $\boldsymbol{p} \in [0,1]^w$ at random.
2. Select a normal $\boldsymbol{n}$ at $\boldsymbol{p}$ with random direction.
3. Choose a scaling k such that

$$\max_{\boldsymbol{x}\in[0,1]^w} \sigma(k(\boldsymbol{x}-\boldsymbol{p})\cdot\boldsymbol{n}) = 1. \tag{8}$$

4. Row $\boldsymbol{w}_i$ of $\boldsymbol{W}^{\boldsymbol{\xi}}$ and $\boldsymbol{b}^{\boldsymbol{\xi}}$ are selected as $b_i = k\boldsymbol{p}\cdot\boldsymbol{n}$ and $\boldsymbol{w}_i = k\boldsymbol{n}^T$.

To initialize the first hidden layer, replace w by the input dimension d in steps 1 and 3 above. A full description of this initialization and an efficient way to calculate the k may be found in Algorithm 2 in Appendix B. With the layer initialized as above, consider feeding a box $[0,1]^w$ as input into a given layer. For a plain neural network, the output $\boldsymbol{x}_{l+1}$ of layer l is given by

$$\boldsymbol{x}_{l+1} = \sigma(\boldsymbol{W}_l\boldsymbol{x}_l + \boldsymbol{b}_l). \tag{9}$$

Then, we have for every component $i \in \{1, 2, \dots, w\}$,

$$\min_{\boldsymbol{x}^l\in[0,1]^w} (\boldsymbol{x}_{l+1})_i = 0; \qquad \max_{\boldsymbol{x}^l\in[0,1]^w} (\boldsymbol{x}_{l+1})_i = 1. \tag{10}$$

Equation 10 implies that layer l maps $[0,1]^w$ into $[0,1]^w$. Moreover, ensuring the extrema are achieved on $[0,1]^w$ guarantees its image intersects each side of the hypercube at least at a point. This does not imply however, that each layer map from $[0,1]^w$ into $[0,1]^w$ is onto. Nor, as we will see, that the the composition of two layer maps will have guaranteed intersections with the boundary. Assuming the input into the first hidden layer is contained in $[0,1]^w$, then box initializion ensures that the hidden layers initially map

$$[0,1]^d \xrightarrow{\text{into}} [0,1]^w \xrightarrow{\text{into}} [0,1]^w \xrightarrow{\text{into}} [0,1]^w \xrightarrow{\text{into}} \dots \xrightarrow{\text{into}} [0,1]^w. \tag{11}$$

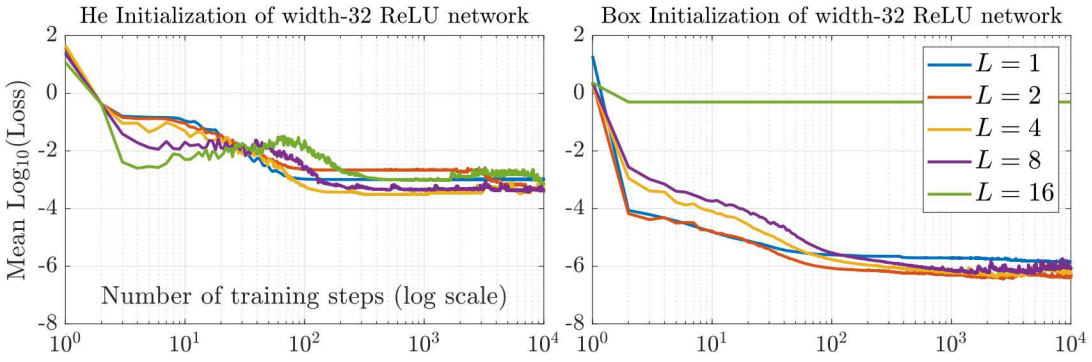

Figure 4: Mean of $\log_{10}(\text{Loss})$ over 16 training runs of plain width-32 ReLU network with $L = 1, 2, 4, 8$ and 16 hidden layers for the He *(left)* and Box *(right)* initializations. The learning rate is 0.005 throughout.

In Figure 4 we compare the the box initialization for a plain ReLU network with width $w = 32$ against the He (see He et al. (2015)) initialization for approximating $\sin(2\pi x)$ on $[0, 1]$. We average over 16 independent training runs. The box initialized basis is significantly richer for up to 8 layers, yielding a loss 2-4 orders of magnitude lower than that of the initialized He basis after the first least squares step. This is borne out by the plots of the initialized basis in Fig. 16. The loss after 10^4 LSGD steps is also lower by 2 orders of magnitude. Despite this promising improvement over He initialization, the box-initialized ReLU network with 16 layers fails to train, and plotting the basis function reveals they are constant over the input to the network; see Appendix C.

To understand why this occurs, consider the image P_L of the unit box $[0, 1]^w$ under L hidden layers of the network, excluding the d-dimensional input layer for now. Fig. 5 shows the evolution of P_L through each layer for different initialization approaches. Because each hidden layer does not map $[0, 1]^w$ *onto* $[0, 1]^w$, as the number of layers increases, we expect P_L to shrink, lose dimension, and eventually collapse to a point. In turn, for input dimension $d \leq w$, the image of the input box $[0, 1]^d$ is a submanifold of P_L, given by the parametrization $(\Phi_1(\boldsymbol{x}), \Phi_2(\boldsymbol{x}), ...\Phi_w(\boldsymbol{x}))$ for $\boldsymbol{x} \in [0, 1]^d$. For example, for the DNNs shown in Fig. 5, with a one-dimensional input this submanifold would be a curve within P_L. The basis function Φ_i is the projection of this submanifold onto the ith coordinate axis; this is illustrated for a width $w = 2$ network with input dimension $d = 1$ in Fig. 6. Once the image P_L is a point, this submanifold within P_L is also a point, so all initial Φ_i will be constant. Fig. 5 demonstrates this for a width-two ReLU network; the He, Glorot, and box initialization suffer from this flaw in the plain network case. While the growth in the magnitude of the basis is controlled (as expected) by box initialization, and the support of the basis does not collapse as quickly in this instance, a statistical study of this approach will indicate that the collapse to a point for all three initializations is inevitable. One possible treatment of this collapse has been proposed in Lu et al. (2019). Issues of training DNNs have also been discussed by Hanin and Rolnick (2018), who proposed a scaling of depth to width as a possible solution. Next, we illustrate how ResNets avoid this issue at higher depth, and propose an analogous box initialization for ResNets.

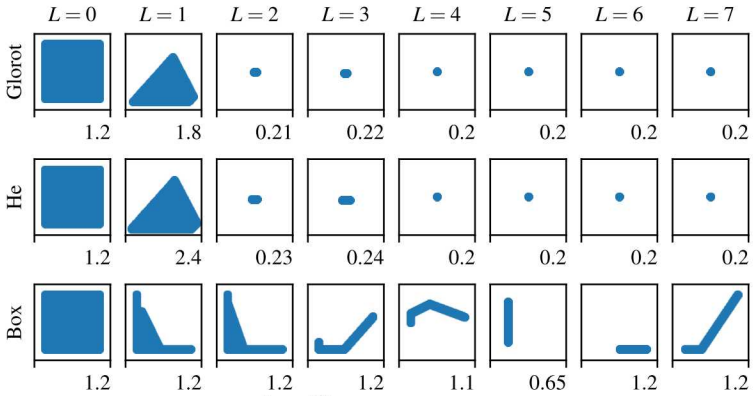

Figure 5: Images P_L of the unit square $[0,1]^2$ under L initialized hidden layers of plain networks for He (*top*) and Box (*bottom*) initializations. Values are presented on the square $[-0.2, H]^2$, where H is denoted to the bottom-right of each image. Collapse to a point corresponds to constant basis functions.

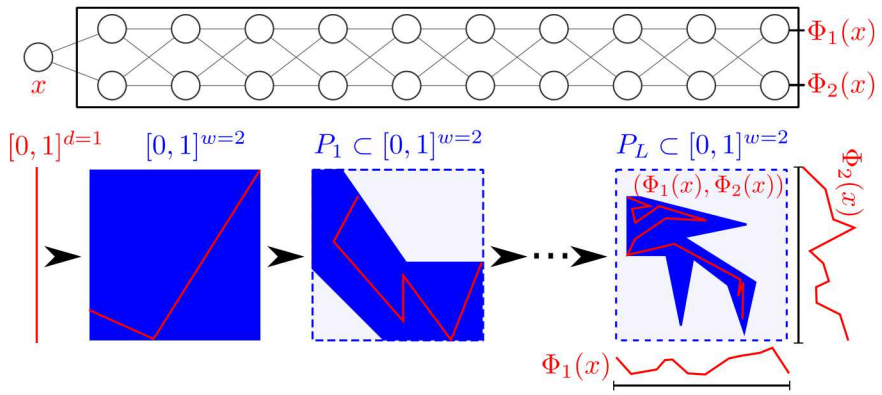

Figure 6: Artist's depiction of the d-dimensional manifold (*red*) parametrized by $(\Phi_1(\boldsymbol{x}), ..., \Phi_w(\boldsymbol{x}))$, which is the image of the input domain $[0,1]^d$ under the input and hidden layers, as a submanifold of the image P_L (*blue*) of the unit box $[0,1]^w$ under the hidden layers. Here $d = 1$ and $w = 2$ to make visualization possible.

4.2. Residual Neural Networks (ResNets)

Consider a residual neural network with input dimension d and hidden layer width w. As usual for a ResNet, unless $d = w$, the first hidden layer is initialized as plain layer as described in Section 4.1 above. Then, for the remaining hidden layers, to initialize the neuron i, $1 \le i \le w$,

1. For m specified later, select $\boldsymbol{p} \in [0, m]^w$ at random.
2. Select a unit normal $\boldsymbol{n}$ at $\boldsymbol{p}$ with random direction.
3. For δ specified later, choose a scaling k such that

$$\max_{[0,m]^w} \sigma(k(\boldsymbol{x} - \boldsymbol{p}) \cdot \boldsymbol{n}) = \delta m. \tag{12}$$

   We again apply Lemma 1 to find the maximal corner.
4. Row $\boldsymbol{w}_i$ of $\boldsymbol{W}^{\boldsymbol{\xi}}$ and $\boldsymbol{b}^{\boldsymbol{\xi}}$ is selected as $b_i = k\boldsymbol{p} \cdot \boldsymbol{n}$ and $\boldsymbol{w}_i = k\boldsymbol{n}^T$.

As for the plain DNN initialization, a more detailed description of the weight and bias initialization procedure can be found in Algorithm 3 in Appendix B. With the layer initialized as above, consider feeding a box $[0, m]^w$ as input into a given layer. For a residual neural network, the output $\boldsymbol{x}^{l+1}$ of layer $l > 1$ is given by $\boldsymbol{x}_{l+1} = \boldsymbol{x}_l + \sigma(\boldsymbol{W}_l\boldsymbol{x}_l + \boldsymbol{b}_l)$ while for the first layer we have $\boldsymbol{x}_2 = \sigma(\boldsymbol{W}_1\boldsymbol{x}_1 + \boldsymbol{b}_1)$. Then, we have for every component $i \in \{1, 2, \ldots, w\}$, $l > 1$

$$\min_{\boldsymbol{x}_l \in [0,m]^w} (\boldsymbol{x}_{l+1})_i \geq \min_{\boldsymbol{x}_l \in [0,m]^w} (\boldsymbol{x}_l)_i + \min_{\boldsymbol{x}_l \in [0,m]^w} \sigma(k(\boldsymbol{x}_l - \boldsymbol{p}) \cdot \boldsymbol{n}) \geq \min_{\boldsymbol{x}_l \in [0,m]^w} (\boldsymbol{x}_l)_i \geq 0 \tag{13}$$

$$\max_{\boldsymbol{x}_l \in [0,m]^w} (\boldsymbol{x}_{l+1})_i \leq \max_{\boldsymbol{x}_l \in [0,m]^w} (\boldsymbol{x}_l)_i + \max_{\boldsymbol{x}_l \in [0,m]^w} \sigma(k(\boldsymbol{x}_l - \boldsymbol{p}) \cdot \boldsymbol{n}) \leq m + m\delta. \tag{14}$$

Thus, layer l maps $[0, m]^w$ into $[0, m(1 + \delta)]^w$ permitting some growth specified by δ. Assuming the input into the first hidden layer is contained in $[0, 1]^w$, initializing the hidden layers with $\delta = \frac{1}{L}$ leads to a network that maps

$$[0,1]^d \xrightarrow{\text{into}} [0,1]^w \xrightarrow{\text{into}} \left[0, 1 + \frac{1}{L}\right]^w \xrightarrow{\text{into}} \left[0, \left(1 + \frac{1}{L}\right)^2\right]^w \xrightarrow{\text{into}} \ldots \xrightarrow{\text{into}} \left[0, \left(1 + \frac{1}{L}\right)^{L-1}\right]^w. \tag{15}$$

This implies the final output of the hidden layer is contained in the box $[0, e]^w$; in other words, the values of each basis function are contained in $[0, e]$. Thus, we use the initialization with parameters

$$\delta = 1 \text{ and } m = 1 \text{ for } l = 1; \quad \delta = \frac{1}{L} \text{ and } m = \left(1 + \frac{1}{L}\right)^{l-1} \text{ for } l > 1. \tag{16}$$

An interesting observation regarding the ResNet initialization is its connection to the recently developed ODE based neural network architectures of Haber and Ruthotto (2017) and Chen et al. (2018). In those cases, a time step size scales the activation function that roughly speaking goes as $1/L$ where L is the number of steps. This ensures that the growth of the network features is a function of the length of the time interval (assuming bounded weights and biases). This is identical to what the analysis above shows for the initialization technique. An important difference; however, is that the ODE architectures retain the scaling through out the training process.

We compare the use of the box initialization for a residual neural network with hidden layer width 32 against the He initialization in Fig 7 for approximating $\sin(2\pi x)$ on $[0, 1]$. We average over 16 independent runs. The box initialized basis is again richer than the He basis and yields an initial LS loss consistently 4 orders of magnitude lower. The loss during training exhibits similar improvements over the He basis. At 128 layers, it is now the He basis which fails to train.

The advantages of the box initialization over the He initialization can be illustrated for a width-2 network by again studying the image P of the unit square $[0, 1]^2$ under both initialization in Fig. 8. Note that the image of the square never collapses to a point due to the ResNet architecture, regardless of initialization. Hence, the initialized basis will not consist of constant functions. This is a new interpretation of the stability provided by residual neural networks; for other perspectives, see Hanin and Rolnick (2018), Haber and Ruthotto (2017), and He et al. (2016a). Nevertheless, both the Glorot and the He initialization exhibit different pathologies in the ResNet case as depth increases: blow-up of the basis function magnitudes and convergence of the image P to lines through the origin. The latter property implies linearly dependent basis functions $\phi_1 = C\phi_2$, again resulting in a decreased expressive power of the initialized basis. All of these properties are illustrated in the basis function plots in Fig 16 in Appendix C. The ResNet box initialization, however, exhibits

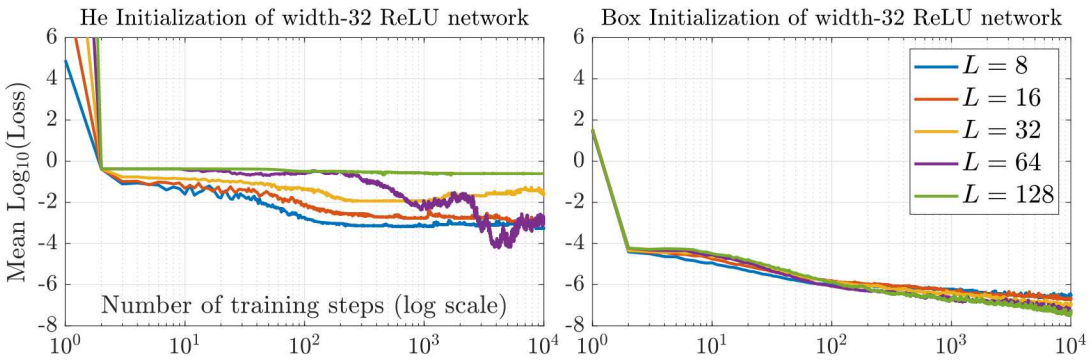

Figure 7: Mean of $\log_{10}(\text{Loss})$ over 16 training runs of residual width-32 ReLU network with $L = 8, 16, 32, 64$ and 128 hidden layers and training rate $2^{-(k+3)}$ for the He *(left)* and Box *(right)* initializations.

both the boundedness of P proven above and a remarkable preservation of the area of P as depth increases. We have not yet found an explanation for the latter property, but these results explain the benefits of the box-initializion for deep networks observed in Fig. 7.

We observe similar properties of the ResNet Box initialization in higher dimensions as well. In Fig. 8 we examine the eigenvalues of the covariance of the image of a set of input points sampled from $\mathcal{U}[0,1]^w$ through networks of increasing depth. We find that for the Glorot and He initializations, the ratio between the smallest and largest eigenvalues quickly become zero with increasing depth. This suggests that one basis function becomes linearly dependent upon the others with only a few layers. Worse, the ratio between the second largest and the largest eigenvalues eventually becomes zero, suggesting that the basis functions all become linearly dependent. In contrast, neither ratio tends toward zero for the Box initialization, indicating that the basis functions remain independent, even for very deep networks.

5. Applications

5.1. One-dimensional regression

In this section, we compare the behavior of the Glorot, He, and Box initializations for regression. We first consider regression on the discontinuous function,

$$u_1(x) = \begin{cases} x & 0 \le x < 0.5 \\ 1 - \frac{3}{4}x^2 & 0.5 \le x \le 1 \end{cases}. \tag{17}$$

With a network width of 2, the three initializations, both Plain and ResNet architectures, and varying depths, we use the LSGD method to fit u_1. Our results are shown in Figure 9 using an ensemble of initial random seeds for each initialization, architecture, and depth. Due to the narrow width of these network, only deep networks are capable of providing good approximations to u_1. However, we find that the Glorot and He initializations fail to find good fits to u_1, particularly for deep networks and regardless of the architecture used. The Box initialization also results in a poor fit, but only for the Plain architecture.

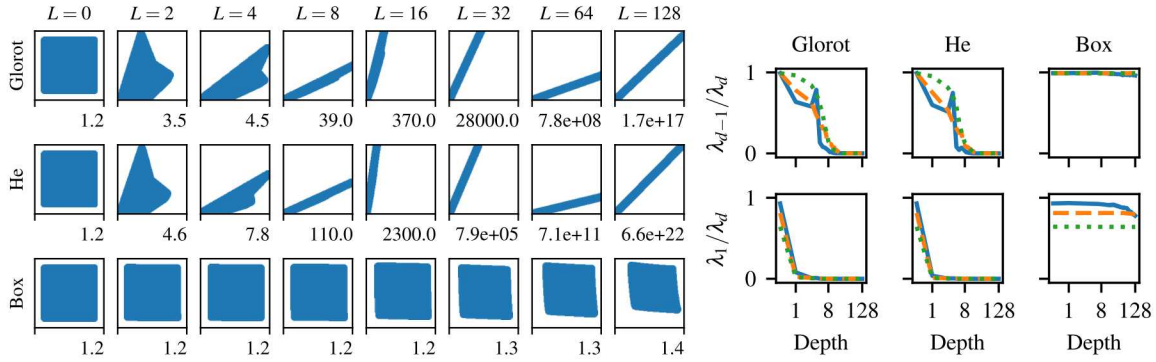

Figure 8: (**Left Subfigure:**) Images of the unit square $[0,1]^2$ under L initialized hidden layers of ResNets for Glorot (*top*), He (*center*) and Box (*bottom*) initializations. Values are presented on the square $[-0.2, H]^2$, where H is denoted to the bottom-right of each image. Collapse to a line through the origin corresponds to linearly dependent basis functions (i.e., $\phi_1 = C\phi_2$) (**Right subfigure:**) Ratio of the second largest to largest eigenvalue (*top*) and smallest to largest (*bottom*) of the covariance of the image of samples from $\mathcal{U}[0,1]^d$. Results are shown for the ResNet architectures using dimensions $w = d = 8$ (——), $w = 32$ (– – –), and $w = 128$ (·····).

Our observations in Figure 5 and 8 suggest that the combination of initialization and architecture can lead to a starting condition in which the span of the basis functions is limited. The results in Figure 9, related to problem 17, indicate that it is difficult to escape from this poor initial starting condition to a good fit. However, the Box initialization for the ResNet does not suffer from this lack of initial expressivity in the basis functions, and we are able to observe improvements increasing the depth of the network.

We next apply the Box initialization for ResNet to regress both u_1 and a smooth function, $u_2(x) = \sin 2\pi x$ for varying widths and depths. We observe first order convergence for the smooth function with respect to both width and depth, but only realize convergence with respect to width for the discontinuous functions (Figure 10).

5.2. Multi-function Regression

The regression problem described above learns the basis for a single function. In this section we modify the loss function so that the basis is defined to approximate a set of N functions:

$$\underset{\boldsymbol{\xi}^{\mathrm{L}}}{\operatorname{argmin}} \sum_{n=1}^{N} \left\| u_n - \sum_i \boldsymbol{\xi}^{\mathrm{L}}_{n,i} \Phi_i(\boldsymbol{x}, \boldsymbol{\xi}^{\mathrm{H}}) \right\|^2_{\ell_2(\mathcal{X})}. \tag{18}$$

Here the target functions are denoted u_n, and each has a corresponding set of linear coefficients $\boldsymbol{\xi}^{\mathrm{L}}_{n,\cdot}$. The basis functions are defined by a single set of nonlinear weights $\boldsymbol{\xi}^{\mathrm{H}}$, that define the output of a neural network as in single function regression described by Equation 2.

Our interest in multi-function regression lies in the fact that the adaptive basis representation of a DNN (3) exposes the problem (2) as seeking a best w-term approximation to u in the $\ell_2(\mathcal{X})$

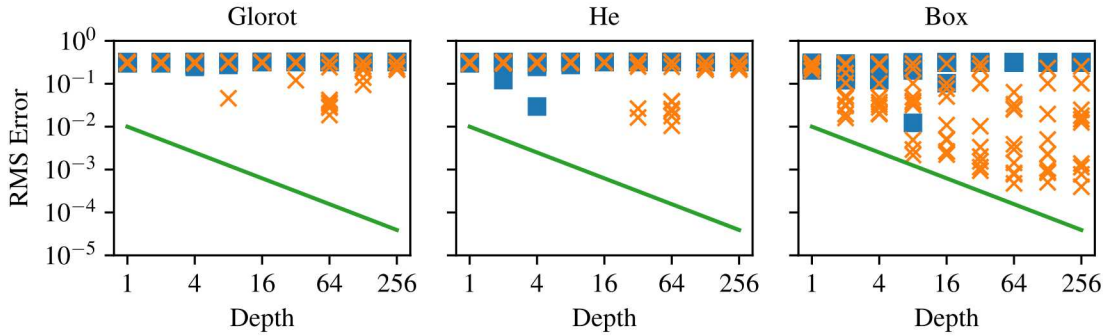

Figure 9: RMS Error for 1D piecewise polynomial polynomial regression (Equation 17) using the three initializations with Plain (■) and ResNet (×) architectures, respectively. Each symbol corresponds to the loss achieved using a different random seed for initialization. The green line (—) indicates first order convergence with respect to depth. Setting: ReLU activation function, network width = 2, learning rate = 0.005.

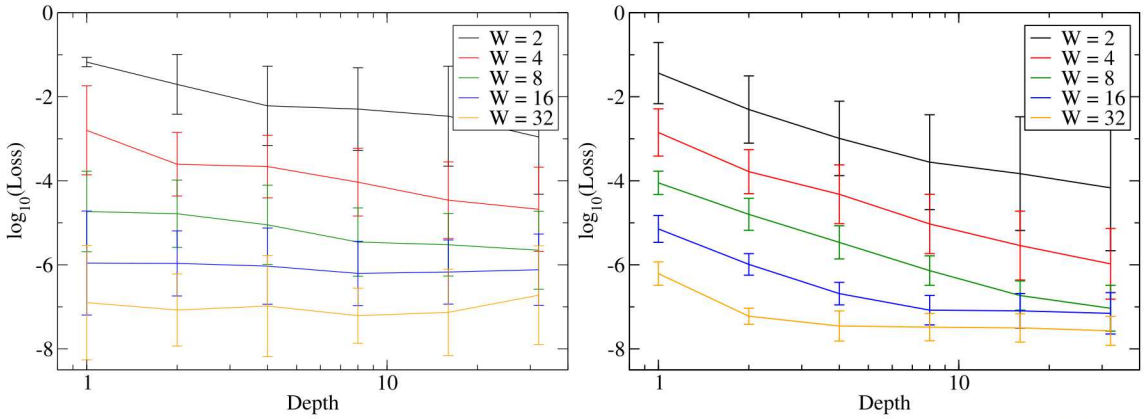

Figure 10: Convergence studies of regression with respect to width and depth on Equation 17 (*left*) and $u = \sin 2\pi x$ (*right*). Setting: ReLU activation function, ResNet architecture.

norm. This is a form of nonlinear approximation that includes, e.g., wavelet and variable-order spline approximation (Cohen et al., 2009; DeVore, 1998). Here, the terms in the approximation Φ_i, $i = 1, ..., w$ belong to the class of depth $L - 1$ DNNs with input dimension d, output dimension 1, and nonlinear in the final layer; see Fig. 1. The multi-function regression problem (18) therefore appears closely related to nonlinear w-widths in approximation theory (DeVore et al., 1989), and has potential for a reduced order modeling strategy (Hesthaven et al., 2016) in which subspaces are found as the span of $\{\Phi_i\}_{i=1}^{w}$ to minimize a loss function of the form (18) given a large collection of data $\{u_n\}$. While the benchmarks considered below are considerably simpler than such an application, this represents a promising direction for future work.

A multi-regression problem is solved targeting the Legendre polynomials in $L^2([0, 1])$, normalized to ensure equal weighting in the loss. The Legendre polynomials were chosen because of the range of structure in the set of polynomials. Note that the algorithm described above has not been

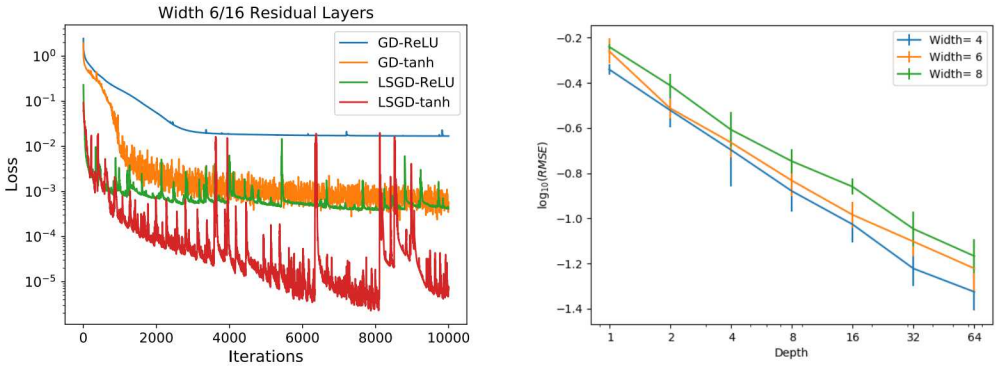

Figure 11: The left image shows the convergence of the loss of several multi-function regression problems with 6 Legendre polynomials. Networks with 16 residual layers and width 6 using ReLU and $\tanh$ activation functions are used. Clearly, the LSGD training algorithm achieves smaller losses and converges more rapidly than GD. The right image shows the convergence as a function of depth of the RMS error. Note that as the width growths, so do the number of Legendre polynomials used in the objective function.

modified to take advantage of their orthonormality. The left image in Fig. 11 shows the convergence of networks with 16 residual layers of width 6 trained to match 6 Legendre polynomials is studied (a one-to-one relationship between width and target functions). Here, the mean loss over 10 repeated simulations is plotted as a function of iteration. The LSGD and GD training algorithms are compared. From the figure, LSGD reaches a smaller magnitude loss in fewer iterations than the equivalent network trained with GD. Furthermore, the usage of $\tanh$ leads to a smaller loss than with ReLU, thus better representing the set of Legendre polynomials. This is attributed to the broader support and greater smoothness of $\tanh$.

For the right image in 11 we use a ReLU ResNet with width w to fit a space of Legendre polynomials of dimension w. For each realization we compute the error as the minimum over all iterations of the maximum RMS errors over the target polynomials and we then plot the mean RMS error over all the realizations. The learning rate for these simulations is set at 0.0005. The image demonstrates more accuracy is achieved as a function of depth.

5.3. Physics-informed neural networks

We consider now a physics-informed neural network (PINN) solution to the linear transport equation $\partial_t u(x,t)+a(x,t)\,\partial_x u(x,t) = 0$ on the unit space-time domain $(x,t) \in [0,1]^2$, with initial condition $u(x,t=0) = u_0(x)$ and homogeneous Dirichlet boundary data $u(x=0,t) = 0$. The loss function considered here is

$$\begin{aligned} \mathcal{J} &= \epsilon\mathcal{J}_1 + \mathcal{J}_2 + \mathcal{J}_3, \qquad \mathcal{J}_1 = \frac{1}{N_1}\sum_{i\in\mathcal{X}_1} |\partial_t \mathcal{NN}_i + \partial_x a(x,t)\mathcal{NN}_i|^2, \\ \mathcal{J}_2 &= \frac{1}{N_2}\sum_{i\in\mathcal{X}_2} |\mathcal{NN}_i(x,0) - u_0|^2, \qquad \mathcal{J}_3 = \frac{1}{N_3}\sum_{i\in\mathcal{X}_3} |\mathcal{NN}_i(0,t)|^2 \end{aligned} \tag{19}$$

where $\mathcal{X}_1, \mathcal{X}_2$ and $\mathcal{X}_3$ are Cartesian point clouds with spacing Δx on the interior, left and bottom boundaries, respectively. We note that the loss function is typically further augmented with a term to match given data (see e.g. Raissi et al. (2019)), and PINNs thus amount to regularizing traditional regression with the least-squares solution of a collocation scheme using the neural network as basis. For all results we will use ResNets and consider as initial condition a tent function $u_0 \in C_0$.

It is an open question how to choose the parameter ϵ scaling the first term of the $\mathcal{J}$ so that the three competing loss functions have the same magnitude under refinement - in the literature this penalty parameter is tuned to a given architecture to demonstrate good agreement, but preventing a formal convergence study. Traditionally in a FEM penalty method, one would scale by a mesh diameter h so that each term in Equation 19 has consistent units, and comparable magnitude. In the current context, the adaptive basis has no inherent lengthscale, as the gradient of the basis may grow arbitrarily large as the hidden weights evolve and cut planes may approach each other.

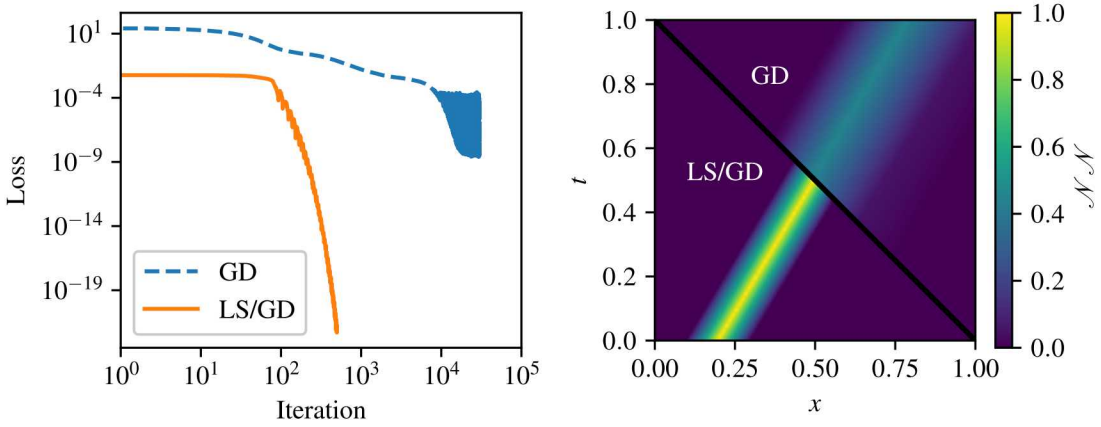

Figure 12: *Left:* PINNs solution for transport equation with constant velocity. Loss evolution over training for GD and LSGD. *Right:* Solution after 5000 iterations for GD and 500 iterations for LSGD. Setting: Box initialization, ReLU activation function, network width = 32, depth = 1, learning rate = 0.005.

We first consider in Figure 12 the case of constant velocity, $a(x,t) = 1$, with corresponding analytic solution $u(x,t) = u_0(x-t)$, and use a shallow one-layer ReLU network. For this case, the exact solution is in the range of the network for width ≥ 3, and at this point $\mathcal{J}_1 = \mathcal{J}_2 = \mathcal{J}_3 = 0$, rendering the choice of ϵ unimportant (we set $\epsilon = 1$). In this case we observe similar trends to the previous sections; the proposed LSGD training strategy converges to 10^{-15} in double precision with orders of magnitude fewer iterations than GD. From the evolution of the cut planes during training (see Appendix D), it is clear that the basis is adapting to the characteristics of the PDE.

We next consider nonconstant velocity, $a(x,t) = x$, with corresponding analytic solution $u(x,t) = u_0(x\exp(-t))$ (Figure 13). In this case we must fix ϵ independent of the neural network size to realize convergence, and we hypothesize $\epsilon = W^{-\alpha}$. Solutions for $\alpha \in \{0, \frac{1}{2}, 1, \frac{3}{2}, 2\}$ reveal $O(W^{\frac{1}{2}})$ convergence for $\alpha = \frac{1}{2}$. Following the FEM interpretation of shallow networks (He et al. (2018)), we interpret $h \sim N^{-\frac{1}{d}}$, and selecting $\epsilon = W^{-\frac{1}{2}}$ corresponds to non-dimensionalizing the loss, allowing a realizatiion of first-order convergence with respect to h. To consider the effect

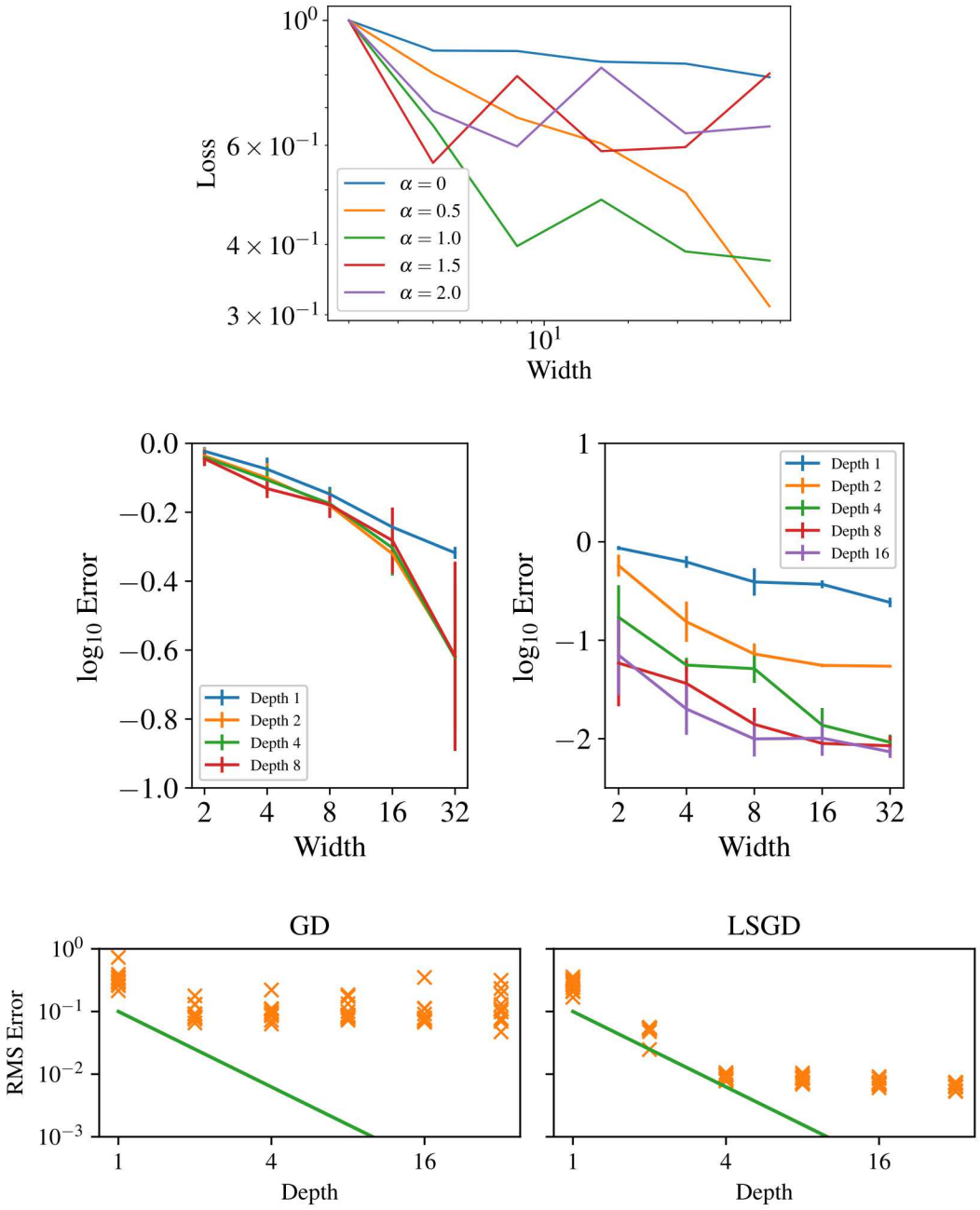

Figure 13: RMS error for ResNet PINNs with Box initialization for the nonconstant velocity case. *Top:* Convergence of ReLU-PINNs solutions with respect to penalty scaling $\epsilon \sim W^{-\alpha}$. *Middle:* Convergence of ReLU (*left*, learning rate 0.001) and tanh (*right*, learning rate 0.01) using $\alpha = -\frac{1}{2}$. *Bottom:* Comparison of between GD (left) and LSGD (right) training for tanh activation functions, learing rate 0.01, width 32 and 5000 epochs. The x's indicate the errors as a function of the number of layers for different realizations of the Box initialization. The line indicates second order convergence with respect to depth.

of depth we repeat the previous experiment for increasingly deep ReLU and tanh networks. Finally, to gauge the effectiveness of our training strategy, we compare using GD only vs LSGD.

While a thorough study of PINNs is beyond the scope of this paper, we conclude that the combination of LSGD, Box initialization, and choice of ϵ provides substantial performance gains relative to traditional GD, and we conclude that depth plays an important role in the convergence of PINNs.

6. Conclusions

Motivated by recent theoretical advances in the approximation theory of DNNs, this work takes an adaptive basis viewpoint of neural network training and initialization. This perspective naturally leads to a hybrid least squares/gradient descent training algorithm. We demonstrate that this approach leads to accelerated training for regression, multi-function regression and physics-informed neural networks, in the context of both ReLU and $\tanh$ activation functions. In a novel development, we proposed a new "box initialization" procedure inspired by the basis viewpoint that dramatically enhances the training of deep ReLU networks. As part of this we analyzed a potential failure mode for certain initializations that leads to a highly linearly dependent initial basis and demonstrated this failure for the Glorot and He initializations that are commonly used to initialize ReLU networks. For ResNets, we showed how the box initialization leads to a significantly improved basis, ultimately leading to more efficient training than the He initialization. Finally, using the combination of both Box initialization and LSGD training, we demonstrate in several scenarios the ability for neural networks to achieve relatively robust convergence as a function of both width and depth, for both single- and multi-function regression problems and PDE applications using physics-informed neural networks.

That machine learning algorithms can be understood as providing an underlying adaptive basis from data is a viewpoint that permeates many areas of deep learning (Murphy, 2012; He et al., 2018; Fokina and Oseledets, 2019; Wang et al., 2019). We believe this viewpoint is amenable to numerical analysis. The techniques developed here, in addition to improving the training of neural networks, demonstrate how an adaptive basis perspective can be used to attack critical issues hindering the robustness of machine learning. Taking a numerical analysis viewpoint has shed new light on the issues confronting neural network training and has provided intuition regarding the use of physics-informed neural networks to solve PDEs. We believe that additional advances are possible when considering the numerical implications of choices made in machine learning. Our work aims to strengthen the numerical properties of existing ML approaches and also provide a mathematical foundation in response to the need suggested in Baker et al. (2019) to obtain rigorous results for use in scientific machine learning.

Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

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

Nathan Baker, Frank Alexander, Timo Bremer, Aric Hagberg, Yannis Kevrekidis, Habib Najm, Manish Parashar, Abani Patra, James Sethian, Stefan Wild, et al. Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence. Technical report, USDOE Office of Science (SC), Washington, DC (United States), 2019.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

Elliott Ward Cheney and William Allan Light. *A course in approximation theory*, volume 101. American Mathematical Soc., 2009.

Albert Cohen, Wolfgang Dahmen, and Ronald DeVore. Compressed sensing and best k-term approximation. *Journal of the American mathematical society*, 22(1):211–231, 2009.

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

Ronald A DeVore. Nonlinear Approximation. *Acta numerica*, 7:51–150, 1998.

Ronald A DeVore, Ralph Howard, and Charles Micchelli. Optimal nonlinear approximation. *Manuscripta mathematica*, 63(4):469–478, 1989.

Daria Fokina and Ivan Oseledets. Growing axons: greedy learning of neural networks with application to function approximation. *arXiv preprint arXiv:1910.12686*, 2019.

C William Gear, Tasso J Kaper, Ioannis G Kevrekidis, and Antonios Zagaris. Projecting to a slow manifold: Singularly perturbed systems and legacy codes. *SIAM Journal on Applied Dynamical Systems*, 4(3):711–732, 2005.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 9:249–256, 13–15 May 2010.

G. H. Golub and V. Pereyra. The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate. *SIAM Journal on Numerical Analysis*, 10(2):413–432, 1973. doi: 10.1137/0710036. URL https://doi.org/10.1137/0710036.

Gene Golub and Victor Pereyra. Separable nonlinear least squares: the variable projection method and its applications. *Inverse Problems*, 19(2):R1–R26, feb 2003. doi: 10.1088/0266-5611/19/2/201. URL https://doi.org/10.1088%2F0266-5611%2F19%2F2%2F201.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 2017.

Boris Hanin. Universal function approximation by deep neural nets with bounded width and ReLU activations. *arXiv preprint arXiv:1708.02691*, 2017.

Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. In *Advances in Neural Information Processing Systems*, pages 571–581, 2018.

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

Juncai He, Lin Li, Jinchao Xu, and Chunyue Zheng. ReLU deep neural networks and linear finite elements. *arXiv preprint arXiv:1807.03973*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016b.

Jan S Hesthaven, Gianluigi Rozza, Benjamin Stamm, et al. *Certified reduced basis methods for parametrized partial differential equations*. Springer, 2016.

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

Lu Lu, Yanhui Su, and George Em Karniadakis. Collapse of deep and narrow neural nets. *arXiv preprint arXiv:1808.04947*, 2018.

Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.

Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

Joost AA Opschoor, Philipp Petersen, and Christoph Schwab. Deep ReLU networks and high-order finite element methods. *SAM, ETH Zürich*, 2019.

Adam Paszke et al. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.

V. Pereyra, G. Scherer, and F. Wong. Variable projections neural network training. *Mathematics and Computers in Simulation*, 73(1):231 – 243, 2006. ISSN 0378-4754. doi: https://doi.org/10.1016/j.matcom.2006.06.017. URL http://www.sciencedirect.com/science/article/pii/S0378475406001753. Applied and Computational Mathematics - Selected Papers of the Fifth PanAmerican Workshop - June 21-25, 2004, Tegucigalpa, Honduras.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

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

Ze Wang, Xiuyuan Cheng, Guillermo Sapiro, and Qiang Qiu. Stochastic conditional generative networks with basis decomposition. *arXiv preprint arXiv:1909.11286*, 2019.

Dmitry Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural Networks*, 94:103–114, 2017.

Appendix A. Properties and Performance of LSGD training

We provide here some supplemental results providing additional insight into the properties and advantages of LSGD and the computational cost relative to GD. We consider first a toy 2D problem in Figure 14, where we compare GD to LSGD for minimizing the loss $5x^2-6xy+5y^2$. This function is quadratic in both x and y, but to make an analogy to (6) we take the x-direction to correspond to the linear activation variable $\boldsymbol{\xi}^L$ and the y-direction to the hidden variable $\boldsymbol{\xi}^H$. We can visualize explicitly that LSGD realizes the global minimum in x at each step, and thus approaches the global minimum in (x, y) along a trajectory (x_k, y_k) where the the coordinate x_k always satisfies the least squares problem $x_k = LS(y_k)$ and is "optimal" for the coordinate y_k.

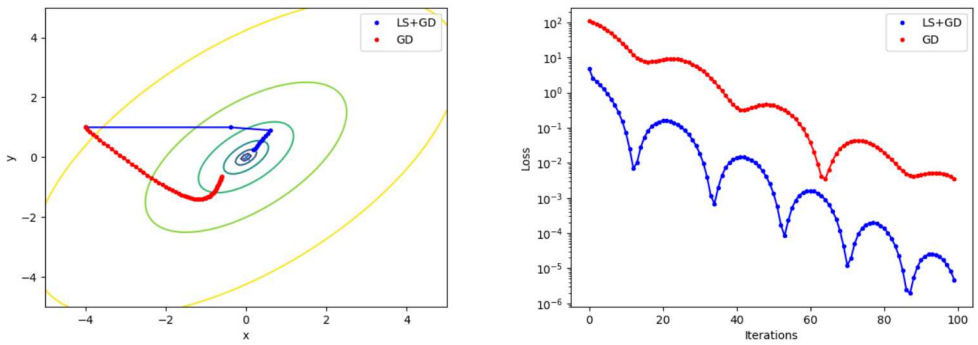

Figure 14: *Left:* Paths (x_k, y_k) taken by LSGD and GD to minimize the function $5x^2 - 6xy + 5y^2$, for learning rate of 0.1 and initial guess of $(x, y) = (-4, 1)$. Least squares optimization corresponds to finding the global minimum in coordinate x for fixed y at each step. Note that for LSGD, after the initial least squares solve, each plotted (x_k, y_k) is the result of gradient descent followed by least squares, rather than either of these steps individually. *Right:* LSGD achieves lower loss for the same number of iterations as GD.

We next provide in Figure 15 a sketch explaining how the LSGD approach may offer gains due to the fact that the dynamics of training are constrained to follow a manifold $\xi^L = LS(\xi^H)$ which necessarily contains all local minima. This figure also makes clear that the paths of GD training and LSGD training are not comparable globally. While training on this manifold may be more stable and lead to faster training, nothing precludes the existence of barriers along this manifold between an initial condition and a "good" local minima, which may be bypassed by GD training, as alluded to in Section 3 during the discussion Figure 2. Figure 15 also illustrates that LSGD can be viewed as type of coordinate descent method (Nocedal and Wright, 2006) in which steps in $\boldsymbol{\xi}^L$ are taken until a global minimum is reached before the variables are alternated, although we find a global minimum in one shot with a least squares solver. We also conjecture that there may be interesting connections with the dynamical system interpretation of training ResNets (Haber and Ruthotto, 2017; Chen et al., 2018) and work on fast/slow manifold dynamics (Gear et al., 2005).

The computational cost of including the least squares step for $\boldsymbol{\xi}^L$ after each gradient descent step in $\boldsymbol{\xi}^H$ depends heavily on the implementation details of both steps – for example, the specific least squares solver, whether GPU acceleration is used for gradient descent, memory access pattern

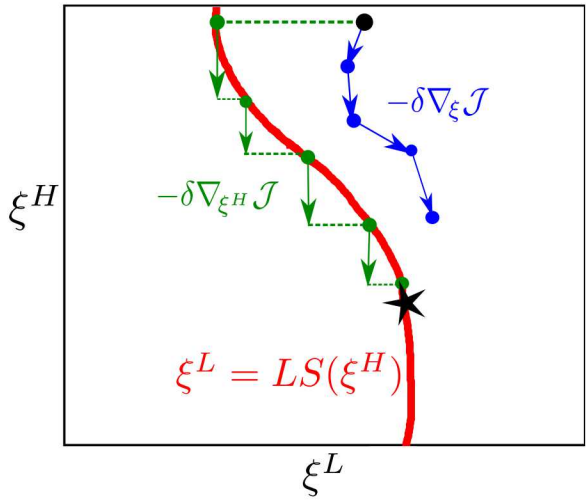

Figure 15: Artist's depiction of the LSGD algorithm. The black dot denotes the initial guess and the black star a minimum that the user wants train the neural network to. The red line represents the submanifold in parameter space $(\boldsymbol{\xi}^H, \boldsymbol{\xi}^L)$ for which $\boldsymbol{\xi}^L$ is a solution to the least squares problem for fixed $\boldsymbol{\xi}^H$. Note that because the local minimum illustrated by the black star must also be a global minimum in $\boldsymbol{\xi}^L$, it must lie on the manifold $\boldsymbol{\xi}^L = LS(\boldsymbol{\xi}^H)$ illustrated by the red line. Note also that $\nabla_{\boldsymbol{\xi}} \mathcal{J} = (\nabla_{\boldsymbol{\xi}^H} \mathcal{J}, \mathbf{0})$ on the manifold $\boldsymbol{\xi}^L = LS(\boldsymbol{\xi}^H)$. The blue curve represents a path of the GD method, while the rectilinear green curve a path of LSGD. An initial least squares solve (dashed green line) moves the neural network parameters to the submanifold $\boldsymbol{\xi}^L = LS(\boldsymbol{\xi}^H)$. In the LSGD algorithm, all gradients are computed from this manifold. Each step of gradient descent can move the parameters off this manifold, but the least squares solve that follows will project back onto the manifold.

used to overwrite linear layer variables, etc. Generally, the least squares solve only increases with the width of the network ($O(W^3)$ for dense solvers), whereas the gradient descent step increases with both the width and the depth, as indicated by Table 1.

	Depth (hidden lyers)				
Width		**4**	**16**	**64**	**256**
	4	1.67	1.60	1.43	1.36
	16	2.08	1.50	1.44	1.41
	64	1.68	1.37	1.37	1.33
	256	2.00	1.65	1.52	1.49

Table 1: Relative increase in wall time for 1000 iterations of LSGD vs pure GD (using the Adam optimizer) for a plain ReLU network, obtained using CPU implementation of Tensorflow on an Intel i7-8700K processor. For deeper networks, the increase is smaller since the computational cost of gradient descent, unlike that of least squares, grows with the depth and dominates the wall time.

Appendix B. Box Initialization Algorithms & Details

We provide in Algorithms 2 and 3 concise definitions of the box initialization algorithm for both plain and ResNet DNNs, respectively.

Algorithm 2 initializes layer l, and takes as input the dimension of the input to this layer, i.e., the width w_{l-1} of the previous layer, and the output dimension w_l. The main points of this algorithm are outlined in Section 4.1. Note that the random normal vector with uniform random direction $\hat{\boldsymbol{n}}$ is conveniently sampled (lines 3 – 4) by sampling from a isotropic multivariate normal of mean $\mathbf{0}$ and then normalizing. Once $\boldsymbol{p}$ and $\hat{\boldsymbol{n}}$ have sampled, a cut plane for a ReLU function is defined. To compute the scaling constant k in Section 4.1 such that the maximum of the ReLU function on $[0,1]^{w_{l-1}}$ is 1, it is necessary to locate the furthest corner $p_{\max} \in [0,1]^{w_{l-1}}$ in the direction of n from the cut plane of the ReLU function. To do this efficiently, we provide a closed form expression in line 5 for the corner of the box where the maximum occurs; this formula is proven in Lemma 1. The scaling factor k is then the inverse of the distance of the cut plane to this corner.

Algorithm 2 Plain Network Box Initialization

1: **function** PLAININIT(w_{l-1}, w_l)
2: $\quad \boldsymbol{p} \sim \mathcal{U}[0,1]^{w_{l-1}\times w_l}$ ▷ Sample w_l points in $[0,1]^{l-1}$
3: $\quad \hat{\boldsymbol{n}} \sim \mathcal{N}[0,1]^{w_{l-1}\times w_l}$ ▷ Sample from a normal distribution
4: $\quad n_{ij} = \hat{n}_{ij}/||\hat{\boldsymbol{n}}_j||_2^2$ ▷ w_l random unit vectors of dimension w_{l-1}
5: $\quad p_{\max} = \max(0, \operatorname{sign}(n_{ij}))$
6: $\quad k_j = 1/\sum_i ((p_{\max} - p_{ij})n_{ij})$
7: $\quad A^l_{ij} = k_j n_{ij}$
8: $\quad b^l_i = \sum_j k_j n_{ij} p_{ij}$
9: $\quad$ **return** $\boldsymbol{A}^l, \boldsymbol{b}^l$
10: **end function**

Lemma 1 *Let $\mathbb{H}$ be a $(d-1)$ dimensional hyperplane in $\mathbb{R}^d$ and let $\boldsymbol{n}$ be a normal to $\mathbb{H}$. Then, the maximum distance along direction $\boldsymbol{n}$ from $\mathbb{H}$ and any point in the unit hypercube $[0,1]^d$ is achieved on*

$$(\max(\operatorname{sgn}(n_1), 0), \max(\operatorname{sgn}(n_2), 0), \ldots, \max(\operatorname{sgn}(n_d), 0)) = \max(\operatorname{sgn}(\boldsymbol{n}), \mathbf{0}). \tag{20}$$

Proof Let us refer to the distance in question as the *directed distance*. The maximum directed distance is achieved on a corner of $[0,1]^d$, not necessarily unique. Let C^* be such a corner; the fact that C^* maximizes the directed distance from $\mathbb{H}$ is invariant under parallel transport of the hyperplane $\mathbb{H}$ in direction $\boldsymbol{n}$. Parallel transport $\mathbb{H}$ in direction $\boldsymbol{n}$ until the plane lands on C^*; then every point in $[0,1]^d$ is either on $\mathbb{H}$ or on the opposite of $\mathbb{H}$ from $\boldsymbol{n}$. Let $C^*_i \in \{0,1\}$ denote the coordinates of C^*. Make C^* the origin. Consider the d unit vectors $\boldsymbol{v}_i$ from C^* to the other d corners of $[0,1]^d$ along one of the axes. If the coordinate C^*_i had been 1, then $\boldsymbol{v}_i = -\boldsymbol{e}_i$; else if C^*_i had been 0, then $\boldsymbol{v}_i = \boldsymbol{e}_i$. Since the other corners are separated from $\boldsymbol{n}$ by $\mathbb{H}$, we have that

$$0 \geq \langle \boldsymbol{n}, \boldsymbol{v}_i \rangle = \begin{cases} n_i \text{ if } C^*_i = 0 \\ -n_i \text{ if } C^*_i = 1. \end{cases} \tag{21}$$

Hence if $0 > n_i$, then $C^*_i = 0$, while if $0 < n_i$, then $C^*_i = 1$. If $n_i = 0$, then both the corner C^* and that corner with the bit C^*_i flipped achieve the same directed distance from $\mathbb{H}$, so we may take $C^*_i = 0$. ■

Algorithm 3 follows from the outline in Section 4.2 in a similar way. We initialize the first hidden layer as a plain hidden layer using Algorithm 2 above (this is necessary for $d \neq w$). Then p_{max} is found by applying the same algorithm in Lemma 1 and scaling by the constant m to yield a corner in the box $[0, m]^{w_{l-1}}$. The scaling constant k now includes the factor $\frac{1}{L-1}$.

Algorithm 3 ResNet Box Initialization

function RESNETINIT(w_{l-1}, w_l, L)
  if $l == 1$ **then**
    return PlainInit(w_{l-1}, w_l)
  else
    $m = (1 + 1/(L-1))^l$
    $\boldsymbol{p} \sim \mathcal{U}[0, m]^{w_{l-1} \times w_l}$
    $\hat{\boldsymbol{n}} \sim \mathcal{N}[0, 1]^{w_{l-1} \times w_l}$
    $n_{ij} = \hat{n}_{ij} / ||\hat{\boldsymbol{n}}_j||_2^2$
    $p_{\max,ij} = m \max(0, \text{sign}(n_{ij}))$
    $k_j = 1 / \sum_i ((p_{\max,ij} - p_{ij}) n_{ij} (L-1))$
    $A_{ij}^l = k_j n_{ij}$
    $b_i^l = \sum_j k_j n_{ij} p_{ij}$
    return $\boldsymbol{A}^l, \boldsymbol{b}^l$
  end if
end function

Appendix C. Basis Function Plots

Figures 16 and 17 show the basis functions at initialization for Plain and ResNet architectures, respectively, in one-dimension.

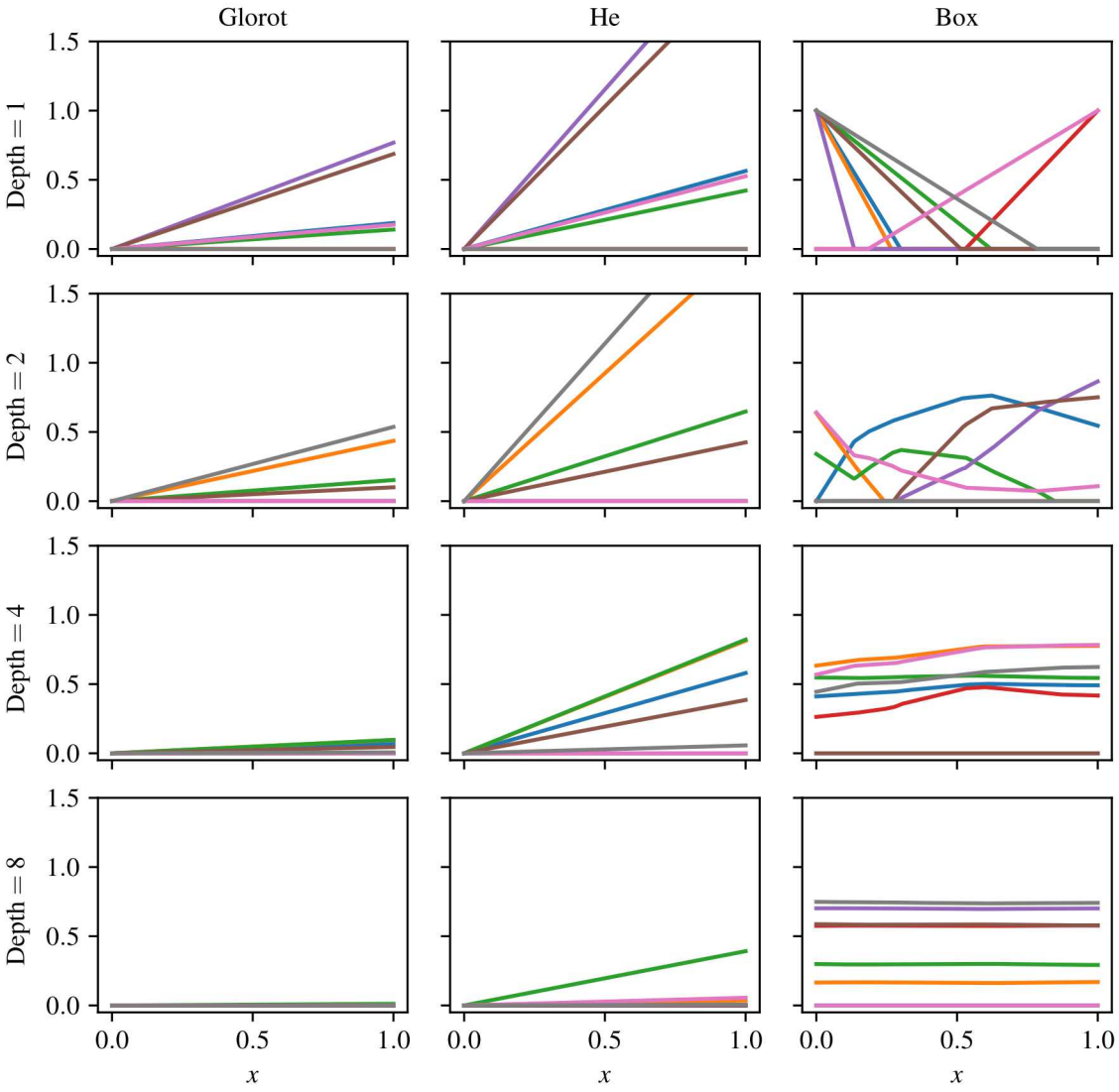

Figure 16: **Plain network basis** functions, Φ_i, after initialization for Glorot (*left column*), He (*middle column*), and Box (*right)* initializations with increasing depth and width 8. The input is one-dimensional. As discussed in Section 4.1, these figures illustrate that the Box initialized basis is richer in features than the He and Glorot initialized bases, but suffers from "collapse" to constant functions as depth increases (notice that this tendency is also visible for the He and Glorot basis functions).

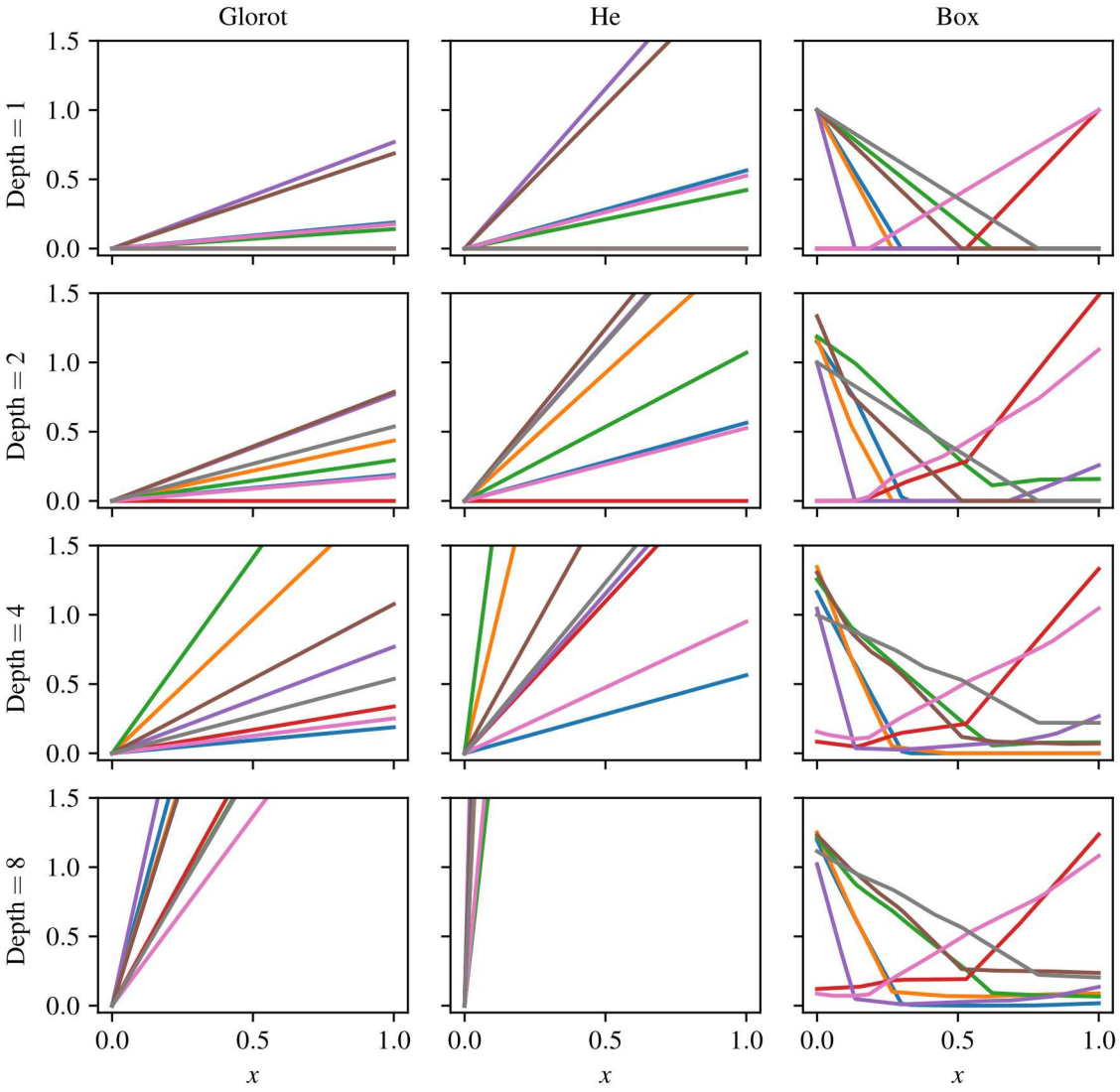

Figure 17: **ResNet basis** functions, Φ_i, after initialization for Glorot (*left column*), He (*middle column*), and Box (*right)* initializations with increasing depth and width 8. The input is one-dimensional. As discussed in Section 4.2, these figures illustrate that for ResNets, the box initialization consistently (with depth) produces basis functions with more features in the input domain than the He and Glorot initializations. The box initialized basis no longer suffers from the collapse to a constant basis as for plain architectures, nor does it exhibit the blow-up evident in the He basis, as depth increases.

Appendix D. PINNs snapshots

The images below depict the PINN solution to the constant coefficient transport equation at training step i with the cut planes of the ReLU basis superimposed as dashed red lines. These training snapshots demonstrate that the LSGD trained PINN (right column) finds the correct characteristics of the PDE with ReLU cutplanes far faster than the GD trained PINN (left column). Note that the i's are different in the two columns, and both networks have identical initializations.