Contents

List of Figures

Chapter 1

Introduction

In the past 5 years there have been breakthroughs on machine learning tasks that were previously considered to be decades away. Machine learning systems can now play Go, chess [1] and even the real-time strategy game Starcraft [2]. They can generate images from text, even those which contain unlikely objects [3], and can perform reasoning with as little as a handful of examples [4]. Some models can optimise traffic [5], predict the structure of proteins [6] and perform relatively complex mathematical manipulations [7]. There is even a system which can perform several unrelated tasks without explicitly being trained for each in part [8]. How can we account for these successes?

At least part of the story is that with increased interest and funding, machine learning has been scaled up by several orders of magnitude. Today's systems are larger, trained on more data and on more powerful, dedicated hardware than their predecessors. In a sense, the field has benefitted from better hardware and engineering.

More recently, there has been a paradigm shift owing to the advent of large language models. Early work showed that language models trained unsupervised on large amounts of data performed well at natural language tasks with only a few examples [9]. This was a major improvement on previous systems which required fine-tuning on datasets for each downstream task; often, this data is infeasible or expensive to collect. Scaling up these models made them even more accurate, with even less task-specific input [10]. Today, we have models with hundreds of billions of parameters that can generate useful outputs simply through prompting – the so-called "zero-shot" setting. What's more, it looks like these large models are under-fitted [11].

In some form or another, these recent advances rely on a particular sort of machine learning approach: neural networks. And yet, when it comes to neural networks, we face something that we could reasonably call "the mystery of generalisation". Simply put, we do not know exactly why deep neural networks work so well. In this paper, we'll look at a possible explanation for this mystery that is related to both the optimisation process and the network architecture.

1.1 Generalisation and inductive bias

In the logical sense, generalisation refers to the process of moving from the particular to the general. It's likely that generalisation is a key component of intelligence, in the sense that it enables the extraction of concepts and rules that can be re-used and re-composed in new ways. In the case of machine learning systems, by the term generalisation we mean the capability of a system to do well not just on training data – the data which the model is fit to – but also on test data (which may be unlike the training data; see Section 3.7).

It is clear that a large factor in generalisation capability is data itself. Without enough data, it is impossible to isolate exactly which concept to learn. But data is not the only factor. There are other mechanisms which influence how a system might learn a particular sort of concept; we collectively call these the "inductive bias" of a system. Intuitively, the inductive bias is anything other than the data that determines the solution. Humans have inductive biases too, some of which we owe to evolution. For example, most people find sweet treats enjoyable without conducting large-scale analyses. This is because sugar is high in calories, and for the majority of our species' history, eating sugar was unambiguously a good idea. We learn the hypothesis "Eating sugar is good" with only a handful of data points (or raspberries).

Neural networks also have inductive biases which determine the sorts of solutions that networks tend to find. We know at a high level that inductive bias differs between architectures, but we are not often in a position where we can characterise this bias. For example, we know that convolutional neural networks applied to image classification are approximately translation invariant, which means that they can correctly label an image of a bicycle, regardless of where in the image the bicycle is. On the other hand, when applied to object localisation, these networks should be translation equivariant: they should "activate" where there is an object in the image, and nowhere else. We know that CNNs are biased in this way because they are explicitly designed to; we cannot usually say this for other architectures. Even so, we do not have a complete picture of the inductive bias of CNNs.

Why do we ultimately want to understand these inductive biases? On one level, we might like to build even better systems, by finding which inductive biases are useful. It might be the case that some inductive biases are more well-suited to the types of tasks we apply machine learning to. These will appear to be more generally capable, but they are likely not universal. For example, in recent years, the transformer architecture, with applications in language [10] and vision [12], seems to be displacing dedicated network architectures. What makes the transformer perform so well across different types of tasks?

Another reason to want to understand inductive biases – and potentially a more consequential one – is that we *need* to know what goes on inside networks. At the moment, neural networks are generally opaque. We do not know what sort of "reasoning", if any,

goes on inside the network. Perhaps for the most part, this is not an issue, as long as it is possible to validate that the network as a whole behaves in a desirable way. But as machine learning systems become more complex, as we deploy them at larger scales and on more open-ended domains, it's likely that humans will no longer be able to validate their outputs as correct or incorrect.

In this work, we evaluate the inductive bias of neural networks on the univariate regression task. We start from an interesting set of results regarding the implicit bias of gradient descent, the optimisation algorithm most often used to train neural networks. In [13], gradient descent is characterised as biased toward particular sorts of smooth functions which solve a closed-form variational problem. In practice, using this characterisation we could control the types of functions a trained network represents by changing how its parameters are initialised.

Our main contributions are as follows:

- we replicate the results in [13] for univariate regression;
- we test the robustness of these results under different experimental conditions by varying key hyperparameters such as learning rate, model size and optimiser
- we additionally evaluate the inductive bias on two separate generalisation tasks, interpolation and extrapolation, by obscuring parts of the training set;

The paper starts with a review of the relevant concepts and literature followed by a description of the problem and experimental set-ups, and finishes with findings and their interpretations.

Chapter 2

Notation and key concepts

Before we review the relevant literature, it's useful to introduce three concepts that will reoccur throughout this paper. We start with univariate regression, since it is the problem for which new results are found in the paper [13], which we aim to replicate here. Then, we introduce the neural network formalism, and end with a short description of natural cubic splines, a class of functions that are useful for describing the inductive bias of some neural networks.

2.1 Univariate regression

Given a set of inputs $x_j \in \mathcal{X}$ and labels $y_j \in \mathcal{Y}$, univariate regression refers to the task of determining a function $f : \mathbb{R} \to \mathbb{R}$, with the property that $f(x) \approx y, \forall x \in \mathcal{X}, y \in \mathcal{Y}$. We will refer to the set of input-label pairs $\{x_j, y_j\}_{j=1}^{M}$ as the dataset $\mathcal{D}$. We denote by $|\mathcal{D}|$ the size of the dataset.

In practice, the dataset is split into the training set $\mathcal{D}_{\text{train}}$ and the test set $\mathcal{D}_{\text{test}}$. The training dataset contains the data a model is fit to, whereas the test set contains data on which its performance is evaluated. It's also common to hold out an additional subset called the validation set for the purposes of tuning a model's hyperparameters. Hyperparameters are numbers that control particular aspects of the training process which themselves do not get updated during training.

A special case of this is linear regression, where f is restricted to being a linear function, and the task is finding two constants $u, v \in \mathbb{R}$ such that $f(x_j) = ux_j + v \approx y_j, \forall x \in \mathcal{X}, y_j \in \mathcal{Y}$. However, if the data follows a non-linear trend, there is a limit to how well a linear estimator can do. Fortunately, there are plenty of alternatives.

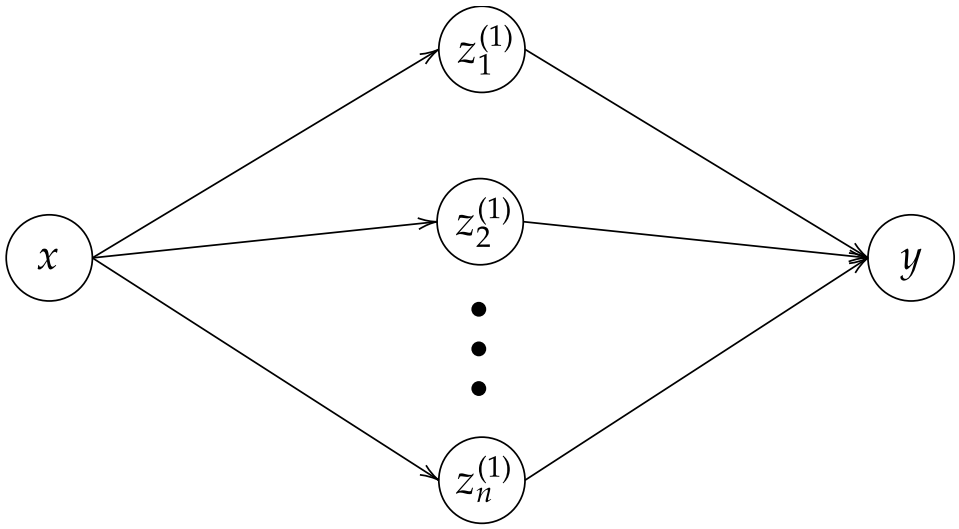


Figure 2.1: Neural network architecture with one input, one output, and a single hidden layer containing n hidden units with activations z_j. The first set of arrows corresponds to the weights in layer 1: $w_j^{(1)}$. The biases $b_j^{(1)}$ by convention are characteristic of each unit in the hidden layer, so they are inside the unit. The second set of arrows corresponds to the weights in layer 2, $w_j^{(2)}$, and the final bias $b^{(2)}$ is in the output unit.

2.2 Neural networks

Going beyond linear regression, one way to determine the function f is to parameterise it with a set of parameters $\boldsymbol{\theta}$ and choose these parameters such that the function fits our data. In a sense, this is what neural networks do: they approximate a function whose closed form is unknown. Loosely following [14], the feed-forward neural network is a collection of such parameters $\boldsymbol{\theta}$, split into a set of weights and biases which implement the relationship:

$$a_j^{(1)} = w_j^{(1)} x + b_j^{(1)}, \tag{2.1}$$

which is called a pre-activation. This linear combination of weight, input and bias is then transformed using a nonlinear activation function ϕ:

$$z_j^{(1)} = \phi(a_j^{(1)}), \tag{2.2}$$

which yields the output (or activation) of a hidden unit j. Overall, a network with one hidden layer implements the following:

$$f(x, \boldsymbol{\theta}) = \sum_{j=1}^{m} w_j^{(2)} \phi(w_j^{(1)} x + b_j^{(1)}) + b^{(2)}, \tag{2.3}$$

with $\boldsymbol{\theta} = \text{vec}(w_1^{(1)}, \ldots, w_m^{(1)},\ b_1^{(1)}, \ldots, b_m^{(1)},\ w_1^{(2)}, \ldots, w_m^{(2)},\ b^{(2)})$.

In practice, networks tend to be have more than one hidden layer, often stacking dozens of layers of different widths (a layer's width is its number of units). One of the most successful networks in recent years, the transformer [15], stacks many "blocks", each of

which contains several deep feed-forward neural networks. Any network with hidden layers is considered "deep", however in practice this usually refers to two or more hidden layers.

A network's parameters are initialised randomly according to some scheme, such as Glorot initialisation [16]. Fitting the network to the data essentially entails updating its parameters via an optimisation process, usually gradient descent, to optimise some measure of the error of the network's predictions on the training dataset.

In the regression case, it's common for the loss function to be mean squared error, defined as:

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) &= \frac{1}{n}\sum_{i=1}^{n} \ell(\hat{y}, y), \\ \ell(\hat{y}, y) &= (\hat{y} - y)^2 \end{aligned} \tag{2.4}$$

where y_i is the ground truth and $\hat{y}_i$ is the network's prediction for x_i, i.e. $f(x_i, \boldsymbol{\theta})$. Given a loss function and a set of parameters, at each optimisation step, gradient descent updates the parameters as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\boldsymbol{\theta}_t) \tag{2.5}$$

where η is the learning rate (or the step size). η is a hyperparameter: a number whose value is not updated by the optimisation process itself, and whose selection is down to modeller's choice. The learning rate has the effect of controlling how large each parameter update is; too large a step means that the update may overshoot a good value for $\boldsymbol{\theta}$, but too small a step and the optimisation takes very long to converge or find a bad solution.

Throughout optimisation, the iterates $\boldsymbol{\theta}_t$ achieve better values for the loss function, such that after some time the optimisation process converges to a solution with minimal loss. For modern neural networks it isn't uncommon for training loss at convergence to be ≈ 0, i.e. the network perfectly fits the data, but this isn't necessarily the case. A more in-depth discussion of the effect of overparameterisation happens in Chapter 3.

2.3 Natural cubic splines

Following the formalism in [17], let's first define a set of points Δ in an interval $[a, b]$:

$$\Delta : a = x_1 < x_2 < \cdots < x_n = b \tag{2.6}$$

We also have the respective values of y: $Y = \{y_1, y_2, \ldots, y_n\}$ for the points in Δ.

A function $S_\Delta(x) \in C^2(a,b)$ is represented by piecewise polynomials:

$$S_\Delta(x) = \begin{cases} C_1(x) & x_1 \leq x \leq x_2 \\ C_2(x) & x_2 < x \leq x_3 \\ \dots & \\ C_n(x) & x_{n-1} < x \leq x_n \end{cases}, \tag{2.7}$$

where each piece is a cubic polynomial of the form: $C_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$. We say that S_Δ is a cubic spline on the set Δ if for all the points in the set, $S(x_i) = y_i$ and the following conditions hold for each pair of polynomials in S_Δ [18]:

$$\begin{aligned} C'_{i-1}(x_i) &= C'_i(x_i), \ i = 2, \dots, n \\ C''_{i-1}(x_i) &= C''_i(x_i), \ i = 2, \dots, n \end{aligned} \tag{2.8}$$

The conditions that each polynomial must pass through exactly two points, together with the conditions regarding first and second derivatives, yield a system of $4n-2$ equations that need to be solved in order to recover the coefficients a_i, b_i, c_i, d_i. But with $4n$ coefficients in total, two additional equations are required, called the boundary conditions. There are multiple types of boundary conditions, yielding different functions. One of these functions is the natural cubic spline, for which the following statements are true:

$$\begin{aligned} C''_1(x_1) &= 0 \\ C''_n(x_n) &= 0. \end{aligned} \tag{2.9}$$

Having been acquainted with these three key concepts, we move onto a survey of the research literature most relevant to the characterisation of implicit bias in [13].

Chapter 3

The mystery of generalisation

In the introduction, we briefly mentioned that there is a mystery regarding the generalisation capability of deep neural networks. In this chapter, we first present why such a mystery exists in the first place. Several approaches are covered which aim to elucidate the question of generalisation, some of which arrive at similar answers through different perspectives.

3.1 Learnability and generalisation

All neural networks carry out the same algorithmic blueprint: empirical risk minimisation (ERM). Under ERM, a generic learning algorithm $\mathcal{A}$ searches over a set of predictors, $\mathcal{H}$, called a hypothesis class, to find a predictor or hypothesis $h \in \mathcal{H}$ which minimises the training error on a dataset comprising inputs and their labels drawn from a distribution $\mathcal{D}$ (see e.g. [19]). The term training error is interchangeable with empirical risk.

The hypothesis class $\mathcal{H}$ is selected before seeing the data, effectively restricting the search space. There is a good reason for doing this: one of the main results of statistical learning theory is the "no free lunch" theorem, which states that there does not exist a universal learner; no predictor has the minimum risk achievable across all distributions. Choosing a suitable hypothesis class is also a way of incorporating prior beliefs regarding which predictors should do well on a given task.

The error of an ERM predictor can be decomposed into two terms: the approximation error (or bias) and the estimation error (or variance). The former is equal to the lowest error achieved by a hypothesis in $\mathcal{H}$; the latter is the difference between the approximation error and the actual error achieved by the predictor.

From this decomposition, a relation arises between the size of the hypothesis class $\mathcal{H}$ and the error of an ERM predictor. Choosing large $\mathcal{H}$ yields low-bias, but high-variance predictors, which in practice "overfit" the data: they achieve good training error, but do poorly on test data. On the other hand, restricting $\mathcal{H}$ to be small yields high-bias,

low-variance predictors; these "underfit" the data, in the sense that they achieve worse training error, but they may generalise better.

To achieve the best possible performance, a learning algorithm needs to find the optimal trade-off between two error terms. This phenomenon is called the bias-variance trade-off, and it is the source of tension between statistical learning theory and modern deep neural networks. Overparameterised networks, that is, networks with many more parameters than data points in the training set, have very large hypothesis classes, which the theory predicts should lead to predictors which overfit and generalise poorly. And yet, deep networks generalise well. Why?

There is no consensus answer to this question. However, there are many proposed explanations, many of which have sound theoretical justifications and *some* empirical backing. In the rest of this chapter, we survey the literature on generalisation, a broad field that resists easy classification. Here, we cover the approaches that seem most promising, and which are closest in reasoning to our main inspiration, the paper by Jin and Montúfar [13].

3.2 Implicit bias of the optimisation process

One of the leading explanations for why overparameterised networks work so well is that the optimisation algorithm is biased toward simpler solutions, which generalise well. That is, despite these networks having a very large hypothesis class, simpler solutions are more likely to be found than complex ones. In this section, we focus on characterisations of the bias of gradient descent due to its ubiquity in neural network training, but similar results exist for other optimisers [20].

If such a bias existed, it would be useful to understand by which mechanism it selects simpler hypotheses. For gradient descent, it seems like this mechanism is an implicit regularisation of the ℓ_2 norm of the network parameters. One of the earliest observations in this direction comes from Neyshabur et al. [21], a primarily empirical examination.

Here, the authors train networks resembling Fig. 2.1 with increasing number of hidden units n on an image classification task. They show that for networks which are large relative to the number data points they are trained on, test loss does not increase with size. In fact, the opposite happens: networks generalise better. This outcome is robust to different experiments they tried, including training on partially randomised labels to force overfitting.

Although they do not hone in on the precise character of the regularisation, [21] demonstrate that something other than the size is controlling the capacity, i.e. the size of the hypothesis class $\mathcal{H}$, of the network, and that it may be related to the norm of the weights. A similar empirical observation is made in [20], who find that gradient descent minimising

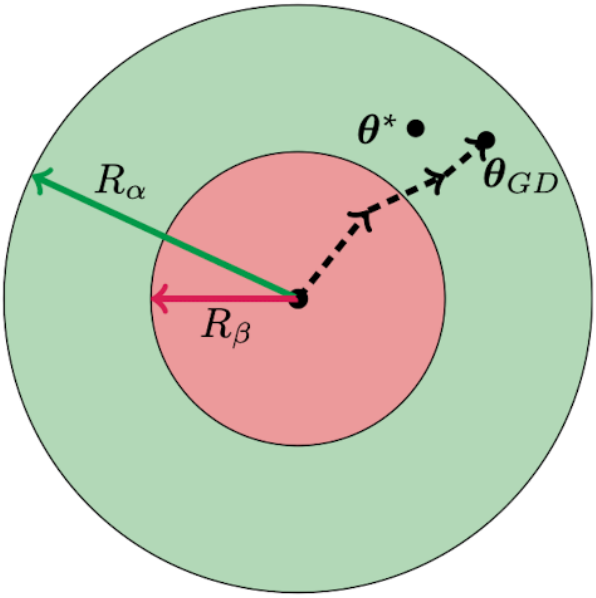


Figure 3.1: Reproduced from [22], who prove that the solution found by gradient descent always lies in the green ring. The red region is excluded by Theorem 2.4 in [22]. The disc is centred on the initial parameters $\boldsymbol{\theta}_0$. The expressions for R_α, R_β are given in equations 3.2; they depend on scalars α, β which are lower and upper bounds, respectively, for the minimum singular value of the Jacobian matrix $\mathcal{J}(\boldsymbol{\theta})$.

mean squared error converges to the solution $f(\cdot, \theta^*)$ that fits the data and whose set of parameters is closest to initialisation:

$$\theta^* = \text{argmin}_\theta \, \|\theta - \theta_0\|_2, \tag{3.1}$$

with $f(\mathcal{X}, \theta^*) = \mathcal{Y}$.

A rigorous proof for the finding that gradient descent implicitly regularises the ℓ_2 comes from Oymak and Soltanolkotabi [22]. Here, the authors are able to set a lower and upper bound for the neighbourhood in which gradient descent searches for solutions. Specifically, gradient descent is guaranteed to converge to a solution $\boldsymbol{\theta}^*$ in the neighbourhood of radius $\frac{4}{\alpha}\|f(\boldsymbol{\theta}_0) - \mathbf{y}\|_2$ around the initial point $\boldsymbol{\theta}_0$. They additionally obtain a result (see Theorem 2.4 in [22]) that shows that iterates cannot lie in a ball of radius R_α of $\boldsymbol{\theta}_0$, so solutions are always on the "ring" surrounding the initial point (see Figure 3.1 for a visualisation). The radii of the discs are:

$$R_\alpha = \frac{\|f(\boldsymbol{\theta}_0) - \mathbf{y}\|_2}{\alpha}, \quad R_\beta = \frac{\|f(\boldsymbol{\theta}_0) - \mathbf{y}\|_2}{\beta}, \tag{3.2}$$

where α, β are lower and upper bounds for the smallest and largest singular values of the Jacobian matrix:

$$\mathcal{J}(\boldsymbol{\theta}) \in \mathbb{R}^{m \times n}, \mathcal{J}_{ij} = \frac{\partial f(\mathbf{x}_i, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j}. \tag{3.3}$$

This set of results raises an interesting observation: if it is indeed the magnitude of the

weights and not their number which controls the capacity of the network, then overparameterised networks should generalise, so long as they have small weight norms. In effect, this means that networks could have an *infinite* number of hidden units and still generalise well.

This line of argument is echoed in [23], who consider infinitely wide ReLU networks in the univariate regression case. They find that as long as the total ℓ_2 norm of the weights of these networks is bounded, the hypotheses the trained networks represent are a class of functions called linear splines. Linear splines are a special case of the cubic spline definition presented in 2.7 with the cubic and quadratic terms equal to 0.

3.3 Approximation theory

Taking a step back, this is an example of a series of results that finds equivalences between deep networks and spline interpolation – learning from data using spline functions. It is a statement about the sorts of functions neural networks can and do represent in practice. But while this type of precise characterisation is recent, results in this area of research date back several decades under the umbrella of approximation theory.

One of the most famous results in approximation theory comes from Kurt Hornik [24], who finds that multi-layer feedforward networks are universal approximators. That is, they can approximate any continuous function arbitrarily well, provided that there are enough hidden layers. If anything, this result makes even more pressing the need to pinpoint a mechanism through which networks find simple, low-complexity functions.

Since then, we've found more precise characterisations of the inductive bias of neural networks, again in the form of splines. For example, [25] find that deep ReLU networks are solutions to a variational problem over functions in a particular function space. Closer to our main focus, [26] find that depending on parameter initialisation shallow ReLU networks represent cubic splines or adaptive linear splines, "where neurons accumulate at the discontinuities and yield piecewise linear approximations" [26].

3.4 Learning regimes and the neural tangent kernel

The scale of parameter initialisation in [26] controls whether a network operates in the so-called "kernel" (or "lazy learning") or "adaptive" regimes. Scale here refers to a simple hyperparameter α used to scale the initial parameters such that $\mathbf{w_0} = \alpha\boldsymbol{\omega}_0$, where ω_0 is a vector of weights initialised according to some known scheme. Networks in the two regimes have different inductive biases, and they yield different solutions after training [27]. This brings us to another perspective on why and how neural networks generalise, that of learning regimes.

The seminal paper from Jacot et al. [28] introduces a connection between the training

dynamics of feedforward neural networks and kernel gradient with regards to the neural tangent kernel (NTK), relying on an analysis in the infinite-width limit. For finite networks, the NTK is random at initialisation and varies during training, but if we make the assumption that $n \to \infty$, the kernel is stable during training and converges to a known limiting kernel. The authors go on to relate the positive-definiteness of the limiting kernel to convergence properties of the network.

Building on this work, Lee et al. [29] show that for sufficiently wide hidden layers, a network's training dynamics are similar to those of an equivalent linearised model. This finding is key to the derivation of the variational problem in [13] – the paper whose results we replicate in Section 5 –, so it is worth spending some time explaining the arguments. The authors start with a fully-connected neural network with L hidden layers and a final readout layer $L+1$, with the network's units implementing the relation:

$$\begin{cases} h^{l+1} = x^l W^{l+1} + b^{l+1} \\ x^{l+1} = \phi(h^{l+1}) \end{cases} \tag{3.4}$$

where the weights and biases are defined as:

$$\begin{cases} W^l_{ij} = \frac{\sigma_\omega}{\sqrt{n_l}} \omega^l_{ij} \\ b^l_j = \sigma_b \beta^l_j \end{cases} \tag{3.5}$$

Equations 3.4 and 3.5 define a non-standard parameterisation referred to as the neural tangent kernel (NTK) parameterisation, where the trainable parameters ω^l_{ij}, β^l_j are drawn from a unit Gaussian distribution: $\omega^l_{ij}, \beta^l_j \sim \mathcal{N}(0,1)$. To simplify the notation, let's take $\theta^l \equiv \text{vec}(\{W^l, b^l\})$ as each layer's parameters and θ to mean all the network's parameters. Additionally, let $f_t(x) \equiv h^{L+1}(x)$ denote the output of the network at time t.

Given the standard supervised learning task on a dataset $\mathcal{D} = \Big\{(x,y)|x \in \mathcal{X}, y \in \mathcal{Y}\Big\}$ by minimising empirical loss $\mathcal{L}$, the parameters θ and outputs f_t evolve during time via continuous gradient descent as:

$$\begin{aligned} \dot{\theta}_t &= -\eta \nabla_\theta f_t(\mathcal{X})^T \nabla_{f_t(\mathcal{X})} \mathcal{L} \\ \dot{f}_t(\mathcal{X}) &= \nabla_\theta f_t(\mathcal{X}) \dot{\theta}_t = -\eta \hat{\Theta}_t(\mathcal{X}, \mathcal{X}) \nabla_{f_t(\mathcal{X})} \mathcal{L} \end{aligned} \tag{3.6}$$

where η is the learning rate, $f_t(\mathcal{X}) = \text{vec}([f_t(x)]_{x \in \mathcal{X}})$ and $\hat{\Theta}_t \equiv \hat{\Theta}_t(\mathcal{X}, \mathcal{X})$ is the neural tangent kernel at time t:

$$\hat{\Theta}_t = \nabla_\theta f_t(\mathcal{X}) \nabla_\theta f_t(\mathcal{X})^T = \sum_{l=1}^{L+1} \nabla_{\theta^l} f_t(\mathcal{X}) \nabla_{\theta^l} f_t(\mathcal{X})^T \tag{3.7}$$

The first important result in [29] regards the training dynamics of linearised networks. Let's define a linearised network as the first order Taylor expansion of a standard neural

network using NTK parameterisation around its initial parameters θ_0:

$$f_t^{\text{lin}} = f_0(x) + \nabla_\theta f_0(x)|_{\theta=\theta_0}\ \omega_t, \tag{3.8}$$

where $\omega_t = \theta_t - \theta_0$. Plugging f_t^{lin} into the ordinary differential equations in 3.6 and choosing mean squared error as our loss we get closed form solutions for the parameters and predictions of the linearised model over time, which depend only on the predictions at initialisation f_0 and the tangent kernel $\hat{\Theta}_0$ (see [29] for the derivation). Having a closed form solution for the training dynamics of these networks means that the values of their parameters and predictions over time are obtainable without carrying out an optimisation process, i.e. without running gradient descent.

Then, the authors go on to show that when training a wide network with a learning rate η less than a threshold η_c which depends on the NTK, as the width n of a network tends to infinity, we have:

$$\begin{aligned} &\sup_{t\geq 0} \|f_t(x) - f_t^{\text{lin}(x)}\|_2 = \mathcal{O}(n^{-\frac{1}{2}}) \\ &\sup_{t\geq 0} \frac{\|\theta_t - \theta_0\|_2}{\sqrt{n}} = \mathcal{O}(n^{-\frac{1}{2}}). \end{aligned} \tag{3.9}$$

This result says that for sufficiently large n, the difference between a network's output and the output of a linearised network is bounded by $\frac{1}{\sqrt{n}}$. As n tends to infinity, this bound becomes tighter and tighter, and the two functions overlap, which means that the infinite-width network is equivalent to a linearised network. Moreover, a similar statement applies to the parameters θ_t: the wider the network, the smaller the difference between θ_t and the initial parameters θ_0.

To summarise, combining the two results we get the insight that infinitely wide networks behave as linearised networks, whose training dynamics are known in closed form. This is a succinct characterisation of not just the bias of these networks, but of their entire behaviour during optimisation.

While this is a fascinating result in itself, it is not one that applies to deep networks in use today. Indeed, [27] recognise that overparameterised networks more often lie in the adaptive regime, which "leads to very different and rich inductive biases, e.g. inducing sparsity or low-rank structure, that allow for generalization in settings where kernel methods would not". What else could explain why these deep networks generalise?

3.5 Loss landscapes

The loss landscape view of generalisation is quite different to the approaches outlined above. It postulates that generalisation occurs due to some characteristics of the loss function, unrelated to the particularities of the optimiser, or to regularisation. Wu et al. [30] frame the problem using two questions:

(a) What differentiates good minima (those with good generalisation capability) from bad minima?

(b) Among all the minima that are possible for an overparameterised neural network, why are trained networks finding the good minima?

The answer to the latter question, [30] say, lies in the fact that in the loss landscape, good minima are surrounded by basins of attraction with large volume relative to the basins around bad minima. The term basin of attraction refers to a region in parameter space such that any parameter $\boldsymbol{\theta}$ lying in that region will eventually converge to an attractor $\mathcal{A}$ (which is some parameter value). The authors find empirically that with different initialisation schemes they still achieve good performance, and conjecture that this is because parameters are more or less always initialised in a good basin of attraction.

They justify this claim analytically for 2-layer networks and through numerical experiments for deeper networks. They find that training with an adversarial objective, i.e. one designed to force the network to overfit, yields models which generalise poorly. However, while the paper offers a new explanation for generalisation which seems to be supported experimentally, it doesn't seem to definitively exclude other explanations for why good solutions are found among bad ones. For example, one claim is that stochastic gradient descent cannot be the answer to the "mystery of generalisation" because plain gradient descent also does relatively well. This rules out stochasticity as the answer, but it does not rule out the implicit bias of gradient descent as an explanation.

Additionally, it is unclear if the empirical findings which justify that all initialisations end up in the basins of good minima hold for other experimental setups. For example, [31] find that it is possible to initialise a network such that SGD finds solutions which do not generalise well, and that these are also in the vicinity of the initial parameters $\boldsymbol{\theta}_0$, in the same region as good solutions.

Another explanation for generalisation that is related to the loss landscape is that networks generalise because they find flat minima. Intuitively, a sharp minimum is one that lies in a "ravine" – a point of low loss that is surrounded by points with much higher loss relative to it. Conversely, a flat minimum is one that is surrounded by points with more or less similar loss. There have been several proposed measures of sharpness, some based on the worst-case increase in loss around a point [32], others around the expected increase in loss induced by a random perturbation in parameter space (as in the PAC-Bayes framework, see e.g. [33]).

In [34], the authors carry out an extensive survey of the correlation between different measures of the complexity of the hypothesis class and generalisation capability (formalised as low generalisation error). This large-scale experimental set-up covers 40 different complexity measures, training convolutional networks with variations along 7 hyperparameters: batch size, dropout probability, learning rate η, network depth, weight decay coefficient

λ, number of hidden units n and optimiser.

They find that sharpness-based measures are the measures best correlated with small generalisation error, with different sharpness measures exhibiting variation in their correlation across hyperparameter values. Another surprising finding is the strong correlation between optimisation-based measures and generalisation, especially the speed of convergence and gradient noise (variance of the gradients) toward the end of training.

Although it is one of the largest experiments to date on theory of generalisation, the authors acknowledge the limitations of such a study. First, that the empirical correlation of a complexity measure with good generalisation does not imply that the measure *causes* good generalisation. They mitigate this shortcoming through the scale and coverage of the experiments, but ultimately there remain some gaps due to computational constraints. They conclude that further experimentation is needed, covering more of the hyperparameter types used in practice, as well as larger architectures that better resemble systems in use today.

One way to exploit sharpness measures to get better generalisation is to deliberately search for flat minima. Two recent approaches with appealing theoretical justifications and seemingly strong empirical backing are sharpness-aware minimisation (SAM) [35] and stochastic weight averaging (SWA) [36]. While their exact mechanisms are out of the scope of this write-up, it is worth stating that the results of using these optimisers as drop-in replacements for plain SGD seem to be positive. For example, as of the time of this writing, SAM holds the state of the art accuracy of 96.08% on the CIFAR-100 image classification task [37].

An important criticism of the sharpness perspective on generalisation is that it is possible for sharp minima to generalise well. In [38], show that it is possible to reparameterise deep ReLU networks such that they exhibit sharp minima where they previously had flat minima, without changing their generalisation capability. This does not invalidate the empirical support for methods that seek out flat minima, but it does challenge the correctness of measures of sharpness in the literature.

3.6 Double descent and grokking

Aside from their surprising capability to generalise, deep neural networks exhibit some perplexing behaviours during training. One fruitful avenue of research concerns two such phenomena: double descent and late generalisation phenomena. Double descent was introduced in [39], and refers to the "double-dip" shape of the test error curve of overparameterised networks (see Figure 3.2). To reiterate, by overparameterised we mean networks whose number of parameters is larger than the number of inputs in the training dataset: $|\boldsymbol{\theta}| > |\mathcal{D}_{\text{train}}$; conversely, underparameterisation is $|\boldsymbol{\theta}| < |\mathcal{D}_{\text{train}}|$. The size of the network where the relationship $|\boldsymbol{\theta}| = |\mathcal{D}_{\text{train}}|$ holds is called the interpolation threshold.

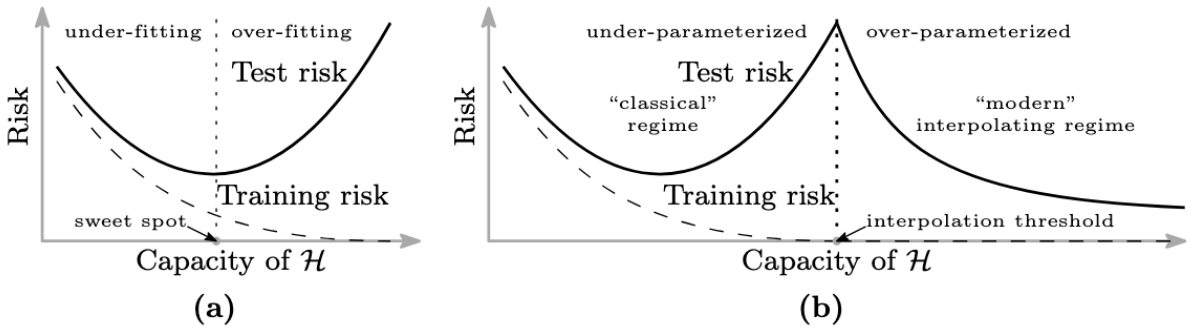


Figure 3.2: Reproduced from [39]. The x-axis is the capacity of $\mathcal{H}$, which the paper measures using the number of parameters of the network. **(Left)** the classic U-shaped figure for empirical risk (equivalent to the loss function of the network) exhibiting the bias-variance trade-off for under-parameterised models. Given a "sweet spot" for the number of parameters, smaller networks underfit the training data, showing higher training loss, while larger networks overfit it and achieve higher loss at test time despite lower training loss. **(Right)** networks larger than the interpolation threshold enter the interpolating regime, with test loss decreasing further despite training loss being approximately zero.

The double descent curve reconciles the apparent contradiction between the predictions of statistical learning theory (SLT) and the empirical success of overparameterised neural networks by claiming that they operate in different regimes. To reiterate, under SLT, neural networks are subject to a bias-variance trade-off, where the following statements are simultaneously true: (1) fitting the training data arbitrarily well does not lead to better performance on test data; (2) underfitting the training data leads to poor performance on both datasets. This leads to the U-shaped curve in 3.2. For networks which operate in this regime, the aim is to find the "sweet spot", the very bottom of the curve where the trade-off between training and test error is optimal.

With larger datasets and more efficient computation becoming available, neural network sizes have increased by several orders of magnitude, entering the so-called interpolation regime. At the interpolation threshold, a neural network trained to convergence memorises the data points and generalises poorly; this explains the peak on the right-hand side of the U-curve. Networks larger than this generalise better, and the test loss reaches new minima that are lower than the minimum in the initial dip of the U-curve.

A later paper, [40], finds that the same double-dip shape also applies epoch-wise: as a model is trained, at first its training and test losses decrease in tandem. Then, after some time, the training loss continues to decrease – eventually reaching 0 – as the test loss increases. Finally, the test loss decreases again while the training loss remains constant at 0.

A related phenomenon is termed "grokking" or late generalisation by [41]. Here, a series of experiments on a small algorithmic dataset outlines a surprising behaviour of neural networks during training. After some amount of optimisation steps, the training error decreases to ≈ 0; at the same time, the test error remains high, with the network only

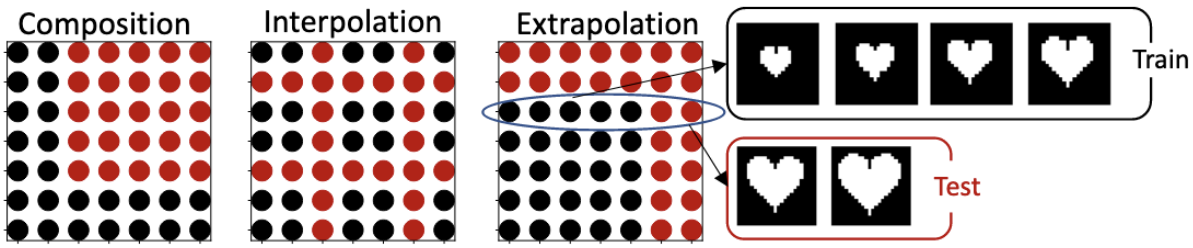


Figure 3.3: Reproduced from [42]. A systematic train-test split where the black dots are training data and the red dots are test data. In the extrapolation example, the test data comprises images displaying hearts that are larger than those in the training set.

doing as good as a random guess on the test set. Many more optimisation steps later, with the training error still at 0, the test error gradually decreases to ≈ 0. In some of the experiments, the difference between when a network converges to ≈ 0 training loss and when it achieves ≈ 0 test loss is striking: on the order of 10^3 additional optimisation steps are needed.

3.7 Out-of-distribution generalisation

So far, we've examined attempts to explain why networks generalise so well. But in practice, there are known limitations to networks' ability to generalise. Specifically, networks do very poorly when the test data is unlike the training data in some systematic way. Consider the training data points $\{(x_j, y_j)\}_{j=1}^{M}$ drawn independently and identically distributed from an underlying distribution $\mathbb{P}_{\text{train}}(X, Y)$. The term *distributional shift* refers to the situation where the test dataset is drawn from a different distribution $\mathbb{P}_{\text{test}}(X, Y)$ that is in some systematic way unlike $\mathbb{P}_{\text{train}}$.

In this setting neural networks generalise very poorly because they learn statistical correlations from the data that may not be robust across different environments. Ideally, we would like these models to have a mechanistic understanding of the world, i.e. one that is grounded in the mechanisms that give rise to observed data. If models had such an understanding, they would do well on unseen data because this data is essentially recombining known factors.

One way to create such a model is to work back from observed data to a set of underlying factors of variation (FOVs) that generate this data. This is the problem setting of [42], a large scale study of different learning approaches on what is sometimes referred to as the disentanglement task (because the aim is to disentangle the FOVs in order to learn them). In [42], the inputs are images generated by a combination of factors of variation, i.e. $\mathbf{x} = g(\mathbf{y})$ where $\mathbf{y}$ is a vector of factors of variation and g is an injective function. The task is then to recover $f(x) = g^{-1}(x) = y$.

The experiments in the paper follow a systematic split of the data into training and test data along the FOVs that compose an image. These splits are interpolation, extrapolation

and composition. For interpolation, given a factor of variation representing the size of a heart shape in an image, the training set might contain small hearts and large hearts, and the test set medium-sized hearts. For extrapolation, images of small and medium-sized hearts are trained on, and images of large hearts are to be predicted.

Composition refers to a combination of two factors of variation such that two particular values for the factors are present, but they do not co-occur. For example, given size and position in the image, the training set might contain small hearts in the bottom-left corner, small and large hearts in the top-right corner, but never large hearts in the bottom-left corner. What these splits aim to highlight is whether the predictor has learned the FOVs that generate the data. If it has done so, then it will use these FOVs to interpolate, extrapolate and compose data points that it has not seen before, thus generalising well.

The benchmark results are not encouraging: all the approaches that were tested had difficulties learning the FOVs, and when presented with FOV values that were outside the training distribution they predicted values that they had seen before, essentially reverting to the mean. None of the 2000 models they trained successfully inverted the generative mechanism $g(\mathbf{y}) = \mathbf{x}$.

Another concrete example of distributional shift comes from Koch et al. [43], who study the problem of objective robustness in the reinforcement learning (RL) context. The term objective robustness is used to refer to an agent that is otherwise capable, but is pursuing an objective other than the one it was rewarded for during training. This is distinct from capabilities robustness in that an agent can have robust capabilities while pursuing the wrong goal.

The paper is structured around multiple experiments with reinforcement learning agents in different settings. One of the settings, CoinRun, is a platformer game where the RL agent controls a player whose goal is to collect a coin while avoiding obstacles and enemies. During training, the coin is always at the end of the level, i.e. at the right-most part of the map. If during test time the coin is instead placed randomly throughout the map, the agent ignores it and goes to the end of the level anyway. Notably, this is not a capability robustness failure; the agent still knows to manage obstacles and enemies, and does so successfully.

In the training set, the features "coin" and "end of level" always co-occurred, and were rewarded together. When they no longer overlapped at test time, it became obvious that what the agent had learned was not to find the coin, but to go to the end of the level. Another example of the same phenomenon comes in the form of a procedurally generated maze, where the player must navigate to a piece of cheese. During training, the piece of cheese is always in the top-right corner of the maze. At test time, the cheese is placed randomly in the maze. The agent reliably navigates to the top-right of the maze.

In both cases, the model is learning spurious correlations from the training environment

and loses reward because those do not hold in the test environment. One way to mitigate this is to learn invariant predictors [44]. Given environments belonging to a perturbation set, $e \in \mathcal{F}$, we first introduce the concept of an equipredictive representation: a function $\Phi(x), x \in X$ with the property that for all environments in the perturbation set, $P_e(Y|\Phi(X))$ is the same. If some relationship between X and Y can be written in terms of an equipredictive representation Φ, then this relationship is invariant across environments, and models that learn this relationship are called invariant predictors. Because invariant predictors learn relationships that are robust across environments, they capture fewer of the spurious correlations that might result in poor generalisation.

Distributional shift has been identified as one of the key open problems for the safety of future AI systems. In [45], the authors note that the problem isn't just that systems make mistakes: it's also that they are overconfident in their predictions. In situations where their outputs are uncertain, we would like AI systems to be more conservative and to ask for human input. This is not always an option, especially for systems that work in time-sensitive domains. Systems that are overconfident and suffer from distributional shift are also prone to runaway effects where one mistake brings the system into a state it has not encountered during training, leading to even worse decisions, and so on.

3.8 Summary

This chapter has introduced the idea that neural networks generalise despite statistical learning theory predicting that they ought to have difficulty. We've covered a few of the interesting threads of research into generalisation, some with very different perspectives on a potential mechanism. We've also explored one of the limitations of today's networks and its implications for future machine learning systems.

What is missing is a unified theory for these seemingly disparate explanations. What is the "recipe" of generalisation? Whatever the solution is, it is clear that further experiments are needed to ascertain the merits of each explanation. In the rest of this paper, we aim to evaluate the robustness of a set of results at the intersection of approximation theory and the implicit bias of gradient descent, presented in [13]. In the next chapter, we present the results themselves, and in Chapter 5 we outline the outcomes of our experiments.

Chapter 4

Gradient descent is biased toward smooth functions

The main aim of this paper is to study and evaluate the results for univariate regression described in [13]. These results refer to the implicit bias of gradient descent when optimising mean squared loss on shallow wide networks on 1-dimensional numerical data. In this section, we introduce the problem formalism following the original paper, summarise their findings, and present the experimental set-up that was used to replicate these results.

4.1 Problem set-up

To start, consider in the general case a fully-connected neural network with one hidden layer of width n, d inputs and a single output. This is parameterised as:

$$f(\mathbf{x}, \theta) = \sum_{i=1}^{n} W_i^{(2)} \phi(\mathbf{W}_i^{(1)} \mathbf{x} + b_i^{(1)}) + b^{(2)}, \tag{4.1}$$

with ϕ being an activation function and $\mathbf{W}^{(1)} = (\mathbf{W}_1^{(1)}, \ldots, \mathbf{W}_n^{(1)})^T \in \mathbb{R}^{n \times d}$, where $\mathbf{W}_i^{(1)} = (W_{i,1}^{(1)}, \ldots, W_{i,d}^{(1)}) \in \mathbb{R}^d$, $\mathbf{W}^{(2)} = (W_1^{(2)}, \ldots, W_n^{(2)})^T \in \mathbb{R}^n, \mathbf{b}^{(1)} = (b_1^{(1)}, \ldots, b_n^{(1)})^T \in \mathbb{R}^n$, $b^{(2)} \in \mathbb{R}$ are the weights and biases of the hidden and output layers. We use $\theta = \text{vec}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, b^{(2)})$ to denote all the parameters of the network.

Let $(\boldsymbol{\mathcal{W}}, \mathcal{B})$ be a joint sub-Gaussian distribution: a distribution whose tails decay at least as fast as those of a Gaussian distribution. Then, the network's parameters are initialised as follows:

$$\begin{aligned} &(\mathbf{W}_i^{(1)}, b_i^{(1)}) = (\mathcal{W}, \mathcal{B}) \\ &W_i^{(2)} = \sqrt{1/n}\, \mathcal{W}^{(2)}, \quad b^{(2)} = \sqrt{1/n}\, \mathcal{B}^{(2)} \end{aligned} \tag{4.2}$$

Additionally, let the activation function be the rectified linear unit: $\phi(x) = \text{ReLU}(x) = \max(0, x)$.

The regression problem is defined on data $\{(\mathbf{x}_j, y_j)\}_{j=1}^M$ where $\mathcal{X} = \{\mathbf{x}_j\}_{j=1}^M$ are the inputs and $\mathcal{Y} = \{y_j\}_{j=1}^M$ are the targets. The loss function is the empirical risk

$$L(\theta) = \frac{1}{M} \sum_{j=1}^{M} \ell(f(\mathbf{x}_j, \theta), y_j), \tag{4.3}$$

where $\ell(f(\mathbf{x}, \theta), y) = \frac{1}{2}\|y - f(\mathbf{x}, \theta)\|_2^2$, making L the mean squared error.

$L(\theta)$ is minimised using full-batch gradient descent with a fixed learning rate η. Each optimisation step carries out the following update:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta) = \theta_t - \eta \nabla_\theta f(\mathcal{X}, \theta_t)^T \nabla_{f(\mathcal{X}, \theta_t)} L, \tag{4.4}$$

where $f(\mathcal{X}, \theta_t) = [f(\mathbf{x}_1, \theta_t) \ldots, f(\mathbf{x}_M, \theta_t)]^T$ is a vector containing network outputs for all training inputs, and $\nabla_{f(\mathcal{X}, \theta_t)} L$ is the gradient of the loss function with respect to the network outputs.

4.2 Results for univariate regression

The main result for univariate regression refers to a characterisation of the bias of gradient descent toward smooth functions.

Theorem 1. Assume a network with a single input, one hidden layer of n rectified linear units and a single output, using the parameterisation in 4.1 and the initialisation scheme in 4.2. Then, for any dataset $\{(\mathbf{x}_j, y_j)\}_{j=1}^M$ and support set $S = \text{supp}(\zeta) \cap [\min_j, x_j, \max_j x_j]$, after minimising the mean squared error using full-batch gradient descent, we have $\sup_{x \in S} \|f(x, \theta^*) - g^*(x)\|_2 = \mathcal{O}(\frac{1}{\sqrt{n}})$, where the function g^* is the solution to a variational problem defined as:

$$\begin{aligned} &\min_{g \in C^2(S)} \int_S \frac{1}{\zeta(x)} (g''(x) - f''(x, \theta_0))^2 dx \\ &\text{subject to } g(x_j) = y_j - u x_j - v, \ j = 1, \ldots, M. \end{aligned} \tag{4.5}$$

where $\rho(x) = \frac{1}{\zeta(x)}$ is a curvature penalty function, with $\zeta(x) = \int_{\mathbb{R}} |W|^3 p_{\mathcal{W}, \mathcal{B}}(W, -Wx) dW$.

To summarise this result, the error between the trained neural network and the solution to the variational problem is bounded, and this bound becomes tighter as the network is scaled up, following the relation $\frac{1}{\sqrt{n}}$. Thus, for wide enough networks, the error is ≈ 0, and the network approximates the function g^*.

There are several assumptions on which this result rests, and which are explored in more depth as part of the experiments:

- The data is linearly adjusted. That is, given the dataset $\{(x_j, y_j)\}_{j=1}^M$, the input to the neural network is $\{(x_j, y_j - u x_j - v)\}_{j=1}^M$, where $u, v \in \mathbb{R}$ are constants.

This detail is motivated by the invariance of the second derivative to addition of linear terms. Intuitively, it removes the component of the data that could easily be interpolated by a linear model.

- Given sufficiently many hidden units n and sufficiently small η, the optimisation process achieves zero training error on the data.

One special case concerns parameter initialisations for which the curvature penalty function $1/\zeta(x)$ is constant. For independent random variables $\mathcal{W}, \mathcal{B}$, $\mathcal{W} \sim \text{Unif}(-a_w, a_w)$ and $\mathcal{B} \sim \text{Unif}(-a_b, a_b)$ with $\frac{a_b}{a_w} \geq I$, the curvature penalty $1/\zeta$ is constant on $[-I, I]$. When the curvature penalty is constant, the variational problem in 4.5 is solved by the natural cubic spline function. We focus the experiments in the current paper on this special case, and recommend that further work investigates other non-closed form solutions to the variational problem.

It is also worth mentioning that for normal initialisation, the curvature penalty is not constant, but has a closed form that depends on the data and the parameter initalisation:

$$\zeta(x) = \frac{2\sigma_w^3 \sigma_b^3}{\pi \left(\sigma_b^2 + x^2 \sigma_w^2\right)^2}, \tag{4.6}$$

where $\mathcal{W} \sim \mathcal{N}(0, \sigma_w^2), \mathcal{B} \sim \mathcal{N}(0, \sigma_b^2)$.

The main result is generalised to activation functions other than ReLU provided they have particular characteristics. Specifically, the activation function ϕ should be k-homogenous: $\phi(ax) = a^k \phi(x), \forall a > 0$, as well as satisfy $L\phi = \delta$, where L is a linear operator and δ is the Dirac delta function. Then, there exists a function p such that $Lp \equiv 0$ which we can use to adjust the data $\{(x_j, y_j - p(x_j))\}_{j=1}^M$. The function g^* now solves:

$$\begin{aligned} &\min_{g \in C^2(S)} \int_S \frac{1}{\zeta(x)} [L(g(x) - f(x, \theta_0))]^2 dx \\ &\text{subject to } g(x_j) = y_j - p(x_j),\ j = 1, \ldots, M. \end{aligned} \tag{4.7}$$

with $\zeta(x) = p_{\mathcal{C}}(x) \mathbb{E}(\mathcal{W}^{2k} | \mathcal{C} = x)$. To recover the original variational problem, we observe that for $\phi(x) = \text{ReLU}(x)$, the linear operator L is the second derivative, i.e. $L\phi = \frac{d^2\phi}{dx^2}$ with $L\phi = \delta$.

An interesting corollary of the main result is that any curvature penalty function $\rho = 1/\zeta$ can be constructed by controlling the distribution from which parameters are initialised. This could have useful practical implications, because it would enable networks to be designed with a particular inductive bias.

Finally, although we do not focus on multivariate regression in this paper, it is worth mentioning that [13] present an analogous result for the multivariate setting, where the $\mathbf{x}$ is a d-dimensional vector. Here, the implicit bias of gradient descent is also toward a class of low-complexity functions, but this measure of complexity is no longer the curvature, and

does not have a straightforward interpretation. Similarly to the natural cubic spline case, for parameter initialisations that yield constant curvature penalties, networks converge to a type of function called a polyharmonic spline (see Theorem 8 in [13]).

4.3 Experimental set-up

Our experiments are designed to test the robustness of the results described in the previous section regarding the bias of gradient descent toward approximating natural cubic spline functions. A first set of experiments follows the set-up in [13] in order to replicate the results. Then, additional experiments examine whether the results hold under various changes to network architecture and hyperparameter choice. In this section, we present the details of which experiments were carried out, and what outcomes are predicted for these experiments according to Theorem 1 and its corollaries. We present our findings in Chapter 5.

4.3.1 Data

The characterisation in Theorem 1 does not depend on the type of dataset. In these experiments, 7 datasets are used, all corresponding to an elementary function. To replicate the conditions in the original paper, only 10 data points are used per training set. Experiments with more data are run as an ablation, and the findings are described in the next section. Data is normalised such that the training set lies in the interval $[-1, 1]$, and the points themselves are selected such that they exhibit interesting behaviour. The datasets are:

(a) *Sine*: $h : [0, 2\pi] \to [-1, 1], h(x) = \sin(x)$

(b) *Parabola*: $h : [-5, 5] \to [0, 25], h(x) = x^2$

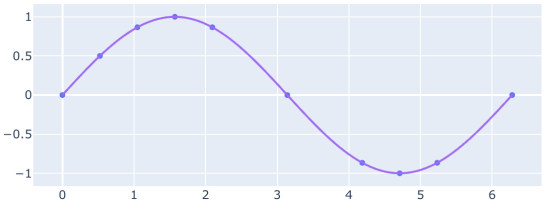
(c) *Piecewise polynomial*: $h : [-2, 2] \to [-\frac{2}{3\sqrt{3}}, 6]$,

$$h(x) = \begin{cases} x^2, & x < 0 \\ x^3 - x, & x \geq 0 \end{cases}$$

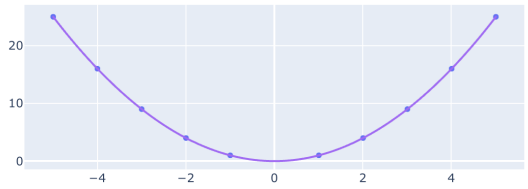
(d) *Chebyshev polynomial*: $h(x) = 16x^4 - 12x^2 + 1$. We study this function over $h : [-1, 1] \to [-\frac{5}{4}, 9]$ because of its interesting behaviour around the origin. The function is symmetrical, creating a rounded "W" shape (Fig. 4.1d).

(e) *Linear*: $h : [0, 9] \to [3, 21], h(x) = 2x + 3$

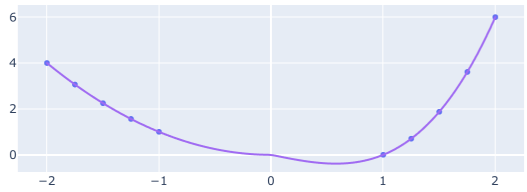
(f) *Constant*: $h : [0, 9] \to \{1\}, h(x) = 1$



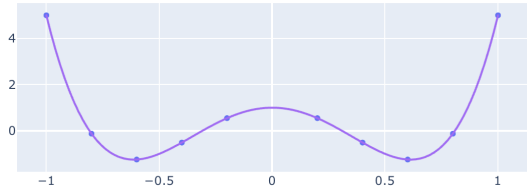
(a) $h(x) = \sin(x)$



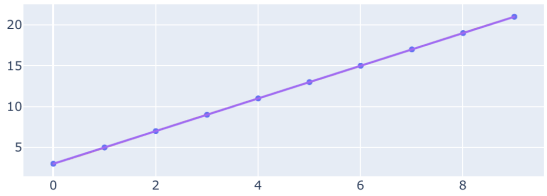
(b) $h(x) = x^2$



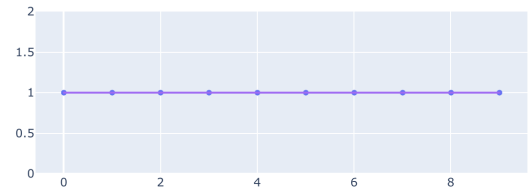
(c) $h(x)$ is a piecewise polynomial.



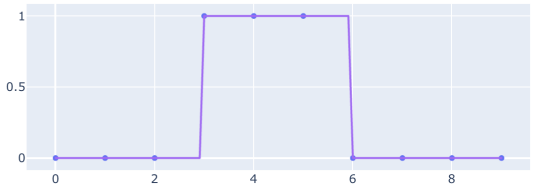
(d) $h(x) = 16x^4 - 12x^2 + 1$



(e) $h(x) = 2x + 3$



(f) $h(x) = 1$



(g) $h(x)$ is a piecewise function where the pieces are constant functions.

Figure 4.1: Plots of the datasets used in the experiments. Each set $\mathcal{X}$ consists of 10 data points, however the range of the values differs such that they capture interesting behaviours of the functions.

(g) *Square*: $h : [0, 9] \to \{0, 1\}$,

$$h(x) = \begin{cases} 0, & x \in [0, 3) \cup [6, 9] \\ 1, & x \in [3, 6) \end{cases}$$

To replicate the findings in Theorem 1 empirically, we run a set of "baseline" experiments. We also experiment with two generalisation tasks other than the baseline: interpolation and extrapolation. For interpolation, we remove the middle third of the original training set and use it as a test set. For example, we split the data set $\mathcal{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, into the training set $\mathcal{D}_{\text{train}} = \{0, 1, 2, 3, 7, 8, 9\}$ and the test set $\mathcal{D}_{\text{test}} = \{4, 5, 6\}$. For extrapolation, we remove the final third of the training set, which would give us $\mathcal{D}_{\text{train}} = \{0, 1, 2, 3, 4, 5, 6\}$ and $\mathcal{D}_{\text{test}} = \{7, 8, 9\}$.

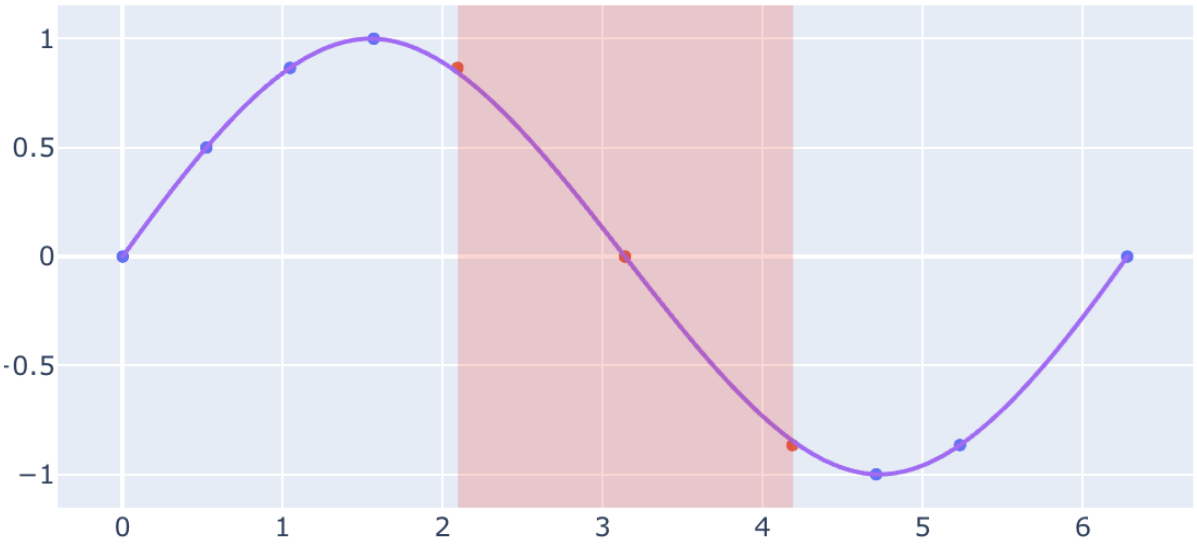


Figure 4.2: Sine interpolation dataset example. Training data in blue, test data in red, $\sin(x)$ ground truth. The training data here is the set $S_1 = \{0, \frac{\pi}{6}, \frac{\pi}{3}, \frac{\pi}{2}, \frac{3\pi}{2}, \frac{5\pi}{3}, 2\pi\}$. The test set is $S_2 = \{\frac{2\pi}{3}, \pi, \frac{4\pi}{3}\}$.

4.3.2 Network architectures

The results in [13] apply to shallow networks – networks with only one hidden layer. To simplify the presentation of the proofs, the authors carry out an anti-symmetric initialisation trick (ASI). ASI was first introduced in [46], who find that a component of the generalisation error is due to the non-zero output of the hypothesis at initialisation, $f(x, \theta_0) \neq 0$. With ASI, the units in the hidden layer are duplicated, and the network is initialised such that the hypothesis at initialisation is equal to 0 on the entire training data: $f(x, \theta_0) = 0, \forall x \in \mathcal{X}$:

$$f_{ASI}(\mathbf{x}, \mathcal{V}_0) = \sum_{i=1}^{n} \frac{\sqrt{2}}{2} \bar{V}_i^{(2)} [\langle \bar{\mathbf{V}}_i^{(1)}, \mathbf{x} \rangle + \bar{a}_i^{(1)}]_+ + \sum_{i=1}^{n} -\frac{\sqrt{2}}{2} \bar{V}_i^{(2)} [\langle \bar{\mathbf{V}}_i^{(1)}, \mathbf{x} \rangle + \bar{a}_i^{(1)}]_+ \equiv 0, \quad (4.8)$$

where $\mathcal{V}_0 = \text{vec}(\bar{\mathbf{V}}_i^{(1)}, \bar{\mathbf{V}}_i^{(1)}, \bar{\mathbf{a}}_i^{(1)}, \bar{\mathbf{a}}_i^{(1)}, \frac{\sqrt{2}}{2}\bar{\mathbf{V}}_i^{(2)}, -\frac{\sqrt{2}}{2}\bar{\mathbf{V}}_i^{(2)}, \frac{\sqrt{2}}{2}\bar{a}_i^{(2)}, -\frac{\sqrt{2}}{2}\bar{a}_i^{(2)})$ is the parameter vector at initialisation.

Because the hypothesis is 0 at initialisation, the variational problem in 4.5 becomes:

$$\begin{aligned} &\min_{h \in C^2(S)} \int_S \frac{1}{\zeta(x)} (h''(x))^2 dx \\ &\text{subject to } h(x_j) = y_j - u x_j - v, \ j = 1, \ldots, M. \end{aligned} \quad (4.9)$$

This means that the initial hypothesis no longer affects the bias of gradient descent, since the solution to the above problem doesn't depend on $f(x, \theta_0)$.

4.3.3 Experiments

Replicating the baseline experiments

Experiments are clustered around replicating the original results ("baseline"), probing their robustness (various ablations) and investigating the generalisation behaviour these networks on interpolation and extrapolation tasks. The baseline experiments are run on three main architectures: feedforward networks with one hidden layer, with and without the ASI trick, and networks with two hidden layers. Their sizes range from 10 to 10,000 hidden units (although for some experiments we did try up to 1 million units). Each run is repeated 3 times, and for the purposes of calculating the variational error the recorded value is averaged and the standard deviation is recorded. In practice, the variational error is calculated as:

$$L_{\text{var}}(f, g^*) = \sup_{x \in G} \|f(x, \theta^*) - g^*(x)\|_2. \tag{4.10}$$

A validation error is also recorded, defined as the L2 norm of the difference between the neural network and the ground truth:

$$L_{\text{val}}(f, h) = \|f(x, \theta^*) - h(x)\|_2. \tag{4.11}$$

Both errors are computed on an interval $G = [\min x, \max x], x \in \mathcal{X}$ which includes the data in the training set, but whose cardinality is equal or greater to $\mathcal{X}$. This is because we are interested in the behaviour of the network both on and off the training data.

Training is stopped when the following condition is true:

$$L(\boldsymbol{\theta}_{t+1}) - L(\boldsymbol{\theta}_t) \leq 10^{-8} \tag{4.12}$$

Additional experiments

To test the robustness of the results, we experiment across variations in some of the key components in the optimisation process. Some of these are directly targeting assumptions made by Theorem 1, such as having a sufficiently small step size and sufficiently many hidden units; others are simply designed to evaluate the behaviour of networks under perturbations in the experimental set-up.

Learning rate and learning rate schedules

We select the learning rate according to two different strategies, one of which is in keeping with the paper [13], and the other which we have found to work well in practice.

In Appendix A of [13], the authors describe the process of finding the learning rate η as follows: they start with a relatively large learning rate, and halve it until they reach a value such that loss decreases during training. For example, we might start with $\eta = 1$ and try $\eta \in \{1, 0.5, 0.25, 0.125, \dots\}$ until we find a value for which the loss decreases.

An alternative way to choose the learning rate is through a form hyperparameter optimisation – for example, grid search. Grid search involves trying out multiple values for the parameter, usually on a logarithmic scale: $\eta \in \{1, 0.1, 0.001, 0.0001, \dots\}$ and then choosing the one that does best according to some metric. This is usually performed on a validation set, however we use the training set directly.

Learning rate schedules are a way to modify the learning rate during training, usually such that the value decreases from or oscillates around an initial value. We apply learning rate decay, which reduces the learning rate by one half whenever the training loss plateaus. A plateau is defined as a change in training loss of less than 10^{-4} over a span of 100 steps/epochs. In other experiments, we apply a cosine annealing schedule [47] whereby the learning rate oscillates around an initial value over a pre-specified number of epochs (we again use 100).

Model size and architecture

Model size is a key hyperparameter to evaluate, because the main result is connected to the number of units in the hidden layer of the network. In [13], the number of hidden units is $n \in \{10, 160, 640, 2560, 10240\}$, sometimes with other intermediate values in the interval $[10, 10240]$. In the experiments, we try to evaluate the behaviour of much larger networks, and go up to 10^6 hidden units. Here it is worth mentioning that for networks that use the ASI trick, the number of parameters is $|\theta| = 6n$, whereas for non-ASI it's $|\theta| = 3n$. We typically use powers of 10 for the hidden units: $n \in \{10, 100, 1000, \dots\}$, though for most experiments we also 50, 500 etc.

We additionally train a deeper, 2-hidden layer network with $n \in \{10, 100, 150, 500\}$ total hidden units (i.e. not n units per hidden layer). Learning rates for these models differ from the ones for the shallow networks (ASI and non-ASI), and we find them via learning rate sweep as described above.

Optimiser

The optimiser in [13] is full-batch gradient descent. For the additional experiments, we maintain the full-batch set-up and try Adam [48] with $\beta_1 = 0.9, \beta_2 = 0.999$ and gradient descent with momentum [49], with $\mu = 0.9$. We do not perform any hyperparameter optimisation.

Non-linearity

A priori we should not expect the natural cubic spline solution to hold for activation functions other than ReLU. This is because for other non-linearities, the variational problem in 4.5 is changed to include a different linear operator L, as detailed in Section 4.2. In the experiments, we replace ReLU with one of the following:

- Gaussian Error Linear Unit (GELU), introduced in [50] as:

$$\text{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\Big[1 + \text{erf}(x/\sqrt{2})\Big]$$

- Leaky ReLU, parameterised by a constant α :

$$R(x) \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

- Exponential Linear Unit (ELU), defined as:

$$R(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leq 0 \end{cases}$$

- Sigmoid: $R(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent, $R(x) = \tanh(x)$

Parameter initialisation

Parameter initialisation affects the bias of gradient descent in two ways. First, through curvature penalty function, which in turn controls the solution to the variational problem. Then, for networks that to not use ASI, they also determine the value of $f(x, \boldsymbol{\theta}_0)$. We study the case where parameters are initialised uniformly in a given interval, resulting in a constant curvature penalty. The motivation for this is two-fold: first, uniform initialisation is the default in PyTorch, a popular deep learning framework [51]; second, for uniform curvature penalty, we know that the solution to the variational problem is a natural cubic spline.

There is, however, a nuance to the uniform initialisation. As a reminder, given a network whose weights initialised uniformly in the interval $[-a_w, a_w]$ and biases initialised uniformly in $[-a_b, a_b]$, the curvature penalty is constant on the interval $[-I, I], I \leq \frac{a_b}{a_w}$. In this interval, the solution to Eq. 4.5 is a natural cubic spline; outside the interval, the curvature depends on x, and the solution is different. So in practice, if we initialise the parameters such that, say, ζ is constant on $[-0.5, 0.5]$, for any data lying outside this interval we should expect $f(x, \theta^*)$ to not track the natural cubic spline.

We run experiments as follows:

(a) all baseline experiments use uniform initialisation $\mathcal{W} \sim U[-1, 1], \mathcal{B} \sim U[-2, 2]$. This makes the interval on which the curvature penalty is constant $I = [-2, 2]$. With data normalised to lie in $[-1, 1]$, the solution to Eq. 4.5 is a natural cubic spline.

(b) unit normal initialisation: $\mathcal{W} \sim \mathcal{N}(0, 1), \mathcal{B} \sim \mathcal{N}(0, 1)$. For normal initialisation, the curvature penalty depends on the data as in 4.6, so we should not expect the

variational problem to be solved by the natural cubic spline for these experiments.

(c) loosely distributed weights, tightly distributed biases: $\mathcal{W} \sim U[-2, 2], \mathcal{B} \sim U[-1, 1]$, with ζ constant on $I = [-0.5, 0.5]$. Because the data is in $[-1, 1]$, the natural cubic spline is the solution to Eq. 4.5 only on the sub-interval I; outside that interval we expect the network to not approximate the spline.

Adjusting data

Theorem 1 holds for data that is adjusted linearly. The motivation for adjusting the data is that the second derivative is invariant to addition of linear terms. In one of the appendices in [13], the authors show that the theorem also approximately holds for the original data.

More data

The numerical experiments in Appendix A of [13] seem to use very few data points – maybe even 5 for some of the plots –, which might be a limitation. In our experiments, we consider up to 50 data points. Some are selected because they are interesting values for the function the dataset represents, and others are generated uniformly in the interval that spans the data. An extension to this could be generating data points non-uniformly, i.e. with more points clustered in a region of the interval.

Loss function

Following the recommendations in Appendix O of [13], we carry out several experiments with loss functions other than mean squared error. Specifically, we use L_1-loss (mean absolute error) as well as the Huber loss [52], where the latter is defined as:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{for } |y - f(x)| \leq \delta \\ \delta \cdot \left(|y - f(x)| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases} \tag{4.13}$$

4.3.4 Hardware and software

All experiments were run on a high-performance computing cluster. Most of the small models did not benefit from running on a GPU, so those experiments were run on CPU nodes only: Intel Xeon Icelake with 6760 MiB of RAM per CPU. Larger networks were trained on nodes running AMD EPYC 7763 64-Core Processor 1.8GHz, 1000 GiB RAM, and NVIDIA A100-SXM-80GB GPUs. Networks were implemented using PyTorch Lightning[53] and experiments were tracked using the Weights and Biases platform[54]. We use an implementation of the natural cubic spline in SciPy[55]. The source code is available on Github[1], and the experiments themselves are publicly available via a dashboard [2].

[1] `https://github.com/inwaves/implicit-bias-of-shallow-wide-nets`
[2] `https://wandb.ai/inwaves/gen2/`

Chapter 5

Findings

Having described the experimental setup used in this paper, in this chapter we present our findings. We start with the outcomes of our attempts to replicate the numerical experiments in Appendix A of [13], which are concerned with the predictions of Theorem 1. Then, we discuss our findings on experiments which vary parts of the setup as presented in the previous chapter. Finally, detail our results on the interpolation and extrapolation tasks.

5.1 Replication

Theorem 1 holds empirically for the sine, parabola, and piecewise polynomial datasets. That is, as the number of hidden units n is increased we see a decrease in the variational error that is roughly proportional to $\frac{1}{\sqrt{n}}$. Figure A.8 shows a log-log plot of the number of units in the hidden layer against the variational error. For each of the plots, the data points are represent an average of three training runs, plotted together with their standard deviations. The orange line in the plots is a linear regressor fit to the data points. The slope is similar to $y = kx^{-0.5}$, and the exponent deviates between ± 0.1. This is consistent with results in Appendix A of [13].

There are four datasets for which the error between the network and the natural cubic spline did not decrease according to Theorem: the constant, linear, square and Chebyshev polynomial datasets.

For the constant and linear functions, there is a straightforward explanation – these function are very simple. Looking at the plot for the constant function in figure 5.2, we see that the variational error for small values of n is already much smaller than the same sized network on other datasets: 0.004 vs. 0.4 on sine. (On some runs, this error has been as low as 4×10^{-7}.) We also see that it decreases much slower than $\frac{1}{\sqrt{n}}$. This is consistent with the idea that the dataset is very simple to interpolate linearly – it actually is a line – which means that we get a very good fit from very small networks, and bigger networks do

not do much better. In fact, these results are reported on data that *has not been linearly adjusted*, because to linearly adjust the constant/linear datasets is equivalent to:

$$\begin{aligned} \{(x_j, y_j)\}_{j=1}^M \to \{(x_j, y_j - ux_j - v)\}_{j=1}^M \\ \text{with } y_j = ux_j - v \text{ for some } u, v \in \mathbb{R}, \end{aligned} \tag{5.1}$$

which results in the dataset $\mathcal{X} = \{(x_j, 0)\}_{j=1}^M$. Because of the anti-symmetric initialisation trick, we know that $f(x, \theta_0) = 0, \forall x \in X$, which means that the loss is 0 without any training at all. We additionally know that the natural cubic spline can trivially represent a linear or constant function, because it is a piecewise polynomial function whose pieces are cubic polynomials, i.e.:

$$S_j(x) = a_j x^3 + b_j x^2 + c_j x + d_j, \tag{5.2}$$

for which $S_j(x) = 0$ and $S_j(x) = c_j x + d_j$ are special cases.

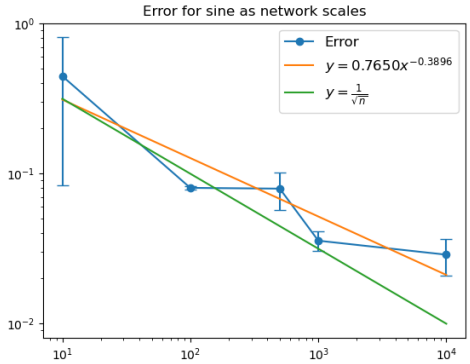
For non-linearly adjusted constant and linear datasets, although the variational error is small, it is non-zero. This is consistent with the analysis in [46], who find that part of the generalisation error is due to the non-zero value of the hypothesis at initialisation.

It's not entirely clear what influences the variational errors on the square and Chebyshev polynomial datasets, both of which decrease at a slower rate than $\frac{1}{\sqrt{n}}$. One likely explanation that is consistent with Theorem 1 in [13] is that the results hold for "sufficiently large n", which for this dataset is larger than any of the values we've experimented with. It's also worth noting that even the larger networks, when applied to this dataset, do not reach 0 training error within the finite time they were trained.

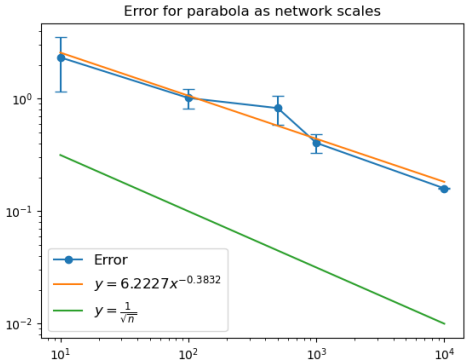
Another particularity of the square function is that it is the only function which the natural cubic spline doesn't perfectly interpolate, i.e. outside the data points, the spline does not match the original function. There reason for this is that the square function is discontinuous:

$$\begin{aligned} \lim_{x \to 3^+} sq(x) = 1 \\ \lim_{x \to 3^-} sq(x) = 0, \end{aligned} \tag{5.3}$$

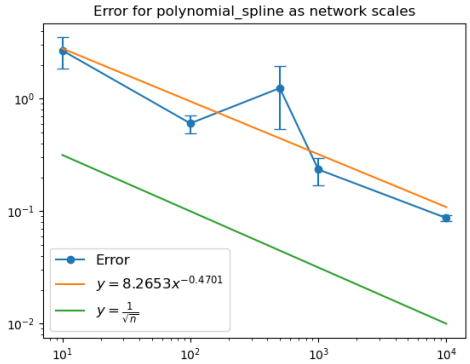
whereas the natural cubic spline $S \in C^2[0, 9]$. This means that the spline can track the square function arbitrarily closely on the two intervals where $sq(x) = 0$ and $sq(x) = 1$, respectively, but it cannot match it at the two discontinuities, where the "bump" starts and ends.



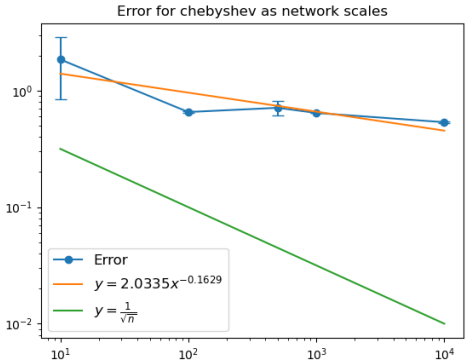
(a) Sine dataset, with network sizes ranging from 10 to 5000 hidden units.



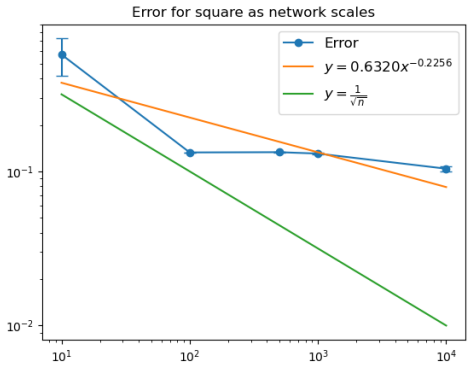
(b) Parabola dataset, with network sizes ranging from 10 to 10000 hidden units.



(c) Piecewise polynomial dataset, with network sizes ranging from 10 to 10000 hidden units.

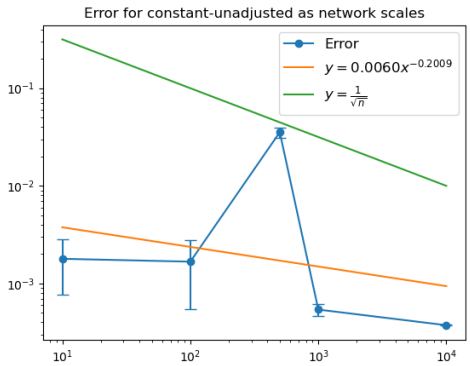


(d) Chebyshev polynomial dataset, with network sizes ranging from 10 to 5000 hidden units.

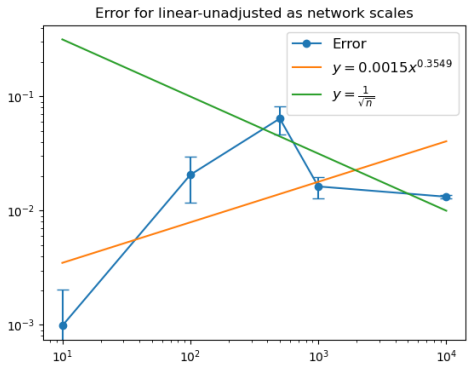


(e) Square dataset, with network sizes ranging from 10 to 10000 hidden units.

Figure 5.1: Results of the baseline experiments on univariate regression. Each image is a log-log plot of the number of units n against the variational error – the difference between the neural network and the natural cubic spline function. The results state that as the network is scaled up, the error decreases as $\frac{1}{\sqrt{n}}$.



(a) Sine dataset, with network sizes ranging from 10 to 5000 hidden units.



(b) Parabola dataset, with network sizes ranging from 10 to 10000 hidden units.

Figure 5.2: Theorem 1 does not seem to replicate for datasets that can be perfectly fit by a linear regressor, $y = ux+v$. Here, the datasets generated by $h(x) = 0$ and $h(x) = 2x+3$ are fit with very low loss even by small networks, $n = 10$. For the constant dataset, error decreases much more slowly than $kx^{-0.5}$. For the linear, error increases, then falls again.

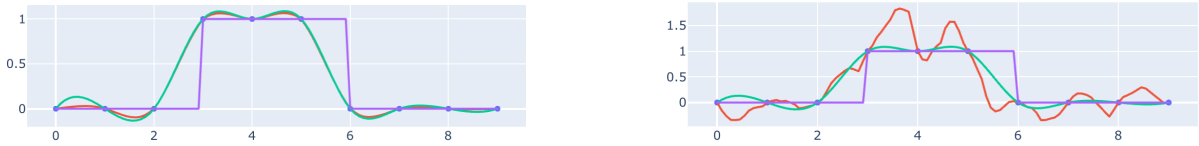
5.2 Other experiments

Learning rate and learning rate schedules

As a reminder, the learning rate in [13] is determined by starting with a larger learning rate and halving it until a value is found that enables the training loss to decrease. We find that this method is not effective in training the network to convergence. A typical outcome is that the largest learning rate that does so only allows the loss to decrease down to a plateau, where it remains until the early stopping condition kicks in or until the computational budget is exhausted. This result is robust across number of hidden units. Further halving the learning rate from the first value that causes a plateau only leads to a different, lower plateau. Eventually, the step size becomes small enough for the network to converge.

We also observed that from one run to another, under the same experimental conditions, one network converges while another identical network does not. One of the assumptions in Theorem 1 is that the networks achieve 0 training loss. In practice, this is very difficult to attain, with many networks plateauing above what could be considered their best performance. In the interest of reporting accurate results, we take these instances into account when calculating the variational loss.

We performed learning rate sweeps for some of the experiment configurations and found that the optimal learning rate depends on both architecture and dataset. Additionally, a good heuristic seems to be $\eta = 1/n$, with smaller networks skewing even smaller (e.g. $\eta = 1/2n$).



(a) Shallow ASI network.

(b) Deeper non-ASI network.

Figure 5.3: Difference in inductive bias for ASI shallow networks and deeper networks of the same total number of parameters.

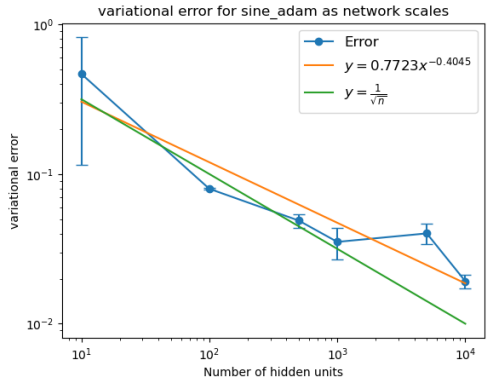
Model size and architecture

We find that wider models train very slowly comparative to their smaller counterparts. Using the early stopping condition in Eq. 4.12, large models often run on the order of 100000 epochs of training, while still achieving worse training and variational losses than their shallower counterparts. We carried out a series of tests with a fixed training duration of 100000 epochs for all networks and found that there are marked differences in wall-clock time taken by networks of different sizes.

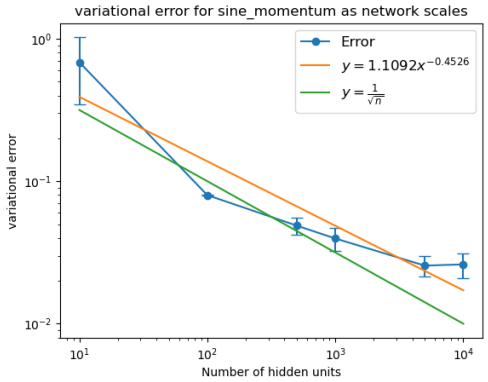
In Table A.1 we detail the findings regarding runtime for networks on the square dataset, with sizes ranging from 10 to 10^6 hidden units. There is a six-fold increase in training time from the smallest to the largest network, and a more than 2x increase from the second-largest network to the largest network we tried.

At the same time, the loss seems to decrease from 0.04 for just 10 hidden units to 2×10^{-7} for 160 units, then steadily increase up to a maximum of 0.009 for the largest network. We find that this is a robust training dynamic for wider networks: they need many more epochs to converge to good training loss than shallower networks. This is counter-intuitive, because for deeper networks the opposite is true: larger networks converge to better loss overall, and they do so faster than smaller networks.

Regarding deeper networks, we find that the predictions from Theorem 1 hold up to $n = 150$ for the 2-hidden layer network. Networks larger than this converge to functions that are less smooth, which no longer approximate the natural cubic spline. This result is robust across datasets, which points to a change in inductive bias as the network becomes deeper. It's worth noting that the deeper networks achieve better training loss than their shallow counterparts – they just don't match the natural cubic spline off-data. Figure 5.3 shows an example on the square dataset. For the full variational error results on all datasets, see Appendix A.2.



(a) Applying the Adam optimiser on the sine dataset.



(b) Applying SGD with momentum on the sine dataset.

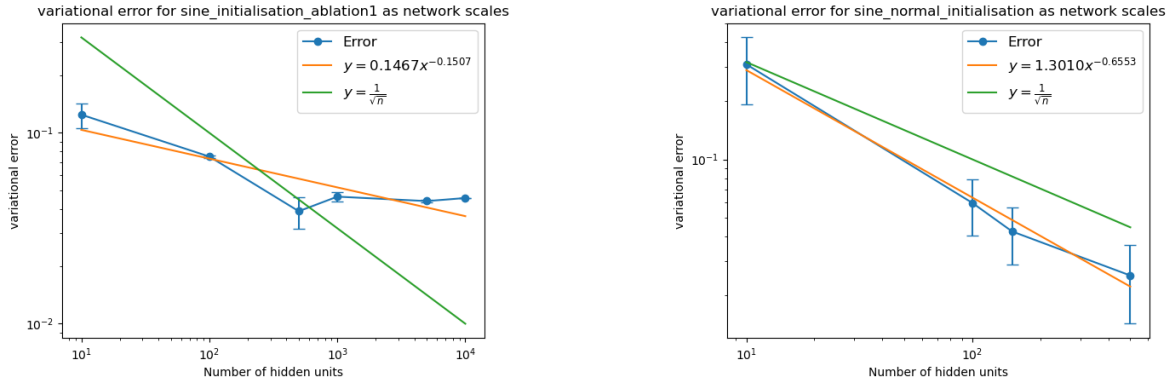
Figure 5.4: The predictions of Theorem 1 seem to be robust to changes in optimiser. With both Adam and SGD with momentum, the trend is very close to the $kx^{-0.5}$ line. The large standard deviation for the smallest network comes from different seeds converging to widely different variational losses, despite the same experimental set-up.

Optimiser

The results are robust to changes in optimiser. Figure 5.4 shows a decrease in the variational error that is proportional to $\frac{1}{\sqrt{n}}$ as the network size is increased. The networks trained faster with Adam, and on some occasions to lower training and variational loss relative to full-batch gradient descent.

Non-linearity

On the sine dataset, the variational error decreases as $\frac{1}{\sqrt{n}}$ as we increase the number of hidden units for all the activation functions we tried. This is in a sense surprising, because only the leaky ReLU function's second derivative is a Dirac delta function (see 4.2). For the other nonlinearities, we should expect the variational problem 4.5 to have a slightly different solution, and so we should get larger error. But in practice it looks like the fit is a natural cubic spline, and the error bound still holds (see Appendix A.2).



(a) $\mathcal{W} \sim U[-2, 2], \mathcal{B} \sim U[-1, 1]$.

(b) $\mathcal{W} \sim \mathcal{N}(0, 1), \mathcal{B} \sim \mathcal{N}(0, 1)$.

Figure 5.5: Change in variational error as networks are scaled up. The networks still track the natural cubic spline function, but the error decreases much slower for the uniform distribution.

Parameter initialisation

One surprising finding is that under initialisation setting (b) – unit normal initialisation – the results from Theorem 1 are robust (Fig. 5.5), despite the curvature penalty not being constant. For the unit normal distribution, equation 4.6 simplifies to:

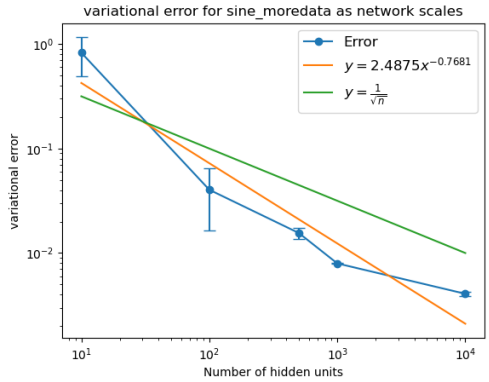
$$\zeta(x) = \frac{2}{\pi(x^2 + 1)^2}$$

which even for relatively small values of $x \in [-1, 1]$ is non-negligible. To better understand why this is the case, in future a more general replication of Theorem 1 should solve the variational problem directly, and measure the error between the solution and a known solution for other initialisations like the natural cubic spline.

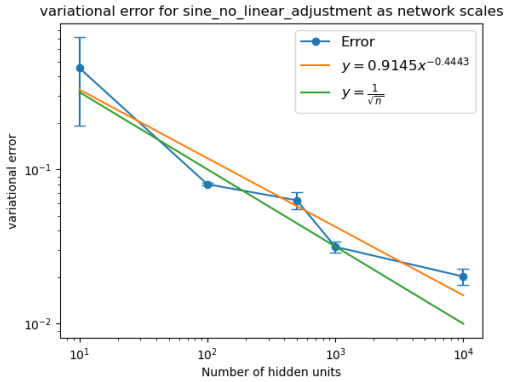
For initialisation (c), the interval where ζ is constant is $I = [-0.5, 0.5]$, while the input data is $x \in [-1, 1]$. Here, we should expect that the natural cubic spline is not the solution to the variational problem over the entire support S in equation 4.5, but only over I. We still observe the neural network approximating the natural cubic spline, albeit with slower decreasing error as the network is scaled up than the theory predicts (Fig. 5.5a). This could be accounted for by the fact that most of the error between the network and the natural cubic spline accumulates on the interval $S \setminus I = J = [-1, -0.5) \cup (0.5, 1]$ where the curvature penalty is not constant. Here again it would be useful to directly compute the solution to the variational problem and calculate the error against the piecewise function:

$$g^*(x) = \begin{cases} g_1^*(x) & x \in I \\ g_2^*(x) & x \in J \end{cases}$$

where $g_1^*(x)$ is the natural cubic spline and $g_2^*(x)$ is the solution for Eq. 4.5 on J.



(a) More data.



(b) Non-linearly adjusted data.

Figure 5.6: The predictions of Theorem 1 hold under various modifications to the input data. On the left, we use 50 data points instead of 10, and the decrease in error happens slightly faster than $\frac{1}{\sqrt{n}}$. On the right, we do not linearly adjust the original 10 data points.

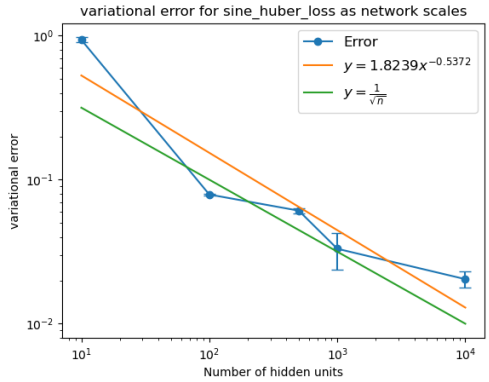
Adjusting data

On the sine dataset, the result seems to be robust to not adjusting the data linearly. In Appendix K of [13], the authors prove that Theorem 1 is still approximately correct on unadjusted data, and upper bound the distance between the solution on adjusted data and the solution on unadjusted data. Figure 5.6b shows the decrease in variational error as the network's hidden units are increased. This modification is robust across different datasets, in the sense that for the sine, parabola and piecewise polynomial datasets the error still decreases as expected, while for the Chebyshev polynomial and square datasets it does not.

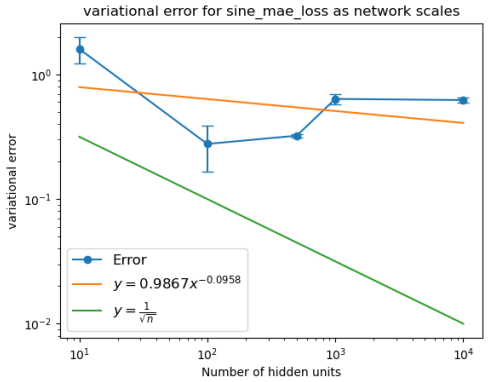
As discussed in an earlier section, for the constant and linear datasets adjusting the data is not a useful experiment. The ASI network, whose hypothesis at initialisation is 0, always converges to 0 loss; it doesn't actually require any training. The non-ASI shallow network and the deeper network converge to very small losses, on the order of 10^{-13}.

More data

Adding more data has the effect of improving the variational error faster than $\frac{1}{\sqrt{n}}$, as in Fig. 5.6a. In a sense, this improvement masks what we actually want to see: the implicit bias of the optimiser. Adding more data means that the network fits more data points which happen to be on the spline (since the latter interpolates the data), potentially obscuring the bias of the optimiser.



(a) Huber loss.



(b) MAE loss

Figure 5.7: Results for different choices of loss function. For Huber loss, the error decreases as expected, potentially because the choice of its hyperparameter δ makes it equivalent to MSE. For MAE, the error decreases, then increases and remains constant.

Loss function

The results are robust under Huber loss, but not under MAE (Fig. 5.7). Because the Huber loss is defined piece-wise (Eq. 4.13), using a $\delta = 1$ it is possible that in most instances the error $-1 \leq y - f(x) \leq 1$, which makes the Huber loss equivalent to MSE. Further work is needed here to evaluate whether this is the case, and whether for different values of δ the results no longer hold.

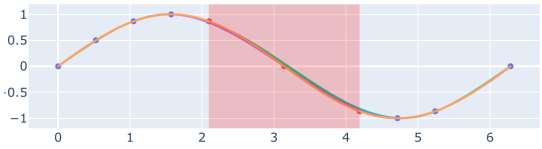
Double descent behaviour

For one of the experiment configurations, we reliably observe a double descent dynamic with regards to the validation error. As a reminder, this is the mean squared error between the neural network and the ground truth (see Eq. 4.11). This occurs on the Chebyshev polynomial dataset, for shallow networks using the ASI trick with $n = 100$ hidden units. An initial experiment with these hyperparameters crashed for unrelated reasons, and while investigating we noticed that its validation error curve did not monotonically decrease (Fig. 5.8a). After resolving the issue, we reran the same experiment three times, all of which show a form of epoch-wise double descent behaviour on the validation loss (Fig. 5.8b). At $|\theta| = 600$ parameters, the network should be in the overparameterised regime relative to the size of the dataset $|\mathcal{D}| = 10$. A similar result is observed for shallow networks which do not use ASI (see Appendix A.2), but not for the deeper, 2-hidden layer networks. We did not observe this for other datasets, however we think it worthwhile exploring this in further work.

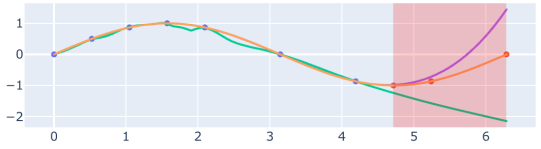
(a) Initial experiment which exhibits double descent-like behaviour with two peaks. Peaks occur at approximately 800,000 and 1,300,000 epochs, respectively.

(b) Additional experiments show that the second peak is followed by a long period of monotonical decrease in validation loss.

Figure 5.8: Double descent behaviour on one of the baseline experiments. For reference, smaller networks often finish within 50,000 epochs of training, which makes the experiments above 2 orders of magnitude longer.



(a) Under interpolation, the network tracks the natural cubic spline function on the test set highlighted in red in the middle of the graph.



(b) For extrapolation, the natural cubic spline and the neural network exhibit completely different behaviours outside the support S.

Figure 5.9: Interpolation and extrapolation experiments on the sine dataset. The orange line is the ground truth, $f(x) = \sin(x)$; the purple line is the natural cubic spline, which in the interpolation case overlaps with the ground truth on the entire graph, but diverges in the extrapolation case. The green line is the neural network.

5.3 Interpolation and extrapolation

For interpolation, the results are consistent with Theorem 1: the neural network approximates the natural cubic spline even on the test set, and the error between the two decreases as the network is scaled up. Since Theorem 1 makes no mention of size of the dataset, we could plausibly use as few as 3-4 training data points that are arbitrarily far apart to evaluate the fit of the network to the natural cubic spline; we leave this as future work. Please see Appendix A.2 for the full results on interpolation.

For the extrapolation case, the results in Theorem 1 does not hold. This is because the variational problem in 4.5 is only defined over the support $S = \text{supp}\,(\zeta) \cap [\min x_j, \max x_j]$, so for any values of $x < \min x_j$ or $x > \max x_j$ we do not actually have a characterisation of the bias. We find that on the extrapolation interval, the variational error does not come from the network failing to track the natural cubic spline well. Instead, they diverge completely. For example, on sine dataset, the natural cubic spline curves up, while the network is a negatively sloped line (Fig. 5.9b).

Chapter 6

Conclusion and further work

In this paper, we find empirical backing for the claim that shallow neural networks trained on the univariate regression task using full-batch gradient descent are biased toward natural cubic splines when their parameters are initialised using a particular scheme. We find that this result is robust across a range of different experimental setups covering different optimisers, nonlinearities, number of hidden units and loss functions, but that it does not hold for deeper networks.

We believe that this sort of practical evaluation of theoretical predictions in the literature on generalisation is useful, as it reveals which explanations have the most predictive power, as well as how robust they are. While the characterisation we replicate here is not a comprehensive answer to the mystery of generalisation, it is a useful starting point for further work to develop a better understanding of how and why deep neural networks work. In the rest of this section, we'd like to outline, in order of importance, a series of proposed directions for future research.

First and most consequentially, we'd like to know if a similar sort of characterisation holds for other deep architectures. Deep learning's recent success with large language models [10, 4] has lent more credibility to the scaling hypothesis [56], which states that to increase performance, all we need to do is use larger versions of the same architecture. Overall, [56] find that scaling the model up is more important than the precise value of the depth/width hyperparameters, and there are arguments for both increasing depth first, then width, as well as vice-versa. Although it is unclear whether there is an optimal scaling strategy, it seems very likely that future networks will be very large relative to the networks for which we can get analytical results regarding the inductive bias. This motivates additional empirical analyses of the biases of large networks.

Understanding the inductive bias of these networks is important beyond the performance of systems that use them. Today, more and more decisions are made based on output from machine learning systems. We would like to ensure that these systems are aligned with human preferences and that they do not behave in ways that causes humans or other

sentient beings to come to harm. There are already examples of ML systems violating principles of fairness, by reflecting or amplifying bias present in their training data [57]. These shortcomings are something we should strive to correct as a field.

Second, it would be valuable to examine the results from [13] regarding multivariate regression. In real-world deployments, it's relatively rare that a neural network would have 1-dimensional input; more often than not, there are hundreds, if not thousands of dimensions to the input and output. This makes results regarding multi-dimensional inputs much more practically relevant and potentially actionable. While useful as a technical analysis, the univariate regression case is unlikely to be directly useful for design choices for systems in production today. We are particularly interested in approaching the results for multivariate regression in [13] using the compositionality framework introduced in [42] to try and shed light on whether shallow networks learn anything like the factors of variation that give rise to a data point.

Third, there are various ways to extend and improve the analysis we carry out here, for example by evaluating the bias of the same networks on the classification task. It would also be useful to plot intermediate outputs of the neural network and look at how the fit to the natural cubic spline changes with training. To better understand why wide networks train more slowly, we could try plotting the variational error as a function of both number of epochs and network size. Finally, it would be interesting to increase the gap between training points in the interpolation setting to see whether the fit of the network to the natural cubic spline deteriorates.

Bibliography

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[2] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II." `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[3] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes, *et al.*, "Photorealistic text-to-image diffusion models with deep language understanding," *arXiv preprint arXiv:2205.11487*, 2022.

[4] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[5] O. Lange and L. Perez, "Traffic prediction with advanced graph neural networks."

[6] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[7] A. Davies, P. Veličković, L. Buesing, S. Blackwell, D. Zheng, N. Tomašev, R. Tanburn, P. Battaglia, C. Blundell, A. Juhász, *et al.*, "Advancing mathematics by guiding human intuition with ai," *Nature*, vol. 600, no. 7887, pp. 70–74, 2021.

[8] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, *et al.*, "A generalist agent," *arXiv preprint arXiv:2205.06175*, 2022.

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[11] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[12] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[13] H. Jin and G. Montúfar, "Implicit bias of gradient descent for mean squared error regression with wide neural networks," *arXiv preprint arXiv:2006.07356*, 2020.

[14] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, vol. 4. Springer, 2006.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[16] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.

[17] J. H. Ahlberg, E. N. Nilson, and J. L. Walsh, "The theory of splines and their applications," *Mathematics in science and engineering*, 1967.

[18] Wikiversity, "Cubic spline interpolation — wikiversity,," 2022. [Online; accessed 12-April-2022].

[19] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[20] S. Gunasekar, J. Lee, D. Soudry, and N. Srebro, "Characterizing implicit bias in terms of optimization geometry," in *International Conference on Machine Learning*, pp. 1832–1841, PMLR, 2018.

[21] B. Neyshabur, R. Tomioka, and N. Srebro, "In search of the real inductive bias: On the role of implicit regularization in deep learning," *arXiv preprint arXiv:1412.6614*, 2014.

[22] S. Oymak and M. Soltanolkotabi, "Overparameterized nonlinear learning: Gradient descent takes the shortest path?," in *International Conference on Machine Learning*, pp. 4951–4960, PMLR, 2019.

[23] P. Savarese, I. Evron, D. Soudry, and N. Srebro, "How do infinite width bounded norm networks look in function space?," in *Conference on Learning Theory*, pp. 2667–2690, PMLR, 2019.

[24] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[25] R. Parhi and R. D. Nowak, "What kinds of functions do deep neural networks learn? insights from variational spline theory," *arXiv preprint arXiv:2105.03361*, 2021.

[26] F. Williams, M. Trager, D. Panozzo, C. Silva, D. Zorin, and J. Bruna, "Gradient dynamics of shallow univariate relu networks," *Advances in neural information processing systems*, vol. 32, 2019.

[27] B. Woodworth, S. Gunasekar, J. D. Lee, E. Moroshko, P. Savarese, I. Golan, D. Soudry, and N. Srebro, "Kernel and rich regimes in overparametrized models," in *Conference on Learning Theory*, pp. 3635–3673, PMLR, 2020.

[28] A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: Convergence and generalization in neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[29] J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, "Wide neural networks of any depth evolve as linear models under gradient descent," *Advances in neural information processing systems*, vol. 32, 2019.

[30] L. Wu, Z. Zhu, *et al.*, "Towards understanding generalization of deep learning: Perspective of loss landscapes," *arXiv preprint arXiv:1706.10239*, 2017.

[31] S. Liu, D. Papailiopoulos, and D. Achlioptas, "Bad global minima exist and sgd can reach them," *Advances in Neural Information Processing Systems*, vol. 33, pp. 8543–8552, 2020.

[32] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[33] D. A. McAllester, "Pac-bayesian model averaging," in *Proceedings of the twelfth annual conference on Computational learning theory*, pp. 164–170, 1999.

[34] Y. Jiang, B. Neyshabur, H. Mobahi, D. Krishnan, and S. Bengio, "Fantastic generalization measures and where to find them," *arXiv preprint arXiv:1912.02178*, 2019.

[35] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur, "Sharpness-aware minimization for efficiently improving generalization," *arXiv preprint arXiv:2010.01412*, 2020.

[36] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson, "Averaging weights leads to wider optima and better generalization," *arXiv preprint arXiv:1803.05407*, 2018.

[37] P. with code, "Papers with code - cifar-100 benchmark (image classification)."

[38] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, "Sharp minima can generalize for deep nets," in *International Conference on Machine Learning*, pp. 1019–1028, PMLR, 2017.

[39] M. Belkin, D. Hsu, S. Ma, and S. Mandal, "Reconciling modern machine-learning practice and the classical bias–variance trade-off," *Proceedings of the National Academy of Sciences*, vol. 116, no. 32, pp. 15849–15854, 2019.

[40] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, "Deep double descent: Where bigger models and more data hurt," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2021, no. 12, p. 124003, 2021.

[41] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra, "Grokking: Generalization beyond overfitting on small algorithmic datasets," *arXiv preprint arXiv:2201.02177*, 2022.

[42] L. Schott, J. von Kügelgen, F. Träuble, P. Gehler, C. Russell, M. Bethge, B. Schölkopf, F. Locatello, and W. Brendel, "Visual representation learning does not generalize strongly within the same domain," *arXiv preprint arXiv:2107.08221*, 2021.

[43] J. Koch, L. Langosco, J. Pfau, J. Le, and L. Sharkey, "Objective robustness in deep reinforcement learning," *arXiv preprint arXiv:2105.14111*, 2021.

[44] D. Krueger, E. Caballero, J.-H. Jacobsen, A. Zhang, J. Binas, D. Zhang, R. Le Priol, and A. Courville, "Out-of-distribution generalization via risk extrapolation (rex)," in *International Conference on Machine Learning*, pp. 5815–5826, PMLR, 2021.

[45] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016.

[46] Y. Zhang, Z.-Q. J. Xu, T. Luo, and Z. Ma, "A type of generalization error induced by initialization in deep neural networks," in *Mathematical and Scientific Machine Learning*, pp. 144–164, PMLR, 2020.

[47] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[49] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, PMLR, 2013.

[50] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[51] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[52] P. J. Huber, "Robust estimation of a location parameter," in *Breakthroughs in statistics*, pp. 492–518, Springer, 1992.

[53] W. Falcon and The PyTorch Lightning team, "PyTorch Lightning," 3 2019.

[54] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com.

[55] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[56] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[57] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–35, 2021.

Appendix A

Supplementary material

A.1 Comparison of runtimes

Table A.1: Increasing the number of hidden units increases the runtime for the same number of steps (100000 epochs). At the same time, training loss seems to increase from a minimum around 160 units as the network is scaled up.

Size	Average runtime (minutes)	Average train loss	Average variational error	Average validation error
10	43	0.04356092	0.20	0.28
160	43	0.00000236	0.06	0.25
640	54	0.00139610	0.07	0.24
2560	48	0.00005435	0.07	0.25
10240	52	0.00873949	0.08	0.25
50000	54	0.00925732	0.09	0.25
100000	54	0.00921736	0.09	0.25
500000	116	0.00923043	0.09	0.25
1000000	246	0.00947294	0.09	0.25

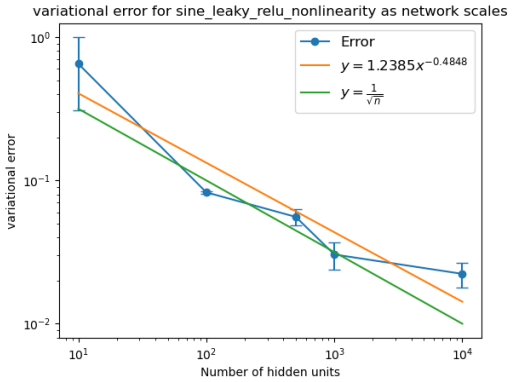
A.2 Additional figures

A.2.1 Other experiments

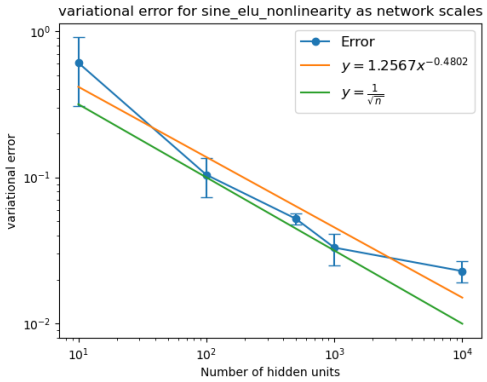
(a) Leaky ReLU activation function.

(b) ELU activation function.

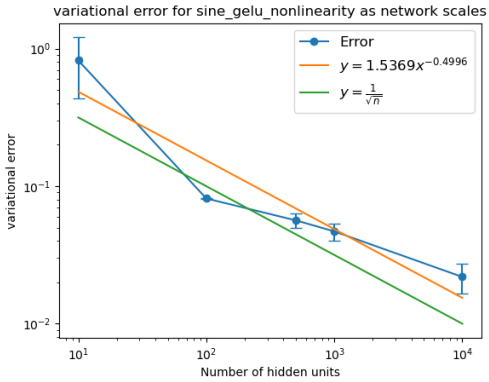
Figure A.1: On the $\sin x$ dataset, the solution to the variational problem is the same (the natural cubic spline) for all the nonlinearities we tried. Here is a comparison of two of them: ReLU and GELU. In red is the neural network, and in purple we have the ground truth, with the natural cubic spline perfectly overlapping.



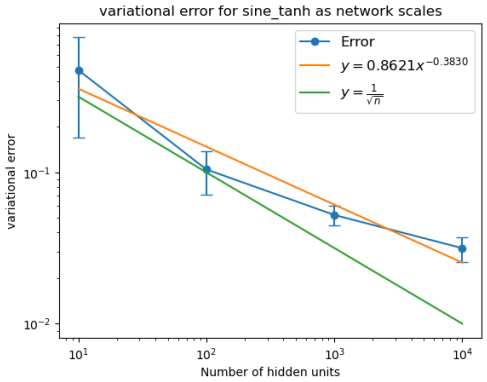
(a) Leaky ReLU activation function.



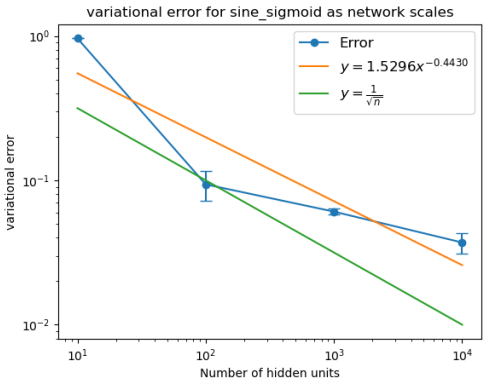
(b) ELU activation function.



(c) GELU activation function.

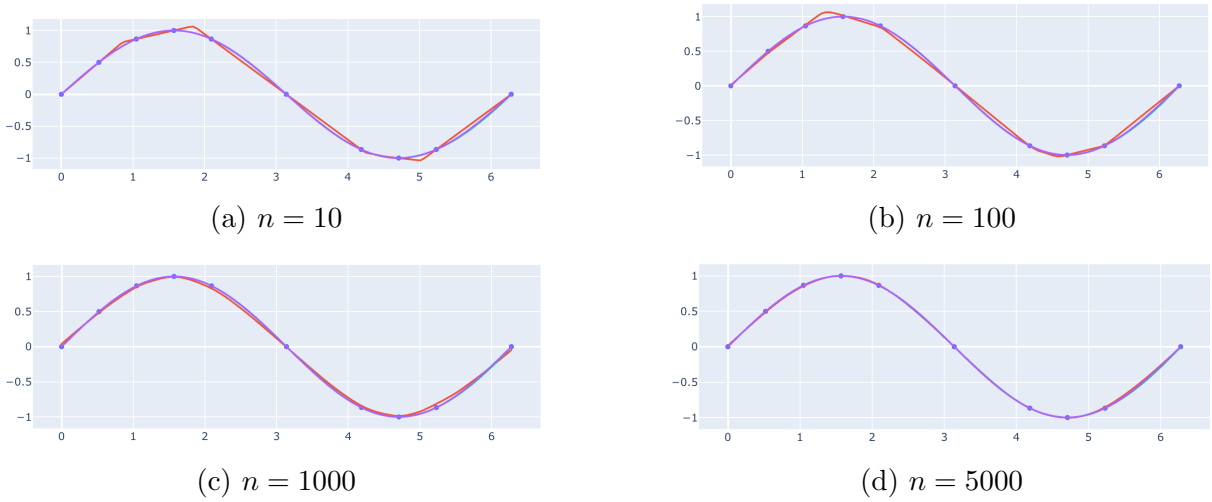


(d) Hyperbolic tangent activation function.



(e) Sigmoid activation function.

Figure A.2: On the $\sin x$ dataset, the predictions of Theorem 1 are robust to changes in the hidden units' activation function.



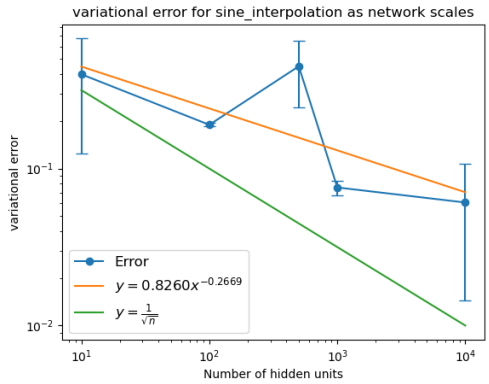
(a) $n = 10$ (b) $n = 100$

(c) $n = 1000$ (d) $n = 5000$

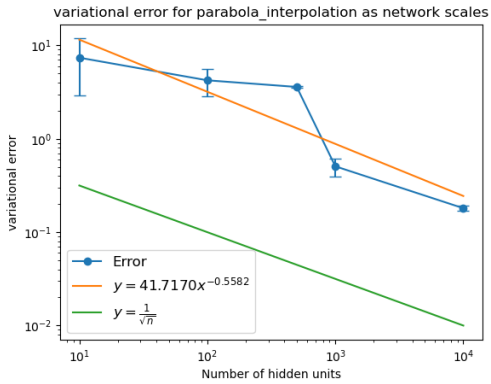
Figure A.3: The fit of networks of different sizes to the sine dataset containing 10 training data points. The purple solid line is the ground truth, i.e. $\sin(x)$ and the red line is $f(x, \theta^*)$, the function represented by the neural network. As the number of hidden units is increased, the quality of the fit to the data increases, and the network better approximates the natural cubic spline. At this scale, the natural cubic spline is not visible because it perfectly overlaps with the ground truth.

Figure A.4: Double descent behaviour with respect to the validation error on the shallow neural network without the ASI trick. These networks have $n = 100$, and since they do not use ASI, $|\theta| = 300$. The validation error curve shows two peaks followed by a monotonical decrease for many epochs.

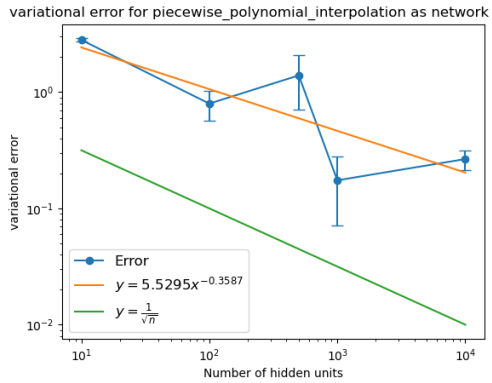
A.2.2 Results on interpolation



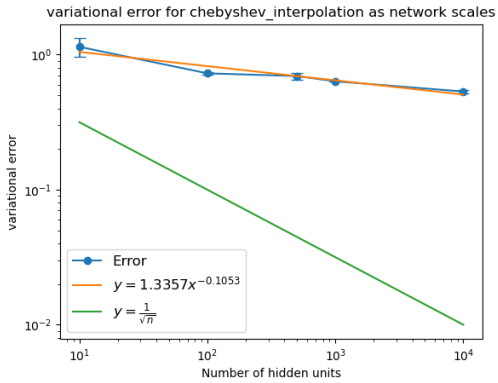
(a) Sine dataset, with network sizes ranging from 10 to 5000 hidden units.



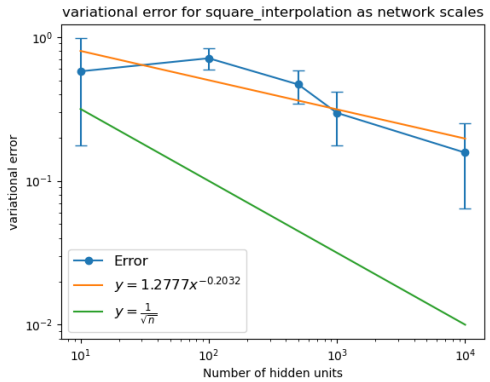
(b) Parabola dataset, with network sizes ranging from 10 to 10000 hidden units.



(c) Piecewise polynomial dataset, with network sizes ranging from 10 to 10000 hidden units.

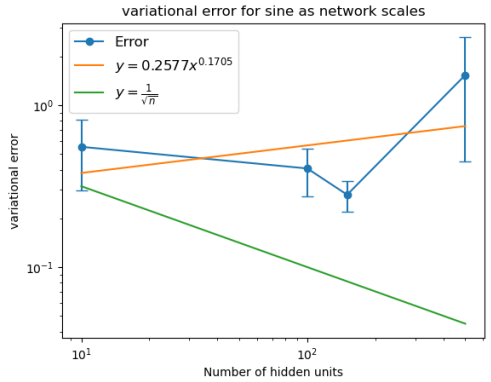


(d) Chebyshev polynomial dataset, with network sizes ranging from 10 to 5000 hidden units.

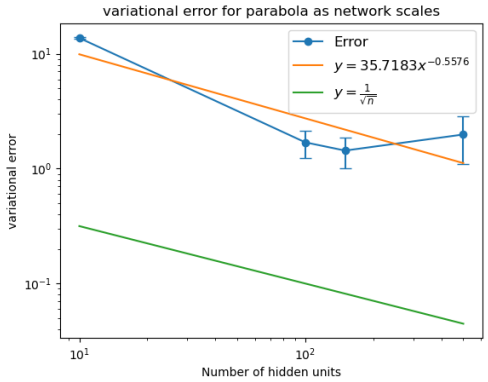


(e) Square polynomial dataset, with network sizes ranging from 10 to 10000 hidden units.

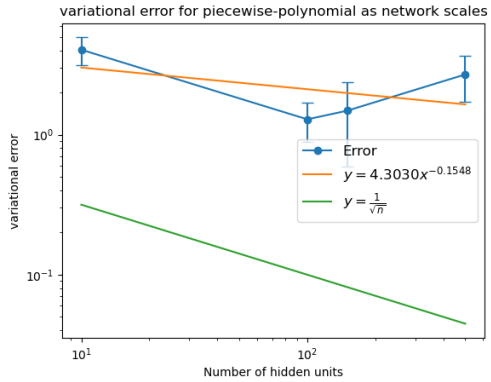
Figure A.5: Interpolation on the shallow, 1-hidden layer network: change in variational error as the hidden layer is scaled up.



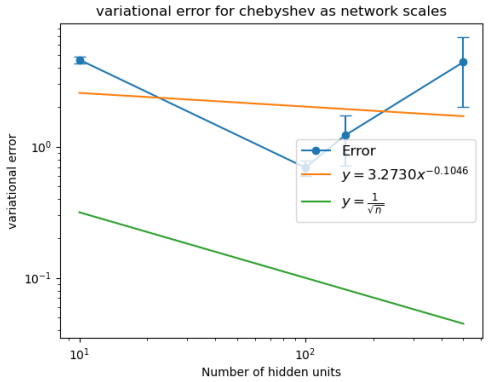
(a) Sine dataset, with network sizes ranging from 10 to 5000 hidden units.



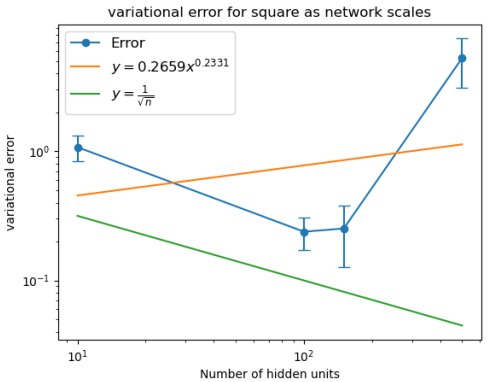
(b) Parabola dataset, with network sizes ranging from 10 to 10000 hidden units.



(c) Piecewise polynomial dataset, with network sizes ranging from 10 to 10000 hidden units.



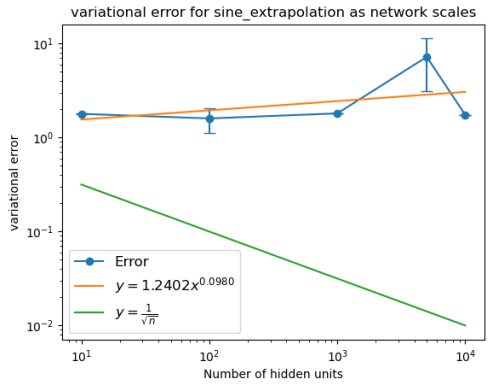
(d) Chebyshev polynomial dataset, with network sizes ranging from 10 to 5000 hidden units.



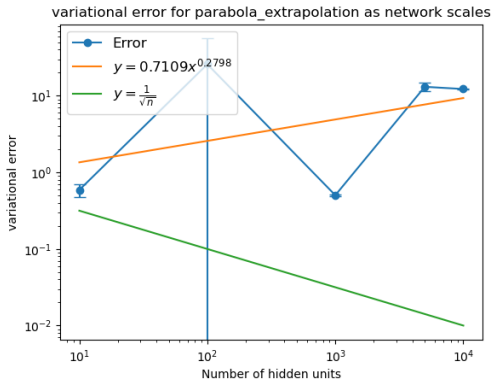
(e) Square dataset, with network sizes ranging from 10 to 10000 hidden units.

Figure A.6: Interpolation on the deeper, 2-hidden layer network: change in variational error as the hidden layers are scaled up.

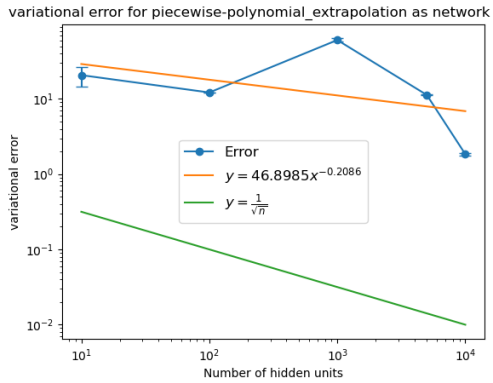
A.2.3 Results on extrapolation



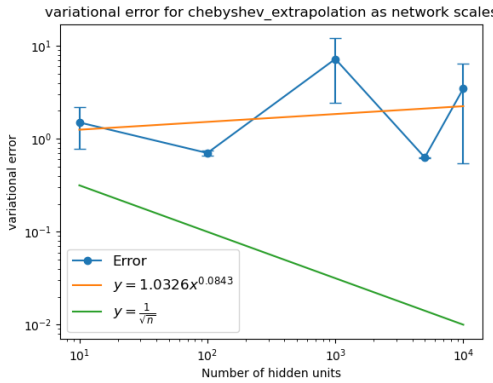
(a) Sine dataset, with network sizes ranging from 10 to 5000 hidden units.



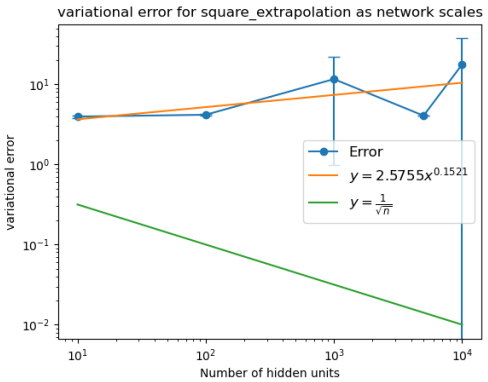
(b) Parabola dataset, with network sizes ranging from 10 to 10000 hidden units.



(c) Piecewise polynomial dataset, with network sizes ranging from 10 to 10000 hidden units.

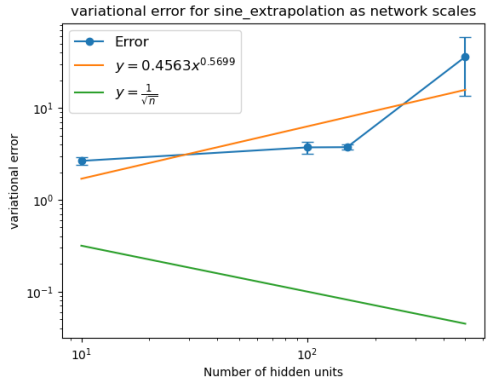


(d) Chebyshev polynomial dataset, with network sizes ranging from 10 to 5000 hidden units.

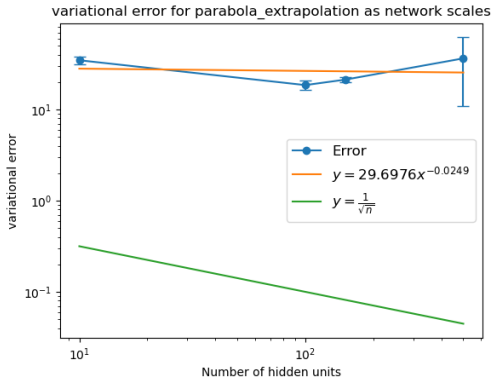


(e) Square dataset, with network sizes ranging from 10 to 10000 hidden units.

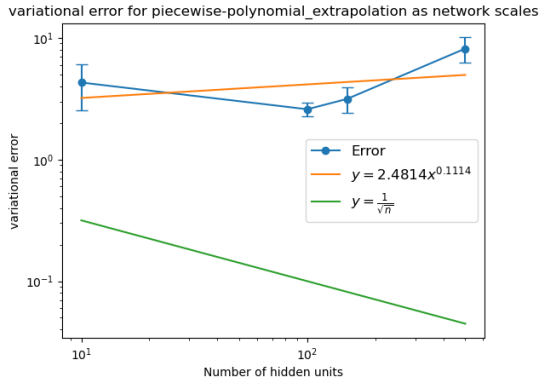
Figure A.7: Extrapolation on the shallow, 1-hidden layer network: change in variational error as the hidden layer is scaled up. Some of the errors increase, and several data points have large standard deviation because of very different functions at convergence.



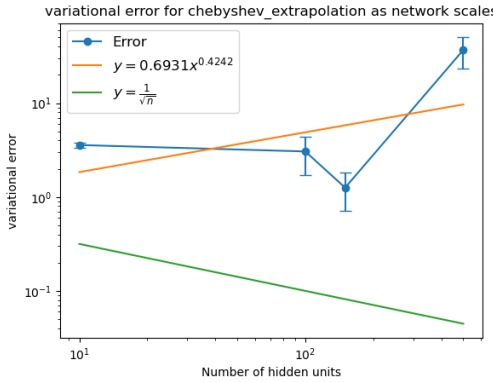
(a) Sine dataset, with network sizes ranging from 10 to 5000 hidden units.



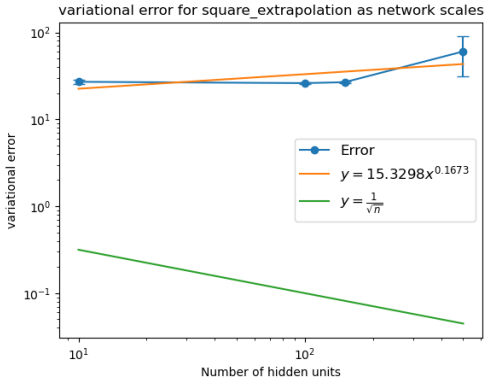
(b) Parabola dataset, with network sizes ranging from 10 to 10000 hidden units.



(c) Piecewise polynomial dataset, with network sizes ranging from 10 to 10000 hidden units.



(d) Chebyshev polynomial dataset, with network sizes ranging from 10 to 5000 hidden units.



(e) Square dataset, with network sizes ranging from 10 to 10000 hidden units.

Figure A.8: Extrapolation on the deeper, 2-hidden layer network: change in variational error as the hidden layers are scaled up.