entropy

Article

Precision Machine Learning

Eric J. Michaud [1,2,*], Ziming Liu [1,2] and Max Tegmark [1,2,3]

[1] Department of Physics, Massachusetts Institute of Technology (MIT), Cambridge, MA 02139, USA
[2] NSF AI Institute for AI and Fundamental Interactions, Cambridge, MA 02139, USA
[3] Center for Brains, Minds and Machines, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
* Correspondence: ericjm@mit.edu

Abstract: We explore unique considerations involved in fitting machine learning (ML) models to data with very high precision, as is often required for science applications. We empirically compare various function approximation methods and study how they *scale* with increasing parameters and data. We find that neural networks (NNs) can often outperform classical approximation methods on high-dimensional examples, by (we hypothesize) auto-discovering and exploiting modular structures therein. However, neural networks trained with common optimizers are less powerful for low-dimensional cases, which motivates us to study the unique properties of neural network loss landscapes and the corresponding optimization challenges that arise in the high precision regime. To address the optimization issue in low dimensions, we develop training tricks which enable us to train neural networks to extremely low loss, close to the limits allowed by numerical precision.

Keywords: machine learning; ML for science; scaling laws; optimization

check for updates

Citation: Michaud, E.J.; Liu, Z.; Tegmark, M. Precision Machine Learning. *Entropy* **2023**, *25*, 175. https://doi.org/10.3390/e25010175

Academic Editor: Friedhelm Schwenker

Received: 28 November 2022
Revised: 5 January 2023
Accepted: 12 January 2023
Published: 15 January 2023

1. Introduction

Most machine learning practitioners do not need to fit their data with much precision. When applying machine learning to traditional tasks in artificial intelligence such as those in computer vision or natural language processing, one typically does not desire to bring training loss all the way down to exactly zero, in part because training loss is just a proxy for some other performance measure like accuracy that one actually cares about, or because there is intrinsic uncertainty which makes perfect prediction impossible, e.g., for language modeling. Accordingly, to save memory and speed up computation, much work has gone into *reducing* the numerical precision used in models without sacrificing model performance much [1–3]. However, modern machine learning methods, and deep neural networks in particular, are now increasingly being applied to science problems, for which being able to fit models *very* precisely to (high-quality) data can be important [4–7]. Small absolute changes in loss can make a big difference, e.g., for the symbolic regression task of identifying an exact formula from data [8,9].

It is therefore timely to consider what, if any, unique considerations arise when attempting to fit ML models very precisely to data, a regime we call *Precision Machine Learning (PML)*. How does pursuit of precision affect choice of method? How does optimization change in the high-precision regime? Do otherwise-obscure properties of model expressivity or optimization come into focus when one cares a great deal about precision? In this paper, we explore these basic questions.

1.1. Problem Setting

We study regression in the setting of supervised learning, in particular the task of fitting functions $f : \mathbb{R}^d \to \mathbb{R}$ to a dataset of $D = \{(\vec{x}_i, y_i = f(\vec{x}_i)\}_{i=1}^{|D|}$. In this work, we mostly restrict our focus to functions f which are given by symbolic formulas. Such functions are appropriate for our purpose, of studying precision machine learning for science applications, since they (1) are ubiquitous in science, fundamental to many fields' descriptions of nature,

(2) are precise, not introducing any intrinsic noise in the data, making extreme precision possible, and (3) often have interesting structure such as *modularity* that sufficiently clever ML methods should be able to discover and exploit. We use a dataset of symbolic formulas from [8], collected from the Feynman Lectures on Physics [10].

Just how closely can we expect to fit models to data? When comparing a model prediction $f_\theta(\vec{x}_i)$ to a data point y_i, the smallest nonzero difference allowed is determined by the numerical precision used. IEEE 754 64-bit floats [11] have 52 mantissa bits, so if y_i and $f_\theta(\vec{x}_i)$ are of order unity, then the smallest nonzero difference between them is $\epsilon_0 = 2^{-52} \sim 10^{-16}$. We should not expect to achieve *relative RMSE loss* below 10^{-16}, where relative RMSE loss, on a dataset D, is:

$$\ell_{\text{rms}} \equiv \left(\frac{\sum_{i=1}^{|D|} |f_\theta(\vec{x}_i) - y_i|^2}{\sum_{i=1}^{|D|} y_i^2} \right)^{\frac{1}{2}} = \frac{|f_\theta(\vec{x}_i) - y_i|_{\text{rms}}}{y_{\text{rms}}}. \tag{1}$$

In practice, precision can be bottlenecked earlier by the computations performed within the model f_θ. The task of precision machine learning is to try to push the loss down many orders of magnitude, driving ℓ_{rms} as close as possible to the numerical noise floor ϵ_0.

1.2. Decomposition of Loss

One can similarly define *relative MSE loss* $\ell_{\text{mse}} \equiv \ell_{\text{rms}}^2$, as well as non-relative (standard) MSE loss $L_{\text{mse}}(f) = \frac{1}{|D|} \sum_{i=1}^{D} (f_\theta(\vec{x}_i) - y_i)^2$, and $L_{\text{rms}} = \sqrt{L_{\text{mse}}}$. Minimizing $\ell_{\text{rms}}, \ell_{\text{mse}}, L_{\text{rms}}, L_{\text{mse}}$ are equivalent up to numerical errors. Note that (relative) expected loss can be defined on a probability distribution $\mathbb{P}_{(\mathbb{R}^d, \mathbb{R})}$, like so:

$$\ell_{\text{rms}}^{\mathbb{P}} = \left(\frac{\mathbb{E}_{(\vec{x},y)\sim\mathbb{P}}\left[(f_\theta(\vec{x}) - y)^2\right]}{\mathbb{E}_{(\vec{x},y)\sim\mathbb{P}}\left[y^2\right]} \right)^{\frac{1}{2}}. \tag{2}$$

When we wish to emphasize the distinction between loss on a dataset D (empirical loss) and a distribution $\mathbb{P}$ (expected loss), we write ℓ^D and $\ell^{\mathbb{P}}$. In the spirit of [12], we find it useful to decompose sources of error into different sources, which we term *optimization error*, *sampling luck*, the *generalization gap*, and *architecture error*. A given model architecture parametrizes a set of expressible functions $\mathcal{H}$. One can define three functions of interest within $\mathcal{H}$:

$$f_{\mathbb{P}}^{\text{best}} \equiv \underset{f \in \mathcal{H}}{\operatorname{argmin}} \{\ell^{\mathbb{P}}(f)\}, \tag{3}$$

the best model on the expected loss $\ell^{\mathbb{P}}$,

$$f_D^{\text{best}} \equiv \underset{f \in \mathcal{H}}{\operatorname{argmin}} \{\ell^D(f)\}, \tag{4}$$

the best model on the empirical loss ℓ^D, and

$$f_D^{\text{used}} = \mathcal{A}(\mathcal{H}, D, L), \tag{5}$$

the model found by a given learning algorithm $\mathcal{A}$ which performs possibly imperfect optimization to minimize empirical loss L on D.

We can therefore decompose the empirical loss as follows:

$$\ell^D(f_D^{\text{used}}) = \underbrace{\left[\ell^D(f_D^{\text{used}}) - \ell^D(f_D^{\text{best}})\right]}_{\text{optimization error}} + \underbrace{\left[\ell^D(f_D^{\text{best}}) - \ell^{\mathbb{P}}(f_D^{\text{best}})\right]}_{\text{sampling luck}} + \underbrace{\left[\ell^{\mathbb{P}}(f_D^{\text{best}}) - \ell^{\mathbb{P}}(f_{\mathbb{P}}^{\text{best}})\right]}_{\text{generalization gap}} + \underbrace{\ell^{\mathbb{P}}(f_{\mathbb{P}}^{\text{best}})}_{\text{architecture error}}, \tag{6}$$

where all terms are positive except possibly the *sampling luck*, which is zero on average, has a standard deviation shrinking with data size $|D|$ according to the Poisson scaling $|D|^{-1/2}$, and will be ignored in the present paper. The generalization gap has been extensively studied in prior work, so this paper will focus exclusively on the optimization error and the architecture error.

To summarize: the architecture error is the best possible performance that a given architecture can achieve on the task, the generalization gap is the difference between the optimal performance on the training set D and the architecture error, and the optimization error is the error introduced by imperfect optimization—the difference between the error on the training set found by imperfect optimization and the optimal error on the training set. When comparing methods and studying their scaling, it useful to ask which of these error sources dominate. We will see that both architecture error and optimization error can be quite important in the high-precision regime, as we will elaborate on in Section 2, Section 3 and Section 4, respectively.

1.3. Importance of Scaling Exponents

In this work, one property that we focus on is how methods *scale* as we increase parameters or training data. This builds on a recent body of work on scaling laws in deep learning [13–21] which has found that, on many tasks, loss decreases predictably as a power-law in the number of model parameters and amount of training data. Attempting to understand this scaling behavior, [22,23] argue that in some regimes, cross-entropy and MSE loss should scale as $N^{-\alpha}$, where $\alpha \gtrsim 4/d$, N is the number of model parameters, and d is the *intrinsic dimensionality* of the *data manifold* of the task.

Consider the problem of approximating some analytic function $f : [0,1]^d \to \mathbb{R}$ with some function which is a piecewise n-degree polynomial. If one partitions a hypercube in $\mathbb{R}^d$ into regions of length ϵ and approximates f as a n-degree polynomial in each region (requiring $N = \mathcal{O}(1/\epsilon^d)$ parameters), absolute error in each region will be $\mathcal{O}(\epsilon^{n+1})$ (given by the degree-$(n+1)$ term in the Taylor expansion of f) and so absolute error scales as $N^{-\frac{n+1}{d}}$. If neural networks use ReLU activations, they are piecewise linear ($n = 1$) and so we may expect $\ell_{\text{rmse}}(N) \propto N^{-\frac{2}{d}}$. Sharma & Kaplan [22] argue that if neural networks map input data onto an intermediate representation, where representations lie on a manifold of intrinsic dimension $d^* < d$, and then perform a piecewise linear fit to a function on this manifold, then RMSE error should scale as N^{-2/d^*}.

If one desires very low loss, then the exponent α, the rate at which methods approach their best possible performance (The best possible performance can be determined either by precision limits or by noise intrinsic to the problem, such as intrinsic entropy of natural language) matters a great deal. Sharma & Kaplan [22] note that $4/d$ ($2/d$ for RMSE loss) is merely a lower-bound on the scaling rate—we consider ways that neural networks can improve on this bound. Understanding model scaling is key to understanding the feasibility of achieving high precision.

1.4. Organization

This paper is organized as follows: In Section 2 we discuss piecewise linear approximation methods, comparing ReLU networks with linear simplex interpolation. We discover that neural networks can sometimes substantially outperform linear simplex interpolation, and hypothesize that NNs do this by exploiting the modularity of the problem, which we call the *modularity hypothesis*. In Section 3 we discuss nonlinear methods, including neural networks with nonlinear activation functions, and find that optimization error, rather than approximation error, can be a major bottleneck to achieving high-precision fits. In Section 4 we discuss the optimization challenge of high-precision neural network training – how optimization difficulties can often make total error far worse than the limits of what architecture error allows. We attempt to develop optimization methods for overcoming these problems and describe their limitations, then conclude in Section 5.

2. Piecewise Linear Methods

We first consider approximation methods which provide a piecewise linear fit to data. We focus on two such methods: linear simplex interpolation and neural networks with ReLU activations.

To review, linear simplex interpolation works as follows: given our dataset of $|D|$ input-output pairs $\{(\vec{x}_i, y_i)\}_{i=1}^{|D|}$, linear simplex interpolation first computes a Delaunay triangulation from $\vec{x}_1, \ldots, \vec{x}_{|D|}$ in the input space $\mathbb{R}^d$, partitioning the space into a collection of d-simplices, each with $d+1$ vertices, whose union is the convex hull of the input points. Since $d+1$ points determine a linear (affine) function $\mathbb{R}^d \to \mathbb{R}$, the function f can be approximated within each d-simplex as the unique linear function given by the value of the function f at the vertices. This gives a piecewise linear function on the convex hull of the training points. Linear simplex interpolation needs to store $N = |D|(d+1)$ parameters: $|D|d$ values for the vertices $\vec{x}_i$, and $|D|$ values for the corresponding function values y_i.

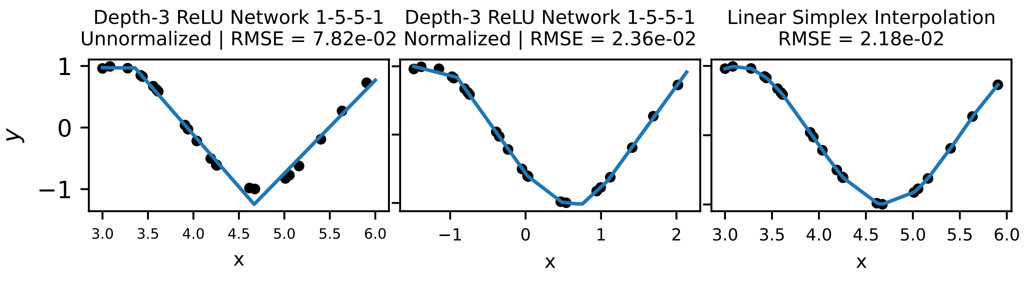
Neural networks with ReLU activations also give a piecewise linear fit f_θ. We consider only fully-connected feedforward networks, a.k.a. multilayer perceptrons (MLPs). Such networks consist of a sequence of alternating affine transformations $T : \vec{x} \mapsto W\vec{x} + b$ and element-wise nonlinearities $\sigma(\vec{x})_i = \sigma(\vec{x}_i)$ for an activation function $\sigma : \mathbb{R} \to \mathbb{R}$:

$$f_\theta = T_{k+1} \circ \sigma \circ T_k \circ \cdots \circ T_2 \circ \sigma \circ T_1.$$

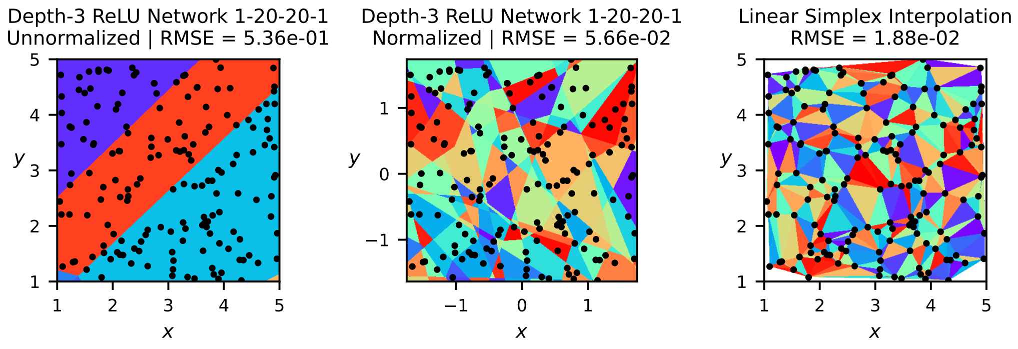
Following [24], we define the depth of the network as the number of affine transformations in the network, which is one greater than the number of hidden layers k. As shown in [24], any piecewise linear function on $\mathbb{R}^d$ can be represented by a sufficiently wide ReLU NN with at most $\lceil \log_2(d+1) \rceil + 1$ depth. Therefore, sufficiently wide and deep networks are able to exactly express functions given by linear simplex interpolation. A natural question then is: given the same amount of data and parameters, how do the two methods compare? We discover that simplex interpolation performs better on 1D and 2D problems, but that neural networks can outperform simplex interpolation on higher-dimensional problems. So although simplex interpolation and ReLU NNs both parametrize the same function class (piecewise linear functions), their performance can differ significantly in practice (Ref. [25] analyzed the expressivity of ReLU NNs in comparison with multivariate adaptive regression splines. Our work conducts a more extensive set of experiments to study how approximators *scale* in practice. We also restrict our focus to fitting physics functions, which have modular structure).

In our experiments, we use the implementation of simplex interpolation from SciPy [26]. When training neural networks, we use the Adam optimizer [27] with a learning rate of 10^{-3}, and train for 20k steps. We use a batch size of $\min(|D|, 10^4)$. While we report loss using RMSE, we train using MSE loss. Training points are sampled uniformly from intervals specified by the AI Feynman dataset [8] (typically $[1, 5] \subset \mathbb{R}$ for each input), but when training neural networks, we normalize the input points [28] so that they have zero mean and unit variance along each dimension. We estimate test loss on datasets of 30k samples (Project code can be found at https://github.com/ejmichaud/precision-ml, accessed on 11 January 2023).

In Section 1, we show for 1D and 2D problems the linear regions given both by simplex interpolation and by neural networks trained with a comparable number of parameters . For 2D problems, Figure 1 illustrates the importance of normalizing input data for ReLU networks. We see that there is a far higher density of linear regions around the data when input data is normalized, which leads to better performance. Neural networks, with the same number of parameters and trained with the same low-dimensional data, often have fewer linear regions than simplex interpolation.



(**a**)



(**b**)

Figure 1. In (**a**) (top), we show the solutions learned by a ReLU network and linear simplex interpolation on the 1D problem $y = \cos(2x)$. In (**b**) (bottom), we visualize linear regions for a ReLU network, trained on unnormalized data (**left**) and normalized data (**center**), as well as linear simplex interpolation (**right**) on the 2D problem $z = xy$. In general, we find that normalizing data to have zero mean and unit variance improves network performance, but that linear simplex interpolation outperforms neural networks on low-dimensional problems by better vertex placement.

In Figure 2, we show how the precision of linear simplex interpolation and neural networks scale empirically. Since simplex interpolation is a piecewise linear method, from the discussion in Section 1.3, we expect its RMSE error to scale as $N^{-2/d}$, and find that this indeed holds (Scaling as $N^{-2/d}$ only holds when the model is evaluated on points not too close to the boundary of the training set. At the boundary, simplices are sometimes quite large, leading to a poor approximation of the target function close to the boundary, large errors, and worse scaling. In our experiments, we therefore compute test error only for points at least 10% (of the width of the training set in each dimension) from the boundary of the training set). To provide a fair comparison with simplex interpolation when evaluating neural networks on a dataset of size D, we give it the same number of parameters $N = |D|(d+1)$. From Figure 2, we see that simplex interpolation outperforms neural networks on low dimensional problems but that neural networks do better on higher-dimensional problems.

For the 1D example in Figure 2 (top left), we know that the amount by which the neural networks under-perform simplex interpolation is entirely due to optimization error. This is because any 1D piecewise linear function $f(x)$ with m corners at $x_1, \ldots, x_m$ can trivially be written as a linear combination of m functions $\mathrm{ReLU}(x - x_i)$.

We hypothesize that the reason why neural networks are able to beat simplex interpolation when fitting functions given by symbolic equations is that neural networks learn to exploit the inherent modularity of these problems. We term this the *modularity hypothesis*. This property of the regression functions we study, which we call *modularity*, has been referred to by other authors as "compositional sparsity" [29,30] or the property of being "hierarchically local compositional" [31]. Such functions consist of many low-dimensional functions composed with each other. For instance, given three two-dimensional functions f_1, f_2, f_3, one can construct a modular four-dimensional function $g(x_1, x_2, x_3, x_4) = f_3(f_1(x_1, x_2), f_2(x_3, x_4))$.

Many natural functions have this structure, notably those we are interested in given by symbolic equations, which compose together binary functions $(+, -, \times, \div)$ and unary functions $(e^x, \ln(x)$, etc.). We are particularly interested in modular functions where the computation graph describing the function has low *maximum arity*—that the constituent functions composing it individually have low input dimension. For symbolic equations, the maximum arity is two.

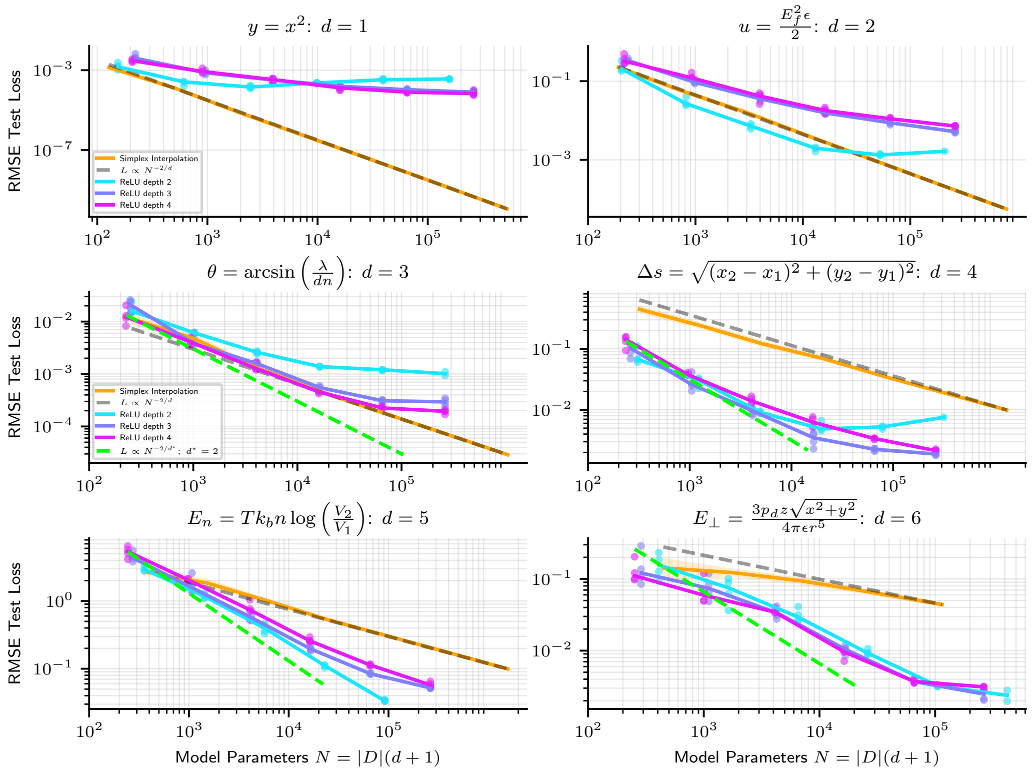


Figure 2. Scaling of linear simplex interpolation versus ReLU NNs. While simplex interpolation scales very predictably as $N^{-2/d}$, where d is the input dimension, we find that NNs sometimes scale better (at least in early regimes) as N^{-2/d^*}, where $d^* = 2$, on high dimensional problems.

On modular problems, it is a theoretical possibility [32,33] that sufficiently deep neural networks could scale as if the problem dimension was the maximum arity of the problem computation graph, rather than the input dimension. This is possible if networks learn a modular solution to the problem, approximating each node in the computation graph and internally composing together these elements according to the computation graph. The error of ReLU NNs should therefore be able to scale as N^{-2/d^*} where d^* is the maximum arity of the computation graph of the target function ($d^* = 2$ for symbolic equations). Scaling in this manner requires excess capacity to be allocated towards improving the approximation of all nodes in the graph, while continuing to compose them properly, rather than approximating a single function on a fixed data manifold.

We observe two tentative pieces of evidence for the modularity hypothesis as the reason why NNs beat linear simplex interpolation. First, we see from Figure 2, at least early in the scaling curves, that neural networks usually appear to scale not as $N^{-2/d}$, but rather as N^{-2/d^*} where $d^* = 2$, the maximum arity of the symbolic regression problems we train on. As an additional test, we train networks where we hard-code the modularity of the problem into the architecture, as depicted in Figure 3. Figure 3 indeed reveals how models for which we enforce the modularity of the problem perform and scale similarly to same-depth dense neural networks without modularity enforced. A modular architecture can be created from a dense one by forcing weight matrices to be block-diagonal (where we do not count off-diagonal entries towards the number of model parameters), but in practice we create modular architectures by creating a separate MLP for each node in the symbolic expression computation graph and connecting them together in accordance with the computation graph. See the diagrams in Figure 3 for an illustration of the modular architecture. In Figure 3, we plot modular and dense network performance against number of model parameters, but we also find that holding width constant, rather than number of parameters, modular networks still slightly outperform their dense counterparts. For instance, depth-6 width-100 modular networks outperform dense networks of the same width and depth, despite dense networks having $\approx$2.5$\times$ fewer parameters. Such "less is more" results are to be expected if the optimal architecture is in fact modular, in which case a fully connected architecture wastes resources training large numbers of parameters that should be zero.

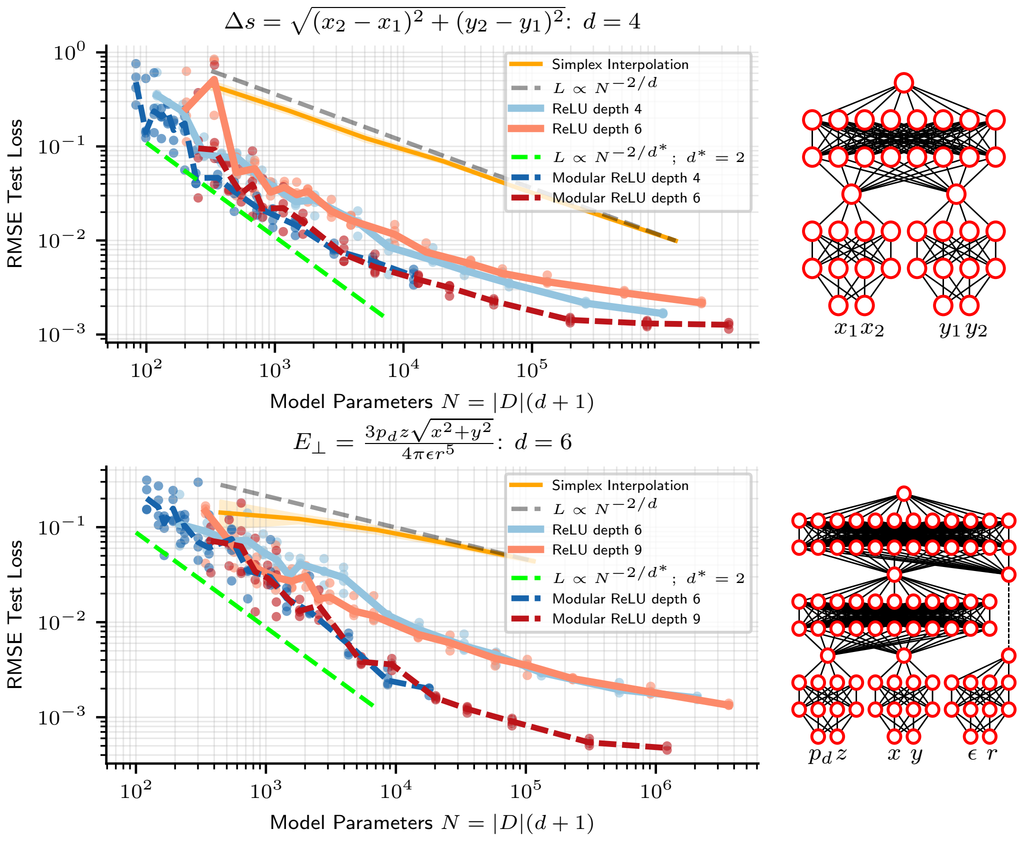


Figure 3. ReLU neural networks are seen to initially scale roughly as if they were modular. Networks with enforced modularity (dark blue and red, dashed line), with architecture depicted on the right, perform and scale similarly, though slightly better, than standard dense MLPs of the same depth (light blue and red).

Our observation that neural networks sometimes appear to scale as if the problem dimension was the maximum arity of the computational graph generalizes and improves upon neural scaling results from Sharma & Kaplan [22], which argued that the effective problem dimension d^* is the intrinsic dimension of a fixed data manifold. In the view from [22], excess capacity is allocated towards approximating a function on this fixed manifold with greater resolution, rather than allocating capacity towards many internal modules and *composing* them in an appropriate manner as we have discussed. It is widely believed in the ML community that the dimensionality of the data manifold or the power-law statistics of the data covariance matrix spectra sets the neural scaling exponent (see for example the excellent treatment by Sharma & Kaplan [22] and Bahri et al. [23]), however, our results show that scaling exponents can be better under the modularity hypothesis, where networks exploit modularity to scale in the maximum arity of a problem's computation graph.

3. Nonlinear Methods

We now turn our attention to approximation methods that are thoroughly nonlinear (as opposed to piecewise linear). As discussed in the introduction, methods approximating the target function f by a piecewise polynomial have a scaling exponent $\alpha = \frac{n+1}{d}$ where n is the degree of the polynomial.

In Figure 4, we plot the performance of approximation methods which are piecewise polynomial, for 1D, 2D and 3D problems. For 1D and 2D problems, we use splines of varying order. For 3D problems, we use the cubic spline interpolation method of [34]. We see empirically that these methods have scaling exponent $\alpha = (n+1)/d$. If the order of the spline interpolator is high enough, and the dimension low enough, we see that relative RMSE loss levels out at $\epsilon_0 \approx 10^{-16}$ at the precision limit. Unfortunately, a basic limitation of these methods is that they are limited to low-dimensional problems.

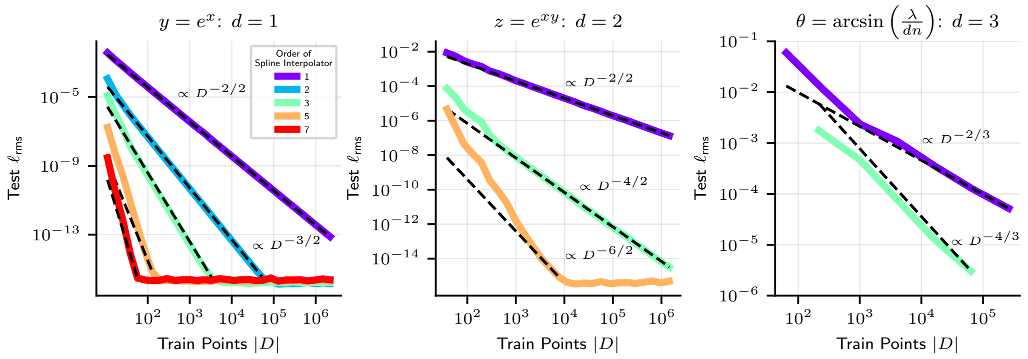


Figure 4. Interpolation methods, both linear and nonlinear, on 2D and 3D problems, seen to approximately scale as $D^{-(n+1)/d}$ where n is the order of the polynomial spline, d is the input dimension.

There are relatively few options for high-dimensional nonlinear interpolation (One method, which we have not tested, is radial basis function interpolation). A particularly interesting parametrization of nonlinear functions is given by neural networks with nonlinear activation functions (and not piecewise linear like ReLUs). In Figure 5, we show how neural networks with tanh activations scale in increasing width. We observe that on some problems, they do better than the ideal scaling achievable with linear methods (shown as a green dashed line). However, in our experiments, they can sometimes scale worse, perhaps the result of imperfect optimization. Also, we find that scaling is typically not nearly as clean as a power law as it was for ReLU networks.

For some problems, one can show theoretically that architecture error can be made arbitrarily low, and that the loss is due entirely to optimization error and the generalization gap. As shown in [35], a two-layer neural network with only four hidden units can perform multiplication between two real numbers, provided that a twice-differentiable activation function is used. See Figure 6b for a diagram of such a network, taken from [35]. Note that this network becomes more accurate in the limit that some of its parameters become very small and others become very large. This result, that small neural networks can express multiplication arbitrarily well, implies that neural network architecture error is effectively zero for some problems. However, actually *learning* this multiplication circuit in practice is challenging since it involves some network parameters *diverging* $\to \infty$ while others $\to 0$ in a precise ratio. This means that for some tasks, neural network performance is mainly limited not by architecture error, but by optimization error.

Indeed, on some problems, a failure to achieve high precision can be blamed entirely on the optimization error. In Figure 6a, we show neural network scaling on the equation $f(x_1, x_2, x_3, y_1, y_2, y_3) = x_1 y_1 + x_2 y_2 + x_3 y_3$. For this problem, a 2-layer network with 12 hidden units (implementing three multiplications in parallel, with their results added in the last layer) can achieve ≈0 architecture error. Yet we see a failure to get anywhere near that architecture error or the noise floor set by machine precision. Instead, as one scales up the network size on this task, we see that despite architecture error abruptly dropping to zero early on, the actually attained loss continues to scales down smoothly.

It is therefore important to analyze the problem of optimization for high precision, which we do in the next section.

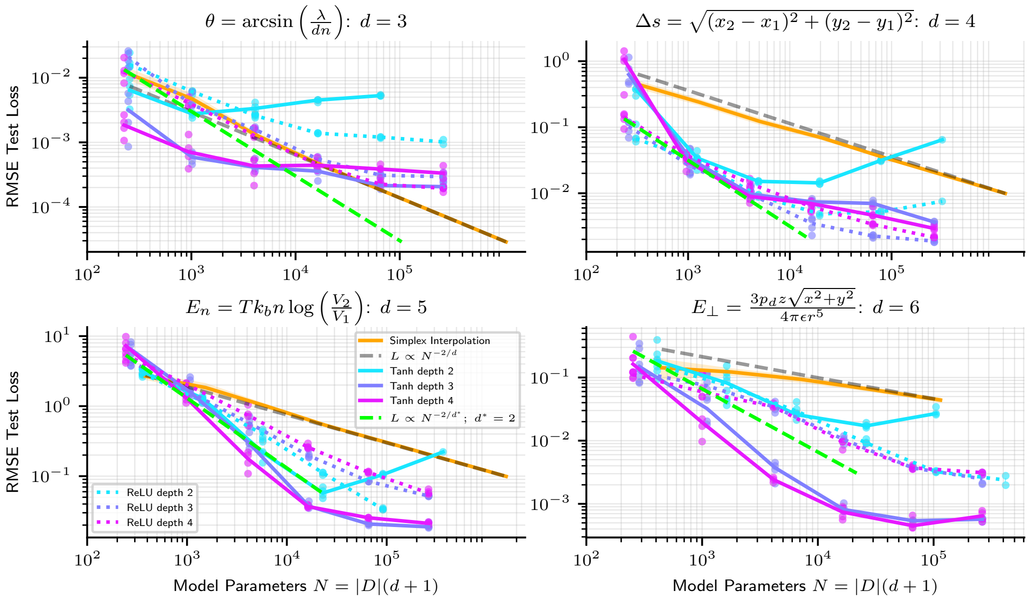
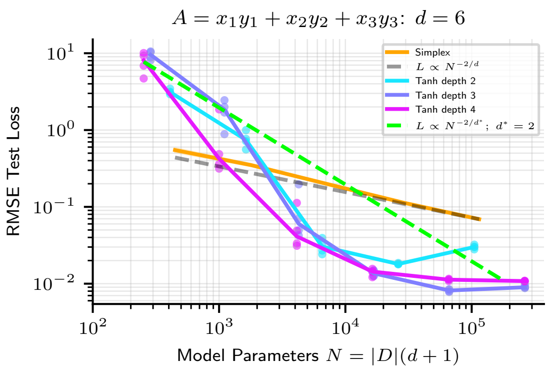
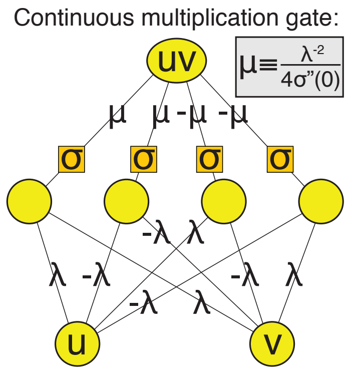


Figure 5. Scaling of linear simplex interpolation vs tanh NNs. We also plot ReLU NN performance as a dotted line for comparison. While simplex interpolation scales very predictably as $N^{-2/d}$, where d is the input dimension, tanh NN scaling is much messier. See Appendix C for a comparison of scaling curves with Adam vs. the BFGS optimizer.



(**a**)


Continuous multiplication gate:
uv
$\mu \equiv \frac{\lambda^{-2}}{4\sigma''(0)}$
μ
μ
-μ
-μ
σ
σ
σ
σ
-λ
λ
λ
-λ
-λ
λ
-λ
λ
u
v

(**b**)

Figure 6. (**a**) Scaling of neural networks on a target function which can be arbitrarily closely approximated by a network of finite width. (**b**) diagram from [35] showing how a four-neuron network can implement multiplication arbitrarily well. Therefore a depth-2 network of width at least 12 has an architecture error at the machine precision limit, yet optimization in practice does not discover solutions within at least 10 orders of magnitude of the precision limit.

4. Optimization

As seen above, when deep neural networks are trained with standard optimizers, they can produce significant optimization error, i.e., fail to find the best approximation. In this section, we discuss the difficulty of optimization in the high-precision regime and explore a few tricks for improving neural network training.

4.1. Properties of Loss Landscape

To understand the difficulty of optimizing in the high-precision regime, we attempt to understand the local geometry of the loss landscape at low loss. In particular, we compute the Hessian of the loss and study its eigenvalues. In Figure 7, we plot the spectrum of the

Hessian, along with the magnitude of the gradient projected onto each of the corresponding eigenvectors, at a point in the loss landscape found by training with the Adam optimizer for 30k steps in a teacher-student setup. The teacher is a depth-3, width-3 tanh MLP and the student is a depth-3, width-40 tanh MLP. In line with [36–38], we find that at low loss, the loss landscape has a top cluster of directions of high curvature (relatively large positive eigenvalues) and a bulk of directions of very low curvature. Furthermore, the gradient tends to point most strongly in directions of higher curvature, and has very little projection onto directions of low curvature magnitude.

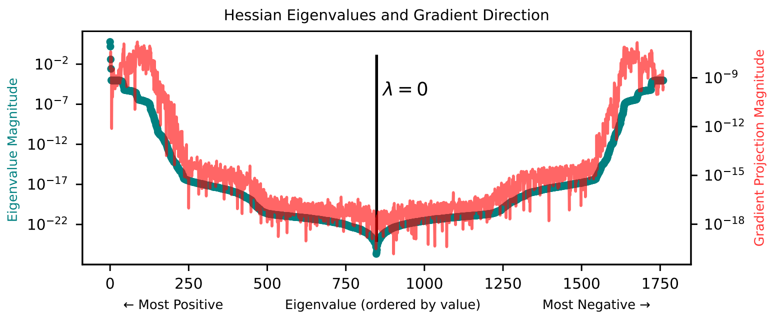


Figure 7. Eigenvalues (dark green) of the loss landscape Hessian (MSE loss) after training with the Adam optimizer, along with the magnitude of the gradient's projection onto each corresponding eigenvector (thin red line). We see a cluster of top eigenvalues and a bulk of near-zero eigenvalues. The gradient (thin jagged red curve) points mostly in directions of high-curvature. See Appendix D for a similar plot after training with BFGS rather than Adam.

The basic picture emerging from this analysis is that of a canyon, i.e., a very narrow, very long valley around a low-loss minimum. The valley has steep walls in high-curvature directions and a long basin in low-curvature directions. Further reducing loss in this environment requires either (1) taking very precisely-sized steps along high-curvature directions to find the exact middle of the canyon or (2) moving along the canyon in low-curvature directions instead, almost orthogonally to the gradient. In this landscape, typical first-order optimizers used in deep learning may struggle to do either of these things, except perhaps if learning rates are chosen extremely carefully.

4.2. Optimization Tricks for Reducing Optimization Error

How can we successfully optimize in such a poorly-conditioned, low-loss regime? We first find that switching from first-order optimizers like Adam to second-order optimizers like BFGS [39] can improve RMSE loss by multiple orders of magnitude. Second-order methods often both (1) employ line searches, and (2) search in directions not strongly aligned with the gradient, allowing optimization to progress within low-curvature subspaces. However, methods like BFGS are eventually bottlenecked by numerical precision limitations. To further lower loss, we tested the following two methods:

4.2.1. Low-Curvature Subspace Optimization

We find that by restricting our optimization to low-curvature subspaces, we can further decrease loss past the point where loss of precision prevented BFGS from taking further steps. Our method has a single hyperparameter τ. The method is as follows: let $g = \nabla_\theta \mathcal{L}$ be the gradient and H be the Hessian of the loss. Denote an eigenvector-eigenvalue pair of H by (e_i, λ_i). Instead of stepping in the direction $-g$, we instead compute $\hat{g} = \sum_{i:\lambda_i<\tau} e_i(e_i \cdot g)$. Essentially, we just project g onto the subspace spanned by eigenvalues of e_i such that $\lambda_i < \tau$. We then perform a line search to minimize loss along the direction $-\hat{g}$, and repeat the process. Note that this requires computing eigenvectors and eigenvalues for the whole Hessian H. See Appendix B for training curves with low-curvature subspace training with varying τ.

4.2.2. Boosting: Staged Training of Neural Networks

We also investigated techniques related to the common practice of "boosting" in ML [40–43]. We found the following method to work quite well in our setting. Instead of training a full network to fit the target f, one can train two networks $f^{(1)}_{\theta_1}, f^{(2)}_{\theta_2}$ sequentially: first train $f^{(1)}_{\theta_1}$ to fit f, then train $f^{(2)}_{\theta_2}$ to fit the residual $\frac{f - f^{(1)}_{\theta_1}}{c}$, where $c \ll 1$ normalizes the residual to be of order unity. One can then combine the two networks into a single model $f(x) \approx f^{(1)}_{\theta_1}(x) + c f^{(2)}_{\theta_2}(x)$. If networks $f^{(1)}_{\theta_1}, f^{(2)}_{\theta_2}$ have widths w_1, w_2 respectively, then they can be combined into one network of width w_1, w_2, with block-diagonal weight matrices, and where the parameters of the last layer of $f^{(2)}_{\theta_2}$ are scaled down by c. While further boosting steps can be performed, we found that there are quickly diminishing returns to training on successive residuals beyond that of the first network.

We find that, for low-dimensional problems, we can achieve substantially lower loss with these techniques. We use the following setup: we train width-40 depth-3 tanh MLPs to fit single-variable polynomials with the BFGS optimizer on MSE loss. The SciPy [26] BFGS implementation achieves 10^{-7} RMSE loss before precision loss prevents further iterations. Subsequently using low-curvature subspace training with a threshold $\tau = 10^{-16}$ can further lower RMSE loss a factor of over $2\times$. On similar low-dimensional problems, as shown in Figure 8, applying boosting, training a second network with BFGS on the residual of the first can lower RMSE loss further by five to six orders of magnitude. In Figure 8, we compare training runs with these tricks to runs with the Adam optimizer for a variety of learning rates. For our Adam training runs, we use width-40 tanh MLPs. When training with boosting, we train a width-20 network for $f^{(1)}_{\theta_1}$ and a width-20 network for $f^{(2)}_{\theta_2}$, for a combined width of 40. We also plot a width-40 network trained solely with BFGS for comparison. We find, unsurprisingly, that BFGS significantly outperforms Adam. With our tricks, particularly boosting, we can sometimes outperform even well-tuned Adam by eight orders of magnitude, driving RMSE loss down to $\approx 10^{-14}$, close to the machine precision limit. See Appendix A for a brief discussion of the origin of the benefit of boosting.

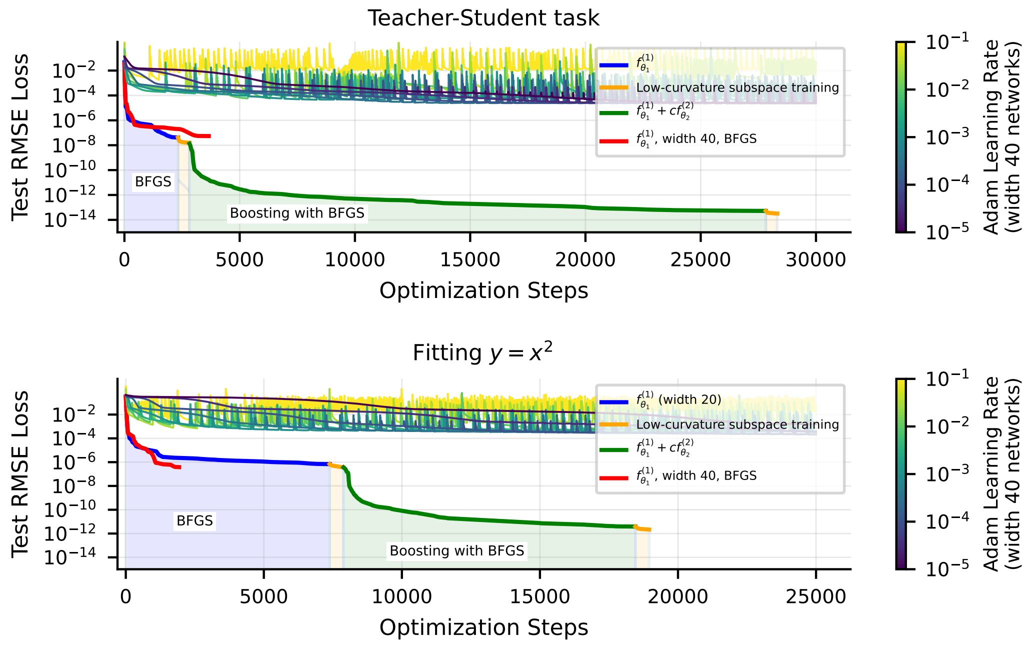


Figure 8. Comparison of Adam with BFGS + low-curvature subspace training + boosting. Using second-order methods like BFGS, but especially using boosting, leads to an improvement of many orders of magnitude over just training with Adam. Target functions are a teacher network (**top**) and a symbolic equation (**bottom**).

4.3. Limitations and Outlook

The techniques we described above are not a silver bullet for fitting neural networks to any data with high precision. Firstly, second-order optimizers like BFGS scale poorly with the number of model parameters N (since the Hessian is an $N \times N$ matrix), limiting their applicability to small models. Also, we find that the gains from boosting diminish quickly as the input dimension of the problem grows. In Figure 9, we see that on the six-dimensional problem discussed earlier (Figure 6a), BFGS + boosting achieves only about a two-order-of-magnitude improvement, bringing the RMSE loss from 10^{-2} to 10^{-4}.

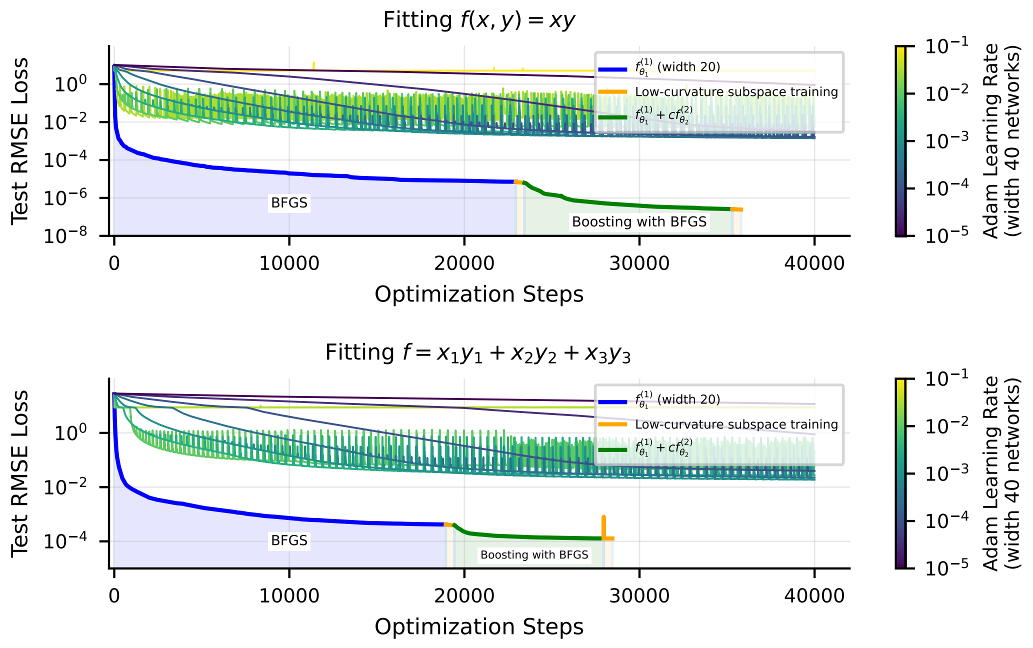


Figure 9. Comparison of Adam with BFGS + low-curvature subspace training + boosting, for a 2D problem (**top**) and a 6D problem (**bottom**), the equation we studied in Figure 6a. As we increase dimension, the optimization tricks we tried in this work show diminishing benefits.

While boosting does not help much for high-dimensional problems, its success on low-dimensional problems is still noteworthy. By training two parts of a neural network separately and sequentially, we were able to dramatically improve performance. Perhaps there are other methods, not yet explored, for training and assembling neural networks in nonstandard ways to achieve dramatically better precision. The solutions found with boosting, where some network weights are at a much smaller scale than others, are not likely to be found with typical training. An interesting avenue for future work would be exploring new initialization schemes, or other ways of training networks sequentially, to discover better solutions in underexplored regions of parameter space.

5. Conclusions

We have studied the problem of fitting scientific data with a variety of approximation methods, analyzing sources of error and their scaling.

- **Linear Simplex Interpolation** provides a piecewise linear fit to data, with RMSE loss scaling reliably as $D^{-2/d}$. Linear simplex interpolation always fits the training points exactly, and so error comes from the generalization gap and the architecture error.
- **ReLU Neural Networks** also provide a piecewise linear fit to data. Their performance (RMSE loss) often scales as D^{-2/d^*}, where d^* is the *maximum arity* of the task (typically $d^* = 2$). Accordingly, they can scale better than linear simplex interpolation when $d > 2$. Unfortunately, they are often afflicted by optimization error making them scale worse than linear simplex interpolation on 1D and 2D problems, and even in higher dimensions in the large-network limit.

- **Nonlinear Splines** approximate a target function piecewise by polynomials. They scale as $D^{-(n+1)/d}$ where n is the order of the polynomial.
- **Neural Networks with smooth activations** provide a nonlinear fit to data. Quite small networks with twice-differentiable nonlinearities can perform multiplication arbitrarily well [35], and so for many of the tasks we study (given by symbolic formulas), their architecture error is zero. We find that their inaccuracy does not appear to scale cleanly as power-laws. Optimization error is unfortunately a key driver of the error of these methods, but with special training tricks, we found that we could reduce RMSE loss on 1D problems down within 2–4 orders of magnitude of the 64-bit machine precision limit $\epsilon_0 \sim 10^{-16}$.

For those seeking high-precision fits, these results suggest the following heuristics, summarized in Figure 10 as a "User's Guide to Precision": If data dimensionality d is low($d \leq 2$), polynomial spline interpolation can provide a fit at machine precision if you (1) have enough data and (2) choose a high enough polynomial order. Neural networks with smooth activations may in some cases also approach machine precision, possibly with less data, if they are trained with second-order optimizers like BFGS and boosted. For higher-dimensional problems ($d \geq 3$), neural networks are typically the most promising choice, since they can learn compositional modular structure that allows them to scale *as if* the data dimensionality were lower.

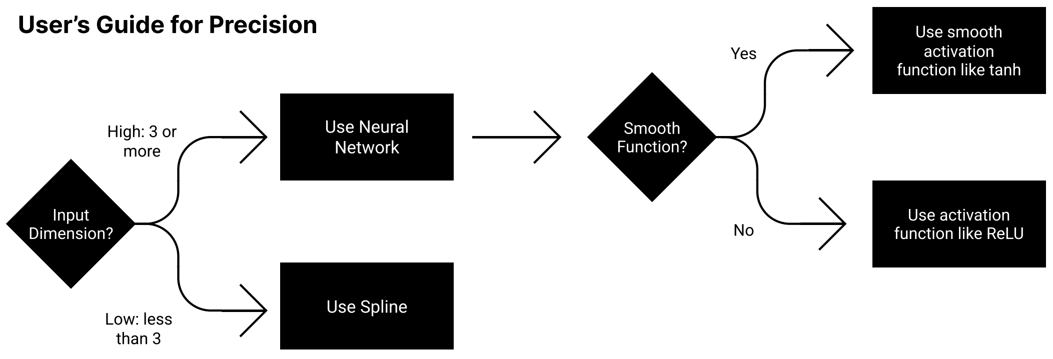

User's Guide for Precision
Input Dimension?
High: 3 or more
Low: less than 3
Use Neural Network
Use Spline
Smooth Function?
Yes
No
Use smooth activation function like tanh
Use activation function like ReLU

Figure 10. User's Guide for Precision: which approximation is best depends on properties of the problem.

In summary, our results highlight both advantages and disadvantages of using neural networks to fit scientific data. We hope that they will help provide useful building blocks for further work towards precision machine learning.

Author Contributions: E.J.M. performed the experiments and made the figures. E.J.M., Z.L. and M.T. wrote the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by The Casey Family Foundation, the Foundational Questions Institute, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant No. 2141064), and IAIFI through NSF grant PHY-2019786.

Data Availability Statement: The data presented in this study can be reproduced with code openly available at https://github.com/ejmichaud/precision-ml (accessed on 4 January 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Boosting Advantage Comes Not Just from Rescaling Loss

When one trains a second network on the residual of a first trained network during boosting, the scale of the loss, at least early in training, lies around order unity, much higher than the loss at the end of training the first network. Does the advantage of boosting then lie in increasing the value of the loss to avoid precision limits within BFGS? To investigate this, instead of performing boosting, we instead simply rescale the loss after training width-40 networks with BFGS and further train the same network with BFGS. In Figure A1, we plot training curves for varying amounts of rescaling of the loss function. We find that the benefits of rescaling the loss are small compared with boosting. At best, we achieved an ≈4× reduction in RMSE loss, compared with the 10,000–100,000× reduction from boosting.

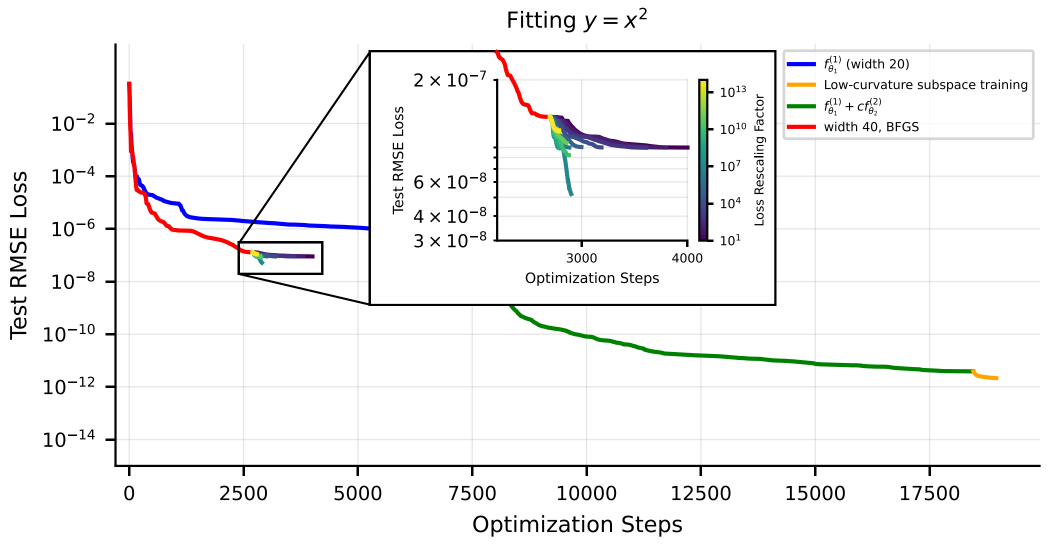


Figure A1. Rescaling loss has minimal benefit relative to boosting.

Appendix B. A Closer Look at Low-Curvature Subspace Optimization

Low-curvature subspace optimization tends to reduce loss much less than boosting; it is difficult to even see improvement from low-curvature subspace training in Figures 8 and 9. In Figure A2, we show training curves for low-curvature subspace optimization for a width-20 depth-3 tanh network after being trained with BFGS. We see that for appropriately chosen curvature thresholds τ, we can reduce RMSE loss by 2–3×, although the method is admittedly expensive, requiring a full computation of the Hessian.

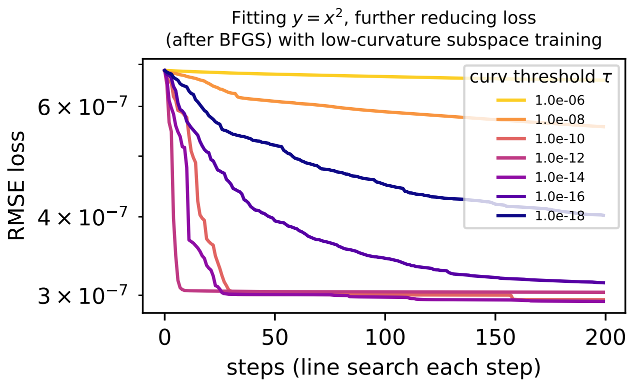


Figure A2. Low-curvature subspace training curves for varying thresholds τ.

Appendix C. Neural Scaling of Tanh Networks with BFGS versus Adam Optimizer

In Section 3, we found that tanh networks sometimes scaled poorly, and suggested that this was due to challenges with optimization rather than the expressivity of the architecture. In Figure A3, we compare neural scaling curves when training with Adam versus the BFGS optimizer for the same problems as Figure 5.

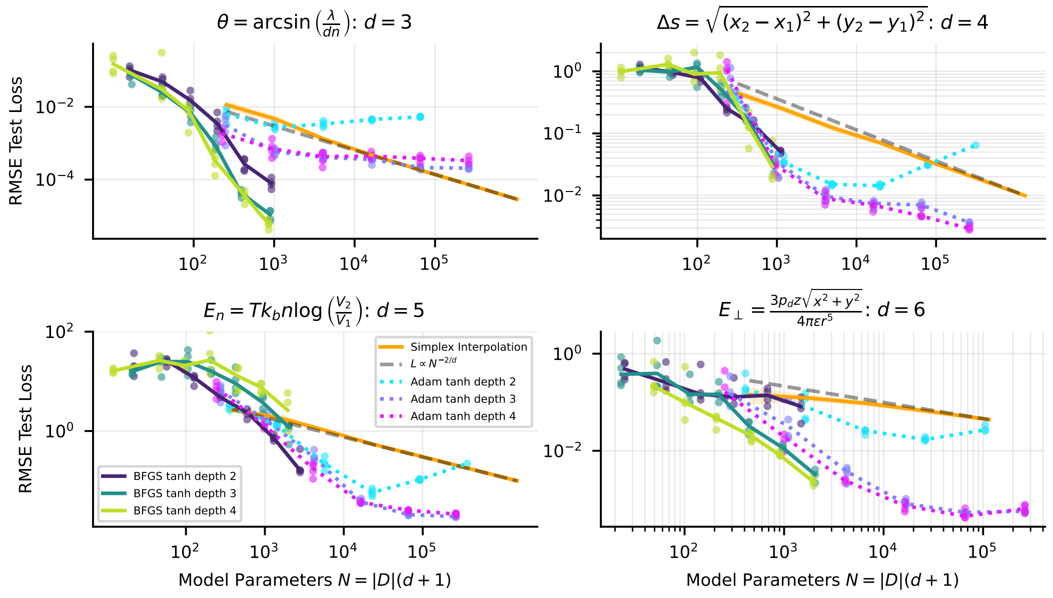


Figure A3. Scaling of tanh networks with the BFGS vs Adam optimizer. We use the same setup as in Figure 5, training tanh MLPs of depth 2–4 of varying width on functions given by symbolic equations. BFGS outperforms Adam on the 3-dimensional example shown (**top left**) and performs roughly similarly to Adam on the other problems.

Appendix D. Loss Landscape at Lower Loss

In Figure 7, we plotted the Hessian eigenvalues at the end of training with the Adam optimizer. In Figure A4, we do the same but for a point in the loss landscape found by the BFGS optimizer rather than Adam. The basic picture of a "canyon" discussed in Section 4 is clearer at lower-loss points in the landscape found by BFGS—there is a clear set of high-curvature directions which the gradient mostly points along.

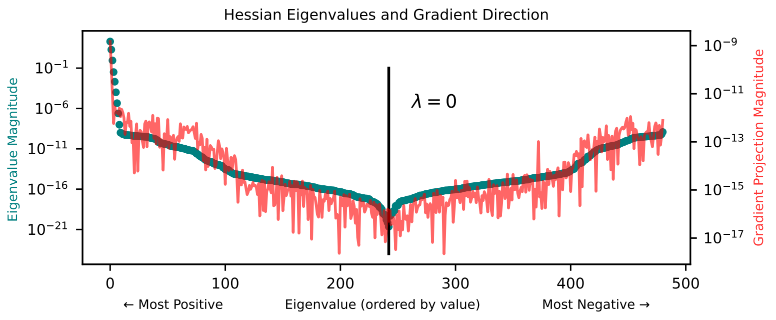


Figure A4. Eigenvalues (dark green) of the loss landscape Hessian (MSE loss) after training a width-20, depth-3 network to fit $y = x^2$ with the BFGS optimizer. Like in Figure 7, we also plot the magnitude of the gradient's projection onto each corresponding eigenvector (thin red line). The "canyon" shape of the loss landscape is more obvious at lower-loss points in the landscape found by BFGS than Adam finds. There is a clear set of top eigenvalues corresponding to a few directions of much higher curvature than the bulk.

References

1. Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; Narayanan, P. Deep Learning with Limited Numerical Precision. In Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 6–11 July 2015 ; Bach, F., Blei, D., Eds.; PMLR: Lille, France, 2015; Volume 37, pp. 1737–1746.
2. Micikevicius, P.; Narang, S.; Alben, J.; Diamos, G.; Elsen, E.; Garcia, D.; Ginsburg, B.; Houston, M.; Kuchaiev, O.; Venkatesh, G.; et al. Mixed precision training. *arXiv* **2017**, arXiv:1710.03740.
3. Kalamkar, D.; Mudigere, D.; Mellempudi, N.; Das, D.; Banerjee, K.; Avancha, S.; Vooturi, D.T.; Jammalamadaka, N.; Huang, J.; Yuen, H.; et al. A study of BFLOAT16 for deep learning training. *arXiv* **2019**, arXiv:1905.12322.

4. Wang, Y.; Lai, C.Y.; Gómez-Serrano, J.; Buckmaster, T. Asymptotic self-similar blow up profile for 3-D Euler via physics-informed neural networks. *arXiv* **2022**. [CrossRef]
5. Jejjala, V.; Pena, D.K.M.; Mishra, C. Neural network approximations for Calabi-Yau metrics. *J. High Energy Phys.* **2022**, *2022*, 105. [CrossRef]
6. Martyn, J.; Luo, D.; Najafi, K. *Applying the Variational Principle to Quantum Field Theory with Neural-Networks*; Bulletin of the American Physical Society; American Physical Society: Washington, DC, USA, 2023.
7. Wu, D.; Wang, L.; Zhang, P. Solving statistical mechanics using variational autoregressive networks. *Phys. Rev. Lett.* **2019**, *122*, 080602. [CrossRef] [PubMed]
8. Udrescu, S.M.; Tegmark, M. AI Feynman: A physics-inspired method for symbolic regression. *Sci. Adv.* **2020**, *6*, eaay2631. [CrossRef] [PubMed]
9. Udrescu, S.M.; Tan, A.; Feng, J.; Neto, O.; Wu, T.; Tegmark, M. AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 4860–4871.
10. Leighton, R.B.; Sands, M. *The Feynman Lectures on Physics*; Addison-Wesley: Boston, MA, USA, 1965.
11. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*; IEEE Standard for Floating-Point Arithmetic. IEEE: Piscataway, NJ, USA, 2019; pp. 1–84. [CrossRef]
12. Gühring, I.; Raslan, M.; Kutyniok, G. Expressivity of deep neural networks. *arXiv* **2020**, arXiv:2007.04759.
13. Hestness, J.; Narang, S.; Ardalani, N.; Diamos, G.; Jun, H.; Kianinejad, H.; Patwary, M.; Ali, M.; Yang, Y.; Zhou, Y. Deep learning scaling is predictable, empirically. *arXiv* **2017**, arXiv:1712.00409.
14. Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T.B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; Amodei, D. Scaling laws for neural language models. *arXiv* **2020**, arXiv:2001.08361.
15. Henighan, T.; Kaplan, J.; Katz, M.; Chen, M.; Hesse, C.; Jackson, J.; Jun, H.; Brown, T.B.; Dhariwal, P.; Gray, S.; et al. Scaling laws for autoregressive generative modeling. *arXiv* **2020**, arXiv:2010.14701.
16. Hernandez, D.; Kaplan, J.; Henighan, T.; McCandlish, S. Scaling laws for transfer. *arXiv* **2021**, arXiv:2102.01293.
17. Ghorbani, B.; Firat, O.; Freitag, M.; Bapna, A.; Krikun, M.; Garcia, X.; Chelba, C.; Cherry, C. Scaling laws for neural machine translation. *arXiv* **2021**, arXiv:2109.07740.
18. Gordon, M.A.; Duh, K.; Kaplan, J. Data and parameter scaling laws for neural machine translation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Punta Cana, Dominican Republic, 7–11 November 2021; pp. 5915–5922.
19. Zhai, X.; Kolesnikov, A.; Houlsby, N.; Beyer, L. Scaling vision transformers. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, New Orleans, LA, USA, 18–24 June 2022; pp. 12104–12113.
20. Hoffmann, J.; Borgeaud, S.; Mensch, A.; Buchatskaya, E.; Cai, T.; Rutherford, E.; Casas, D.d.L.; Hendricks, L.A.; Welbl, J.; Clark, A.; et al. Training Compute-Optimal Large Language Models. *arXiv* **2022**, arXiv:2203.15556.
21. Clark, A.; de Las Casas, D.; Guy, A.; Mensch, A.; Paganini, M.; Hoffmann, J.; Damoc, B.; Hechtman, B.; Cai, T.; Borgeaud, S.; et al. Unified scaling laws for routed language models. In Proceedings of the International Conference on Machine Learning, Baltimore, MD, USA, 17–23 July 2022; pp. 4057–4086.
22. Sharma, U.; Kaplan, J. A neural scaling law from the dimension of the data manifold. *arXiv* **2020**, arXiv:2004.10802.
23. Bahri, Y.; Dyer, E.; Kaplan, J.; Lee, J.; Sharma, U. Explaining neural scaling laws. *arXiv* **2021**, arXiv:2102.06701.
24. Arora, R.; Basu, A.; Mianjy, P.; Mukherjee, A. Understanding deep neural networks with rectified linear units. *arXiv* **2016**, arXiv:1611.01491.
25. Eckle, K.; Schmidt-Hieber, J. A comparison of deep networks with ReLU activation function and linear spline-type methods. *Neural Netw.* **2019**, *110*, 232–242. [CrossRef]
26. Virtanen, P.; Gommers, R.; Oliphant, T.E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nat. Methods* **2020**, *17*, 261–272. [CrossRef]
27. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
28. LeCun, Y.A.; Bottou, L.; Orr, G.B.; Müller, K.R. Efficient backprop. In *Neural Networks: Tricks of the Trade*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 9–48.
29. Poggio, T. *Compositional Sparsity: A Framework for ML*; Technical Report; Center for Brains, Minds and Machines (CBMM): Cambridge, MA, USA, 2022.
30. Dahmen, W. Compositional Sparsity, Approximation Classes, and Parametric Transport Equations. *arXiv* **2022**, arXiv:2207.06128.
31. Poggio, T.; Mhaskar, H.; Rosasco, L.; Miranda, B.; Liao, Q. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *Int. J. Autom. Comput.* **2017**, *14*, 503–519. [CrossRef]
32. Kohler, M.; Langer, S. On the rate of convergence of fully connected deep neural network regression estimates. *Ann. Stat.* **2021**, *49*, 2231–2249. [CrossRef]
33. Bauer, B.; Kohler, M. On deep learning as a remedy for the curse of dimensionality in nonparametric regression. *Ann. Stat.* **2019**, *47*, 2261–2285. [CrossRef]
34. Lekien, F.; Marsden, J. Tricubic interpolation in three dimensions. *Int. J. Numer. Methods Eng.* **2005**, *63*, 455–471. [CrossRef]
35. Lin, H.W.; Tegmark, M.; Rolnick, D. Why does deep and cheap learning work so well? *J. Stat. Phys.* **2017**, *168*, 1223–1247. [CrossRef]
36. Sagun, L.; Bottou, L.; LeCun, Y. Eigenvalues of the hessian in deep learning: Singularity and beyond. *arXiv* **2016**, arXiv:1611.07476.

37. Sagun, L.; Evci, U.; Guney, V.U.; Dauphin, Y.; Bottou, L. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv* **2017**, arXiv:1706.04454.
38. Gur-Ari, G.; Roberts, D.A.; Dyer, E. Gradient descent happens in a tiny subspace. *arXiv* **2018**, arXiv:1812.04754.
39. Nocedal, J.; Wright, S.J. *Numerical Optimization*; Springer: New York, NY, USA, 1999.
40. Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794.
41. Schwenk, H.; Bengio, Y. Training methods for adaptive boosting of neural networks. *Adv. Neural Inf. Process. Syst.* **1997**, *10*, 647–650.
42. Schwenk, H.; Bengio, Y. Boosting neural networks. *Neural Comput.* **2000**, *12*, 1869–1887. [CrossRef] [PubMed]
43. Badirli, S.; Liu, X.; Xing, Z.; Bhowmik, A.; Doan, K.; Keerthi, S.S. Gradient boosting neural networks: Grownet. *arXiv* **2020**, arXiv:2002.07971.