ETH Zürich

Deep Learning in Scientific Computing

Lecture notes

February 25, 2022

T. De Ryck
S. Mishra
R. Molinaro
T. K. Rusch

Preface

These lecture notes contain material delivered at the master level course *Deep Learning in Scientific Computing*, taught by Prof. Dr. Siddhartha Mishra at ETH Zürich in the spring semesters of 2021 and 2022. It is important to note that these lecture notes are still under construction and will be adapted and expanded continuously.

If you find typos or other mistakes, or if you have suggestions to improve these lecture notes, we would be thankful if you could let us know by sending an email to `tim.deryck@sam.math.ethz.ch`.

Zürich,
February 2022

Tim De Ryck
Siddhartha Mishra
Roberto Molinaro
T. Konstantin Rusch

Contents

Chapter 1
Deep learning

1.1 Introduction

Deep learning has brought forward many breakthroughs in a very diverse array of disciplines. An illustrous example is the ImageNet challenge, a visual object recognition software contest with the goal to automatically classify millions of pictures into thousands of categories, based on the objects on the pictures. In 2012, a large drop in the error rate was accomplished by AlexNet, a deep convolution network, which sparked a boom in the interest in neural networks for image recognition. Other deep learning algorithms of notable importance include AlphaGo, an AI-based computer program that won games of Go from professional go player Lee Sedol, and AlphaFold that predicts protein structures at a level of accuracy that dramatically exceeds that of the alternatives.

Although neural networks are only being heavily used since a couple of years, they have been around for much longer. The first neural network was divised by McCollough and Pitts in 1943 [50] and was created in the image of the biological neurons in our brains. This network, the *threshold logic unit*, only consisted of one artificial unit. Taking input $x \in \{0,1\}^d$, the network would give as output 1 if $\sum_i w_i x_i + b > 0$ and 0 otherwise. The weights $w \in \mathbb{R}^d, b \in \mathbb{R}$ are adjustable depending on the goal of the ask, but are not learned based on data. Note that the thresholding operation is equivalent with using the Heaviside function H as an *activation function* i.e., computing $H(\sum_i w_i x_i + b)$. In 1957, Rosenblatt proposed the *perceptron* [65]. Although it is structurally quite similar to the threshold logic unit, the perceptron distinguishes itself from its ancestor by accepting a real vector $x \in \mathbb{R}^d$ instead of binary vector as input, and by learning the weights based on training data. The perceptron was particularly used to classify data into two classes. A direct descendant of the perceptron is the *multilayer perceptron*, a composition of multiple perceptrons, which are stacked in layers. We will study the mathematical properties of multilayer perceptrons in great detail in Chapter 2.

After years of little progress, the so-called 'first AI winter', neural network research regained strength in the 1980s when other, differentiable activation functions

were considered. This allowed the use of gradient-based optimization algorithms to learn the weights of the networks. Moreover, the invention of *backpropagation* (Section 1.3.2.3) allowed practitioners to efficiently evaluate the needed gradients. On the theory side, the first results approximation capabilities of neural networks became available in the form of universal approximation theorems, e.g. [11], see also Chapter 2. After a second AI winter, the final breakthrough of neural networks happened around 2010, initiated by the availability of large data sets, powerful GPUs to process them, faster computers and better training algorithms.

1.2 Neural network architectures

Neural networks come in many forms and sizes. To distinguish between different neural networks, a lot of terminology has seen the light. In what follows, we discuss some common neural network architectures.

1.2.1 Multilayer perceptrons

Multilayer perceptrons get their name from their definition as a concatenation of affine maps and activation functions. As a consequence of this structure, the computational graph of an MLP is organized in layers. At the k-th layer, the activation function is component-wise applied to obtain the value

$$z^k = \sigma(C_{k-1}(z^{k-1})) = \sigma(W_{k-1}z^{k-1} + b_{k-1}), \tag{1.1}$$

where z^0 is the input of the network, $W_k \in \mathbb{R}^{d_{k+1}\times d_k}$, $b_k \in \mathbb{R}^{d_{k+1}}$ and $C_k : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}} : x \mapsto W_k x + b_k$ for $d_k \in \mathbb{N}$ is an affine map. A **multilayer perceptron (MLP)** with L layers, activation function σ, output function σ_o, input dimension d_{in} and output dimension d_{out} then is a function $f_\theta : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ of the form

$$f_\theta(x) = (\sigma_o \circ C_L \circ \sigma \circ \cdots \circ \sigma \circ C_0)(x) \quad \text{for all } x \in \mathbb{R}^{d_{in}}, \tag{1.2}$$

where σ is applied element-wise and $\theta = ((W_0, b_0), \ldots, (W_L, b_L))$. The dimension d_k of each layer is also called the *width* of that layer. The width of the MLP is defined as $\max_k d_k$. Furthermore, the matrices W_k are called the *weights* of the MLP and the vectors b_k are called the *biases* of the MLP. Together, they constitute the (trainable) parameters of the MLP, denoted by $\theta = ((W_0, b_0), \ldots, (W_L, b_L))$. Finally, an output function σ_o can be employed to increase the interpretability of the output. In classification problems, the **softmax** function, defined as

$$(\sigma_o(x))_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \tag{1.3}$$

is used to ensure that $\sum_i (f_\theta(x))_i = 1$. As a consequence, $(f_\theta(x))_i$ can be interpreted as the probability that x belongs to class i. In regression settings, no output function is needed, or equivalently one can set $\sigma_o(x) = x$. An example of an MLP can be seen in Figure 1.1.

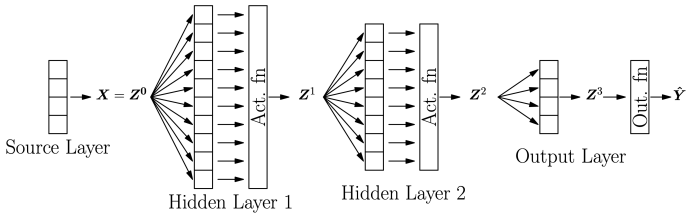


Fig. 1.1: Structure of a multilayer perceptron (MLP).

Remark 1.1 In some texts, the number of layers L corresponds to the number of affine maps in the definition of f_θ. Of these L layers, the first $L-1$ layers are called hidden layers. An L-layer network in this alternative definition hence corresponds to a $(L-1)$-layer network in our definition. It is important to be aware of this.

A lot of the properties of a neural network depend on the choice of activation function. We discuss the most common activation functions below and visualize them in Figure 1.2.

- The **Heaviside** function is the activation that was used in the McCullocks-Pitts neuron and is defined as

$$H(x) = \begin{cases} 0, & x < 0 \\ 1 & x \geq 0, \end{cases} \tag{1.4}$$

  Because the gradient is zero wherever it exists, it is not possible to train the network using a gradient-based approach. For this reason, the Heaviside function is not used anymore.
- The **logistic** function is defined as

$$\rho(x) = \frac{1}{1+e^{-x}} \tag{1.5}$$

  and can be thought of as a smooth approximation of the Heaviside function as $\rho(\lambda x) \to H(x)$ for $\lambda \to \infty$ for almost every x. It is a so-called sigmoidal function, meaning that the function is smooth, monotonic and has horizontal asymptotes for $x \to \pm\infty$. We will see in Chapter 2 that for neural networks with a sigmoidal activation function many theoretical results are available.
- The hyperbolic tangent or **tanh** function, defined by

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{1.6}$$

is another smooth, sigmoidal activation function that is symmetric, unlike the logistic function. One problem with the tanh (and all other sigmoidal functions) is that the gradient vanished away from zero, which might be problematic when using a gradient-based optimization approach.

- The Rectified Linear Unit, rectifier function or **ReLU** function, defined as

$$\mathrm{ReLU}(x) = \max\{x, 0\}, \tag{1.7}$$

is a very popular activation function. It is easy to compute, scale-invariant and reduces the vanishing gradient problem that sigmoidal functions have. One common issue with ReLU networks is that of 'dying neurons', caused by the fact that the gradient of ReLU is zero for $x < 0$. This issue can however be overcome by using the so-called **leaky ReLU** function,

$$\mathrm{ReLU}_\nu(x) = \max\{x, -\nu x\}, \tag{1.8}$$

where $\nu > 0$ is small. Other adaptations are the Gaussian Error Linear Unit, the Sigmoid Linear Unit, Exponential Linear Unit and softplus function.

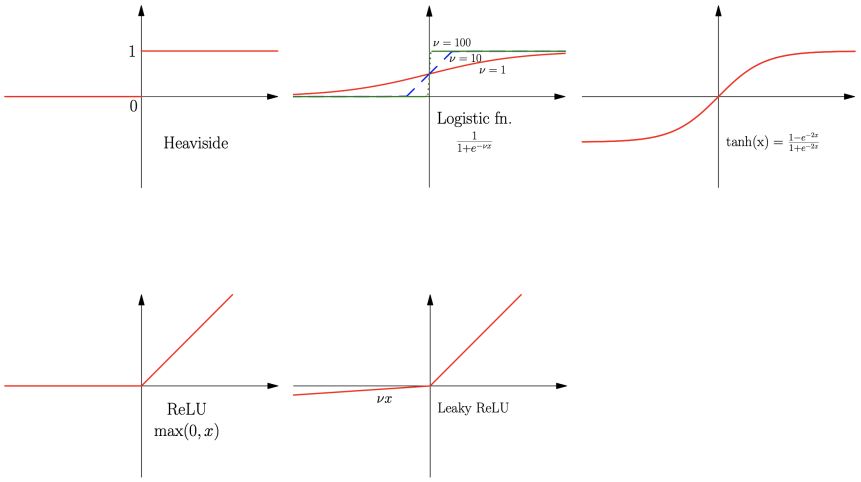


Fig. 1.2: Overview of the most popular neural network activation functions.

1.2.2 Convolutional neural networks

A convolutional neural network (CNN) is a regularized version of a multi-layer perceptron that is often applied in the analysis of images. For images, the input space often has a very high dimension, e.g. a colour picture with 32×32 pixels can be seen as a 3072-dimensional vector. For such high input dimensions, the number of weights in a fully-connected MLP quickly becomes very large. One can drastically reduce the size of Θ by requiring that a lot of weights are identical. The weight-sharing used in CNNs is inspired by the discrete convolution operation, known from signal processing and image analysis. A simple example of such a convolution operation is the calculation of the moving average y of a d-dimensional vector x,

$$y_i = \sum_{k=1}^{K} v_{K-k} x_{i+k}, \qquad \text{for all } 1 \leq i \leq d, \tag{1.9}$$

where in this case $v_k = \frac{1}{K}$ for all k and where $K \ll d$. The vector v can be interpreted as a convolution kernel of size K. The formula (1.9) can be slightly generalized to

$$y_i = \sigma\left(\sum_{k=1}^{K} v_{K-k} x_{i+k} + v_0\right), \qquad \text{for all } 1 \leq i \leq d, \tag{1.10}$$

for some scalar v_0, a vector $v \in \mathbb{R}^K$ and a (possibly nonlinear) scalar function σ. A comparison with (1.1) reveals that the above formula can be interpreted as the output of a hidden layer of an MLP where the weight matrix is quite sparse and the non-zero weights are shared. A layer of such a form (1.10) is called a *convolutional layer*. In traditional image processing, convolution vectors (also masks or kernels) can be used to sharpen or blur images or extract features such as edges from images. The basic idea of using convolutional layers is that the neural networks automatically learn which features should be extracted.

To enforce shift invariance (or rather equivariance), convolutional neural networks often consist of combinations of convolutional layers and so-called *pooling layers*. They reduce the dimensionality of the output of the previous layer by performing a pooling operation over neightbouring neurons, e.g. taking the maximum or average of these neurons. In this way, the CNN becomes more robust to translations. In general, pooling layers are not trainable.

1.2.3 Residual neural networks

Residual neural networks are feedforward neural networks where also non-consecutive layers can be connected. A simple example of a so-called ResNet is a network where the input for each layer is given by the output of the two previous layers,

$$z^k = \sigma\big(W^1_{k-1} z^{k-1} + b^1_{k-1} + W^2_{k-2} z^{k-2} + b^2_{k-2}\big). \tag{1.11}$$

Compared with the definition of an MLP (1.1), there are now skip connections or jumps between non-consecutive layers (Figure 1.3). There is a lot of empirical evidence [28] that these shortcuts ease the training of deep networks by mitigating the vanishing gradient problem, allowing a substantial increase in accuracy.

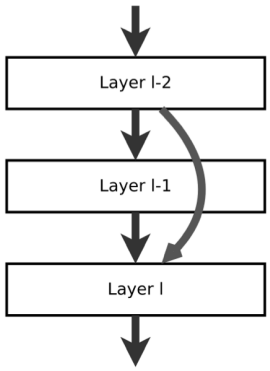


Fig. 1.3: Structure of a residual neural network (ResNet).

Residual neural networks where all layers are connected in a feedforward manner are referred to as DenseNets [32]. The output of a hidden layer of such a network is given by

$$z^k = \sigma\left(\sum_{\ell=1}^{k} (W^\ell_{k-\ell} z^{k-\ell} + b^{k-\ell}_\ell)\right). \tag{1.12}$$

1.2.4 Recurrent neural networks

Finally, we discuss a neural network architecture that is tailored for sequential data. We motivate the need of such an architecture based on a simple example. Consider the ODE $h'(t) = F(h(t), X(t))$, where $X(t)$ is a source term and F is a differential operator. The goal is to retrieve $\{h(t)\}_t$ from $\{X(t)\}_t$. Using a multilayer perceptron, this task could be solved by training a network that approximates the map $\{X(t_n)\}_{n=1}^N \mapsto \{h(t_n)\}_{n=1}^N$ i.e., by training a network that maps a vector to another vector. A major drawback of this approach is that the temporal structure is not taken into account: from the ODE it follows that $h(t_{n+1})$ can be obtained (or approximated) from $h(t_n)$ and $X(t_{n+1})$ in the exact same way for every n. This structure can be built in the structure of a neural network by setting

$$h(t_{n+1}) = A(h(t_n), X(t_{n+1})) = \overline{W}\sigma(W_h h(t_n) + W_X X(t_{n+1}) + b), \tag{1.13}$$

where $\overline{W}, W_h, W_X$ are weight matrices and b is a bias vector. In this way, both the input and output of the network can be an arbitrarily long sequence. The structure of an RNN is visualized in Figure 1.4.

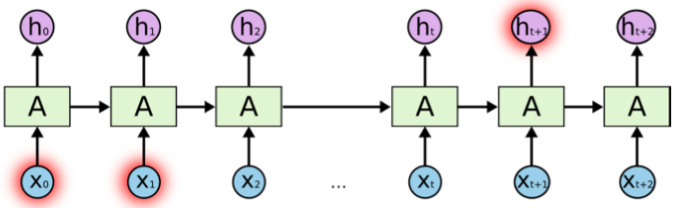


Fig. 1.4: Structure of a recurrent neural network (RNN).

Figure 1.4 and equation 1.13 only show an example of a very simple RNN. Many more complicated variants are used in practice, such as the gated recurrent unit (GRU) [10] and long short-term memory (LSTM) [30] networks.

1.3 Training a neural network

In this section, we discuss how one can (try to) determine the optimal set of weights and biases to maximize the accuracy of the neural network based on an available data set. This procedure is also called the *training* of the neural network. The accuracy of a neural network can be quantified using a setting-dependent *loss function* (Section 1.3.1). This loss function can then be minimized using a (gradient-based) optimizer (Section 1.3.2). Finally, one wants the neural network to not only perform well on the training data set, but also on unseen data i.e., one wants that it *generalizes* well to unseen data. This is the topic of Section 1.3.3.

1.3.1 Training set and loss function

A first crucial element in the training of a neural network is the availability of a *training data set*, which will serve as input for the neural network. If the training set also includes the intended output of the neural network, we say that the training set is *labeled* and we say that the algorithm is a *supervised learning* algorithm. Image classification is a well-known example of this setting, as the training data sets consists of images for which it is known to which category they belong. Another important example is that of function approximation, where the neural network is trained to approximate some function $f : D \to \mathbb{R}$. In this case the training set will be of the form $\mathcal{S} = \{(x_i, f(x_i)\}_{i=1}^{N}$, where $x_i \in D$. It is also possible that the training

data set is *unlabeled*, in that case one speaks of *unsupervised learning*. Examples of unsupervised learning algorithms are clustering methods, principal component analysis and anomaly detection methods.

The performance of a neural network crucially depends on the nature, quality and size of the training data set. In many fields, one is generally restricted to a fixed data set. Although it can be slightly altered using data augmentation techniques, there is little freedom in the choice of training data. In scientific computing, the situation is completely different as training sets can be generated on demand based on simulations. Time, computing power and cost are the only restrictions in this setting. In Chapter 3 we will discuss how to exploit this large degree of freedom.

The next step is to choose a suitable loss function that quantifies for every set of weights and biases θ the error that the neural network f_θ makes. As the loss function $\mathcal{J}$ depends on the training set $\mathcal{S}$ of size $N = |\mathcal{S}|$, the loss function has the form $\theta \mapsto \mathcal{J}(\theta, \mathcal{S})$. For function regression, a popular choice is given by

$$\mathcal{J}(\theta, \mathcal{S}) = \frac{1}{N} \sum_{i=1}^{N} |f(x_i) - f_\theta(x_i)|^p + \lambda \|\theta\|^q, \tag{1.14}$$

for $\theta \in \Theta$ and where usually $p, q \in \{1, 2\}$. The term $\|\theta\|^q$ is a regularization term that usually has a positive influence on the training procedure and the hyperparameter $\lambda \geq 0$ controls the magnitude of the regularization term. The case $p = 2$ corresponds to least squares minimization. Setting $q = 1$ induces sparsity in the weights, cf. lasso regularization in statistics. For classification problems and unsupervised learning problems other loss functions are used. An overview can be found in [24].

1.3.2 Optimization

Once a loss function is chosen, training a network corresponds to finding an approximation of the minimizer of that loss function,

$$\theta^*(\mathcal{S}) \approx \arg\min_{\theta \in \Theta} \mathcal{J}(\theta, \mathcal{S}). \tag{1.15}$$

In this section we discuss the details of how this minimizer is approximated in practice.

1.3.2.1 Optimization algorithms

Since Θ is high-dimensional and the map $\theta \mapsto \mathcal{J}(\theta, \mathcal{S})$ is generally non-convex, solving the minimization problem (1.15) can be very challenging. Fortunately, the loss function $\mathcal{J}$ is almost everywhere differentiable, so that gradient-based iterative optimization methods can be used.

The simplest example of such an algorithm is **gradient descent**. Starting from an initial guess θ_0, the idea is to take a small step in the parameter space Θ in the direction of the steepest descent of the loss function to obtain a new guess θ_1. Note that this comes down to taking a small step in the opposite direction of the gradient evaluated in θ_0. Repeating this procedure yields the following iterative formula,

$$\theta_{\ell+1} = \theta_\ell - \eta_\ell \nabla_\theta \mathcal{J}(\theta_\ell, \mathcal{S}), \qquad \forall \ell \in \mathbb{N}, \tag{1.16}$$

where the learning rate η_ℓ controls the size of the step and is generally quite small. The gradient descent formula yields a sequence of parameters $\{\theta_\ell\}_{\ell \in \mathbb{N}}$ that converges to a local minimum of the loss function under very general conditions. Because of the non-convexity of the loss function, convergence to a global minimum can not be ensured. Another issue lies in the computation of the gradient $\nabla_\theta \mathcal{J}(\theta_\ell, \mathcal{S})$. A simple rewriting of the definition of this gradient (up to regularization term),

$$\nabla_\theta \mathcal{J}(\theta_\ell, \mathcal{S}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{J}_i(\theta_\ell) \quad \text{where } \mathcal{J}_i(\theta_\ell) = \mathcal{J}(\theta_\ell, \{(x_i, f(x_i))\}), \tag{1.17}$$

reveals that one actually needs to evaluate $N = |\mathcal{S}|$ gradients. As a consequence, the computation of the full gradient will be very slow and memory intensive for large training data sets.

Stochastic gradient descent (SGD) overcomes this problem by only calculating the gradient in one data point instead of the full training data set. More precise, for every ℓ a random index i_ℓ, with $1 \leq i_\ell \leq N$ is chosen, resulting in the following update formula,

$$\theta_{\ell+1} = \theta_\ell - \eta_\ell \nabla_\theta \mathcal{J}_{i_\ell}(\theta_\ell), \qquad \forall \ell \in \mathbb{N}. \tag{1.18}$$

Similarly to gradient descent, stochastic gradient descent provably converges to a local minimum of the loss function under rather general conditions, although the convergence will be slower. In particular, the convergence will be noisier and there is no guarantee that $\mathcal{J}(\theta_{\ell+1}) \leq \mathcal{J}(\theta_\ell)$ for every ℓ. Because the cost of each iteration is much lower than that of gradient descent, it might still makes sense to use stochastic gradient descent.

Mini-batch gradient descent tries to combine the advantages of gradient descent and SGD by evaluating the gradient of the loss function in a subset of $\mathcal{S}$ in each iteration (as opposed to the full training set for gradient descent and one training sample for SGD). More precise, for every $\ell \in \mathbb{N}$ a subset $\mathcal{S}_\ell \subset \mathcal{S}$ of size n is chosen. These subsets are referred to as *mini-batches* and their size n as the mini-batch size. In contrast to SGD, these subsets are not entirely randomly selected in practice. Instead, one randomly partitions the training set into $\lceil N/n \rceil$ mini-batches, which are then all used as a mini-batch $\mathcal{S}_\ell$ for consecutive ℓ. Such a cycle of $\lceil N/n \rceil$ iterations is called an *epoch*. After every epoch, the mini-batches are reshuffled, meaning that a new partition of the training set is drawn. In this way, every training sample is used one time during every epoch. The corresponding update formula reads as,

$$\theta_{\ell+1} = \theta_\ell - \eta_\ell \nabla_\theta \mathcal{J}(\theta_\ell, \mathcal{S}_\ell), \qquad \forall \ell \in \mathbb{N}. \tag{1.19}$$

By setting $n = N$, resp. $n = 1$, one retrieves gradient descent, resp. SGD. For this reason, gradient descent is also often referred to as *full batch* gradient descent.

The performance of mini-batch gradient descent can be improved in many ways. One particularly popular example of such an improvement is the **Adam** optimizer [36], proposed by Kingma and Ba in 2015. Adam, of which the name is derived from *adaptive moment estimation*, calculates at each ℓ (exponential) moving averages of the first and second moment of the mini-batch gradient g_ℓ,

$$m_\ell = \beta_1 m_{\ell-1} + (1-\beta_1) g_\ell, \quad v_\ell = \beta_2 v_{\ell-1} + (1-\beta_2) g_\ell^2, \quad g_\ell = \nabla_\theta \mathcal{J}(\theta_\ell, \mathcal{S}_\ell), \tag{1.20}$$

where β_1 and β_2 are close to 1. However, one can calculate that these moving averages are biased estimators. This bias can be removed using the following correction,

$$\widehat{m}_\ell = \frac{m_\ell}{1-\beta_1^\ell}, \quad \widehat{v}_\ell = \frac{v_\ell}{1-\beta_2^\ell}, \qquad \forall \ell \in \mathbb{N}, \tag{1.21}$$

where $\widehat{m}_\ell, \widehat{v}_\ell$ are unbiased estimators. One can obtain Adam from the basic mini-batch gradient descent update formula (1.19) by replacing $\nabla_\theta \mathcal{J}(\theta_\ell, \mathcal{S}_\ell)$ by the moving average m_ℓ and by setting $\eta_\ell = \frac{\alpha}{\sqrt{\widehat{v}_\ell + \epsilon}}$, where $\alpha > 0$ is small and $\epsilon > 0$ is close to machine precision. This leads to the update formula

$$\theta_{\ell+1} = \theta_\ell - \alpha \frac{\widehat{m}_\ell}{\sqrt{\widehat{v}_\ell + \epsilon}}, \qquad \forall \ell \in \mathbb{N}. \tag{1.22}$$

Compared to mini-batch gradient descent, the convergence Adam will be less noisy for two reasons. First, a fraction of the noise will be removed since a moving average of the gradient is used. Second, the step size of Adam decreases if the gradient g_ℓ varies a lot i.e., when $\widehat{v}_\ell$ is large. These design choices make that Adam performs well on a large number of tasks, making it one of the most popular optimizers in deep learning.

Finally, the interest in **second-order** optimization algorithms has been steadily growing over the past years. Most second-order methods used in practice are adaptations of the well-known Newton method,

$$\theta_{\ell+1} = \theta_\ell - \eta_\ell (\nabla_\theta^2 \mathcal{J}(\theta, \mathcal{S}))^{-1} \nabla_\theta \mathcal{J}(\theta, \mathcal{S}), \qquad \forall \ell \in \mathbb{N}, \tag{1.23}$$

where $\nabla_\theta^2 \mathcal{J}(\theta, \mathcal{S})$ denotes the Hessian of the loss function. The generalization to a mini-batch version is immediate. As the computation of the Hessian and its inverse is quite costly, one often uses a *quasi*-Newton method that computes an approximate inverse by solving the linear system $\nabla_\theta^2 \mathcal{J}(\theta, \mathcal{S}) h_\ell = \nabla_\theta \mathcal{J}(\theta, \mathcal{S})$. An example of a popular quasi-Newton method in deep learning for scientific computing is **L-BFGS** [41]. In many other application areas, however, first-order optimizers remain the dominant choice for a myriad of reasons.

1.3.2.2 Weight initialization

All iterative optimization algorithms from the previous section require an initial guess θ_0. Making this initial guess is referred to as the weight initialization. Although it is often overlooked, a bad initialization can cause a lot of problems, especially for very deep networks.

Exercise 1.1 For a wide and deep enough neural network of choice, initialize all the weights according to the standard normal distribution and generate an input vector from the same distribution. Calculate the output of the network. What do you observe? What will be the consequences when you try to optimize the weights of this network using a gradient-based optimizer?

The previous exercise showed an instance of the so-called *exploding gradient problem*, which occurs when the initial weights are chosen too large. Similarly, when the initial weights are too small, the gradients might also be very close to zero, leading to the *vanishing gradient problem*. Fortunately, it is possible to choose the variance of the initial weights in such a way that the output of the network has the same order of magnitude as the input of the network. One such possible choice is **Xavier initialization** [23], meaning that the weights are initializated as

$$(W_k)_{ij} \sim N\left(0, \frac{2g^2}{d_{k-1}+d_k}\right) \text{ or } (W_k)_{ij} \sim U\left(-g\sqrt{\tfrac{6}{d_{k-1}+d_k}}, g\sqrt{\tfrac{6}{d_{k-1}+d_k}}\right), \quad (1.24)$$

where N denotes the normal distribution, U the uniform distribution, d_k is the output dimension of the k-th layer and the value g depends on the activation function. For the ReLU activation one sets $g = \sqrt{2}$ and for the tanh activation function $g = 1$ is a valid choice.

Finally, it is customary to retrain neural network with different starting values θ_0 (drawn from the same distribution) as the gradient-based optimizer might convergence to a different local minimum for each θ_0. One can then choose the 'best' neural network based on some suitable criterion or combine the different neural networks if the setting allows this. Also note that this task can be performed in parallel.

1.3.2.3 Backpropagation

Recall that using mini-batch gradient descent instead of gradient descent ensured that the number of gradients that need to be computed to optimize the network weights stays computationally feasible. This is of course of no use if the computation of one single gradient $\nabla_\theta \mathcal{J}(\theta, \mathcal{S})$ already is expensive. In fact, it turns out that a naive application of the chain rule indeed leads to a computationally costly computation of the gradient of the loss function. For simplicity, we consider a neural network f_θ of width one, depth L, activation function σ and no biases. For $x \in \mathbb{R}$, we write

$$f_\theta(x) = \theta_{L+1} \cdot (f_{\theta_L} \circ \cdots \circ f_{\theta_1})(x), \quad \text{where } f_{\theta_\ell}(y) = \sigma(\theta_\ell \cdot y) \text{ for all } y \in \mathbb{R}. \quad (1.25)$$

For notational convenience later on, we also define the function

$$g_\ell(x) = (f_{\theta_\ell} \circ \cdots \circ f_{\theta_1})(x) \text{ for all } y \in \mathbb{R} \text{ and } 1 \leq \ell \leq L. \tag{1.26}$$

We now calculate the gradients. First, one can easily see that $\partial_{\theta_{L+1}} f_\theta(x) = (f_{\theta_L} \circ \cdots \circ f_{\theta_1})(x)$. This requires $O(L)$ multiplications or function evaluations, which is computationally feasible. On the other hand, we also find from the chain rule that

$$\partial_{\theta_\ell} f_\theta(x) = \theta_{L+1} \cdot \prod_{k=\ell}^{L} (f'_{\theta_k} \circ g_{k-1})(x). \tag{1.27}$$

Since the calculation of $(f'_{\theta_\ell} \circ g_{\ell-1})(x)$ requires $O(\ell)$ multiplications or function evaluations, we find that the calculation of $\partial_{\theta_1} f_\theta(x)$ requires $O(\prod_{l=1}^{L} \ell) = O(L!)$ operations. Since $L! \sim L^{L-\frac{1}{2}}$ it is clear that this approach becomes infeasible when the number of weights L is very large.

Fortunately, the complexity of $O(L!)$ can be drastically reduced by making use of the chain formula in a clever way and storing some intermediate values. The resulting algorithm is called **backpropagation**. The first step includes the forward propagation of the input x through the network. By making use of the recursion formulas

$$g_\ell(x) = (f_{\theta_\ell} \circ g_{\ell-1})(x) \text{ for } \ell = 1, \ldots, L \text{ and } f_\theta(x) = \theta_{L+1} \cdot g_L(x), \tag{1.28}$$

we can see that we can simultaneously calculate the values $g_1(x), \ldots, g_L(x), f_\theta(x)$ with only $O(L)$ operations. The second step is the *backwards* calculation of the gradients, based on the recursion formula

$$\partial_{\theta_\ell} f_\theta(x) = \theta_{L+1} \cdot (f'_{\theta_\ell} \circ g_{\ell-1})(x) \cdot \partial_{\theta_{\ell+1}} f_\theta(x) \text{ for } \ell = L, \ldots, 1, \tag{1.29}$$

where we again store each $\partial_{\theta_\ell} f_\theta(x)$. Hence, we can calculate all the gradients $\partial_{\theta_1} f_\theta(x), \ldots, \partial_{\theta_{L+1}} f_\theta(x)$ at the cost of only $O(L)$ operations!

The only thing needed to improve the complexity from $O(L!)$ to $O(L)$ was to change the order of calculations and to store $O(L)$ values, which is the essence of the backpropagation algorithm. The calculation we did for a very simple network also generalizes to wider neural networks. Finally, some important misconceptions about the backpropagation algorithm are that it refers to an optimization method and that it yields approximations of the gradients, whereas it merely is an efficient way to *exactly* calculate gradients of a neural network (or other functions with a compositional structure).

1.3.3 Generalization

When training a neural network, the accuracy of the trained network can be monitored using the training loss. In the first place, the training error can be decreased by training for a higher number of epochs. If this does not work, it might be possible that the hypothesis space is too restrictive and that for no neural network of the chosen architecture the loss function can be made sufficiently small. This scenario is called *underfitting*. When a model is underfitting, one needs to make the hypothesis space larger. In the context of deep learning, this comes down to increasing the network size or changing the network architecture until the neural network approximates the objective function well on the training samples. This is however no guarantee that the network will perform well on unseen data as the network might have 'memorized' the training data. This scenario is called *overfitting*.

The performance of a neural network on unseen data is called the **generalization** of the network. If a well-trained network also performs well on unseen data, one says that it *generalizes well*. The error of a trained neural network on the whole domain (i.e. generally on an infinite amount of unseen data) is called the **generalization error** and can not be calculated directly. It is however common practice to approximate the generalization error by calculating the error on a finite validation set $\mathcal{V}$ that is disjunct of the training set $\mathcal{S}$. If such a validation set can not be generated on demand, one can sacrifice a small part (e.g. $5 - 10\%$) of the the training set as validation set. The validation loss

$$\mathcal{E}_{\text{val}}(\theta) = \frac{1}{|\mathcal{V}|} \sum_{x \in \mathcal{V}} |f(x) - f_\theta(x)|^p \tag{1.30}$$

can then be used as a proxy for the generalization error. If the data points of the validation set are chosen as quadrature points, resp. random points, then the accuracy of this approximation can be quantified using arguments from deterministic numerical integration, resp. Monte Carlo integration. If the neural network is overfitting, then the validation error will start increasing at some point eventhough the training error will keep decreasing (Figure 1.5). As a consequence, the generalization error will also start increasing i.e., the neural network does not generalize well to unseen data.

In Section 2.4, the connection between the training loss and the generalization error will be further discussed.

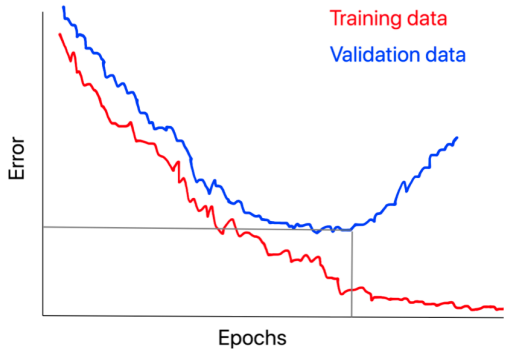


Fig. 1.5: Evolution of the training and validation error when a neural network is overfitting.

Chapter 2
Neural network theory

In this chapter, we investigate the approximation capabilities of neural networks. After stating some basic properties of neural networks in the first section, we prove a number of rigourous results on the expressivity of neural networks. Next, we introduce and discuss the so-called curse of dimensionality, and how to overcome it. In the last section, we prove an upper bound on the error of a trained neural network.

2.1 Introduction

In these lecture notes, we will use the general term *neural networks* to refer to multilayer perceptrons, unless we explicitly state otherwise.

Definition 2.1 Let $d_{in}, d_{out}, L \in \mathbb{N}$ and let $\sigma : \mathbb{R} \to \mathbb{R}$ be a function. Let $d_0 = d_{in}$, $d_{L+1} = d_{out}$, $W_k \in \mathbb{R}^{d_{k+1} \times d_k}$, $b_k \in \mathbb{R}^{d_{k+1}}$ and $C_k : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}} : x \mapsto W_k x + b_k$ for $d_k \in \mathbb{N}$, $0 \leq k \leq L$. A **multilayer perceptron (MLP)** with L layers, activation function σ, input dimension d_{in} and output dimension d_{out} is a function $f_\theta : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ of the form

$$f_\theta(x) = (C_L \circ \sigma \circ \cdots \circ \sigma \circ C_0)(x) \quad \text{for all } x \in \mathbb{R}^{d_{in}}, \tag{2.1}$$

where σ is applied element-wise and $\theta = ((W_0, b_0), \ldots, (W_L, b_L))$.

In mathematics literature, contrary to computer science literature, not f_θ but θ is called a **neural network**. Instead, f_θ is called the *realization* of neural network θ. This might seem superfluous, but making this distinction can be important as very often there might be different networks $\theta_1 \neq \theta_2$ that realize the same function $f_{\theta_1} = f_{\theta_2}$. As a consequence, it technically does not make sense to speak of the number of neurons and layers of f_θ as their might be many $\theta_1, \theta_2 \ldots$ of different sizes that realize the same function $f_{\theta_1} = f_{\theta_2} = \ldots$

Example 2.1 Consider the neural networks $\theta_1 = ((2, 0), (1, 0), (\frac{1}{2}, 0))$ and $\theta_2 = ((1, 0), (1, 0))$ with the ReLU activation function. Clearly, $\theta_1 \neq \theta_2$, but on the other

hand it holds for all $x \in \mathbb{R}$ that

$$f_{\theta_1}(x) = \frac{1}{2}\mathrm{ReLU}(\mathrm{ReLU}(2x) = \mathrm{ReLU}(x) = f_{\theta_2}(x). \tag{2.2}$$

Nevertheless, we will follow common practice and also use the term *neural network* for the function f_θ. When speaking of the number of neurons or layers of f_θ, we will actually mean the number of neurons or layers of θ.

Instead of a tuple of matrices and vectors, θ can also be thought of as one single vector $\theta \in \Theta \subseteq \mathbb{R}^{d_\Theta}$. Often the parameter space Θ will just be $\mathbb{R}^{d_\Theta}$, but sometimes we will need to consider a bounded parameter space $[-R, R]^{d_\Theta}$ for some $R > 0$.

Exercise 2.1 For an MLP as in Definition 2.1, the dimension of the space of trainable parameters Θ is equal to $d_\Theta = \sum_{k=0}^{L}(d_k + 1)d_{k+1}$.

An important consequence of the definition of an MLP (Definition 2.1) is that the composition of two MLPs is again an MLP. Similarly, the parallelization and even the sum of two MLPs with the same number of layers is again an MLP. As we will use these two properties of MLPs frequently throughout this chapter, we will formally state them.

Lemma 2.1 *Let $L, L', d_0, \ldots, d_L, d'_0, \ldots, d'_{L'} \in \mathbb{N}$ and $d_L = d'_0$. Let $f_{\theta_1} : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ be an MLP with L layers with layer widths $d_0, \ldots, d_L$ and let $f_{\theta_2} : \mathbb{R}^{d'_0} \to \mathbb{R}^{d'_{L'}}$ be an MLP with L' layers with layer widths $d'_0, \ldots, d'_{L'}$. Then there is an MLP $f_{\theta_3} : \mathbb{R}^{d_0} \to \mathbb{R}^{d'_{L'}}$ with $L + L'$ layers with layer widths $d_0, \ldots, d_L, d'_1, \ldots d'_{L'}$ such that*

$$f_{\theta_3}(x) = (f_{\theta_2} \circ f_{\theta_1})(x) \quad \textit{for all } x \in \mathbb{R}^{d_0}. \tag{2.3}$$

Proof Check that the MLP f_{θ_3} where the weights and biases are defined as

$$\theta_3 = ((W_0, b_0), \ldots (W_{L-1}, b_{L-1}), (W'_0 \cdot W_L, W'_0 \cdot b_L + b'_0), (W'_1, b'_1), \ldots (W'_L, b'_L)) \tag{2.4}$$

satisfies that $f_{\theta_3}(x) = f_{\theta_2} \circ f_{\theta_1}$. □

Lemma 2.2 *Let $L, d_0, \ldots, d_L, d'_0, \ldots, d'_{L'} \in \mathbb{N}$. Let $f_{\theta_1} : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ be an MLP with L layers with layer widths $d_0, \ldots, d_L$ and let $f_{\theta_2} : \mathbb{R}^{d'_0} \to \mathbb{R}^{d'_{L'}}$ be an MLP with L layers with layer widths $d'_0, \ldots, d'_L$. Then there is an MLP $f_{\theta_3} : \mathbb{R}^{d_0+d'_0} \to \mathbb{R}^{d_L+d'_L}$ with L layers with layer widths $d_0 + d'_0, \ldots, d_L + d'_L$ such that*

$$f_{\theta_3}(x) = (f_{\theta_1}(x_1, \ldots, x_{d_0}), f_{\theta_2}(x_{d_0+1}, , \ldots, x_{d_0+d'_0})) \quad \textit{for all } x \in \mathbb{R}^{d_0+d'_0}. \tag{2.5}$$

Exercise 2.2 Prove Lemma 2.2 by explicitly writing down the parameter vector θ_3.

Exercise 2.3 Prove that the sum of two MLPs with the same number of layers, denoted by $f_{\theta_1} : \mathbb{R}^{d_0} \to \mathbb{R}$ and $f_{\theta_2} : \mathbb{R}^{d_0} \to \mathbb{R}$, is again an MLP. In other words, prove that there is an MLP $f_{\theta_3} : \mathbb{R}^{d_0} \to \mathbb{R}$ such that

$$f_{\theta_3}(x) = f_{\theta_1}(x) + f_{\theta_2}(x) \quad \text{for all } x \in \mathbb{R}^{d_0}. \tag{2.6}$$

2.2 Neural network approximation

In this section we will investigate how well neural networks (i.e. MLPs) can approximate functions that belong to a given function class. We first focus on approximation results by neural networks with only one hidden layer, also called **shallow neural networks** (corresponding to $L = 1$ in Definition 2.1). Neural networks with two or more hidden layers (i.e. $L \geq 2$) are referred to as **deep neural networks** and their expressivity will be discussed in the next section.

2.2.1 Shallow neural networks

In the last decades, many papers have been published that show that neural networks are universal approximators for a multitude of function classes, under varying conditions on the network architecture and the choice of the activation function. Whereas earlier results focus on neural networks of only one hidden layer, more recent results investigate the expressive power of deep neural networks. One of the first famous results is due to Cybenko, who proved in [11] that shallow neural networks are universal approximators for continuous functions when a so-called sigmoidal activation function is used (e.g. tanh, sigmoid). Leshno, Lin, Pinkus and Schocken [39] generalized his result to the following:

Theorem 2.1 *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a function that is locally essentially bounded with the property that the closure of the set of points of discontinuity has zero Lebesgue measure. For $1 \leq j \leq n$, let $\alpha_j, \theta_j \in \mathbb{R}$ and $y_j \in \mathbb{R}^d$. Then finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \quad x \in \mathbb{R}^d \tag{2.7}$$

are dense in $C(\mathbb{R}^d)$ if and only if σ is not an algebraic polynomial.

One can check that all popular activation functions satisfy the stated conditions. Theorem 2.1 therefore proves that shallow neural networks are universal approximators, since density in $C(\mathbb{R}^d)$ implies that for every function $g \in C(\mathbb{R}^d)$ and every $\epsilon > 0$ there will be a neural network G as in the theorem such that $\|g - G\|_\infty < \epsilon$. This result, together with many other similar results from that time, are *existence results*: they only guarantee the existence of a neural network that achieves the wanted accuracy, but they convey no information on the values of N, α_j or y_j. For all we know, N may be equal to 10^{100}, which makes the result useless in practice. Hence, it is clear that approximation results as Theorem 2.1 are not satisfactory enough. In particular since from a practical point of view, the following question is of extreme importance:

Given a function class $\mathcal{F}$, a function $f \in \mathcal{F}$ and a desired approximation accuracy $\epsilon > 0$, how large should the neural network f_θ be such that $\|f - f_\theta\|_{\mathcal{F}} < \epsilon$?

One way to answer this question is to explicitly construct the neural network f_θ for every function f in a given function class. A function class we will frequently encounter is that of k times continuously differentiable functions. For functions with domain D, we denote this space by $C^k(D)$. For $k = 0$, $C^0(D)$ refers to the space of continuous functions with domain D. The space $C^0(D)$ is equipped with the following norm,

$$\|f\|_{C^0(D)} = \|f\|_\infty = \max_{x \in D} |f(x)| \tag{2.8}$$

for $f \in C^0(D)$. We will frequently use this norm to measure the approximation error. An example of an approximation result for shallow neural networks is given in Lemma 2.3 below, where we prove that a shallow ReLU neural network with $O(N)$ neurons can approximate a twice continuously differentiable function to an accuracy of $O(N^{-2})$.

Lemma 2.3 *Let $f \in C^2([0,1])$. For any $N \in \mathbb{N}$, there exists a shallow ReLU neural network $\widehat{f_N}$ with $N + 3$ neurons such that*

$$\left\| f - \widehat{f_N} \right\|_{C^0([0,1])} \leq \frac{1}{8N^2} \|f''\|_{C^0([0,1])}. \tag{2.9}$$

Proof We will explicitly construct the shallow ReLU neural network $\widehat{f_N}$ as a linear interpolation of f on a uniform grid and prove that the resulting network satisfies the claimed bounds. Let $N \in \mathbb{N}$ a natural number and define $x_j = j/N$ for all $-1 \leq j \leq N + 1$. For every $j \in \{0, \dots, N\}$ we define the hat function ρ_j through

$$\rho_j(x) = \frac{\mathrm{ReLU}(x - x_{j-1}) - \mathrm{ReLU}(x - x_j)}{x_j - x_{j-1}} - \frac{\mathrm{ReLU}(x - x_j) - \mathrm{ReLU}(x - x_{j+1})}{x_{j+1} - x_j}. \tag{2.10}$$

Note that the functions ρ_j form a partition of unity on $[0,1]$, i.e. $\sum_j \rho_j \equiv 1$ on $[0,1]$. We then define the approximation

$$\widehat{f_N}(x) = \sum_{j=0}^{N} f(x_j)\rho_j(x). \tag{2.11}$$

This formula corresponds to a function that linearly interpolates f at the grid points x_j. This can be seen since every ρ_j is piecewise affine and since $\rho_j(x_i) = 0$ for all $i \neq j$ and $\rho_j(x_j) = 1$. The maximum error of an approximation by linear interpolation is given by

$$\left\| f - \widehat{f_N} \right\|_{C^0([0,1])} \leq \frac{1}{8N^2} \|f''\|_{C^0([0,1])}. \tag{2.12}$$

We now slightly rewrite (2.11) by collecting all terms with $\mathrm{ReLU}(x - x_j)$ and using that $x_j - x_{j-1} = 1/N$,

$$\begin{aligned}\widehat{f_N}(x) = &\sum_{j=0}^{N} \left(\frac{f(x_{j+1}) - 2f(x_j) - f(x_{j-1})}{1/N} \right) \mathrm{ReLU}(x - x_j) \\ &+ \frac{f(0)}{1/N}\mathrm{ReLU}(x - x_{-1}) + \frac{f(1)}{1/N}\mathrm{ReLU}(x - x_{N+1}).\end{aligned} \tag{2.13}$$

This rewriting reveals that $\widehat{f_N}$ is a ReLU neural network with one hidden layer consisting of $N+3$ neurons. The weights and biases can also be read off from the above formula. Following the notation of Definition 2.1, we find that $W_0, b_0, W_1 \in \mathbb{R}^{N+3}$ and $b_1 \in \mathbb{R}$ are defined as

$$W_0 = (1, \dots, 1), \quad (b_0)_{j+2} = -x_j \text{ for } -1 \le j \le N+1, \tag{2.14}$$

$$(W_1)_{j+2} = \frac{f(x_{j+1}) - 2f(x_j) - f(x_{j-1})}{1/N} \text{ for } 0 \le j \le N, \tag{2.15}$$

$$(W_1)_1 = \frac{f(0)}{1/N}, \quad (W_1)_{N+3} = \frac{f(1)}{1/N}, \quad b_1 = 0. \tag{2.16}$$

□

Now suppose we want to approximate the function $f : [0,1] \to \mathbb{R} : x \mapsto \sin(2\pi x)$ to an accuracy of 1% in C^0-norm. Lemma 2.3 guarantees the existence of a shallow ReLU neural network with 26 neurons that reaches this accuracy. It is also important to realize what Lemma 2.3 does *not* claim:

- Lemma 2.3 does *not* guarantee that if we train a ReLU network with 26 neurons that the trained network will only have an error of 1%. The error of the trained network depends on many factors, including the training set size, the optimization procedure etc.
- Lemma 2.3 does *not* claim that there is no ReLU network with strictly less than 26 neurons that still reaches an accuracy of 1%. In the proof of the lemma, we only used that $f \in C^2([0,1])$, but in this example f is infinitely many times continuously differentiable. One can imagine that this additional information can be used to prove sharper bounds.

Lemma 2.3 is only a first example of the kind of results we are interested in proving. We want to obtain results for general input dimension d, other activation functions than ReLU and for more general function classes, in particular $C^k([0,1]^d)$. Finally, we want to investigate whether *deep* neural networks are more efficient at approximating functions than shallow neural networks. This is the topic of the next section.

2.2.2 Deep neural networks

Almost all famous neural network applications involve deep neural networks, i.e. networks with two or more hidden layers, rather than shallow ones. Shallow neural networks, however, are already provably universal approximators, so why would it be advantageous to add more layers to the neural network? In this section we try to answer this question from the point of view of function approximation. In particular, we will formulate a general, rigorous result on the expressivity of deep neural networks. Afterwards, we apply this result to ReLU and tanh neural networks and discuss some specific details.

Below, we will prove a general result on the expressivity of neural networks regarding the approximation of functions in the space $C^k([0,1]^d)$. Instead of focusing on a particular neural networks with a particular activation function, we state three very general assumptions that should be met in order for the result to hold. We will show later on that ReLU neural networks and tanh neural networks satisfy these assumptions. In short, the three requirements are that the map $x \mapsto x$, the map $x \mapsto x^2$ and a partition of unity can be approximated using neural networks with some chosen activation function. The precise formulation of the assumptions is as follows:

1. *Approximation of $x \mapsto x$.* For every $\epsilon > 0$, there has to exist a neural network $f_1^\epsilon : [-1,1] \to [-1,1]$ of width W_1^ϵ and depth L_1^ϵ such that $\max_{x\in[1,1]} \left|f_1^\epsilon(x) - x\right| < \epsilon$. Moreover, we require that $\mathrm{Lip}(f_1^\epsilon) \leq 2$.
2. *Approximation of $x \mapsto x^2$.* For every $\epsilon > 0$, there has to exist a neural network $f_2^\epsilon : [-1,1] \to [-1,1]$ of width W_2^ϵ and depth $L_2^\epsilon = L_1^\epsilon$ such that $\max_{x\in[-1,1]} \left|f_2^\epsilon(x) - x^2\right| < \epsilon$. Moreover, we require that $\mathrm{Lip}(f_2^\epsilon) \leq 2$.
3. *Approximation of partition of unity.* For every $\epsilon > 0$, there exist a neural network $\Psi^\epsilon : \mathbb{R} \to [0,1]$ of width W_Ψ^ϵ and depth L_Ψ^ϵ, such that for any $N \in \mathbb{N}, 1 \leq j \leq N$, it holds for $\Psi_j^{N,\epsilon} = \Psi^\epsilon(N(x - x_j^N))$ that

$$\max_{x\in I_j^N} \left| \sum_{l=-1,0,1} \Psi_{j+l}^{N,\epsilon}(x) - 1 \right| < \epsilon \quad \text{and} \quad \max_{x\in I_j^N} \sum_{\substack{l\neq -1,0,1,\\ 1\leq j+l\leq N}} \left|\Psi_{j+l}^{N,\epsilon}(x)\right| < \epsilon, \quad (2.17)$$

   where $x_j^N = j/N$ and $I_j^N = [x_{j-1}^N, x_j^N]$.

Under these three assumptions, we prove an upper bound on the width and depth of a neural network so that it can approximate a $C^k([0,1]^d)$-function to a specified accuracy.

Theorem 2.2 *Let $d, k \in \mathbb{N}$ and $f \in C^k([0,1]^d)$. Assume that for every $\epsilon > 0$, there exists neural networks f_1^ϵ, f_2^ϵ and Ψ^ϵ that satisfy assumptions 1,2 and 3. Then for every $\epsilon > 0$, there exists a neural network $\widehat{f}^\epsilon$ of width $O(\epsilon^{-d/k}(W_1^\epsilon + W_2^{\epsilon^{(k+d)/k}} + W_\Psi^\epsilon))$ and depth $O((L_1^\epsilon + L_2^{\epsilon^{(k+d)/k}} + L_\Psi^\epsilon))$ such that*

$$\left\| f - \widehat{f}^{\epsilon} \right\|_{C^0([0,1])} < \epsilon. \tag{2.18}$$

Proof We will prove the theorem for $d = 1$. The full proof is similar to that of [14, Theorem 5.1].

We divide the unit interval into N subintervals of length $1/N$. On each of these subintervals, f can be approximated by a (Taylor) polynomial. The global approximation can then be constructed by multiplying each polynomial with the indicator function of the corresponding subinterval and summing over all subintervals. We then prove that replacing these polynomials, multiplications and indicator functions with suitable neural networks results in a new approximation that has approximately the same accuracy. In the last step we calculate the size of the required neural network.

Step 1: Construction of the approximation. For any $N \in \mathbb{N}$, $1 \leq j \leq N$, let $x_j^N = j/N$, $I_j^N = [x_{j-1}^N, x_j^N]$ and $J_j^N = [x_{j-2}^N, x_{j+1}^N]$. By Taylor's theorem, there exists a polynomial p_j^N of degree at most $k-1$ such that

$$\left\| f - p_j^N \right\|_{C^0(J_j^N)} < \frac{1}{k!} \left(\frac{3}{2N} \right)^k \|f\|_{C^k([0,1])}. \tag{2.19}$$

In order to approximate every polynomial p_j^N, we need to approximate all monomials x^p with $1 \leq p \leq k-1$. Already, we have that $x \approx f_1^{\epsilon}(x)$ and $x^2 \approx f_2^{\epsilon}(x)$. To approximate the higher-order monomials, we first approximate the multiplication of two real numbers. Using the trick that $xy = \frac{1}{4}((x+y)^2 - (x-y)^2)$, for $x, y \in \mathbb{R}$, we define the neural network for $\delta, M > 0$,

$$\widehat{\times}^{\delta}(x,y) = M^2 \left(f_2^{\delta} \left(\frac{x+y}{2M} \right) - f_2^{\delta} \left(\frac{x-y}{2M} \right) \right) \quad \text{for } x, y \in [-M, M]. \tag{2.20}$$

The rescaling with a factor $\frac{1}{2M}$ is necessary to make sure that $\frac{x \pm y}{2M}$ lies in $[-1, 1]$, the domain of f_2^{δ}. We are now ready to construct neural networks that approximate the monomials x^p for $1 \leq p \leq k-1$. We can for instance make the following approximation,

$$x^p \approx f_p^{\epsilon} := \widehat{\times}^{\epsilon}(f_{p-1}^{\epsilon}(x), f_1^{\epsilon}(x)). \tag{2.21}$$

There is however one problem with these approximations: we cannot directly approximate the polynomial $\sum_{i=0}^{k-1} a_i x^i$ by $\sum_{i=0}^{k-1} a_i f_p^{\epsilon}(x)$, as all the f_p^{ϵ} have a different depth. We will therefore construct neural networks $m_p^{\epsilon}(x) \approx x^p$ that all have the same depth for $1 \leq p \leq k-1$. Recall that f_1^{ϵ} has L_1^{ϵ} hidden layers and f_p^{ϵ}, $p \geq 2$, has $(p-1)L_1^{\epsilon}$ hidden layers. Therefore, we must make sure that every m_p^{ϵ} has $\max\{1, (k-2)\}L_1^{\epsilon}$ hidden layers. We do this by setting for

$$m_1^\epsilon(x) = \underbrace{(f_1^\epsilon \circ \cdots \circ f_1^\epsilon)}_{\max\{1,k-2\} \text{ times}}(x), \qquad m_2^\epsilon(x) = \underbrace{(f_1^\epsilon \circ \cdots \circ f_1^\epsilon)}_{k-3 \text{ times}}(f_2^\epsilon(x)), \quad (2.22)$$

$$m_p^\epsilon(x) = \underbrace{(f_1^\epsilon \circ \cdots \circ f_1^\epsilon)}_{k-1-p \text{ times}}\left(\widehat{\times}^\epsilon\left(m_{p-1}^\epsilon(x), \underbrace{(f_1^\epsilon \circ \cdots \circ f_1^\epsilon)}_{p-2 \text{ times}}(x)\right)\right), \quad (2.23)$$

for $k-1 \geq p \geq 3$. Using these m_p^ϵ, we will approximate every polynomial $p_j^N(x) = \sum_{i=0}^{k-1} a_i x^i$ by the neural network $q_j^N(x) = \sum_{i=0}^{k-1} a_i m_p^\epsilon(x)$. Using assumptions 1 and 2, we find that

$$\max_{x\in[-1,1]} \left|m_p^\epsilon(x) - x^p\right| \leq \max\{1,(k-2)\}2^{k-2}\epsilon. \quad (2.24)$$

And as a consequence we find that

$$\max_{1\leq j\leq N} \left\|p_j^N - q_j^N\right\|_{C^0([0,1])} \leq C\epsilon, \quad (2.25)$$

for a constant $C > 0$ that depends only on f and k. We then define our neural network approximation $\widehat{f^N}$ by multiplying each q_j^N by $\Psi_j^N := \Psi_j^{N,\epsilon}$ and summing over j,

$$\widehat{f^N} = \sum_{j=1}^N \widehat{\times}^\epsilon\left(q_j^N(x), \Psi_j^N(x)\right). \quad (2.26)$$

Step 2: Error estimate. Using the triangle inequality we find

$$\left\|f - \widehat{f^N}\right\|_{C^0([0,1])} \leq \left\|f - \sum_{j=1}^N f\cdot\Psi_j^N\right\|_{C^0([0,1])} + \left\|\sum_{j=1}^N (f - q_j^N)\cdot\Psi_j^N\right\|_{C^0([0,1])} \quad (2.27)$$

$$+\left\|\sum_{j=1}^N q_j^N\cdot\Psi_j^N - \sum_{j=1}^N \widehat{\times}^\epsilon\left(q_j^N, \Psi_j^N\right)\right\|_{C^0([0,1])} \quad (2.28)$$

We bound each term on the right-hand side separately. Let $1 \leq i \leq N$ be arbitrary. We can bound the first term as

$$\left\|f - \sum_{j=1}^N f\cdot\Psi_j^N\right\|_{C^0(I_i^N)} \leq \left\|f - \sum_{l=-1}^1 f\cdot\Psi_{i+l}^N\right\|_{C^0(I_i^N)} + \left\|\sum_{\substack{l\neq -1,0,1,\\ 1\leq i+l\leq N}} f\cdot\Psi_{i+l}^N\right\|_{C^0(I_i^N)} \quad (2.29)$$

$$\leq 2\|f\|_{C^0([0,1])}\epsilon, \quad (2.30)$$

where we used assumption 3. For the second term it holds that

$$\left\|\sum_{j=1}^{N}(f-q_j^N)\cdot\Psi_j^N\right\|_{C^0(I_i^N)} \leq \left\|\sum_{l=-1}^{1}(f-q_{i+l}^N)\cdot\Psi_{i+l}^N\right\|_{C^0(I_i^N)} \tag{2.31}$$

$$+\left\|\sum_{\substack{l\neq-1,0,1,\\ 1\leq i+l\leq N}}(f-q_{i+l}^N)\cdot\Psi_{i+l}^N\right\|_{C^0(I_i^N)} \tag{2.32}$$

$$\leq \max_{l=-1,0,1}\left\|f-q_{i+l}^N\right\|_{C^0(I_i^N)}\left\|\sum_{l=-1}^{1}\Psi_{i+l}^N\right\|_{C^0(I_i^N)} \tag{2.33}$$

$$+\max_{1\leq j\leq N}\left\|f-q_j^N\right\|_{C^0(I_i^N)}\sum_{\substack{l\neq-1,0,1,\\ 1\leq i+l\leq N}}\left\|\Psi_{i+l}^N\right\|_{C^0(I_i^N)} \tag{2.34}$$

$$\leq \left(\frac{1}{k!}\left(\frac{3}{2N}\right)^k\|f\|_{C^k([0,1])}+C\epsilon\right)(1+\epsilon) \tag{2.35}$$

$$+\left(\max_{0\leq s\leq k-1}\frac{\|f\|_{C^s([0,1])}}{s!}+\epsilon\right)\epsilon, \tag{2.36}$$

where we used assumption 3 and (2.19). For the final term, we find that

$$\left\|\sum_{j=1}^{N}q_j^N\cdot\Psi_j^N-\sum_{j=1}^{N}\widehat{\times}^\delta\left(q_j^N,\Psi_j^N\right)\right\|_{C^0([0,1])}\leq N\left\|\widehat{\times}^\delta-\times\right\|_\infty\leq N\delta \tag{2.37}$$

Combining everything, we find that

$$\left\|f-\widehat{f}^N\right\|_{C^0([0,1])}\leq C'(N^{-k}+\epsilon+N\delta), \tag{2.38}$$

for some $C'>0$. Therefore we should set $N=\epsilon^{-1/k}$ and $\delta=\epsilon^{\frac{k+1}{k}}$ in order to obtain that $\left\|f-\widehat{f}^N\right\|_{C^0([0,1])}=O(\epsilon)$.

Step 3: Network size. For the approximation of the monomials, we need a subnetwork of width $O(W_1^\epsilon+W_2^\epsilon)$ and depth $O(L_1^\epsilon)$. For the construction of all Ψ_j^N, we need a (paralellel) subnetwork of width $O(NW_\Psi^\epsilon)=O(\epsilon^{-1/k}W_\Psi^\epsilon)$ and depth $O(L_\Psi^\epsilon)$. We might need another subnetwork of width $O(NW_1^\epsilon)$ to make the two previous of the same depth. For the multiplication of all q_j^N and Ψ_j^N, we need a subnetwork of width $O(NW_2^{\epsilon^{(k+1)/k}})=O(\epsilon^{-1/k}W_2^{\epsilon^{(k+1)/k}})$ and depth $O(L_1^{\epsilon^{(k+1)/k}})$. The total width of the network is therefore $O(\epsilon^{-1/k}(W_1^\epsilon+W_2^{\epsilon^{(k+1)/k}}+W_\Psi^\epsilon))$ and the total depth is $O((L_1^\epsilon+L_2^{\epsilon^{(k+1)/k}}+L_\Psi^\epsilon))$. □

Exercise 2.4 Prove equation (2.24).

2.2.2.1 ReLU neural networks

It is now natural to apply the results of Theorem 2.2 for ReLU neural networks. Therefore, we must construct ReLU networks $f_1^\epsilon, f_2^\epsilon$ and Ψ^ϵ (cf. the assumptions of the previous section) and quantify their sizes. From this, an expressivity result for ReLU neural networks directly follows.

1. Approximation of $x \mapsto x$. For any $x \in \mathbb{R}$, it holds that $f_1^\epsilon(x) := \mathrm{ReLU}(x) - \mathrm{ReLU}(-x) = x$. Therefore, the identity function can be exactly represented using a shallow ReLU neural network of width 2.

Exercise 2.5 Prove that for any $L \in \mathbb{N}$ there exists a ReLU neural network of depth L and width 2 that exactly represents the identity function.

2. Approximation of $x \mapsto x^2$. The square function cannot be exactly represented by a ReLU neural network. The universal approximation theorem guarantees the existence of an arbitrarily accurate approximation, but does not provide any bounds on the needed network size. Alternatively, Lemma 2.3 could be used, but it is possible to do better. Yarotksy [72] proved that the square function can be very efficiently approximated by a ReLU neural network. This result, presented below, is still a key ingredient in most constructive proofs on ReLU neural networks.

Lemma 2.4 *For any $\epsilon > 0$, there exists a ReLU neural network $f_2^\epsilon : [-1,1] \to [0,1]$ of width $O(\ln(1/\epsilon))$ and depth $O(\ln(1/\epsilon))$ such that f_2^ϵ satisfies the error bound*

$$\max_{x\in[-1,1]} \left| f_2^\epsilon(x) - x^2 \right| < \epsilon. \tag{2.39}$$

Proof In [72, Proposition 2], a parametrized class of ReLU neural networks q_m, $m \in \mathbb{N}$, that approximate $q : [0,1] \to [0,1] : x \mapsto x^2$ is constructed. For $m \in \mathbb{N}$, q_m is piecewise affine and for $x \in I_k^m = [k/2^m, (k+1)/2^m]$, $0 \le k \le 2^m - 1$, given by

$$q_m(x) = \left(\frac{k}{2^m}\right)^2 + \left(x - \frac{k}{2^m}\right)\frac{2k+1}{2^m}. \tag{2.40}$$

The function q_m is a piecewise linear interpolant of q on the dyadic grid with grid size 2^{-m} and therefore approximates q to an accuracy of 2^{-2m-2}. Moreover, it can be efficiently written as a ReLU neural network using the following argument due to Telgarsky [70]. Consider the 'tooth' function $g : [0,1] \to [0,1]$,

$$g(x) = \begin{cases} 2x, & x < \frac{1}{2}, \\ 2(1-x), & x \ge \frac{1}{2}, \end{cases} \quad \text{and} \quad g_s(x) = \underbrace{(g \circ \cdots \circ g)}_{s \text{ times}}(x). \tag{2.41}$$

Telgarsky showed that $q_{m-1}(x) - q_m(x) = 2^{-2m} g_m(x)$ and hence that

$$q_m(x) = x - \sum_{s=1}^{m} \frac{g_s(x)}{2^{2s}}. \tag{2.42}$$

Since $x = \mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$ and $g(x) = 2\mathrm{ReLU}(x) - 4\mathrm{ReLU}(x - \frac{1}{2}) + 2\mathrm{ReLU}(x - 1)$, q_m can be written as a ReLU neural network of depth and width $O(m)$. We can then define $f_2^\epsilon(x) = q_m(|x|)$ with e.g. $m = \lceil \ln(1/\epsilon) \rceil$ to obtain the statement of the lemma. □

Exercise 2.6 Argue that $\mathrm{Lip}(f_2^\epsilon) \leq 2$, as required by the second assumption needed for Theorem 2.2.

Exercise 2.7 Explain why the function $x \mapsto q_m(|x|)$ from the last part of the proof of Lemma 2.4 can be represented by a ReLU neural network of the stated size.

A comparison of Lemma 2.4 with Lemma 2.3 for $x \mapsto x^2$ reveals already one advantage of using a deep ReLU network over a shallow one: whereas a shallow ReLU network needs $O(\sqrt{\epsilon})$ neurons to approximate the square function to an accuracy of $\epsilon > 0$, a deep ReLU network only needs $O(\ln(1/\epsilon))$ neurons.

3. Approximation of partition of unity. Next, we show that it is possible to create an *exact* partition of unity using ReLU neural networks. For every $\epsilon > 0$, we define

$$\Psi^\epsilon(x) = \mathrm{ReLU}\left(x + \frac{1}{2}\right) - 2\mathrm{ReLU}\left(x - \frac{1}{2}\right) + \mathrm{ReLU}\left(x - \frac{3}{2}\right), \tag{2.43}$$

such that $L_\Psi^\epsilon = 1$ and $W_\Psi^\epsilon = 3$. Then for $N \in \mathbb{N}$, $1 \leq j \leq N$, the functions $\Psi_j^{N,\epsilon} = \Psi^\epsilon(N(x - j/N))$ constitute a partition of unity.

Exercise 2.8 Sketch the ReLU neural network Ψ^ϵ and prove that it indeed satisfies (2.17).

Exercise 2.9 Give an example of a different choice of Ψ^ϵ such that $L_\Psi^\epsilon = 1$ and $W_\Psi^\epsilon = 4$.

Applying the bounds established above to Theorem 2.2, we find the following result on the expressivity of ReLU neural networks.

Corollary 2.1 *Let $d, k \in \mathbb{N}$ and $f \in C^k([0,1]^d)$. For every $\epsilon > 0$, there exists a ReLU neural network $\widehat{f^\epsilon}$ of width $O(\epsilon^{-d/k} \ln(1/\epsilon))$ and depth $O(\ln(1/\epsilon))$ such that*

$$\left\| f - \widehat{f^\epsilon} \right\|_{C^0([0,1]^d)} < \epsilon. \tag{2.44}$$

Exercise 2.10 Check that Corollary 2.1 indeed follows from Theorem 2.2.

We see that Corollary 2.1 provides a much better approximation rate than Lemma 2.3 if k is high enough.

2.2.2.2 Tanh neural networks

Next, we apply the results of Theorem 2.2 to tanh neural networks. We thus must construct tanh networks f_1^ϵ, f_2^ϵ and Ψ^ϵ (cf. the assumptions of the previous section) and quantify their sizes. Let in the following σ denote the tanh function.

1. Approximation of $x \mapsto x$. Unlike with ReLU neural networks, the identity function cannot be exactly represented using a tanh neural network. Therefore, for every $\epsilon > 0$, we have to construct a neural network $f_1^\epsilon : [-1, 1] \to [-1, 1]$ such that $\max_{x\in[-1,1]} |f_1(x) - x| < \epsilon$ and $\mathrm{Lip}(f_1^\epsilon) \leq 2$. Using a Taylor expansion, we make the following observation,

$$\frac{\sigma(xh) - \sigma(-xh)}{2xh} = \sigma'(0) + O((xh)^2). \tag{2.45}$$

Taking into account that $x \in [-1, 1]$ then gives that

$$\frac{\sigma(xh) - \sigma(-xh)}{2h\sigma'(0)} = \frac{\sigma(xh)}{h\sigma'(0)} = x + O(h^2). \tag{2.46}$$

Hence, we can take $f_1^\epsilon(x) = \frac{\sigma(xh)}{h\sigma'(0)}$ for h small enough. We find that $W_1^\epsilon = L_1^\epsilon = 1$ if $h = O(\sqrt{h})$.

2. Approximation of $x \mapsto x^2$. We can define f_2^ϵ in a similar way, by letting $x_0 \in \mathbb{R}$ be such that $\sigma''(x_0) \neq 0$,

$$f_2^\epsilon(x) = \frac{\sigma(x_0 + xh) - 2\sigma(x_0) + \sigma(x_0 - xh)}{4h^2\sigma''(x_0)} = x^2 + O(h^2) \tag{2.47}$$

We find that $W_2^\epsilon = 3$ and $L_2^\epsilon = 1$.

Exercise 2.11 Argue that the conditions $\mathrm{Lip}(f_2^\epsilon) \leq 2$ and $\mathrm{Lip}(f_2^\epsilon) \leq 2$ are satisfied when $h > 0$ is small enough, as required by the first two assumptions for Theorem 2.2.

Using similar techniques, one can give another argument as to why deep neural networks are more expressive than shallow ones. In [40], one has proven that it is impossible to approximate the multiplication of d real numbers, i.e. the map $(x_1, \ldots, x_d) \mapsto \prod_{i=1}^d x_i$, with a shallow network with strictly less than 2^d neurons. On the other hand, for d even, $\prod_{i=1}^d x_i$ can be approximated using a deep neural network with $4d$ neurons distributed over $\log_2(d)$ hidden layers (see Exercise 2.12). In this case, a deep neural network thus needs exponentially less neurons to achieve the same accuracy, which clearly indicates the advantages of depth.

Exercise 2.12 Let d be an even natural number and let $x_0 \in \mathbb{R}$ such that $\sigma''(x_0) \neq 0$. First prove that for $x, y \in [-1, 1]$ the expression

$$\widehat{g}(x, y) = \frac{\sigma(x_0 + xh + yh) - \sigma(x_0 + xh - yh) - \sigma(x_0 - xh + yh) + \sigma(x_0 - xh - yh)}{4h^2\sigma''(x_0)} \tag{2.48}$$

is a suitable approximation of xy. Use $\widehat{g}$ to construct a network with $4d$ neurons distributed over $\log_2(d)$ hidden layers that approximates $\prod_{i=1}^d x_i$. Hint: generalize the observation that $x_1x_2x_3x_4 \approx \widehat{g}(\widehat{g}(x_1, x_2), \widehat{g}(x_3, x_4))$.

3. Approximation of partition of unity. In constrast to ReLU neural networks, an exact partition of unity cannot be created using tanh neural networks. Luckily,

the sigmoidal shape of the tanh activation function makes it easy to construct an approximate partition of unity, which we define by

$$\Psi_j^{N,\epsilon}(x) = \frac{1}{2}\sigma\left(\alpha_\epsilon N(x-(j-1)/N)\right) - \frac{1}{2}\sigma\left(\alpha_\epsilon N(x-j/N)\right) \tag{2.49}$$

where $\alpha_\epsilon = \sigma^{-1}\left(1-\frac{\epsilon}{N}\right)$ for any $\epsilon > 0$ and $N \in \mathbb{N}$. Recalling the exact statement (2.17) of the third requirement for Theorem 2.2, we see that it must be true that

$$\sum_{l=-1,0,1} \Psi_{j+l}^{N,\epsilon}(x) \approx 1 \quad \text{and} \quad \sum_{\substack{l\neq -1,0,1,\\ 1\leq j+l\leq N}} \Psi_{j+l}^{N,\epsilon}(x) \approx 0 \quad \text{for all } x \in I_j^N = \left[\frac{j-1}{N}, \frac{j}{N}\right]. \tag{2.50}$$

Figure 2.1 visualizes that this is indeed the case by showing an example where $N = 7$ and $j = 4$. Thin blue lines correspond to the $\Psi_{j+l}^{N,\epsilon}$ in the sum of the left, whereas the sum itself is shown as a thick blue line, and similarly for the right sum (in red). The region of interest (I_4^7) is shown in gray. Note that for the leftmost and rightmost interval a slightly different definition of $\Psi_j^{N,\epsilon}$ is used, in order to account for border phenomena. More specifically, one can define

$$\Psi_1^{N,\epsilon}(x) = \frac{1}{2} - \frac{1}{2}\sigma\left(\alpha_\epsilon(Nx-1)\right), \quad \Psi_N^{N,\epsilon}(x) = \frac{1}{2} + \frac{1}{2}\sigma\left(\alpha_\epsilon(Nx-N+1)\right). \tag{2.51}$$

Another approach would be to introduce a ghost cell on each side of the grid and letting j range from -1 to $N+1$ instead of 0 to N.

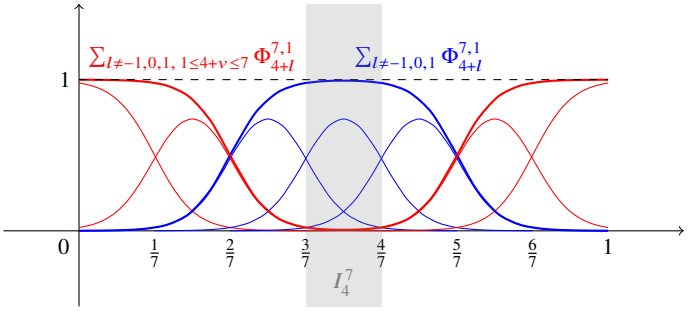


Fig. 2.1: Example of an approximate partition of unity on $[0,1]$ with $N = 7$. The thin lines represent the $\Psi_j^{7,1}$, $1 \leq j \leq 7$.

The following lemma proves that the constructed approximate partition of unity indeed satisfies requirement (2.17).

Lemma 2.5 *For $\epsilon > 0$, it holds that,*

$$\left\| \sum_{l=-1}^{1} \Psi_{j+l}^{N,\epsilon} - 1 \right\|_{C^0(I_j^N)} \leq \epsilon \quad \textit{and} \quad \sum_{\substack{l \neq -1,0,1, \\ 1 \leq j+l \leq N}} \left\| \Psi_j^{N,\epsilon} \right\|_{C^0(I_j^N)} \leq \epsilon. \tag{2.52}$$

Proof First we see that for all $x \in I_j^N$ it holds that

$$\sum_{l=-1}^{1} \Psi_{j+l}^{N,\epsilon}(x) = \frac{1}{2}\sigma\left(\alpha_\epsilon N(x - x_{j-2}^N)\right) - \frac{1}{2}\sigma\left(\alpha_\epsilon N(x - x_{j+1}^N)\right) \leq 1. \tag{2.53}$$

On the other hand, for all $x \in I_j^N$, it holds that

$$\frac{1}{2}\sigma\left(\alpha_\epsilon N(x - x_{j-2}^N)\right) - \frac{1}{2}\sigma\left(\alpha_\epsilon N(x - x_{j+1}^N)\right) \geq \sigma(\alpha_\epsilon) \geq 1 - \epsilon. \tag{2.54}$$

These inequalities already prove the first part of the lemma. Now, for $|l| \geq 2$, it holds that

$$\left|\Psi_{j+l}^{N,\epsilon}(x)\right| \leq \frac{1}{2}\sigma(2\alpha_\epsilon) - \frac{1}{2}\sigma(\alpha_\epsilon) \tag{2.55}$$

$$= \frac{1}{2}\sigma(\alpha_\epsilon)\left(1 - \sigma(2\alpha_\epsilon)\,\sigma(\alpha_\epsilon)\right) \tag{2.56}$$

$$\leq \frac{1}{2}\left(1 - \sigma^2(\alpha_\epsilon)\right) = \frac{1}{2}\left(1 - \left(1 - \frac{\epsilon}{N}\right)^2\right) \leq \frac{\epsilon}{N}. \tag{2.57}$$

□

We are now able to apply Theorem 2.2 to the setting of tanh neural networks. Using the constructed f_1^ϵ, f_2^ϵ and Ψ^ϵ, the upper bound on the size of $\widehat{f^\epsilon}$ becomes remarkable simple.

Corollary 2.2 *Let $d, k \in \mathbb{N}$ and $f \in C^k([0,1]^d)$. For every $\epsilon > 0$, there exists a tanh neural network $\widehat{f^\epsilon}$ of width $O(\epsilon^{-d/k})$ and depth $O(1)$ such that*

$$\left\| f - \widehat{f^\epsilon} \right\|_{C^0([0,1]^d)} < \epsilon. \tag{2.58}$$

Exercise 2.13 Check that Corollary 2.2 indeed follows from Theorem 2.2.

One consequence of Corollary 2.2 is that, unlike for ReLU neural networks, the depth of the neural network does not need to increase to obtain a better accuracy. Using similar arguments to the ones presented here, it is possible to upgrade the result from Corollary 2.2: in [14] it is proven that two hidden layers suffice to obtain the same rate. In addition, it is even possible to measure the accuracy in a higher-order C^s-norm (where $s \geq 1$),

$$\|f\|_{C^s([0,1]^d)} = \max_{\substack{\alpha_1,\dots\alpha_d \geq 0 \\ \sum_{i=1}^d \alpha_i \leq s}} \max_{x \in [0,1]^d} \left| \frac{\partial^{\sum_{i=1}^d \alpha_i} f}{\partial x_1^{\alpha_1} \cdots \partial x_d^{\alpha_d}} \right|. \tag{2.59}$$

If $\left\|f - \widehat{f}\right\|_{C^s([0,1]^d)}$ is small, then this means that not only that $\widehat{f}$ is a good approximation of f, but also that the partial derivatives (up to order s) of $\widehat{f}$ are good approximations of the partial derivatives of f. Theoretical results like this will be useful in the mathematical analysis of so-called physics-informed neural networks. Below we state a result on the approximation of C^k-function by tanh neural networks in higher-order norms. The explicit constants implied in the Landau notation are worked out in [14, 43]. The result also holds when the C^s norm

Theorem 2.3 *Let $d, k \in \mathbb{N}$, $s \in \mathbb{N}_0$ with $s < k$ and $f \in C^k([0,1]^d)$. For every $\epsilon > 0$, there exists a tanh neural network $\widehat{f^\epsilon}$ with two hidden layers and width $O(\epsilon^{-d/(k-s)})$ such that*

$$\left\|f - \widehat{f^\epsilon}\right\|_{C^s([0,1]^d)} < \epsilon. \tag{2.60}$$

Note that Corollary 2.2 follows from Theorem 2.3 by setting $s = 0$. If we choose s to be higher, then more neurons are needed to guarantee the same accuracy. This make sense, as it is more difficult to obtain a good approximation in higher-order C^s-norms.

In summary, we have proven a result that quantifies how large a neural network should be to guarantee some specified accuracy when approximating a C^k function (Theorem 2.2). We then applied this result to the setting of ReLU neural networks (Corollary 2.1) and tanh neural networks (Corollary 2.2). In both cases (up to logarithmic terms) $O(\epsilon^{-k/d})$ neurons were needed to guarantee an accuracy of $\epsilon > 0$. Hence, the number of needed neurons decreases if the function is smoother (higher k) and increases with the input dimension d is higher. More precisely, the number of needed neurons increases *exponentially* in d. Because of these exponential growth, we say that these approximation results suffer from the **curse of dimensionality** when k is small compared to d: to decrease the error with a factor of 10, the number of neurons needs to be multiplied by a factor of $10^{d/k}$.

Example 2.2 Suppose that $k = 1$, $d = 6$ and that the desired accuracy is $\epsilon = 0.01$, all of which are reasonable parameters in practice. However, we get that $\epsilon^{-d/k} = 10^{12}$. In comparison: the largest neural network trained so far (in May 2021) 'only' consists of 10^{11} parameters.

Exercise 2.14 Choose a function $f \in C^1([0,1]^6)$ and train neural networks of different sizes to approximate f. How many neurons do you need to approximate f to an accuracy of $\epsilon = 0.01$? Compare with the theoretical estimate.

As the previous exercise probably showed, the theoretical results we established are large overestimates of the neural network sizes that are needed in practice, in particular when d is high. This can be explained as follows: the network sizes of Theorem 2.2 and its corollaries are *worst-case* network sizes, i.e. sizes needed to approximate the functions $f \in C^k([0,1]^d)$ that are the hardest to approximate. Fortunately, we rarely encounter such functions in practice. Instead, most functions of interest have a lot of additional structure that allows a considerably more efficient neural network approximation. This is the topic of the next section.

2.3 Curse of dimensionality

This section gives an overview of some assumptions under which the number of neurons provably does not grow exponentially in the input dimension and hence the neural network can *overcome the curse of dimensionality*. We first give examples of assumptions on the structure of functions that are widely applicable such as compositionality, manifold and regularity assumptions. Finally, we give examples of PDEs for which neural networks can provably approximate their solution without suffering from the curse of dimensionality. This is a theoretical cornerstone of the use of deep learning in scientific computing.

2.3.1 Compositionality assumptions

Consider the high-dimensional function $f \in C^1([0,1]^d)$ that is defined as

$$f(x) = \sum_{i=1}^{d} f_i(x_i) \qquad \text{for } x \in [0,1]^d, \tag{2.61}$$

where each $f_i \in C^1([0,1])$. A direct application of Corollary 2.2 to f leads to a worst-case network size of $O(\epsilon^{-d})$ to guarantee an accuracy of ϵ. On the other hand, the same corollary tells us that only $O(\epsilon^{-1})$ neurons are needed to approximate any of the f_i to an error of ϵ. Given that the sum of neural networks is again a neural network (see Exercise 2.3) and that there are d one-dimensional functions f_i, it follows that the worst-case network size is in fact only $O(d\epsilon^{-1})$.

Example 2.3 As in Example 2.2, we take again $d = 6$ and $\epsilon = 0.01$. On the one hand, we have $\epsilon^{-d} = 10^{12}$ and on the other hand $d\epsilon^{-1} = 600$. We conclude that taking additional compositionality assumptions into account leads to a drastic improvement of the worst-case network size.

In this toy example, f was a composition of d one-dimensional functions and the d-dimensional sum, i.e. the mapping $x \mapsto \sum_{x=1}^{d} x_i$, which can be exactly represented by neural networks. In practice, most C^k-functions we encounter are compositions of low-dimensional functions on the one hand, and high-dimensional functions that can be approximated without the curse of dimensionality (e.g. the sum or product of d real numbers). As a result, the worst-case network size can generally be improved from $O(\epsilon^{-d/k})$ to $O(d^\alpha \epsilon^{-\beta})$ for some $\alpha > 0$ and $0 < \beta \ll d/k$.

One way to formalize the concept of compositional functions is to consider their computational graph $\mathcal{G}$. For an explicitly defined function, this is a *directed acyclic graph* (DAG) with nodes V and edges E. Consider for instance a function of the form $f(x_1, x_2, x_3) = f_c(f_a(x_1), f_b(x_2, x_3))$. The corresponding DAG is given by $V = \{x_1, x_2, x_3, a, b, c\}$ and $E = \{(x_1, a), (x_2, b), (x_3, b), (a, c), (b, c)\}$, visualized in Figure 2.2. We say that f is a (V, E)-compositional function and for every node

$v \in V$ we assume that the function f_v is a $C^{k_v}([0,1]^{d_v})$, with d_v being the number of source nodes of v.

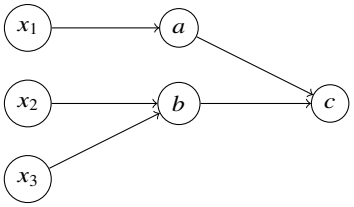


Fig. 2.2: Computational graph of a function of the form $f(x_1, x_2, x_3) = f_c(f_a(x_1), f_b(x_2, x_3))$.

For any DAG $\mathcal{G}$, the following theorem states the worst-case neural network size of a $\mathcal{G}$-compositional function [61, Theorem 3].

Theorem 2.4 *Let $\mathcal{G} = (V, E)$ be a DAG, d the number of source nodes and for every $v \in V$, let d_v be the number of incoming edges of v. Let $f : [0,1]^d \to \mathbb{R}$ be a compositional $\mathcal{G}$-function, where each of the constituent functions is in $C^{k_v}([0,1]^{d_v})$. Then for every $\epsilon > 0$, there is a deep neural network $\widehat{f^\epsilon}$ with smooth non-polynomial activation function and complexity $O\left(\sum_{v \in V} \epsilon^{-d_v/k_v}\right)$ such that $\left\| f - \widehat{f^\epsilon} \right\|_{C^0([0,1]^d)} < \epsilon$.*

This theorem is an improvement over the general result for C^k functions if $\max_{v \in V} \frac{d_v}{k_v} < \frac{d}{k}$.

Example 2.4 Consider again the example from Figure 2.2 and assume that $k_a = 1$, $k_b = 3$ and $k_c = 4$. In this case $\max_{v \in V} \frac{d_v}{k_v} = 1$. The worst-case neural network size is therefore $O(\epsilon^{-1})$ (according to Theorem 2.4) instead of $O(\epsilon^{-3})$ (according to e.g. Corollary 2.2).

2.3.2 Manifold assumptions

Another assumption is that only a subset of the domain $[0,1]^d$ of a function is of practical interest. For example, in face recognition the input of the neural network is a d-dimensional vector that represents an image of a face. One can however imagine that the vectors that correspond to images of faces are a very small subset of $[0,1]^d$. Moreover, a slightly deformed picture of face will still resemble a face (and a slightly deformed picture of something else than a face will still not resemble a face). Based on these observations, it is imaginable that the subspace of $[0,1]^d$ corresponding to face images is some kind of d^*-dimensional manifold. If $d^* \ll d$, then the bound on the worst-case network size can be drastically reduced. This is the topic of the following theorem [66, Theorem 3]. Many similar results are available in literature.

Theorem 2.5 *Let $\mathcal{M} \subset [0,1]^d$ be a compact d^*-dimensional manifold with smooth local coordinates and let $f \in C^k(\mathcal{M})$. Then for every $\epsilon > 0$, there exists a deep ReLU neural network $\widehat{f}^\epsilon$ of depth $O(\ln(1/\epsilon))$ and width $O(\epsilon^{-d^*/k})$ such that $\left\| f - \widehat{f}^\epsilon \right\|_{C^0(\mathcal{M})} < \epsilon$.*

2.3.3 Regularity assumptions

We already established that functions with lower regularity are more difficult to approximate by neural networks and therefore require more neurons. In particular, the theoretical bounds suffer from the curse of dimensionality as the number of neurons grows exponentially in the input dimension. In this section we investigate whether the curse of dimensionality can be overcome if we only consider very smooth functions.

A first example of very smooth functions are analytic functions. Analytic functions are functions that can be locally defined as a convergent power series and are infinitely differentiable. The following result [14, Corollary 5.6] should be interpreted as the counterpart of Corollary 2.2 for analytic functions.

Theorem 2.6 *Let $\Omega \subset \mathbb{R}^d$ open with $[0,1]^d \subset \Omega$ and let f be analytic on Ω. For every $\epsilon > 0$, there exists a deep tanh neural network $\widehat{f}^\epsilon$ with two hidden layers and width $O(\ln(1/\epsilon)^{d+1})$ such that $\left\| f - \widehat{f}^\epsilon \right\|_{C^0([0,1]^d)} < \epsilon$.*

Ergo, by assuming that f is analytic, the worst-case network size reduces from $O(\epsilon^{-d/k})$ to $O(\ln(1/\epsilon)^{d+1})$. The dependence on the dimension hence has not completely disappeared, but it is still a major improvement as $\ln(1/\epsilon)$ grows much slower than $\epsilon^{-1/k}$ for $\epsilon \to 0$.

Example 2.5 As in Example 2.2, we take again $d = 6$ and $\epsilon = 0.01$. In this case, $\ln(1/\epsilon)^{d+1} = 2^7 = 128$.

Another example is the famous work of Andrew Barron [3], where he defined a function class for which the neural network approximation rate is independent of the input dimension of the function. His result holds for neural networks with sigmoidal activation functions[1] and functions $f : \mathbb{R}^d \to \mathbb{R}$ for which there is a Fourier presentation $f(x) = \int_{\mathbb{R}^d} e^{i\omega \cdot x} \widetilde{f}(\omega) d\omega$ such that

$$C_f = \int_{\mathbb{R}^d} |\omega| \left| \widetilde{f}(\omega) \right| d\omega < \infty. \tag{2.62}$$

This condition can be interpreted as the integrability of the Fourier transform of the gradient of f, and is therefore a regularity assumption. Under these assumptions, the following theorem states that the worst-case network size grows as $O(\epsilon^{-2})$ independent of the input dimension d, as in [3, Proposition 1].

[1] A sigmoidal function is bounded, measurable function on the real line with $\lim_{x \to -\infty} \sigma(x) \to -1$ and $\lim_{x \to \infty} \sigma(x) \to 1$.

Theorem 2.7 *Let $B = \{x \in \mathbb{R}^d : |x| \leq 1\}$ be the d-dimensional unit ball, let $f : \mathbb{R}^d \to \mathbb{R}$ be a function with $C_f < \infty$ and let μ a probability measure on B. Then for every $\epsilon > 0$, there exists a shallow network with sigmoidal activation function and width $O(\epsilon^{-2})$ such that*

$$\left\| f - \widehat{f^\epsilon} \right\|_{L^2(B,\mu)} = \left(\int_B (f(x) - \widehat{f^\epsilon}(x)) \mu(dx) \right)^{\frac{1}{2}} < \epsilon. \tag{2.63}$$

Note however that the constant in the Landau notation is still allowed to depend on d and that the error is measured with respect to another norm than the C^0-norm from all previous results. The dependence on d of the constant in the Landau notation has been worked out for some function classes in [3].

2.3.4 PDE solutions

A central problem in scientific computing is the numerical approximation of solutions of partial differential equations (PDEs). Often these PDE solutions depend on many parameters or have a high input dimension. In both cases, approximating the PDE solution with classical numerical methods might be very expensive as convential discretization-based methods unavoidably suffer from the curse of dimensionality. Fortunately, many experiments suggest that neural networks overcome the curse of dimensionality for the approximation of the solution of many PDEs. The goal of this section is to give some examples of results that theoretically confirm the performance observed in the experiments. We make the distinction between settings where the parameter dimension is high and settings where the (physical) space dimension is high.

2.3.4.1 High parameter dimension

PDEs can depend on a high number of parameters. If one is only interested in the solution of the PDE for one fixed set of these parameters, then this does not bring any problems. In some applications, e.g. uncertainty quantification, one needs the PDE solution for many different sets of parameters, which is generally expensive using classical numerical methods. We show some examples where neural networks can provably overcome the curse of dimensionality in the parameter dimension.

As a first example, we consider the linear transport equation in one dimension in space,

$$\partial_t u(t,x,y) = a \partial_x u(t,x,y), \quad u(0,x,y) = u_0(x,y), \tag{2.64}$$

for $(t,x,y) \in [0,T] \times [-M,M] \times [0,1]^d$, where $a, T, M > 0$. In this example, the initial condition u_0 is parametrized by the (possibly high-dimensional) vector

y. One practical use case is when u_0 is random. As many well-behaved stochastic processes admit a Karhunen-Loève decomposition[2], it makes sense to consider as initial conditions a class of truncated Karhunen-Loève expansions,

$$u_0(x, y) = \overline{u}(x) + \gamma \sum_{i=1}^{d} \sqrt{\lambda_i} y_i \varphi_i(x), \tag{2.65}$$

where $\gamma > 0$, $y \in [0, 1]^d$ and $\{\varphi_i\}_i$ is some set of (orthogonal) basis functions, e.g. $\varphi_i(x) = \sin(2\pi i x)$. Using the method of characteristics, we find that the solution of the PDE is given by

$$u(t, x, y) = \overline{u}(x - at) + \gamma \sum_{i=1}^{d} \sqrt{\lambda_i} y_i \varphi_i(x - at). \tag{2.66}$$

We see that u is a composition of one-dimensional functions and the d-dimensional sum. Following the results of Section 2.3.1 and assuming that $\overline{u}, \varphi_i \in C^k([0, 1])$, we find that the (worst-case) numbers of neurons required to approximate u to accuracy $\epsilon > 0$ is $O(d\epsilon^{-1/k})$.

As a second example, we consider the one-dimensional heat equation,

$$\partial_t u(t, x, y) = \partial_x^2 u(t, x, y), \quad u(0, x, y) = u_0(x, y) \tag{2.67}$$

for $(t, x, y) \in [0, T] \times [0, 1] \times [0, 1]^d$, where $T > 0$. We assume that u_0 can be approximated with an error $\epsilon > 0$ by a neural network with $O(d^\alpha \epsilon^{-\beta})$ for some $\alpha, \beta > 0$. This means that the network size grows only polynomially in d and that the actual rate is even independent of d, i.e. that u_0 can be approximated without the curse of dimensionality. This assumption is for example met if u_0 a compositional function is in the form of a truncated Karhunen-Loève expansion (2.65). More precisely, we assume that there is a neural network $\widehat{u}_0^\epsilon$ with $O(d^\alpha \epsilon^{-\beta})$ neurons such that $\left\| u_0(\cdot, y) - \widehat{u}_0^\epsilon(\cdot, y) \right\|_{L^2} < \epsilon$ for all $y \in [0, 1]^d$. Now let u^* be the exact solution to

$$\partial_t u^*(t, x, y) = \partial_x^2 u^*(t, x, y), \quad u^*(0, x, y) = \widehat{u}_0^\epsilon(x, y) \tag{2.68}$$

for $(t, x, y) \in [0, T] \times [0, 1] \times [0, 1]^d$. Then from the maximum principle it follows that

$$\| u(t, \cdot, y) - u^*(t, \cdot, y) \|_{L^2} \leq \left\| u_0(\cdot, y) - \widehat{u}_0^\epsilon(\cdot, y) \right\|_{L^2} < \epsilon \tag{2.69}$$

for all $(t, y) \in [0, T] \times [0, 1]^d$. Next, we will construct a neural network $\widehat{u}$ by emulating a finite difference scheme (FDS). We discretize time and space by defining $t_n = \frac{Tn}{N}$ and $x_j = j/J$ for all $1 \leq n \leq N$ and $0 \leq j \leq J$ for some $J, N \in \mathbb{N}$ that satisfy the CFL condition. The FDS approximations are then defined for every y, n, j

[2] A Karhunen-Loève decomposition is quite similar to what a Fourier series is for functions on a bounded interval. It is also related to principal component analysis.

as

$$U_j^0(y) = \widehat{u}_0^\epsilon(x_j, y) \tag{2.70}$$

$$U_j^{n+1}(y) = U_j^n(y) + \frac{\Delta t}{(\Delta x)^2}(U_{j+1}^n(y) - 2U_j^n(y) + U_{j-1}^n(y)), \tag{2.71}$$

with $\Delta t = TN^{-1}$ and $\Delta x = J^{-1}$. The discrete L^2-error of the FDS approximation is then given by

$$\sqrt{\frac{\Delta x}{2}\sum_{j=1}^{J}\left|u^*(t_n, x_j, y) - U_j^n(y)\right|^2} = O(N^{-1} + J^{-2}) = O(J^{-2}), \tag{2.72}$$

where the latter equality follows from the CFL condition. We now define our neural network as

$$\widehat{u} : [0,1]^d \to \mathbb{R} : y \mapsto (U_0^N(y), \dots, U_J^N(y)) \approx (u(T, x_0, y), \dots, u(T, x_J, y)). \tag{2.73}$$

As for every n the map $(U_j^n(y))_j \mapsto (U_{j+1}^n(y))_j$ is affine, $\widehat{u}$ is indeed a neural network. The neural network $\widehat{u}$ maps every parameter y to a vector of values that is equivalent with a piecewise constant function. Combining (2.69) and (2.72), we find that the accuracy of $\widehat{u}$ is equal to $O(\epsilon + J^{-2})$. Therefore we should set $J \sim 1/\sqrt{\epsilon}$ to obtain an accuracy of $O(\epsilon)$. Moreover, the total number of required neurons is J times the number of neurons of $\widehat{u}_0^\epsilon$. We conclude that in the worst case $O(d^\alpha \epsilon^{-\beta-1/2})$ neurons are needed to approximate u to an L^2-error $\epsilon > 0$.

Remark 2.1 The approach presented above does not only hold for the heat equation, but can in fact be generalized to any PDE for which (1) the finite difference scheme (FDS) converges to the real solution, (2) the coefficient functions in the PDE can be efficiently approximated by neural networks and (3) small changes in the initial condition lead to small changes in the PDE solution (i.e. well-posedness). In general, the composition of multiple FDS steps will not be affine and therefore the depth of the network will grow linearly in N. The other arguments remain identical to the ones presented above.

Remark 2.2 Using some additional work, one can also construct a (ReLU) neural network that takes as input (t, x, y) instead of y and returns as output a scalar that approximates $u(t, x, y)$, instead of a vector with approximations at grid points. A complete proof can be found in [15] in the case of hyperbolic conservation laws, but the arguments given there can be used to obtain a space-time network from a vectorial output on a grid for any type of PDE.

Remark 2.3 It is possible to generalize the approach to settings where the space dimension is higher than one. As the number of grid points of a FDS grows exponentially in the space dimension, the curse of dimensionality will only be overcome in the parameter dimension.

2.3.4.2 High space dimension

For some PDEs, the input dimension of the solution itself is high, without taking parameters into account. For example, the 3D radiative transfer equations have 7 variables, in the Schrödinger equation the input dimension depends on the number of particles and in the Black-Scholes equation the input dimension is equal to the number of assets in a portfolio. In this section, we follow the approach proposed in e.g. [25, 31] to prove that the curse of dimensionality can be overcome for linear Kolmogorov equations, an important subclass of PDEs. A linear Kolmogorov equation with drift $\mu : \mathbb{R}^d \to \mathbb{R}^d$, diffusion $\sigma : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ and initial condition $\varphi : \mathbb{R}^d \to \mathbb{R}$ is given by,

$$\partial_t u(t,x) = \frac{1}{2}\text{Trace}(\sigma(x)\sigma(x)^T \text{Hess}_x[u](t,x)) + \langle \mu(x), \nabla_x[u](t,x)\rangle \quad (2.74)$$

$$u(0,x) = \varphi(x) \quad (2.75)$$

for all $(t,x) \in [0,T] \times [0,1]^d$. The existence and regularity of its classical/viscosity solution depend on the properties of μ and σ. Examples of linear Kolmogorov equations are the heat equation, the Black-Scholes equation and more general advection-diffusion equations.

An interesting property of linear Kolmogorov equations is that they are linked to stochastic differential equations. Let $(\Omega, \mathcal{F}, P, (\mathbb{F}_t)_{t\in[0,T]})$ be a stochastic basis, and for every $x \in D$, let $X^x : \Omega \times [0,T] \to \mathbb{R}^d$ be the solution process of the following stochastic differential equation,

$$dX_t^x = \mu(X_t^x)dt + \sigma(X_t^x)dB_t, \quad X_0^x = x, \quad x \in D, t \in [0,T], \quad (2.76)$$

where B_t is a standard d-dimensional Brownian motion on $(\Omega, \mathcal{F}, P, (\mathbb{F}_t)_{t\in[0,T]})$. Then under some technical assumptions, the Feynman-Kac formula states that

$$u(t,x) = \mathbb{E}[\varphi(X_t^x)], \quad (2.77)$$

where u is the solution of the linear Kolmogorov equation (2.74).

We now discuss how the Feynman-Kac formula can be used in order to construct a neural network that approximates $u(T,x)$ without the curse of dimensionality. Like in the previous section, we assume that φ can be approximated by a neural network $\widehat{\varphi}^\epsilon$ with $O(d^\alpha \epsilon^{-\beta})$ neurons to an accuracy ϵ. For simplicity, we focus on the d-dimensional heat equation (i.e. $\mu = 0$ and $(\sigma(x))_{ij} = \delta_{ij}$), more details and the general case can be found in [25]. First note that it holds that $X_t^x = x + B_t$ for the heat equation. Let $B_T^1, \dots, B_T^M$ be M independent realizations of the random variable B_T. We then define our neural network approximation as a Monte Carlo approximation of the Feynman-Kac formula with $\widehat{\varphi}^\epsilon$ as initial condition,

$$\widehat{u}_T(x) = \frac{1}{M}\sum_{m=1}^{M} \widehat{\varphi}^\epsilon(x + B_T^m). \quad (2.78)$$

We then quantify the error of this approximation using the triangle inequality,

$$\|u(T,\cdot) - \widehat{u}_T\|_{L^2} \leq \|u(T,\cdot) - \mathbb{E}[\widehat{\varphi}^\epsilon(\cdot + B_T)\|_{L^2} + \|\mathbb{E}[\widehat{\varphi}^\epsilon(\cdot + B_T) - \widehat{u}_T\|_{L^2}. \quad (2.79)$$

The first term on the right-hand side is $O(\epsilon)$ and the second term is of size $O(M^{-1/2})$, using a Monte Carlo argument. Hence, we should choose $M \sim \epsilon^{-2}$ to obtain a total accuracy of $O(\epsilon)$. Moreover, we need M times $O(d^\alpha \epsilon^{-\beta})$ neurons in order to construct $\widehat{u}_T$. This amounts to $O(d^\alpha \epsilon^{-\beta-2})$ neurons, meaning that the curse of dimensionality has indeed been overcome.

Remark 2.4 We constructed an approximation for a fixed time $T > 0$. It is also possible to construct a time-dependent ReLU neural network approximation. This has been proven in [31] using an Euler-Maruyama approximation of the solution process X_t^x of the SDE.

2.4 Error of a trained neural network

In scientific computing, we are often interested in approximating the solution $f : D \to \mathbb{R}$ of a (possibly high-dimensional) PDE. So far we have focussed on proving that there exists a neural network $\overline{f} = f_{\overline{\theta}}$, with weights and biases $\overline{\theta} \in \Theta$, such that $\left\|f - \overline{f}\right\| < \epsilon$. In particular, we have shown that for many PDEs only $O(d^\alpha \epsilon^{-\beta})$ neurons are needed to achieve this accuracy, thereby overcoming the curse of dimensionality. These theoretical results are a crucial element in the mathematical analysis of deep learning methods in scientific computing: if it were provably impossible to approximate f, then these lectures notes would not have existed. However, as crucial as they are, they do not explain how to find $\overline{f}$ and therefore cannot fully explain the good performance of deep learning methods. In this section, we attempt to do so by discussing the error of trained neural networks.

To train a neural network, one needs a training set $\mathcal{S} = \{x_1, \ldots, x_N\}$ consisting of $N \in \mathbb{N}$ training samples. For every training set, one can define a loss function $\theta \mapsto \mathcal{J}(\theta, \mathcal{S})$. A popular choice is given by

$$\mathcal{J}(\theta, \mathcal{S}) = \frac{1}{N} \sum_{i=1}^{N} |f(x_i) - f_\theta(x_i)|^p + \lambda \|\theta\|^q, \quad (2.80)$$

for $\theta \in \Theta$ and where usually $p, q \in \{1, 2\}$. The term $\|\theta\|^q$ is a regularization term that usually has a positive influence on the training procedure and the hyperparameter $\lambda \geq 0$ controls the magnitude of the regularization term. Training a network then corresponds to finding an approximation of the minimizer of the loss function $\theta^*(\mathcal{S}) \approx \arg\min_{\theta \in \Theta} \mathcal{J}(\theta, \mathcal{S})$. We then denote the trained network as $f^* = f_{\theta^*}$. The performance of a neural network f_θ on the training set is quantified by the *training error*,

$$\mathcal{E}_T(\theta, \mathcal{S}) = \frac{1}{N} \sum_{i=1}^{N} |f(x_i) - f_\theta(x_i)|^p. \tag{2.81}$$

However, what we are actually interested in is the *error* of the neural network on the whole domain D,

$$\mathcal{E}(\theta) = \int_D |f(x) - f_\theta(x)|^p d\mu(x), \tag{2.82}$$

where μ is a probability measure on D. If $\mathcal{S}$ contains iid training samples, then it is tempting to infer that $\mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}) \approx \mathcal{E}(\theta^*(\mathcal{S}))$ if N is large enough, as the former appears to be a Monte Carlo approximation of the latter. This is however not true. An erroneous justification could go as follows. Let g be some function and let $X_i \sim \mu$ be iid random variables. It holds that if all $g(X_i)$ are independent and have finite variance, then $\int g(x)d\mu(x) = \frac{1}{N}\sum_{i=1}^{N} g(X_i) + O(\frac{1}{\sqrt{N}})$. This principle is the foundation of Monte Carlo integration. Taking $g(x) = \left|f(x) - f_{\theta^*(\mathcal{S})}(x)\right|^p$ then seems to suggest that $\mathcal{E}(\theta^*(\mathcal{S})) = \mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}) + O(\frac{1}{\sqrt{N}})$ if $\mathcal{S}$ contains iid training samples. The problem is that $\theta^*(\mathcal{S})$ generally depends on the entire training set, and as a result $f_{\theta^*(\mathcal{S})}(X_i)$ and $f_{\theta^*(\mathcal{S})}(X_j)$ are not independent for $i \neq j$. As a result, Monte Carlo integration is not applicable and a small training error $\mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S})$ does not automatically imply that the error $\mathcal{E}(\theta^*(\mathcal{S}))$ is small. In general, it is actually impossible to prove such an implication for an arbitrary training set $\mathcal{S}$. For example, if the training samples are not properly distributed over D then there is no hope that the error is small, no matter how small the training error is. Fortunately, the probability of generating such an ill-behaved training set is small. This hints at the possibility of proving results that hold with a large probability, or that hold averaged over all possible training sets. For this reason, we define the cumulative training error and cumulative error for the trained network as follows,

$$\overline{\mathcal{E}}_T = \mathbb{E}\left[\mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S})\right], \qquad \overline{\mathcal{E}} = \mathbb{E}\left[\mathcal{E}(\theta^*(\mathcal{S}))\right]. \tag{2.83}$$

The goal is now to bound $\overline{\mathcal{E}}$ in terms of $\overline{\mathcal{E}}_T$. We start by proving a decomposition of the error.

Lemma 2.6 *For any $\theta_1, \theta_2 \in \Theta$ and training set $\mathcal{S}$ it holds that*

$$\mathcal{E}(\theta_1) \leq \mathcal{E}(\theta_2) + 2 \sup_{\theta \in \Theta} |\mathcal{E}_T(\theta, \mathcal{S}) - \mathcal{E}(\theta)| + \mathcal{E}_T(\theta_1, \mathcal{S}) - \mathcal{E}_T(\theta_2, \mathcal{S}). \tag{2.84}$$

Proof Fix $\theta_1, \theta_2 \in \Theta$ and generate a random training set $\mathcal{S}$. The proof consists of the repeated adding, subtracting and removing from terms. It holds that

$$\begin{aligned}
\mathcal{E}(\theta_1) &= \mathcal{E}(\theta_2) + \mathcal{E}(\theta_1) - \mathcal{E}(\theta_2) && (2.85)\\
&= \mathcal{E}(\theta_2) - (\mathcal{E}_T(\theta_1, \mathcal{S}) - \mathcal{E}(\theta_1)) + (\mathcal{E}_T(\theta_2, \mathcal{S}) - \mathcal{E}(\theta_2)) && (2.86)\\
&\quad + \mathcal{E}_T(\theta_1, \mathcal{S}) - \mathcal{E}_T(\theta_2, \mathcal{S}) && (2.87)\\
&\leq \mathcal{E}(\theta_2) + 2 \max_{\theta \in \{\theta_2, \theta_1\}} |\mathcal{E}_T(\theta, \mathcal{S}) - \mathcal{E}(\theta)| && (2.88)\\
&\quad + \mathcal{E}_T(\theta_1, \mathcal{S}) - \mathcal{E}_T(\theta_2, \mathcal{S}). && (2.89)
\end{aligned}$$

□

If we now take $\theta_1 = \theta^*(\mathcal{S})$ and $\theta_2 = \overline{\theta}$ (i.e. the weights and biases corresponding to a provably accurate neural network) and take the expectation over all training sets, we find that

$$\overline{\mathcal{E}} \leq \underbrace{\mathcal{E}(\overline{\theta})}_{\text{approximation error}} \underbrace{+2\,\mathbb{E}\left[\sup_{\theta \in \Theta} |\mathcal{E}_T(\theta, \mathcal{S}) - \mathcal{E}(\theta)|\right]}_{\text{generalization error}} + \underbrace{\overline{\mathcal{E}}_T - \mathbb{E}\left[\mathcal{E}_T(\overline{\theta}, \mathcal{S})\right]}_{\text{optimization error}}. \quad (2.90)$$

The cumulative error is now decomposed into three error sources. The *approximation error* $\mathcal{E}(\vartheta)$ should be provably small, e.g. using results from the previous section. The second term of the RHS, termed *generalization error*, quantifies how well the training error approximates the error. Finally, an *optimization error* is incurred due to the inability of the optimization procedure to find $\overline{\theta}$ based on a finite training data set. Indeed one can see that if $\theta^*(\mathcal{S}) = \overline{\theta}$, then the optimization error vanishes. If we can prove an upper bound on the generalization error, then we will have successfully bounded the cumulative error $\overline{\mathcal{E}}$. We will prove such a bound based on Hoeffding's inequality.

Lemma 2.7 (Hoeffding's inequality) *Let $\epsilon, c > 0$, $N \in \mathbb{N}$, let $(\Omega, \mathcal{A}, \mathbb{P})$ be a probability space and let $X_n : \Omega \to [0, c]$ be independent random variables. Then it holds that*

$$\mathbb{P}\left(\frac{1}{N}\left(\sum_{i=1}^{N}(X_i - \mathbb{E}[X_i])\right) \geq \epsilon\right) \leq \exp\left(\frac{-2\epsilon^2 N}{c^2}\right). \quad (2.91)$$

Hoeffding's inequality is a so-called concentration inequality: it states that if N is large enough, the average of bounded independent random variables will be close to the average of their expectations with a large probability. Using this, we can state and prove a bound on the generalization error for ReLU neural networks. A similar theorem can be proven for neural networks with smooth activation functions, using results from e.g. [16].

Theorem 2.8 *Let $d, L, W, d_\Theta \in \mathbb{N}$, with $W \geq 2$, $B, F \geq 1$, $D = [0,1]^d$, $\Theta = [-B, B]^{d_\Theta}$. Let $\{f_\theta : \theta \in \Theta\}$ be ReLU neural networks with at most L layers and width W. Assume that $\max_{x \in D, \theta \in \Theta} \max\{|f(x)|, |f_\theta(x)|\} \leq F$. Then it holds that for large enough $N \in \mathbb{N}$ that,*

$$\mathbb{E}\left[\sup_{\theta\in\Theta}|\mathcal{E}_T(\theta,\mathcal{S})-\mathcal{E}(\theta)|\right]\leq 2(2F)^p(L+1)(W+1)\sqrt{\frac{p\ln\left(8B(W+1)\sqrt{N}\right)}{N}}. \tag{2.92}$$

Proof We follow the proof presented in [6]. A first ingredient of the proof is the fact that the map $\theta\mapsto f_\theta$ is Lipschitz continuous with constant $\mathcal{L}=(L+1)B^L(W+1)^{L+1}$ (see [6]), i.e.

$$\left\|f_{\theta_1}(x)-f_{\theta_2}(x)\right\|_{C^0(D)}\leq (L+1)B^L(W+1)^{L+1}\|\theta_1-\theta_2\|_\infty. \tag{2.93}$$

As a consequence, we also find that

$$\begin{aligned}
&\left\||\mathcal{E}(\theta_1)-\mathcal{E}_T(\theta_1,\{x\})|-|\mathcal{E}(\theta_2)-\mathcal{E}_T(\theta_2,\{x\})|\right\|_{C^0(D)} && (2.94)\\
&\quad\leq p(2F)^{p-1}\left\|f_{\theta_1}(x)-f_{\theta_2}(x)\right\|_{C^0(D)} && (2.95)\\
&\quad\leq p(2F)^{p-1}\mathcal{L}\|\theta_1-\theta_2\|_\infty && (2.96)
\end{aligned}$$

Next, since Θ is compact, we can cover it using a finite number of balls. For any $\delta>0$, the δ-covering number of Θ is the smallest number $C_\Theta(\delta)$ such that there exist $\theta_1,\dots\theta_{C_\Theta(\delta)}\in\Theta$ such that for all $\theta\in\Theta$ there is an i such that $\|\theta-\theta_i\|_\infty\leq\delta$. One can calculate that

$$C_\Theta(\delta)\leq\left(\frac{B}{\delta}\right)^{d_\Theta}\quad\text{and therefore}\quad C(\delta)\leq\left(\frac{Bp(2F)^{p-1}\mathcal{L}}{\delta}\right)^{d_\Theta}, \tag{2.97}$$

where $C(\delta)$ is the δ-covering number of $\{|\mathcal{E}(\theta)-\mathcal{E}_T(\theta,\{\cdot\})|:\theta\in\Theta\}$.

Next, using Hoeffding's inequality with $X_i=\mathcal{E}_T(\theta,\{x_i\})$ for fixed $\theta\in\Theta$ and $x_i\in\mathcal{S}$ and $c=(2F)^p$ we find that for every $\epsilon>0$,

$$\mathbb{P}\left(|\mathcal{E}(\theta)-\mathcal{E}_T(\theta,\mathcal{S})|\geq\epsilon\right)\leq 2\exp\left(\frac{-\epsilon^2N}{(2F)^{2p}}\right). \tag{2.98}$$

Using the definition of $C(\delta)$ and a union bound, we find that

$$\begin{aligned}
\mathbb{P}\left(\sup_{\theta\in\Theta}|\mathcal{E}(\theta)-\mathcal{E}_T(\theta,\mathcal{S})|\geq\epsilon\right)&\leq\sum_{i=1}^{C(\epsilon/2)}\mathbb{P}\left(|\mathcal{E}(\theta_i)-\mathcal{E}_T(\theta_i,\mathcal{S})|\geq\frac{\epsilon}{2}\right) && (2.99)\\
&\leq 2C(\epsilon/2)\exp\left(\frac{-\epsilon^2N}{(2F)^{2p}}\right). && (2.100)
\end{aligned}$$

Next, we use that for any random variable bounded by some constant $c>0$ it holds that

$$\mathbb{E}[X]=\mathbb{E}[X1_{X\leq\epsilon}]+\mathbb{E}[X1_{X>\epsilon}]\leq\epsilon+c\mathbb{P}(X>\epsilon). \tag{2.101}$$

Applying this equation to our setting gives us that

$$\mathbb{E}\left[\sup_{\theta\in\Theta}|\mathcal{E}(\theta)-\mathcal{E}_T(\theta,\mathcal{S})|\right] \leq \epsilon + (2F)^p \cdot 2C(\epsilon/2)\exp\left(\frac{-\epsilon^2 N}{(2F)^{2p}}\right) \tag{2.102}$$

$$\leq \epsilon + 2(2F)^p \left(\frac{2Bp(2F)^{p-1}\mathcal{L}}{\epsilon}\right)^{d_\Theta} \exp\left(\frac{-\epsilon^2 N}{(2F)^{2p}}\right). \tag{2.103}$$

If we now choose N such that $\epsilon^2 = \frac{(2F)^{2p}}{N}\ln\left(4F\left(\frac{2Bp(2F)^{p-1}\mathcal{L}}{\epsilon}\right)^{d_\Theta+1}\right)$, then it holds that

$$\mathbb{E}\left[\sup_{\theta\in\Theta}|\mathcal{E}(\theta)-\mathcal{E}_T(\theta,\mathcal{S})|\right] \leq 2\epsilon = 2\sqrt{\frac{(2F)^{2p}}{N}\ln\left(2(2F)^p\left(\frac{2Bp(2F)^{p-1}\mathcal{L}}{\epsilon}\right)^{d_\Theta+1}\right)}. \tag{2.104}$$

Filling in ϵ again in the right-hand side of the previous inequality, with ϵ small enough and N large enough, gives that

$$\mathbb{E}\left[\sup_{\theta\in\Theta}|\mathcal{E}(\theta)-\mathcal{E}_T(\theta,\mathcal{S})|\right] \leq 2(2F)^p\sqrt{\frac{p(d_\Theta+1)\ln\left(4Bp\mathcal{L}\sqrt{N}\right)}{\sqrt{N}}} \tag{2.105}$$

$$\leq 2(2F)^p(L+1)(W+1)\sqrt{\frac{p\ln\left(8B(W+1)\sqrt{N}\right)}{N}}, \tag{2.106}$$

where we used the definition of $\mathcal{L}$ and the fact that $d_\Theta + 1 \leq L(W+1)^2$. This proves the claim. □

We can now bound every term on the right hand size of the error decomposition (2.90). Using the results of Section 2.3.4, we find that there is a neural network with parameter $\overline{\theta} \in \Theta$ such that $\overline{\mathcal{E}}(\overline{\theta}) < \epsilon$ and $(L+1)(W+1) = O(d^\alpha \epsilon^{-\beta})$ for some $\alpha, \beta > 0$. Using Theorem 2.8, the generalization error can be bounded by $O(d^\alpha \epsilon^{-\beta} N^{-1/2})$. Finally, we bound the optimization error by the cumulative training error, $\overline{\mathcal{E}}_T - \mathbb{E}\left[\mathcal{E}_T(\overline{\theta},\mathcal{S})\right] \leq \overline{\mathcal{E}}_T$. Under the assumption that we can train (and retrain) the neural network arbitrarily well, $\overline{\mathcal{E}}_T$ will indeed be small. Putting everything together, we find that the error of the trained neural network is of the order of

$$\overline{\mathcal{E}} = O\left(\epsilon + \frac{d^\alpha \epsilon^{-\beta}}{\sqrt{N}} + \overline{\mathcal{E}}_T\right). \tag{2.107}$$

This formula shows a trade-off between the approximation error and generalization error. The smaller we choose $\epsilon > 0$, the larger the generalization error gets, as a more richer class of neural networks is considered. To obtain a small total error, a large value of N must be chosen. Using a different error decomposition than (2.90), one can also avoid the need for results on the approximation error and use the actual

values for L and W in Theorem 2.8 instead of $O(d^\alpha \epsilon^{-\beta})$, leading to a bound of the form [16],

$$\overline{\mathcal{E}} = O\left(\frac{LW}{\sqrt{N}} + \overline{\mathcal{E}}_T\right), \tag{2.108}$$

up to logarithmic terms. Again, a large value of N, albeit much smaller than in (2.107), needs to be chosen to guarantee a small error: at least $O(10^4)$ training samples are needed to achieve an accuracy of 1%. This means that (2.107) and (2.108) bound the error of a neural network in the *underparametrized regime*, where the number of training samples is (much) larger than the number of training parameters. The same holds for error bounds based on e.g. the Rademacher complexity or Vapnik-Chervonenkis dimension of the class of neural networks. This might be a limitation in some settings:

- In computer science and other areas, one often uses neural networks for which the number of parameters is much larger than the number of training samples. Empirical observations show that in general the error of the trained network $\overline{\mathcal{E}}$ is still small in this *overparametrized regime*. In particular, it is observed that the generalization first increases with decreasing training error (much like (2.107) and (2.108) for fixed N), but afterwards the generalization error starts decreasing as well. This phenomenon, coined *double descent* [7], can however not be explained by error bounds like (2.107), (2.108). A possible explanation might lie in the use of stochastic gradient descent algorithms to train the network.
- In scientific computing, the size of the used neural networks are in general quite small. At the same time, the generation of training data using PDE solvers is very expensive, such that N is generally not very large either. Therefore, the convergence rate in N of $\frac{1}{2}$ that we obtain using random training samples, as in (2.107) and (2.108), might not be good enough. In the text chapter, we present techniques to further decrease the error of a trained neural network in the setting of scientific computing, e.g. by increasing the convergence rate in N.

Chapter 3
Supervised learning in scientific computing

In this chapter, we give an overview of recently proposed techniques to improve the performance of deep learning techniques in the setting of scientific computing. In Section 3.2 and Section 3.3, special choices of training points are considered in order to increase the convergence rate of the error of the trained network in terms of the number of training samples. In Section 3.4, the supervised learning problem is reformulated in a way that can lead to a speed-up of the training. Next, in Section 3.5, an ensemble training technique is discussed in order to find the optimal choice of hyperparameters for the network. Finally, two applications of using neural networks as surrogates for observables are presented: uncertainty quantification (Section 3.6) and PDE-constrained optimization (Section 3.7).

3.1 Introduction

A fundamental goal in scientific computing is the efficient simulation of *observables* (also referred to as functionals, quantities of interest or figures of merit) of systems that arise in physics and engineering. A prototypical example is provided by the simulation of flows past aerospace vehicles where the observables of interest are body forces such as the lift and the drag coefficients and the underlying PDEs are the compressible Euler or Navier-Stokes equations of fluid dynamics. Other interesting examples include the run-up height for a tsunami (with the shallow water equations modeling the flow) or loads (stresses) on structures, with the underlying system being modeled by the equations of elasticity or visco-elasticity.

We assume that the observable of interest is defined in terms of the solution of the following very generic system of time-dependent parametric PDEs,

$$\begin{aligned} \partial_t U(t,x,y) &= L\left(y, U, \nabla_x U, \nabla_x^2 U, \dots\right), \quad \forall\, (t,x,y) \in [0,T] \times D(y) \times Y, \\ U(0,x,y) &= \overline{U}(x,y), \quad \forall\, (x,y) \in D(y) \times Y, \\ L_b U(t,x,y) &= U_b(t,x,y), \quad \forall\, (t,x,y) \in [0,T] \times \partial D(y) \times Y. \end{aligned} \tag{3.1}$$

Here, Y is the underlying parameter space and without loss of generality, we assume it to be $Y = [0,1]^d$, for some $d \in \mathbb{N}$. The spatial domain is labeled as $y \to D(y) \subset \mathbb{R}^{d_s}$ and $U : [0,T] \times D \times Y \to \mathbb{R}^m$ is the (unknown) solution of the PDE. The differential operator L is in a very generic form and can depend on the gradient and Hessian of U, and possibly higher-order spatial derivatives. Moreover, L_b is a generic operator for imposing boundary conditions. For the parametrized PDE (3.1), we consider the following generic form of observables,

$$L_g(y,U) := \int_0^T \int_{D_y} \psi(x,t) g(U(t,x,y)) dx dt, \quad \text{for } \mu \text{ -a.e } y \in Y. \tag{3.2}$$

Here, $\psi \in L^1_{\text{loc}}(D_y \times (0,T))$ is a *test function* and $g \in C^k(\mathbb{R}^m)$, for $k \geq 1$. For fixed functions ψ, g, we can then define the *parameters-to-observable* map:

$$\Psi : y \in Y \to \Psi(y) = L_g(y,U), \tag{3.3}$$

with L_g being defined by (3.2).

In practice, computing observables involves first solving the underlying PDE by suitable numerical methods such as finite difference, finite element, finite volume or spectral methods and then evaluating the corresponding observable by another numerical method, usually by approximating the integral in (3.2) with a quadrature rule. We denote the numerical approximation of the PDE solution for every parameter vector $y \in Y$ by $U^\Delta(y) \approx U(y)$, with Δ denoting the grid resolution (e.g. mesh size, time step) and similarly we numerically approximate the integral in the observable $L_g^\Delta(y,u) \approx L_g(y,u)$. Hence, there exists an approximation to the *parameters-to-observable* map Ψ^Δ of the form,

$$\Psi^\Delta : y \in Y \to \Psi^\Delta(y) = L_g^\Delta(y, U^\Delta). \tag{3.4}$$

The computation of observables can be very expensive as solving the underlying (nonlinear) PDE, especially in three space dimensions, entails a large computational cost, even on state of the art high performance computing (HPC) platforms. This high computational cost is particularly evident when one considers large scale problems such as uncertainty quantification, (Bayesian) inverse problems, data assimilation or optimal control/design. All these problems are of the *many query* type i.e, the underlying PDE has to be solved for a very large number of instances, each corresponding to a particular realization of the input parameter space, in order to compute the parameters-to-observable map. Querying the computationally costly PDE solver multiple times renders these problems prohibitively expensive.

Our objective is to approximate observables with neural networks, which are very cheap to evaluate and therefore well suited for the many query problems described above. In the last chapter it was proven that neural networks can approximate the solution of many PDEs without the curse of dimensionality. The bottleneck is that the supervised learning of neural networks still requires (a lot of) training data, which

needs to be generated using expensive PDE solvers. This brings us to the central question of this chapter:

> How can we approximate an observable with a neural network to a high accuracy with as little as possible training data?

3.2 Training with low-discrepancy sequences

This section is based on the results in the paper 'Enhancing accuracy of deep learning algorithms by training with low-discrepancy sequences' by S. Mishra and T.K. Rusch [56].

In many machine learning applications, the choice of training samples is restricted to pre-existing datasets. Famous examples include the MNIST database of handwritten digits and the CIFAR-10 database for image classification. In scientific computing, the training samples generally are generated using numerical simulations, which gives a high degree of control on the size and properties of the training set. This luxury however (literally) comes with a price: as these simulations often involve the numerical solution of PDEs, which can be very expensive for high-dimensional problem. In contrast to the usual big data applications of machine learning, in scientific computing we are interested in achieving high accuracy with as little data as possible. A careful choice of the mechanism to generate the training data set is crucial.

In Section 2.4, it was shown that $\overline{\mathcal{E}} \sim N^{-1/2}$ for randomly generated training samples. Because of this slow convergence rate, many training samples are needed to achieve a high accuracy, which is not ideal for scientific computing purposes. In Figure 3.1a, we show 256 randomly generated points from the uniform distribution on $[0,1]^2$. Observe that there are some large areas with very few training samples. Intuitively, these gaps in the training data will have a negative effect on the error of the trained network. The use of points taken from a uniform grid will avoid these gaps, but will incur the curse of dimensionality as N^d points are needed to fill $[0,1]^d$ using a grid with grid size $1/N$. Ideally, we can generate evenly distributed points that share some properties with random points, such that they do not suffer from the curse of dimensionality. In this section, we will show that **low-discrepancy sequences** (LDS) satisfy these requirements for moderately high dimensions.

3.2.1 Low-discrepancy sequences

We first introduce the notion of low-discrepancy sequences and give a few examples of such sequences. Let μ be the Lebesgue measure and let $y = \{y_n\}_{n\in\mathbb{N}}$ be a sequence

of points with $y_n \in Y = [0,1]^d$. For any Lebesgue-measurable set $A \subset Y$ and $N \in \mathbb{N}$, let

$$R_N(A) = \left| \frac{1}{N} \sum_{n=1}^{N} \chi_A(y_n) - \mu(A) \right|, \tag{3.5}$$

where χ_A is the indicator function on the set A. Note that (3.5) is the error of approximating the Lebesgue measure of a set $\mu(A)$ with the number of points y_n, $1 \leq n \leq N$, that are in A divided by N. The discrepancy of the sequence y is then defined as

$$D_N = \sup_{A \in E} R_N(A) \quad \text{where } E = \left\{ \prod_{i=1}^{d} [a_i, b_i) : 0 \leq a_i < b_i \leq 1 \right\}. \tag{3.6}$$

Roughly speaking, the discrepancy measures how well a sequence fills the underlying domain (i.e. how uniform, without gaps). A related quantity is that of the star-discrepancy D_N^*,

$$D_N^* = \sup_{A \in E^*} R_N(A) \quad \text{where } E^* = \left\{ \prod_{i=1}^{d} [0, b_i) : 0 < b_i \leq 1 \right\}. \tag{3.7}$$

Because of its definition, the star-discrepancy is slightly easier to work with than the discrepancy. The discrepancy and star-discrepancy of a sequence are related by $D_N^* \leq D_N \leq 2^d D_N^*$. As a consequence, we will be able to deduce the complexity in N of D_N from that of D_N^*.

Exercise 3.1 Let $Y = [0,1]^2$ and let $N \in \mathbb{N}$ be such that $\sqrt{N} \in \mathbb{N}$. Let $\{y_n\}_{1 \leq n \leq N}$ be the points of a uniform 2D grid with grid size $1/\sqrt{N}$. Compute the star-discrepancy of $\{y_n\}_{1 \leq n \leq N}$. Note these points technically do not constitute a sequence, since when we increase N to $N+1$, all previous values change as well, instead of only adding a new one at the end. Nevertheless, the notion of star-discrepancy is well-defined for any set of points.

Exercise 3.2 Let y be a sequence of iid uniformly distributed points in $[0,1]^d$. Show that $D_N^* = O(N^{-1/2})$.

We are now in a position to follow [8] and define low-discrepancy sequences below.

Definition 3.1 A sequence of points $\{y_n\}_{n \in \mathbb{N}}$ be a sequence of points with $y_n \in Y = [0,1]^d$ is termed as a **low-discrepancy sequence** if there exists a constant $C > 0$ such that for all $N \in \mathbb{N}$ it holds that

$$D_N^* \leq \frac{C(\log N)^d}{N}. \tag{3.8}$$

The constant C is independent of N, but might depend on d.

From this definition, it is clear that neither uniform grid points, nor random points give rise to a low-discrepancy sequence. In the following, we give some examples of low-discrepancy sequence and how to construct them. Many of the examples are based on number theoretic considerations.

The simplest low-discrepancy sequence is the one-dimensional **van der Corput sequence**. It is based on the on the representation of natural numbers in a numeral system using base $b \in \mathbb{N}$. For every $n \in \mathbb{N}$ there exist digits $0 \leq d_k(n) < b$ such that $n = \sum_{k=0}^{K-1} d_k(n) b^k$ for some $K \in \mathbb{N}$. The van der Corput sequence with base b is then defined as

$$y_n = \sum_{k=1}^{K} d_k(n) b^{-k}, \quad n \in \mathbb{N}. \tag{3.9}$$

For base $b = 10$, the van der Corput sequence starts as

$$\frac{1}{10}, \frac{2}{10}, \ldots, \frac{9}{10}, \frac{1}{100}, \frac{11}{100}, \frac{21}{100}, \ldots, \frac{91}{100}, \frac{2}{100}, \frac{12}{100}, \ldots \tag{3.10}$$

The natural generalization of the van der Corput sequence to higher dimensions is the **Halton sequence** [27]. Let $b_1, \ldots, b_d \in \mathbb{N}$ be coprime bases, i.e. for every pair of bases the only positive integer that evenly divides both of them is 1. Then let $y^{b_1}, \ldots, y^{b_d}$ be one-dimensional van der Corput sequence with the respective bases. A Halton sequence y is then generated as

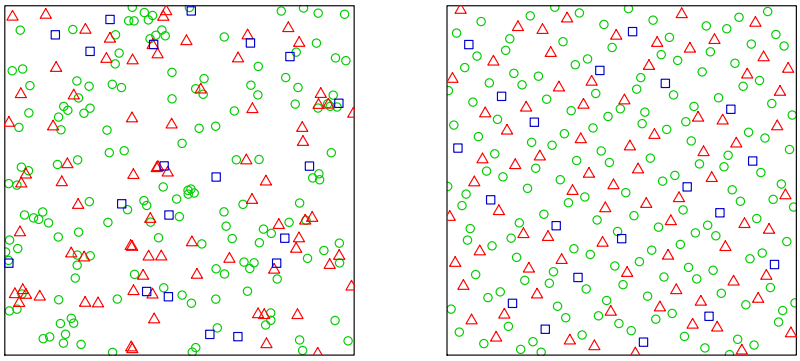
$$y_n = (y_n^{b_1}, \ldots, y_n^{b_d}), \quad n \in \mathbb{N}. \tag{3.11}$$

In Figure 3.1, we compare 256 random points generated from the uniform distribution on $[0, 1]^2$ with the first 256 points of a two-dimensional Halton sequence. Whereas the random points sometimes form clusters and leave large gaps, the Halton points are evenly distributed over the whole space.

For very high dimensions d, the definition of a Halton sequence requires that very high bases are used. As a result, the Halton sequence might show correlations between some coordinate dimensions if N is not very large and hence might not fill the space evenly. In this case, **Sobol sequences** [68] perform better. The definition of Sobol sequences is rather technical and out of scope for these lecture notes, but can be found in e.g. [8]. Other low-discrepancy sequences are Owen [58] and Niederreiter [57] sequences.

3.2.2 DL-LDS

We can now propose a deep learning algorithm that uses low-discrepancy sequences. In short, the algorithm corresponds to the one described in Section 2.4, but instead of random training samples, points from low-discrepancy sequences are used. An overview is given below.



(a) Random points. (b) Halton points.

Fig. 3.1: Example of 256 random and Halton points. The first 20 points are displayed as blue squares, the next 80 points as red triangles and the remaining 156 points as green circles.

DL-LDS: Deep Learning using Low-Discrepancy Sequences

Goal: Find a neural network Ψ^* that approximates the function $\Psi : Y \to \mathbb{R}$.

Step 1: Choose the training set $\mathcal{S} = \{y_n\}_{n=1}^N$ with $y_n \in Y$ for all $1 \leq n \leq N$, such that the sequence $y = \{y_n\}_{n\in\mathbb{N}}$ is a low-discrepancy sequence in the sense of Definition 3.1. Evaluate $\Psi(y_n)$ for all $y_n \in \mathcal{S}$ by a suitable numerical method.

Step 2: Choose a loss function based on $\mathcal{S}$ (see e.g. (2.80)), fix a neural network architecture and initialize the network Ψ_θ by choosing a parameter $\theta \in \Theta$.

Step 3: Run a stochastic gradient descent algorithm until an approximate local minimum $\theta^*(\mathcal{S})$ of the loss function is reached. The map $\Psi^* = \Psi_{\theta^*(\mathcal{S})}$ is the desired neural network approximating the function Ψ.

3.2.3 Generalization error of DL-LDS

Recall that our motivation for considering low-discrepancy sequences as training data was to improve the convergence rate of the generalization error for random training data, which is $\overline{\mathcal{E}} \sim N^{-1/2}$ (see Section 2.4). In this section, we will prove

that when DL-LDS is used the error of the trained network (for $p = 1$) can be bounded by

$$\mathcal{E}(\theta^*(\mathcal{S})) \leq \mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}) + \frac{C(\log N)^d}{N}, \tag{3.12}$$

for $C > 0$ independent of N.

A first key ingredient of the proof is the concept of the *Hardy-Krause variation* of a function. The general definition is rather technical and can be found in e.g. [56]. For one-dimensional, smooth enough functions $f : [0, 1] \to \mathbb{R}$ the Hardy-Krause variation $V_{HK}(f)$ is equal to the total variation of the function,

$$V_{HK}(f) = \int_0^1 \left| \frac{df}{dy}(y) \right| dy. \tag{3.13}$$

In general, the Hardy-Krause variation of a smooth enough function $f : [0, 1]^d \to \mathbb{R}$ is recursively defined in terms of the restrictions $f_1^{(i)}$ of the function f to the boundary $y_i = 1$,

$$V_{HK}(f) = \int\limits_{[0,1]^d} \left| \frac{\partial^d f(y)}{\partial y_1 \partial y_2 \cdots \partial y_d} \right| dy + \sum_{i=1}^{d} V_{HK}(f_1^{(i)}). \tag{3.14}$$

One sufficient smoothness assumption is that all the mixed partial derivatives that appear in (3.14) are continuous [8].

Example 3.1 For $d = 2$, the Hardy-Krause variation (3.14) of a smooth enough function $f : [0, 1]^2 \to \mathbb{R}$ is given by

$$V_{HK}(f) = \int\limits_{[0,1]^2} \left| \frac{\partial^2 f(y)}{\partial y_1 \partial y_2} \right| dy + \int_0^1 \left| \frac{\partial f}{\partial y_2}(1, y_2) \right| dy_2 + \int_0^1 \left| \frac{\partial f}{\partial y_1}(y_1, 1) \right| dy_1 \tag{3.15}$$

Next, we present a central result in quasi-Monte Carlo (QMC) integration theory; the Koksma-Hlawka inequality. It quantifies the error made when an integral is approximated using the quasi-Monte Carlo method.

Lemma 3.1 *Let $g \in L^1([0, 1]^d)$ such that $V_{HK}(g) < \infty$. Then it holds that*

$$\left| \int_{[0,1]^d} g(y) dy - \frac{1}{N} \sum_{n=1}^{N} g(y_n) \right| \leq V_{HK}(g) D_N^*. \tag{3.16}$$

Recalling the definitions of the training error $\mathcal{E}_T$ (2.81) and error $\mathcal{E}$ (2.82) of a neural network, we see that we could obtain our wanted bound (3.12) from the Koksma-Hlawka inequality with $g(y) = |f(y) - f^*(y)|$ as long as $V_{HK}(|f - f^*|) < \infty$. Indeed, then we could calculate that

$$|\mathcal{E}(\theta^*(\mathcal{S})) - \mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S})| \leq V_{HK}(|f - f^*|) D_N^* \tag{3.17}$$

$$\leq C V_{HK}(|f - f^*|) \frac{(\log N)^d}{N}, \tag{3.18}$$

where we used Lemma 3.1 and Definition 3.1. Unfortunately, directly proving this is difficult because of the absolute value in the definition of the (training) error. Instead, we prove the following result [56, Lemma 3.5].

Theorem 3.1 *Let $d \in \mathbb{N}$ and let $f : [0,1]^d \to \mathbb{R}$ be a function with bounded mixed first-order partial derivatives and $V_{HK}(f) < \infty$. Let f^* be a neural network with activation function $\sigma \in C^d(\mathbb{R})$. Then it holds for any given tolerance $\delta > 0$ (independent of dimension d) that*

$$\mathcal{E}(\theta^*(\mathcal{S})) \leq \mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}) + \frac{C(\log N)^d}{N} + 2\delta. \tag{3.19}$$

Here, the constant $C > 0$ may depend on the tolerance δ and the dimension d, but is independent of the number of training samples N.

Proof To prove (3.19) for any given $\delta > 0$, we use an auxiliary function $\rho_\delta \in C^d(\mathbb{R})$ for which

$$\| |u| - \rho_\delta(u) \|_{L^\infty(\mathbb{R})} \leq \delta. \tag{3.20}$$

The existence of such a function ρ_δ can easily be verified by mollifying the absolute value function in the neighbourhood of the origin. Next, we denote,

$$\mathcal{E}^\delta(\theta^*(\mathcal{S})) = \int_{[0,1]^d} \rho_\delta\left(f(y) - f^*(y)\right) dy, \tag{3.21}$$

$$\mathcal{E}_T^\delta(\theta^*(\mathcal{S}), \mathcal{S}) = \frac{1}{N} \sum_{i=1}^{N} \rho_\delta\left(f(y_n) - f^*(y_n)\right). \tag{3.22}$$

Then, it is easy to see that $\mathcal{E}_T^\delta(\theta^*(\mathcal{S}), \mathcal{S})$ is the QMC approximation of $\mathcal{E}^\delta(\theta^*(\mathcal{S}))$ and we can apply the Koksma-Hlawska inequality as in (3.17) to obtain that,

$$|\mathcal{E}^\delta(\theta^*(\mathcal{S})) - \mathcal{E}_T^\delta(\theta^*(\mathcal{S}), \mathcal{S})| \leq C \frac{V_{HK}\left(\rho_\delta\left(f - f^*\right)\right) (\log N)^d}{N}. \tag{3.23}$$

From the assumption that the activation function $\sigma \in C^d(\mathbb{R})$ and structure of the artificial neural network, we see that for any $\theta \in \Theta$, the neural network f^* is a composition of affine (hence C^d) and sufficiently smooth C^d functions, with each function in the composition being defined on compact subsets of $\mathbb{R}^{\overline{d}}$, for some $\overline{d} \geq 1$. Thus, for this composition of functions, we can directly apply Theorem 4 of [4] to conclude that $V_{HK}(f^*) < \infty$. We have also assumed that $V_{HK}(f) < \infty$. It is straightforward to check that

$$V_{HK}(f - f^*) < \infty. \tag{3.24}$$

Next, we observe that $\rho_\delta \in C^d(\mathbb{R})$ by our assumption (3.20). Combining this with (3.24), we use Theorem 4 of [4] (with an underlying multiplicative Faà di Bruno's formula for compositions of multivariate functions) to conclude that

$$V_{HK}\left(\rho_\delta\left(f - f^*\right)\right) \leq C < \infty, \tag{3.25}$$

where the constant $C > 0$ depends on the dimension and the tolerance δ. Applying (3.25) in (3.23) and identifying all the constants as C yields,

$$|\mathcal{E}^\delta(\theta^*(\mathcal{S})) - \mathcal{E}_T^\delta(\theta^*(\mathcal{S}), \mathcal{S})| \leq C\frac{(\log N)^d}{N}, \tag{3.26}$$

where $C > 0$ is a combination of the previous constants that might depend on d and δ. It is straightforward to conclude from assumption (3.20) that

$$\max\left\{|\mathcal{E}(\theta^*(\mathcal{S})) - \mathcal{E}^\delta(\theta^*(\mathcal{S}))|, |\mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}) - \mathcal{E}_T^\delta(\theta^*(\mathcal{S}), \mathcal{S})|\right\} \leq \delta. \tag{3.27}$$

Combining (3.26) with (3.27) and a straightforward application of the triangle inequality leads to the desired inequality (3.19) on the generalization error. □

Some remarks about the result from Theorem 3.1 are in order.

- First of all, Theorem 3.1 presents a deterministic bound on the error of the trained network. It holds for any training set generated using a low-discrepancy sequences. This is in contrast with the results from Section 2.4 for random training points, where the bounds only hold either with a certain probability or for the cumulative error. In principle, the constant $C > 0$ of Theorem 3.1 can be explicitly calculated from the definition of a neural network and the generalized Faà di Bruno formula, provided that a bound on $V_{HK}(f)$ is available. It is however well known that estimates on the QMC integration error in terms of the Koksma-Hlawka inequality can be very large overestimates [8]. The role of (3.19) is therefore primarily to illustrate the convergence rate of DL-LDS.
- Second, the estimate of Theorem 3.1 requires a certain regularity of the underlying function, namely that $V_{HK}(f) < \infty$. As this condition only requires the boundedness on the mixed first-order partial derivatives, it is a significantly weaker requirement than assuming that the function is d times continuously differentiable.
- Next, we see from (3.19) that as long as the number of training samples $N \geq 2^d$, the error for the deep learning algorithm with a low-discrepancy sequence training set decays at much faster (linear) rate than the corresponding $1/\sqrt{N}$ decay for the deep learning algorithm with a randomly chosen training set. We can expect that for low to moderately high input dimensions and sufficiently regular functions, DL-LDS will be significantly more accurate than the standard deep learning algorithm with a randomly chosen training set. On the other hand, either for maps with low-regularity or for problems in very high dimensions (e.g. $d > 20$), the theory developed here suggests no significant advantage using

a low-discrepancy sequence as the training set over randomly chosen points, in the context of deep learning.

- Finally, we investigate the issue of activation functions in this context. In deriving the bound (3.19), we assume that the activation function σ of the neural network is sufficiently smooth, i.e. $\sigma \in C^d(\mathbb{R})$. Thus, standard choices of activation functions such as sigmoid and tanh are admissible under this assumption.
  On the other hand, the ReLU function, which is only Lipschitz continuous, is widely used in deep learning. It can be shown that ReLU neural networks can have infinite Hardy-Krause variation and thus might not be suitable for DL-LDS. Numerical experiments confirm this suspicion.

3.2.4 Numerical experiments

We present numerical evidence that the theoretical results from the previous sections are indeed observed in practice. For $d, r \in \mathbb{N}$, consider the function

$$f_{d,r} : [0,1]^d \to \mathbb{R} : x \mapsto \left(\max \left\{ \sum_{i=1}^{d} x_i - \frac{1}{2}, 0 \right\} \right)^r . \tag{3.28}$$

In our example, we let $d = 3$, $r = 4$. As it can be calculated that $V_{HK}(f_{3,4}) < \infty$, using a neural network with sigmoid activation function and applying the error estimate (3.19) suggests a linear decay of the error of the trained network. On the other hand, (3.19) does not directly apply if the underlying activation function is ReLU. We approximate $f_{3,4}$ with the following three algorithms:

- DL-LDS with Sobol sequences as the training set and sigmoid activation function. This configuration is referred to as DL_{sob}.
- DL-LDS with Sobol sequences as the training set and ReLU activation function. This configuration is referred to as ReLU-DL_{sob}.
- The standard deep learning algorithm with random points as the training set and sigmoid activation function. This configuration is referred to as DL_{rand}.

For each configuration, we train deep neural networks with architectures and hyperparameters specified in [56] with exactly 1 hidden layer. We train 100 of these one-hidden layer networks, each with a different random initialization of the stochastic gradient descent algorithm, and present the average (over the ensemble) error for each of the above 3 configurations, for different numbers of training points, and present the results in figure 3.2. The error of the trained network is approximated by testing the output of the networks on a test set of 8192 Sobol points for DL_{sob} and 8192 random points for DL_{rand}, which are independent of the training set. As seen from the figure, the error with the DL_{sob} algorithm is significantly smaller than the other 2 configurations and decays superlinearly with respect to the number of training samples. On the other hand, there is no significant difference between the ReLU-DL_{sob} and DL_{rand} algorithms for this example. The error decays much more

slowly than for the DL_{sob} algorithm. Thus, this example illustrates that it might not be advisable to use ReLU as an activation function in DL-LDS with low-discrepancy sequences. Smooth activation functions, such as sigmoid and tanh should be used instead.

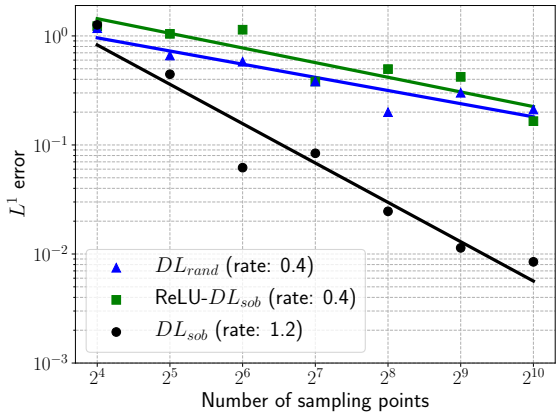


Fig. 3.2: L^1 generalization errors for approximating $f_{3,4}$ (3.28) using DL_{sob} based on ReLU and sigmoid activation functions compared to the generalization error when using DL_{rand}.

3.3 Higher-order quasi-Monte Carlo training

This section is based on the results in the paper 'Higher-order Quasi-Monte Carlo Training of Deep Neural Networks' by M. Longo, S. Mishra, C. Schwab and T.K. Rusch [42].

In the previous section, it was shown that the use of low-discrepancy sequences as training points leads to a massive improvement (over random points) of the error of the trained network for moderately high input dimensions. This improvement provably holds for the large class of functions with bounded Hardy-Krause variation. In this section we discuss how we can improve the error decay even more for very high dimensions through the choice of the training points, at the price of restricting ourselves to holomorphic functions.

Like in the previous section, results from quasi-Monte Carlo (QMC) integration theory provide inspiration. Recent developments in QMC theory, namely the design of *higher-order QMC* (HoQMC) rules for numerical integration [18, 17, 21], establish a dimension-independent, superlinear decay of the quadrature error as long as the integrand appearing in the loss function is holomorphic. This is achieved by

using **extrapolated lattice points** or **interlaced lattice points**, of which the definition is rather technical and can be found in [42, Section 2.1]. We term the resulting algorithm **DL-HoQMC**.

DL-HoQMC: Deep Learning using higher-order QMC training points

Goal: Find a neural network Ψ^* that approximates the function $\Psi : Y \to \mathbb{R}$.
Step 1: Choose for training set $\mathcal{S}$ extrapolated lattice points or interlaced lattice points. Evaluate $f(y_n)$ for all $y_n \in \mathcal{S}$ by a suitable numerical method.
Step 2: Choose a loss function based on $\mathcal{S}$ (see e.g. (2.80)), fix a neural network architecture and initialize the network Ψ_θ by choosing a parameter $\theta \in \Theta$.
Step 3: Run a stochastic gradient descent algorithm until an approximate local minimum $\theta^*(\mathcal{S})$ of the loss function is reached. The map $\Psi^* = \Psi_{\theta^*(\mathcal{S})}$ is the desired neural network approximating the function Ψ.

In [42], it is proven that the generalization error decays at the rate that is independent of the dimension. The rate itself depends on the properties of the function of interest. Without going into details, we (informally) summarize the main result [42, Theorem 3.14] below.

Theorem 3.2 *Let $\alpha \in \mathbb{N}$ with $\alpha \geq 2$. If f is a holomorphic function that satisfies some technical assumptions depending on α, then (under some conditions) the generalization error of a neural network with a holomorphic activation function decays as $N^{-\alpha}$.*

To show this higher-order decay in practice, we consider a test case that arises as a prototype for uncertainty quantification (UQ) in elliptic PDEs with uncertain coefficient. On the bounded physical domain $D = (0,1)^2$ and with parameters $y \in \left[-\frac{1}{2}, \frac{1}{2}\right]^d$, we consider the following elliptic equation with homogeneous Dirichlet boundary conditions

$$\begin{cases} -\operatorname{div}\left(a(x,y)\nabla u(x,y)\right) = f(x) & x \in D, \\ u(x,y) = 0 & x \in \partial D. \end{cases} \tag{3.29}$$

We choose a deterministic source $f(x) = 10x_1$, and the observable $g\colon [-\frac{1}{2}, \frac{1}{2}]^d \to \mathbb{R}$ given by

$$\mathcal{L}(y) := \frac{1}{|\widetilde{D}|} \int_{\widetilde{D}} u(\cdot, y), \tag{3.30}$$

for $\widetilde{D} = (0, \frac{1}{2})^2$. We assume that the diffusion coefficient $a(x,y)$ is affine-parametric, that is

$$a(x,y) = \overline{a}(x) + \sum_{j=1}^{d} y_j \psi_j(x). \tag{3.31}$$

Here we choose $\overline{a} \equiv 1$, $\psi_j(x) := \psi_{(k_1,k_2)}(x) = \frac{1}{(k_1^2+k_2^2)^\eta} \sin(k_1\pi x_1)\sin(k_2\pi x_2)$ and the ordering is defined by $(k_1, k_2) < (\overline{k}_1, \overline{k}_2)$ when $k_1^2+k_2^2 < \overline{k}_1^2+\overline{k}_2^2$ and is arbitrary when equality holds.

The resulting generalization error for approximating the observable $\mathcal{L}$ (or more precisely, a finite element approximation of $\mathcal{L}$) on parameter domains of dimension 16 and 32, with extrapolated lattice training points, can be seen in Figure 3.3. We observe that for both choices of the parameter dimension d, the generalization error not only has the same convergence rate of around 2.1, but also has almost the same absolute error. This is again in accordance with our theoretical findings, where we claimed the generalization error to be independent of the dimension of the underlying problem. More details on this experiment can be found in [42, Section 4].

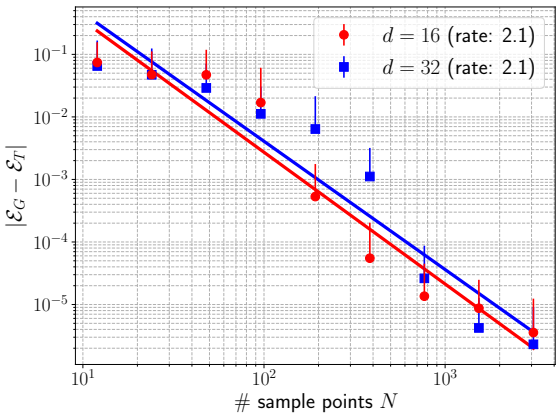


Fig. 3.3: The gap between the total error and the training error $|\mathcal{E} - \mathcal{E}_T|$ (mean and standard deviation of trained ensemble) for approximating the observable $\mathcal{L}$ corresponding to the elliptic PDE (3.29) in 16 dimensions as well as in 32 dimensions.

3.4 Multi-level training

This section is based on the results in the paper 'A multi-level procedure for enhancing accuracy of machine learning algorithms' by K.O. Lye, S. Mishra and R. Molinaro [44].

Multi-level methods were introduced in the context of numerical quadrature by Heinrich in [29] and for numerical solutions of stochastic differential equations (SDEs) by Giles in [22]. They have been heavily used in recent years for uncertainty quantification in PDEs, solutions of stochastic PDEs, data assimilation and Bayesian

inversion. Multi-level methods are inspired by multi-grid and multi-resolution techniques, which have been used in numerical analysis over many decades.

3.4.1 Multi-level DL-LDS

Our goal in this section is to propose a multi-level version of the deep learning algorithm DL-LDS (Section 3.2.2). The basis of this algorithm is the observation that the underlying parameters-to-observable map $\Psi : Y \to \mathbb{R}$ (3.3) can be approximated on a sequence of mesh resolutions Δ_ℓ, for $0 \leq \ell \leq L$ for some $L > 0$, where Ψ^Δ is as in (3.4). We require that $\Delta_\ell < \Delta_{\ell-1}$ for each $1 \leq \ell \leq L$, see Figure 3.4 for a diagrammatic representation of this sequence of grids.

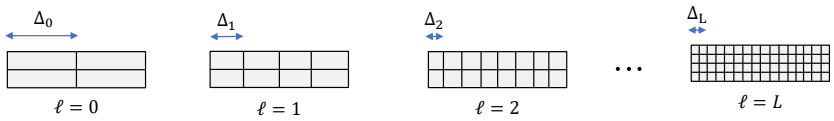


Fig. 3.4: A schematic for a sequence of nested grids used in defining the multi-level algorithms.

For each ℓ, we assume that the underlying parameters-to observable-map Ψ can be approximated by the map Ψ^{Δ_ℓ}, computed on resolution Δ_ℓ. Given such a sequence of resolutions $\{\Delta_\ell\}_{\ell=0}^L$ and approximate parameters-to-observable maps Ψ^{Δ_ℓ}, we have the following telescopic decomposition,

$$\Psi^{\Delta_L}(y) = \Psi^{\Delta_0}(y) + \sum_{\ell=1}^{L} \left(\Psi^{\Delta_\ell}(y) - \Psi^{\Delta_{\ell-1}}(y) \right), \quad \forall y \in Y. \tag{3.32}$$

Introducing the so-called *details*,

$$d_\ell(y) = \Psi^{\Delta_\ell}(y) - \Psi^{\Delta_{\ell-1}}(y), \quad \forall y \in Y,\ 1 \leq \ell \leq L, \tag{3.33}$$

we rewrite the telescopic decomposition (3.32) as

$$\Psi^{\Delta_L}(y) = \Psi^{\Delta_0}(y) + \sum_{\ell=1}^{L} d_\ell(y), \quad \forall y \in Y. \tag{3.34}$$

The multi-level deep learning algorithm will be based on independently learning the following maps

$$\Psi^{\Delta_0}(y) \approx \Psi_0^*(y), \quad d_\ell(y) \approx d_\ell^*(y), \quad \forall y \in Y,\ 1 \leq \ell \leq L. \tag{3.35}$$

Here, Ψ_0^* and d_ℓ^*, for $1 \le \ell \le L$, are neural networks. If the underlying numerical method converges to a solution of the PDE, then it holds for instance that

$$V_{HK}(|d_\ell - d_\ell^*|) \sim \Delta_\ell^s \quad \text{for some } s > 0, \tag{3.36}$$

resulting in a smaller value of the upper bound on the error of DL-LDS and allowing us to learn the details with significantly fewer training samples. By carefully balancing the Hardy-Krause variation of the details with the computational cost of generating the observable at each level of resolution, we aim to reduce the overall generalization error, while keeping the cost of generating the training samples small. The resulting algorithm is summarized below.

ML-DL-LDS: Multi-Level Deep Learning using Low-Discrepancy Sequences

Goal: Find a neural network Ψ^* that approximates the function $\Psi : Y \to \mathbb{R}$.
Step 1: Choose $N_0 < N_1 < \ldots < N_L \in \mathbb{N}$ and select a training sets $\mathcal{S}_\ell = \{y_n^\ell\}_{n=1}^{N_\ell}$ based on a low-discrepancy sequences and compute $\Psi^{\Delta_\ell}(y_n^\ell)$ and $d_\ell(y_n^\ell)$ for $1 \le n \le N_\ell$ and $0 \le \ell \le L$.
Step 2: Train neural networks $\Psi_0^*, d_1^*, \ldots, d_L^*$ using DL-LDS (Section 3.2.2) such that $\Psi_0^* \approx \Psi^{\Delta_0}, d_1^* \approx d_1, \ldots, d_L^* \approx d_L$.
Step 3: For every $y \in Y$, define the final approximation as

$$\Psi^*(y) = \Psi_0^*(y) + \sum_{\ell=1}^{L} d_\ell^*(y).$$

3.4.2 Computational gain of multi-level training

We now present a heuristical argument why multi-level training can lead to a speed-up over single-level training. In Section 3.2 the following bound (3.17) was proven,

$$\mathcal{E}(\theta^*(\mathcal{S})) \le \mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}) + V_{HK}(|f - f^*|)\frac{C(\log N)^d}{N}, \tag{3.37}$$

where f is some function, f^* is a neural network approximation and N is the number of (low-discrepancy) training samples. To simplify the calculations, we make the assumption (1) that the training error is negligible, (2) that d is low enough so that we can neglect $(\log(N))^d$ as well and (3) that the Hardy-Krause variation of the details grows as in (3.36). We then find the following estimates for the error of a single-level network trained on N samples, denoted by $\mathcal{E}_{SL}$, and the error of a multi-level network trained on $N_0 < \ldots < N_L$ samples, denoted by $\mathcal{E}_{ML}$,

$$\mathcal{E}_{SL} \approx \frac{C\Delta_0^s}{N}, \qquad \mathcal{E}_{ML} \approx \sum_{\ell=0}^{L} \frac{C\Delta_\ell^s}{N_\ell}. \tag{3.38}$$

To make sure that $\mathcal{E}_{SL} \approx \mathcal{E}_{ML} \approx \epsilon$, we must choose $N = \frac{C\Delta_0^s}{\epsilon}$ and $N_\ell = \frac{C(L+1)\Delta_\ell^s}{\epsilon}$ for $0 \leq \ell \leq L$. We now have to quantify the cost of both single-level and multi-level training. We assume that the cost is dominated by the generation of the training data using an expensive PDE solver. For a PDE with spatial dimension d_s (not to be confused with the parameter dimension d), the cost C_Δ of solving the PDE one time using a numerical method based on a grid with size Δ is of the order of $\Delta^{-\overline{d}}$ where $\overline{d} = d_s + 1$ (spatial dimension plus the time). This assumption is justified for first-order time-dependent PDEs (due to the CFL condition) and the arguments below can be readily extended to a more general case. Assuming that $\Delta_\ell = 2^{L-\ell}\Delta$ for some $\Delta > 0$, one can calculate that the single-level cost is given by

$$C_{SL} = NC_{\Delta_L} \sim N\Delta_L^{-\overline{d}} \approx \frac{C\Delta_0^s}{\epsilon}\Delta^{-\overline{d}} = \frac{C2^{Ls}\Delta^{s-\overline{d}}}{\epsilon} \tag{3.39}$$

and the multi-level cost by

$$C_{ML} = \sum_{\ell=0}^{L} N_\ell C_{\Delta_\ell} \sim \sum_{\ell=0}^{L} \frac{C(L+1)\Delta_\ell^s}{\epsilon}\Delta_\ell^{-\overline{d}} \tag{3.40}$$

$$= \frac{C(L+1)(2^L\Delta)^{s-\overline{d}}}{\epsilon}\frac{1-2^{(\overline{d}-s)(L+1)}}{1-2^{(\overline{d}-s)}} \tag{3.41}$$

$$\approx \frac{C(L+1)(2^L\Delta)^{s-\overline{d}}}{\epsilon}2^{(\overline{d}-s)L} \tag{3.42}$$

$$= \frac{C(L+1)\Delta^{s-\overline{d}}}{\epsilon}, \tag{3.43}$$

where we assumed that $s < \overline{d}$. This is in general fulfilled if $\overline{d} = 3, 4$. The speed-up or computation gain G of multi-level training over single-level training is then given by

$$\text{gain } G = \frac{C_{SL}}{C_{ML}} \approx \frac{2^{Ls}}{L+1}. \tag{3.44}$$

We find an exponential gain in the number of levels L. Note however that we made many simplifications and that we neglected some terms. The calculations made above should therefore be taken with a grain of salt.

3.4.3 Numerical experiments

In [44], the computational gain of multi-level training over single-level training is demonstrated using the example of projectile motion. The dynamical system

modeling the motion of a projectile, subjected to both gravity as well as air drag is described by the nonlinear system of ODEs,

$$\begin{aligned} \frac{d}{dt}\boldsymbol{x}(t;y) &= \boldsymbol{v}(t;y), \quad \frac{d}{dt}\boldsymbol{v}(t;y) = -F_D\big(\boldsymbol{v}(t;y);y\big)\boldsymbol{e}_1 - g\boldsymbol{e}_2 \\ \boldsymbol{x}(y;0) &= \boldsymbol{x}_0(y), \quad \frac{d\boldsymbol{x}(y;0)}{dt} = \boldsymbol{v}_0(y). \end{aligned} \tag{3.45}$$

Here, $F_D = \frac{1}{2m}\rho C_d \pi r^2 ||\boldsymbol{v}||^2$ denotes the drag force, with ρ being the air density and m, C_d, r, the mass, the drag coefficient and the radius of the object, respectively. Let further $\boldsymbol{x}_0(y) = [0,\ h]$, $\boldsymbol{v}_0(y) = [v_0\cos(\alpha),\ v_0\sin(\alpha)]$ be the initial position and velocity of the object (see Figure 3.5a for a schematic representation). We assume that m, ρ, C_d, r, h, v_0 and α are all parametrized by one variable. This then gives rise to a seven-dimensional parameter $y \in [0,1]^7$. The objective of the simulation is to compute and quantify uncertainty with respect to the observable corresponding to the horizontal range x_{max} (see Figure 3.5a),

$$\Psi(y) = x_{max}(y) = x_1(t_f; y), \quad \text{with } t_f = x_2^{-1}(0). \tag{3.46}$$

The observable Ψ is then approximated using both DL-LDS and ML-DL-LDS (see [44, Section 6] for details on the training procedure) and we display the computational gain in Figure 3.5b. We plot the mean and the maximum value of the gain G[1] with respect to multi-level hyperparameters for a range of computational costs. From this figure, we see a mean gain between 2 and 8 and a maximum gain of 12 for the multi-level algorithm over the single-level deep learning algorithm. Larger gains were obtained for lower computational costs (less number of training samples), which is the case of practical interest.

3.5 Ensemble training

This section is based on the results in the paper 'Deep learning observables in computational fluid dynamics' by K.O. Lye, S. Mishra and D. Ray [45].

All the deep learning frameworks we have introduced (and will introduce) require us to specify a number of hyperparameters, ranging from the network architecture to the choice of the optimizer. A priori, it is unclear how much the network accuracy will depend on the exact choice of these hyperparameters, as often little theoretical considerations are available. In practice, one often uses the straight-forward approach of trying out a number of hyperparameter combinations and choosing the combination that performs the best. This **ensemble training** technique will be introduced below.

[1] In [44] the gain G is defined as the error obtained using single-level training divided by the multi-level training error.

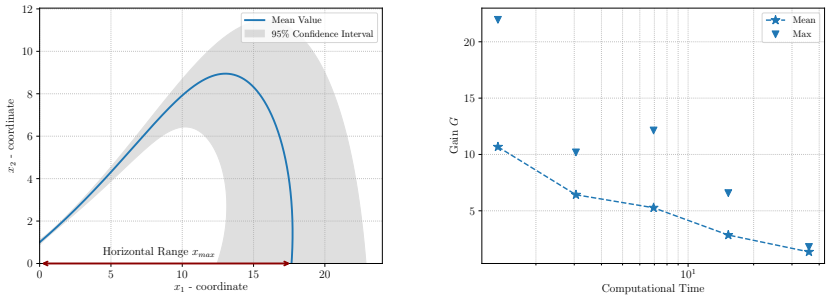


Fig. 3.5: Left: An illustration of two-dimensional projectile motion, with the mean value and the envelope of trajectories, corresponding to the 95% interval shown. Right: Gain (defined as the improvement of the error for a fixed cost) vs. cost.

We first list the most important hyperparameters from the deep learning frameworks considered in these lecture notes.

- *Neural network size:* number of hidden layers and number of neurons in each layer.
- *Loss function:* the choice of the exponents $p, q \in \{1, 2\}$ in the definition of the loss function (2.80), as well as the value of the regularization parameter $\lambda > 0$.
- *Training set:* the number of training points, as well as their nature (e.g. random points, low-discrepancy sequences (Section 3.2), higher-order QMC (Section 3.3)).
- *Optimizer:* an optimization algorithm (e.g. ADAM, RMSprop, L-BFGS...), their hyperparameters (e.g. learning rate) and a stop condition (e.g. number of epochs, early stopping).
- *Other:* in the case of multi-level training (Section 3.4) the number of levels is an additional hyperparameter. In the ensemble training that will be introduced below the selection criterion is an hyperparameter as well.

An exhaustive search of all hyperparameter combinations is often not desirable. For example, if we consider 6 hyperparameters with 5 possible values each, then there are $5^6 = 15625$ combinations that need to be considered, i.e. 15625 networks that need to be trained. Even though these networks can be trained independently (i.e. in a parallel way), this is not done in practice. Instead, a large number of hyperparameters are fixed based on previous experience or theoretical considerations, or a chosen number of hyperparameter combinations is randomly selected.

The next step is to select the optimal hyperparameter combination. The decision of which combination is optimal is based on some selection criterion. Ideally, one would select the network with the smallest error on the whole domain of interest, but this is not possible. A first approach is to approximate this error using k-fold **cross-validation**. In this approach, the training set is partitioned into k subsets of

approximately equal size. One then trains k different networks, each time with only $k-1$ of those subsets as training data and the remaining set as a validation set, on which the validation error is calculated. The average of these k validation errors is a statistically sound approximation of the error on the whole domain and can therefore be used as selection criterion. A second selection criterion involves choosing the network with the lowest training error. This method is faster and simpler than the validation error approach, but one must take into account that a low training error does not necessarily imply that the network performs well on the whole domain.

We summarize the ensemble training algorithm below. More details and numerical experiments can be found in the lecture slides and the paper 'Deep learning observables in computational fluid dynamics' by K.O. Lye, S. Mishra and D. Ray [45].

Ensemble training

Goal: Choose optimal neural network hyperparameters
Step 1: Identify all hyperparameters that should be optimized and choose a (small) number of possible values for each hyperparameter.
Step 2: For (a subset of) all possible hyperparameter combinations, train a neural network.
Step 3: Select the optimal hyperparameter combination based on a suitable selection criterion.

3.6 Uncertainty quantification

This section is based on the results in the paper 'Deep learning observables in computational fluid dynamics' by K.O. Lye, S. Mishra and D. Ray [45].

In many applications, observables are calculated based on data from uncertain measurements or approximations. Given statistical information on these measurements, one is often interested in statistical information about the observable itself. This is the topic of uncertainty propagation. In this section, we demonstrate how the presented deep learning algorithms can be used in this form of (forward) uncertainty quantification (UQ).

First recall that the parameter-to-observable map Ψ maps an input parameter $y \in Y$ to the observable $\Psi(y)$. Now assume that we access to the probability distribution of the input parameters through the input measure μ on Y. In the context of forward UQ, we are interested in how this initial measure is changed by Ψ. Hence, we consider the pushforward measure $\nu := \Psi\#\mu$, which is defined such that for every measurable function g it holds that

$$\langle \nu, g \rangle := \int_{\mathbb{R}} g(z) d\nu(z) = \int_{Y} g(\Psi(y)) d\mu(y). \tag{3.47}$$

In particular, this formulation allows us to compute any statistical moment of interest. From the first moment (by setting $g(z) = z$) and second moment (by setting $g(z) = z^2$), the mean and variance can easily be calculated,

$$\overline{\nu} := \int_Y \Psi(y) d\mu(y), \qquad \mathrm{Var}(\nu) := \int_Y (\Psi(y))^2 d\mu(y) - \overline{\nu}^2. \tag{3.48}$$

The task of efficiently computing the probability distribution ν is however significantly harder than just estimating the mean and variance of the underlying map.

Classically, ν is approximated using (quasi-)Monte Carlo algorithms. In these algorithms, M samples $\{y_m\}_{m=1}^M \subset Y$ are chosen and then the pushforward measure ν is approximated by

$$\nu \approx \nu_M = \frac{1}{M} \sum_{m=1}^{M} \delta_{\Psi(y_m)}, \tag{3.49}$$

and therefore we also get the approximation

$$\langle \nu, g \rangle \approx \langle \nu_M, g \rangle = \frac{1}{M} \sum_{m=1}^{M} g(\Psi(y_m)). \tag{3.50}$$

In Monte Carlo algorithms, the samples $\{y_m\}_{m=1}^M$ are iid distributed, whereas in quasi-Monte Carlo algorithms low-discrepancy sequences as in Section 3.2 are used. The caveat of these approaches is that Ψ needs to be evaluated M times, which can be very expensive.

Similar to the previous sections, we propose to replace the exact parameter-to-observable map Ψ by a neural network surrogate Ψ^*. The resulting algorithm is summarized below.

DLQMC: Deep Learning Quasi-Monte Carlo

Goal: Find an approximation ν^* of the pushforward measure $\nu = \Psi\#\mu$.
Step 1: Generate a training set $\{\overline{y}_n\}_{n=1}^N \subset Y$ based on low-discrepancy sequences and evaluate all $\Psi(\overline{y}_n)$.
Step 2: Train a neural network Ψ^* to approximate Ψ using DL-LDS.
Step 3: Define $\nu_M^* = \frac{1}{M} \sum_{m=1}^M \delta_{\Psi^*(y_m)}$ as an approximation to ν.

We demonstrate why the use of DLQMC can be advantageous by analysing its cost and comparing it with the cost of the classical QMC algorithm. Suppose we want to approximate $\langle \nu, g \rangle$ to an accuracy of $\epsilon > 0$. Based on the results of Section 3.2, we find that the choice $M \sim \epsilon^{-1}$ suffices (up to polylogarithmic factors). Denoting the cost of evaluating Ψ one time by C, we find that the cost of QMC is given by $C_{\mathrm{QMC}} \sim CM$. In order to reach the same accuracy using DLQMC, one can for instance require the following,

$$\int_Y |\Psi - \Psi^*| d\mu(y) \leq \frac{\epsilon}{2} \quad \text{and} \quad \left| \langle \nu, g \rangle - \langle \nu_M^*, g \rangle \right| \leq \frac{\epsilon}{2}. \tag{3.51}$$

Neglecting the training error, the first requirement boils down to setting $N \sim \epsilon^{-1}$. Denoting the cost of evaluating the neural network Ψ^* one time by C^*, we find that the second requirement brings us $M \sim \epsilon^{-1}$. As a result, the cost of DLQMC is given by $C_{\text{DLQMC}} \sim CN + C^*M + C_{\text{train}}$, where C_{train} is the cost of training Ψ^*. Since evaluating a neural network is very cheap and C is the cost of using an expensive PDE solver, it is clear that $C \ll C^*$. Consequently, we find that DLQMC is faster than QMC if the training cost is sufficiently low and most importantly if $N < M$. This again confirms the importance of being able to find accurate neural network approximations with little data.

More details and numerical experiments can be found in the lecture slides and the paper 'Deep learning observables in computational fluid dynamics' by K.O. Lye, S. Mishra and D. Ray [45].

3.7 PDE-constrained optimization

This section is based on the results in the paper 'Iterative surrogate model optimization (ISMO): An active learning algorithm for PDE constrained optimization with deep neural networks' by K.O. Lye, S. Mishra, D. Ray and P. Chandrashekar [46].

We conclude the chapter by discussing how deep learning can be used to solve PDE-constrained optimization problems, i.e. optimization problems where one or more constraints are formulated in terms of a partial differential equation. We demonstrate the importance of these problems using the example of shape optimization of airfoils. The shape S of an airfoil can be parametrized using so-called Hicks-Henne basis functions $\phi_i = \phi(y_i)$,

$$S(y) = S_{\text{ref}} + \sum_{i=1}^{d} \phi(y_i), \tag{3.52}$$

where S_{ref} is the reference shape and $y \in Y = [0,1]^d$ is a (shape) parameter. The flow around an airfoil can be modeled by e.g. the Euler equations and the observables of interest are the lift $C_L(y)$ and the drag $C_D(y)$ for every shape $S(y)$. The goal of aerodynamic shape optimization is to find the shape parameter y that minimizes the drag while keeping the lift higher than a reference value C_L^{ref}. The constraint on the lift is indeed a PDE constraint as the lift needs to be calculated from the solution to the Euler equations. In an attempt to solve this problem, one can try to minimize the (simplified) cost function, also called objective function,

$$J(y) = C_D(y) + P \max\{C_L^{\text{ref}} - C_L, 0\}, \tag{3.53}$$

where $P > 0$ is a penalization parameter. Standard shape optimization algorithms are often iterative algorithms that require the the gradient $\nabla_y J(y)$ during each iteration. As this gradient involves multiple calls to an PDE solver, the classical approach

often is very expensive and can be even infeasible for shape optimization under uncertainty.

In the general case, J is a cost function that is based on an observable Ψ with parameter y, i.e. $J(y) = G(\Psi(y))$ for some function G. A first algorithm to efficiently approximate

$$\widehat{y} = \arg\min_{y \in Y} J(y) \tag{3.54}$$

using deep learning consists of replacing the observable Ψ by a neural network, in a similar spirit to the previous algorithms of this chapter. We term this algorithm *DNNopt* and summarize it below.

DNNopt: deep learning-based PDE-constrained optimization

Goal: Compute approximate minimizers for the PDE constrained optimization problem (3.54).

Step 1: Construct a training set $\{y_m\}_{m=1}^M \subset Y$ (random points or LDS points depending on the input dimension) and generate a deep neural network surrogate map Ψ^* for the observable Ψ.

Step 2: Draw N random starting points $\{\overline{y}_n\}_{n=1}^N \subset Y$. For every starting value $\overline{y}_n$, run a standard iterative optimization algorithm to obtain an approximate minimizer y_n^* of the cost function $J^*(y) = G(\Psi^*(y))$.

Step 3: The set $\{y_n^*\}_{n=1}^N$ contains approximate minimizers for the PDE constrained optimization problem (3.54).

As neural networks (and their gradients) are very cheap to evaluate, it is much faster to optimize the *DNNopt* cost function J^* (with Ψ replaced by the DNN surrogate Ψ^*) than the original cost function J. However, there might be a problem with the current approach. We demonstrate this using the example of the shape optimization of airfoils from the beginning of this section, where the goal was to maximize the lift-to-drag ratio. We select 64 and 2048 Sobol points as shape parameters and calculate for each of these parameters the lift-to-drag ratio (see Figure 3.6). None of the 64 Sobol points lie in the region of Y that corresponds to the highest lift-to-drag ratio, meaning that a small training set might not represent the extrema well and that therefore DNNopt might not reach the desired extrema. Increasing the training set size would solve this issue, but then the computational advantage over the standard optimization algorithm is lost.

3.7.1 Iterative Surrogate Model Optimization

In [46], one proposes to overcome this problem by constructing the training set in an iterative way, through a feedback loop between the neural networks and the optimization algorithm. Their algorithm, called **Iterative Surrogate Model Optimization (ISMO)**, is based on the simple idea that in order for the training points to better

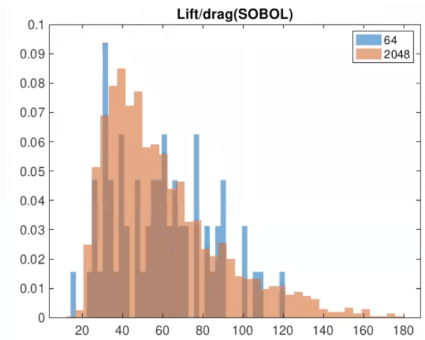


Fig. 3.6: The distribution of lift-to-drag ratio for 64 and 2048 different shape parameters. The shape parameters are taken to be the first 64 (or 2048) points of a Sobol sequence.

represent the extrema, it suffices to add more training points near those extrema. As a consequence, ISMO is likely to be more efficient than DNNopt, where the only option is to add more training points that are evenly distributed in Y. We present ISMO below.

ISMO: Iterative Surrogate Model Optimization	
Goal:	Compute approximate minimizers for the PDE constrained optimization problem (3.54).
Step 0:	Construct an initial training set $\mathcal{S}_0 = \{y_n^0\}_{n=1}^N \subset Y$ (random points or LDS points depending on the input dimension) and generate a deep neural network surrogate map Ψ_0^* for the observable Ψ. Draw ΔN random starting points $\{\overline{y}_n^0\}_{n=1}^{\Delta N} \subset Y$. For every starting value $\overline{y}_n^0$, run a standard iterative optimization algorithm to obtain an approximate minimizer y_n^1 of the cost function $J_0^*(y) = G(\Psi_0^*(y))$.
Step 1 $\leq$ **k** $\leq$ **K**:	Set $\mathcal{S}_k = \mathcal{S}_{k-1} \cup \{y_n^{k-1}\}_{n=1}^{\Delta N}$ and generate a deep neural network surrogate map Ψ_k^* for the observable Ψ. Draw ΔN random starting points $\{\overline{y}_n^k\}_{n=1}^{\Delta N} \subset Y$. For every starting value $\overline{y}_n^k$, run a standard iterative optimization algorithm to obtain an approximate minimizer y_n^{k+1} of the cost function $J_k^*(y) = G(\Psi_k^*(y))$. Repeat for $1 \leq k \leq K$.
Step K+1:	The set $\{y_n^{K+1}\}_{n=1}^{\Delta N}$ contains approximate minimizers for the PDE constrained optimization problem (3.54).

Because of its internal feedback loop, ISMO can be thought of as an example of an **active learning** algorithm, where the learner (the neural network) queries the teacher (the optimization algorithm) to iteratively identify training data that provides a better

approximation of the local optima. We demonstrate the evolution of the training sets $\mathcal{S}_k$, $1 \leq k \leq K$, for an optimization problem where there the set of minimizers of the cost function $\arg\min_{y \in Y} J(y)$ is a parabola. In Figure 3.7, we show the evolution of the training sets $\mathcal{S}_k$. Note how most of the newly added training points lie near this parabola, which will result in a more accurate neural network surrogate model in this region of interest.

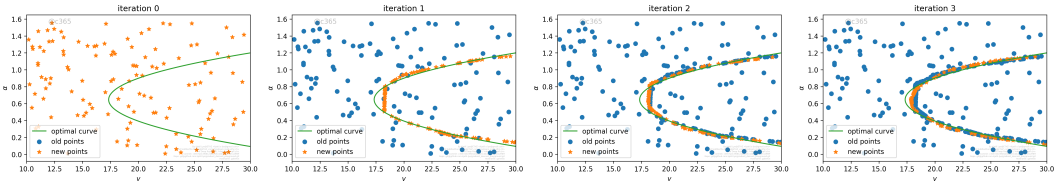


Fig. 3.7: Evolution of training sets $\mathcal{S}_k$, for different iterations k, for the ISMO algorithm. The newly added training points are the orange dots, while existing training points are blue dots. The exact minimizers are displayed as a green parabola.

Finally, we compare the performance of ISMO and DNNopt for the example of the shape optimization of airfoils from the beginning of this section. The goal is to find minimizers of the objective function defined in (3.53). We use ISMO with $N_0 = 64$ and $\Delta N = 16$ and DNNopt with training sets of size $N_k = N_0 + k\Delta N$. Since the computational cost of both algorithms is dominated by the generation of training data, the cost of ISMO and DNNopt is comparable for every k. In Figure 3.8, we show for both algorithms the average value of the objective function J evaluted at the approximate minimizers, and its standard deviation. One can see that ISMO provides better minimizers with a lower variation than DNNopt.

More details and numerical experiments can be found in the lecture slides and the paper 'Iterative surrogate model optimization (ISMO): An active learning algorithm for PDE constrained optimization with deep neural networks' by K.O. Lye, S. Mishra, D. Ray and P. Chandrashekar [46].

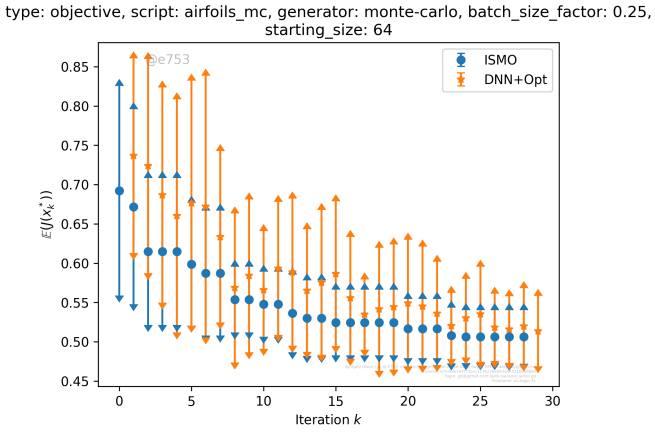


Fig. 3.8: The objective function J (3.53) vs. number of iterations of ISMO (with $N_0 = 64$ and $\Delta N = 16$) and DNNopt (with $N = N_0 + k\Delta N$) for the airfoil shape optimization problem. The average of the objective function J evaluated in the approximate minimizers and the (average ± standard deviation) are shown.

Chapter 4
Unsupervised learning in scientific computing

This chapter is based on the results in the papers:

- *'Estimates on the generalization error of physics informed neural networks (PINNs) for approximating PDEs' by S. Mishra and R. Molinaro [52],*
- *'Physics Informed Neural Networks for Simulating Radiative Transfer' by S. Mishra and R. Molinaro [53],*
- *'Physics Informed Neural Networks (PINNs) for approximating nonlinear dispersive PDEs' by G. Bai, U. Koley, S. Mishra and R. Molinaro [2],*
- *'Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for PDEs' by S. Mishra and R. Molinaro [54].*

.

The previous chapter was dedicated to techniques to train neural networks to a high accuracy with as little training data as possible, motivated by the high cost of evaluating a PDE solver that is needed to generate training data when approximating the solution (or an observable) of a PDE. In this chapter, we go one step further and present an approach that does not even necessarily require the generation of training data. Instead, the PDE formulation, and hence the underlying physics, is directly used to train neural networks.

In this chapter we will consider the following setting. Let X, Y, Z be separable Banach spaces with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$, respectively, and analogously let $X^* \subset X$ and $Y^* \subset Y$ be closed subspaces with norms $\|\cdot\|_{X^*}$ and $\|\cdot\|_{Y^*}$, respectively. Typical examples of such spaces include L^p and Sobolev spaces. We consider PDEs of the following abstract form,

$$\mathcal{D}(u) = f, \qquad \mathcal{B}(u) = 0 \tag{4.1}$$

where $\mathcal{D} : X^* \to Y^*$ is a *differential operator* and $f \in Y^*$ is an *input* or *source* function. We assume that the operator $\mathcal{D}$ also describes the boundary and initial conditions and that for all $f \in Y^*$ there exists a unique $u \in X^*$ such that (4.1) holds. Finally, we assume that for all $u \in X^*$ and $f \in Y^*$ it holds that

$$\|\mathcal{D}(u)\|_{Y^*} < +\infty, \qquad \|f\|_{Y^*} < +\infty. \tag{4.2}$$

In what follows, we present the framework of physics-informed neural networks, which allows to obtain a surrogate model for the solution of (4.1) with little or even no training data.

4.1 Physics-informed neural networks (PINNs)

First proposed by [38, 37] and popularized by [62, 63], *physics-informed neural networks* (PINNs) have emerged as a very successful paradigm for approximating different aspects of solutions of PDEs. They only distinguish themselves from the neural networks of the previous chapters in the way they are trained, as a different loss function is used. The PINN itself is just a plain MLP, as we defined in Definition 2.1, and we denote it by u_θ, with $\theta \in \Theta$, where Θ is still the space of weights and biases. In addition, we assume that $u_\theta \in X^*$ for all $\theta \in \Theta$.

We now define the PDE residual,

$$\mathfrak{R}_\theta = \mathfrak{R}(u_\theta) := \mathcal{D}(u_\theta) - f. \tag{4.3}$$

By the assumptions in (4.2), we see that $\mathfrak{R}_\theta \in Y^*$ and $\|\mathfrak{R}_\theta\|_{Y^*} < +\infty$ for all $\theta \in \Theta$. Note that $\mathfrak{R}(u) = \mathcal{D}(u) - f \equiv 0$, for the solution u of the PDE (4.1). Hence, the term *residual* is justified for (4.3). The strategy of PINNs, following [38], is to minimize the residual (4.3), over the admissible set of tuning parameters $\theta \in \Theta$ i.e., find $\theta^* \in \Theta$ such that

$$\theta^* = \arg\min_{\theta\in\Theta} \|\mathfrak{R}_\theta\|_Y. \tag{4.4}$$

In most practical applications it will be the case that $Y = Y^* = L^p(D)$ for some $1 \le p < \infty$, such that one can equivalently minimize,

$$\theta^* = \arg\min_{\theta\in\Theta} \|\mathfrak{R}_\theta\|^p_{L^p(D)} = \arg\min_{\theta\in\Theta} \int_D |\mathfrak{R}_\theta(y)|^p dy. \tag{4.5}$$

As it will not be possible to evaluate the integral in (4.5) exactly, we need to approximate it numerically by a quadrature rule. To this end, we will use numerical quadrature rules and select the *training set* $\mathcal{S} = \{y_n\}$ with $y_n \in D$ for all $1 \le n \le N$ as the quadrature points and consider the following approximation,

$$\sum_{n=1}^{N} w_n |\mathfrak{R}_\theta(y_n)|^p = \sum_{n=1}^{N} w_n \, |\mathcal{D}(u_\theta(y_n)) - f(y_n)|^p \approx \|\mathfrak{R}_\theta\|^p_{L^p(D)}, \tag{4.6}$$

where the w_n are quadrature weights. We can then define a loss function $\mathcal{J}$ as,

$$\mathcal{J}(\theta, \mathcal{S}) = \sum_{n=1}^{N} w_n |\mathfrak{R}_\theta(y_n)|^p + \lambda_{reg} \|\theta\|_q^q. \tag{4.7}$$

Here, the last term is a *regularization* (penalization) term. A popular choice for q is either $q = 1$ (to induce sparsity) or $q = 2$. The parameter $0 \leq \lambda_{reg} \ll 1$ balances the regularization term with the residual loss (4.6). We then approximate the exact minimizer θ^* as in (4.5) by

$$\theta^*(\mathcal{S}) = \arg\min_{\theta \in \Theta} \mathcal{J}(\theta, \mathcal{S}). \tag{4.8}$$

As in the previous chapters, the above minimization problem amounts to finding a minimum of a possibly non-convex function over a subset of $\mathbb{R}^M$ for possibly very large M. Following standard practice in machine learning, one solves this minimization problem approximately by either (first-order) stochastic gradient descent methods such as ADAM [36] or even higher-order optimization methods such as LBFGS [19].

For notational simplicity, we denote the (approximate, local) minimum in (4.8) as θ^*. The underlying deep neural network $u^* = u_{\theta^*}$ will be our physics-informed neural network (PINN) approximation for the solution u of the PDE (4.1). The proposed algorithm for computing this PINN is given below.

PINN: training of a physics-informed neural network

Goal: Find PINN $u^* = u_{\theta^*}$ for approximating the PDE (4.1).

Step 1: Choose the training set $\mathcal{S} = \{y_n\}$ for $y_n \in D$, for all $1 \leq n \leq N$ such that $\{y_n\}$ are numerical quadrature points. For high input dimensions, the use of (quasi-) random points is advised.

Step 2: For an initial value of the weight vector $\overline{\theta} \in \Theta$, evaluate the neural network $u_{\overline{\theta}}$, the PDE residual (4.3), the loss function (4.8) and its gradients to initialize the underlying optimization algorithm.

Step 3: Run the optimization algorithm until an approximate local minimum θ^* of (4.8) is reached. The map $u^* = u_{\theta^*}$ is the desired PINN for approximating the solution u of the PDE (4.1).

Now recall that the standard practice in machine learning is to approximate the solution u of (4.1) from training data $\mathcal{S}_d = \{(z_j, u(z_j))\}_{j=1}^{N_d}$, for $z_j \in D$ and $1 \leq j \leq N_d$, then one would like to minimize the so-called *data loss*:

$$\mathcal{J}_d(\theta, \mathcal{S}_d) := \frac{1}{N_d} \sum_{j=1}^{N_d} |u(z_j) - u_\theta(z_j)|^p. \tag{4.9}$$

Comparing the loss functions (4.6) and (4.9) reveals the essence of PINNs: for PINNs, one does not necessarily need any training data for the solution u of (4.1), only the residual (4.3) needs to be evaluated, for which knowledge of the differential

operator and inputs for (4.1) suffices. Here, we distinguish initial and boundary data, which are implicitly included into the formulation of the PDE (4.1) and which are necessary for any numerical solution of (4.1), from other types of data, for instance values of the solution u, from the interior of the domain D. Thus, the PINN in this formulation, which is closer in spirit to the original proposal of [38], can be purely thought of as a numerical method for the PDE (4.1). In this form, one can consider PINNs as an example of *unsupervised learning* [47], as no explicit data is necessary.

On the other hand, the authors of [63] and subsequent papers, have added further flexibility to PINNs by also including training data by augmenting the loss function (4.8) with the data loss (4.9) and seeking neural networks with tuning parameters defined by,

$$\theta^*(\mathcal{S}, \mathcal{S}_d) = \arg\min_{\theta\in\Theta} \left(\mathcal{J}_d(\theta, \mathcal{S}_d) + \lambda \mathcal{J}(\theta, \mathcal{S})\right), \tag{4.10}$$

with an additional hyperparameter λ that balances the data loss with the residual. This paradigm has been very successful for inverse problems, where boundary and initial data may not be known [63, 64]. However, we will focus on the forward problem and only consider PINNs that minimize the residual-based loss function 4.8, without needing additional data.

Remark 4.1 The PINN training procedure requires that the residual (4.3) for the neural network u_θ can be evaluated pointwise for every training step. Hence, one needs the neural network to be sufficiently regular. Depending on the order of derivatives in $\mathcal{D}$ of (4.1), this can be ensured by requiring sufficient smoothness for the activation function. Hence, the ReLU activation function, which is only Lipschitz continuous, might not be admissible in this framework. On the other hand, smooth activation functions such as logistic and tanh are always admissible.

Remark 4.2 Rooted in the success of the initial definitions PINNs, many adaptations have been proposed. Some variants are inspired by seasoned techniques from numerical mathematics, such as domain decomposition and Galerkin methods based on the variational formulation of a PDE. These techniques were the inspiration for respectively *extended PINNs* (XPINNs) [33] and *hp-variational PINNs* (hp-VPINNs) [35]. Another method using the variational formulation of PDEs is the Deep Ritz method [73]. Other PINN variants are adapted to a specific subclass of PDEs, like *conservative PINNs* (cPINNs) [34] for conservation laws, *nonlocal PINNs* (nPINNs) [59] for PDEs involving a nonlocal universal Laplacian operator and *fractional PINNs* (fPINNs) [60] for fractional advection-diffusion equations. A a significant speed-up for long-time integration of PDEs can be achieved using *parareal PINNs* (PPINNs) [51]. Finally, *Bayesian PINNs* (B-PINNs) [71] make use of both physical laws and scattered noisy measurements to provide predictions and quantify the aleatoric uncertainty arising from the noisy data in the Bayesian framework.

4.2 Theory for PINNs

The aim of this section is to develop error estimates for PINNs, similar to those of Section 2.4 and Chapter 2. The relevant concept of error is the error emanating from approximating the solution u of (4.1) by the PINN u^*,

$$\mathcal{E}(\theta) := \|u - u_\theta\|_X, \quad \mathcal{E}^* := \mathcal{E}(\theta^*(\mathcal{S})). \tag{4.11}$$

Clearly, the error depends on the chosen training set $\mathcal{S}$ and the trained neural network with tuning parameters $\theta^*(\mathcal{S})$. However, we will suppress this dependence due to notational convenience. Note that there is no computation of $\mathcal{E}$ during the training process. On the other hand, we monitor the so-called *training error* given by,

$$\mathcal{E}_T(\theta, \mathcal{S}) := \left(\sum_{n=1}^{N} w_n |\mathfrak{R}_\theta(y_n)|^p \right)^{\frac{1}{p}}, \quad \mathcal{E}_T^* := \mathcal{E}_T(\theta^*(\mathcal{S}), \mathcal{S}). \tag{4.12}$$

Hence, the training error $\mathcal{E}_T^*$ can be readily computed, after training has been completed, from the loss function (4.8). Similar to the definitions in the previous chapters, we investigate how well the error of the PINN in the training data generalizes to the whole domain through the *generalization* error $\mathcal{E}_G$,

$$\mathcal{E}_G(\theta) := \|\mathfrak{R}_\theta\|_{L^p}, \quad \mathcal{E}_G^* := \mathcal{E}_G(\theta^*(\mathcal{S}), \mathcal{S}). \tag{4.13}$$

Given these definitions, some fundamental theoretical questions arise immediately:

Q1. **Existence**: Given an error tolerance $\epsilon > 0$, does there exist a PINN $\widehat{u} = u_{\widehat{\theta}}$ for some $\widehat{\theta} \in \Theta$ such that the corresponding generalization error $\mathcal{E}_G(\widehat{\theta})$ is small i.e., $\mathcal{E}_G(\widehat{\theta}) < \epsilon$?

Q2. **Stability**: Given that a PINN $\widehat{u}$ has a small generalization error, will the corresponding total error $\mathcal{E}(\widehat{\theta})$ be small as well? In other words, is there a function $\delta : \mathbb{R}_+ \to \mathbb{R}_+$ with $\lim_{\epsilon \to 0} \delta(\epsilon) = 0$ such that $\mathcal{E}_G(\widehat{\theta}) < \epsilon$ implies that $\mathcal{E}(\widehat{\theta}) < \epsilon$ for any $\epsilon > 0$?

Q3. **Generalization**: Given a small training error $\mathcal{E}_T^*$ and a sufficiently large training set $\mathcal{S}$, will the corresponding generalization error also be small?

The above questions are of fundamental importance as affirmative answers to them certify that, in principle, there exists a PINN, corresponding to the parameter $\widehat{\theta}$, such that the resulting PDE residual (4.13) is small (by Q1), and consequently also the overall error (4.11) in approximating the solution of the PDE (by Q2). Moreover, the smallness of the generalization error $\mathcal{E}_G(\widehat{\theta})$ (4.13) can imply that the training error $\mathcal{E}_G(\widehat{\theta})$ (4.12), which is an approximation of the generalization error, is also small. Hence, in principle, the (global) minimization of the optimization problem (4.8) should result in a proportionately small training error. However, the optimization problem (4.8) involves the minimization of a non-convex, very high-dimensional objective function. Hence, it is unclear if a global minimum is attained

by a gradient-descent algorithm. An affirmative answer to question Q3, together with question Q2, will imply that the trained PINN u_{θ^*} is an accurate approximation of the solution u of the underlying PDE (4.1).

4.2.1 Existence of PINNs

The universal approximation theorem for neural networks (Theorem 2.1) and the constructive proofs from Section 2.2 guarantee that any measurable function can be approximated by a neural network in supremum-norm, where the convergence rate depends on the properties of the function (e.g. input dimension, regularity). They however do not imply an affirmative answer to question Q1 i.e., PINNs might not even be guaranteed to *exist*. A neural network that approximates a function well in supremum-norm might be highly oscillatory, such that the derivatives of the network and that of the function are very different, giving rise to a large PDE residual. Hence, we require results on the existence of neural networks that approximate functions in a *stronger norm* than the supremum-norm. In particular, the norm should quantify how well the derivatives of the neural network approximate those of the function.

For solutions of PDEs, a very natural norm that satisfies this criterion is the *Sobolev norm*. For $s \in \mathbb{N}$ and $p \in [1, \infty]$, the Sobolev space $W^{s,q}(D; \mathbb{R}^m)$ is the space of all functions $f : D \to \mathbb{R}^m$ for which f, as well as the (weak) derivatives of f up to order s belong to $L^q(D; \mathbb{R}^m)$. Readers unfamiliar with Sobolev spaces can replace the space $W^{s,\infty}(D; \mathbb{R}^m)$ with the space of s times continuously differentiable functions $C^s(D; \mathbb{R}^m)$ or can find more information in Appendix A.1.

Fortunately, it turns out that Theorem 2.2 can be generalized from C^0-norm to $W^{k,\infty}$-norm for any $k \in \mathbb{N}_0$. We state a result that is tailored to tanh neural networks [14, Theorem 5.1], but more general results are also available [26].

Theorem 4.1 *For every $N \in \mathbb{N}$ and every $f \in W^{s,\infty}([0,1]^d)$, there exists a tanh neural network $\widehat{f}$ with 2 hidden layers of width N^d such that for every $0 \leq k < s$ it holds that,*

$$\left\| f - \widehat{f} \right\|_{W^{k,\infty}} \leq C(\ln(cN))^k N^{-s+k}, \tag{4.14}$$

where $c, C > 0$ are independent of N and explicitly known.

Using this theorem, it is easy to prove that the PDE residual can be made arbitrary small by a neural network. In addition, an estimate on the size of the neural network is provided. We will demonstrate this approach in detail for the semilinear heat equation (4.27) in Section 4.3.

4.2.2 Stability of PINNs

Next, we want to investigate whether a small PDE residual implies that the total error (4.11) will be small as well (Question Q2). Such a stability bound can be formulated

as the requirement that for any $u, v \in X^*$, the differential operator $\mathcal{D}$ satisfies

$$\|u - v\|_X \leq C_{pde} \|\mathcal{D}(u) - \mathcal{D}(v)\|_Y, \tag{4.15}$$

where the constant $C_{pde} > 0$ is allowed to depend on $\|u\|_{X^*}$ and $\|v\|_{X^*}$.

As a first example of a PDE with solutions satisfying (4.15), we consider a linear differential operator $\mathcal{D} : X \mapsto Y$, i.e. $\mathcal{D}(\alpha u + \beta v) = \alpha \mathcal{D}(u) + \beta \mathcal{D}(v)$, for any $\alpha, \beta \in \mathbb{R}$. For simplicity, let $X^* = X$ and $Y^* = Y$. By the assumptions on the existence and uniqueness of the underlying linear PDE (4.1), there exists an *inverse* operator $\mathcal{D}^{-1} : Y \mapsto X$. Note that the assumption (4.15) is satisfied if the inverse is bounded i.e, $\|\mathcal{D}^{-1}\| \leq C < +\infty$, with respect to the natural norm on linear operators from Y to X. Thus, the assumption (4.15) on stability boils down to the boundedness of the inverse operator for linear PDEs. Many well-known linear PDEs possess such bounded inverses [12].

As a second example, we will consider a nonlinear PDE (4.1), but with a well-defined linearization i.e, there exists an operator $\overline{\mathcal{D}} : X^* \mapsto Y^*$, such that

$$\mathcal{D}(u) - \mathcal{D}(v) = \overline{\mathcal{D}}_{(u,v)} (u - v), \quad \forall u, v \in X^*. \tag{4.16}$$

Again for simplicity, we will assume that $X^* = X$ and $Y^* = Y$. We further assume that the inverse of $\overline{\mathcal{D}}$ exists and is bounded in the following manner,

$$\left\| \left(\overline{\mathcal{D}}_{(u,v)} \right)^{-1} \right\| \leq C \left(\|u\|_X, \|v\|_X \right) < +\infty, \quad \forall u, v \in X, \tag{4.17}$$

with the norm of $\overline{\mathcal{D}}^{-1}$ being an operator norm, induced by linear operators from Y to X. Then a straightforward calculation shows that (4.17) suffices to establish the stability bound (4.15).

We will provide an explicit proof that the stability bound (4.15) is satisfied for the semilinear heat equation (4.27) in Section 4.3. Similar explicit bounds are available for viscous scalar conservation laws [52], the Navier-Stokes equations [13] and linear Kolmogorov equations [16].

4.2.3 Generalization of PINNs

We will provide an affirmative answer to question Q3 by showing how the generalization error $\mathcal{E}_G^*$ (4.13) can be estimated in terms of the training error $\mathcal{E}_T^*$ (4.12) using standard results on numerical integration, which we recall now. For a mapping $g : D \mapsto \mathbb{R}^m$, $D \subset \mathbb{R}^d$, such that $g \in L^1(D)$, we are interested in approximating the integral,

$$\overline{g} := \int_D g(y) dy,$$

with dy denoting the Lebesgue measure on D. In order to approximate the above integral by a quadrature rule, we need the quadrature points $y_i \in D$ for $1 \leq i \leq N$, for some $N \in \mathbb{N}$ as well as weights w_i, with $w_i \in \mathbb{R}_+$. Then a quadrature is defined by,

$$\overline{g}_N := \sum_{i=1}^{N} w_i g(y_i), \tag{4.18}$$

for weights w_i and quadrature points y_i. We further assume that the quadrature error is bounded as,

$$\left|\overline{g} - \overline{g}_N\right| \leq C_{quad} N^{-\alpha}, \tag{4.19}$$

for some $\alpha > 0$ and where C_{quad} depends on g and its properties. As long as the domain D is in reasonably low dimension, i.e. $\overline{d} \leq 4$, we can use standard (composite) Gauss quadrature rules on an underlying grid. In this case, the quadrature points and weights depend on the order of the quadrature rule [69] and the rate α depends on the regularity of the underlying integrand. On the other hand, these grid based quadrature rules are not suitable for domains in high dimensions. For moderately high dimensions, i.e. $4 \leq \overline{d} \leq 20$, we can use low-discrepancy sequences, such as the Sobol and Halton sequences, as quadrature points. As long as the integrand g is of bounded Hardy-Krause variation, the error in (4.19) converges at a rate $(\log(N))^{\overline{d}} N^{-1}$, as was shown in Section 3.2.

It is clear that the error bound for a quadrature rule immediately proves that the generalization error can be bounded in terms of the training error and the training set size, thereby answering question Q3, at least for deterministic quadrature rules. We can now combine this generalization result with the stability bounds from Section 4.2.2 to prove an upper bound on the total error of the trained pinn $\mathcal{E}^*$ (4.11) [52, Theorem 2.6].

Theorem 4.2 *Let $u \in X^*$ be the unique solution of the PDE* (4.1) *and assume that the stability hypothesis* (4.15) *holds. Let $u^* \in X^*$ be the PINN generated based on the training set $\mathcal{S}$ of quadrature points corresponding to the quadrature rule* (4.18)*. Further assume that the residual $\mathfrak{R}_{\theta^*}$, defined in* (4.3)*, be such that $\mathfrak{R}_{\theta^*} \in Y = L^p(D)$ and the quadrature error satisfies* (4.19)*. Then the following estimate on the error holds,*

$$\mathcal{E}^* \leq C_{pde}\mathcal{E}_T^* + C_{pde} C_{quad}^{\frac{1}{p}} N^{-\frac{\alpha}{p}}, \tag{4.20}$$

with constants C_{pde} and C_{quad} stemming from (4.15) *and* (4.19)*, respectively.*

Proof In the following, we denote by $\mathfrak{R} = \mathfrak{R}_{\theta^*}$ the residual (4.3), corresponding to the trained neural network u^*. As u solves the PDE (4.1) and $\mathfrak{R}$ is defined by (4.3), we easily see that,

$$\mathfrak{R} = \mathcal{D}(u^*) - \mathcal{D}(u). \tag{4.21}$$

Hence, we can directly apply the stability bound (4.15) to yield,

$$\mathcal{E}^* = \|u - u^*\|_X \leq C_{pde}\|\mathcal{D}(u^*) - \mathcal{D}(u)\|_Y = C_{pde}\|\mathfrak{R}\|_Y. \tag{4.22}$$

By the fact that $Y = L^p(D)$, the definition of the training error (4.12) and the quadrature rule (4.18), we see that,

$$\|\mathfrak{R}\|_Y^p \approx \left(\sum_{n=1}^{N} w_n |\mathfrak{R}_{\theta^*}|^p \right) = (\mathcal{E}_T^*)^p . \tag{4.23}$$

Hence, the training error is a quadrature for the residual (4.3) and the resulting quadrature error, given by (4.19) translates to,

$$\|\mathfrak{R}\|_Y^p \leq (\mathcal{E}_T^*)^p + C_{quad} N^{-\alpha}. \tag{4.24}$$

Therefore, we find that

$$(\mathcal{E}^*)^p \leq C_{pde}^p \left((\mathcal{E}_T^*)^p + C_{quad} N^{-\alpha} \right), \tag{4.25}$$

from which the desired estimate (4.20) follows. □

The previously described approach only makes sense for low to moderately high dimensions ($d < 20$) since the convergence rate of numerical quadrature rules with deterministic quadrature decreases with increasing d. For problems in very high dimensions, $d \gg 20$, Monte-Carlo quadrature is the numerical integration method of choice. In this case, the quadrature points are randomly chosen, independent and identically distributed (with respect to a scaled Lebesgue measure). Since in this case the trained network is correlated with all training points, one must be careful when one calculates the convergence rate. Similar to Section 2.4, we can only bound the so-called cumulative error. The cumulative total, training and generalization error are then denoted by $\overline{\mathcal{E}}$, $\overline{\mathcal{E}}_T$ and $\overline{\mathcal{E}}_G$, respectively, and they are defined similarly to (2.83). Using these quantities, it is possible to prove error bounds for PINNs with a smooth activation function in the spirit of

$$\overline{\mathcal{E}}^* \leq \overline{\mathcal{E}}_T^* + C N^{-\frac{1}{(2+\gamma)p}}, \tag{4.26}$$

where $\gamma > 0$ can be made arbitrarily small and $C > 0$ is a constant independent of N. An exact proof can be found in [16]. Note that the convergence rate is essentially the same as that in Theorem 2.8.

4.3 Case study: semilinear heat equation

We now apply the abstract framework and estimates from the previous sections to the semilinear heat equation. We will work out all definitions and error bounds in detail.

We first recall the set-up of the semilinear heat equation. Let $D \subset \mathbb{R}^d$ be an open connected bounded set with a C^1 boundary ∂D. The semi-linear parabolic equation is then given by,

$$\begin{cases} u_t = \Delta u + f(u), & \forall x \in D,\ t \in (0,T), \\ u(x,0) = \overline{u}(x), & \forall x \in D, \\ u(x,t) = 0, & \forall x \in \partial D,\ t \in (0,T). \end{cases} \tag{4.27}$$

Here, $u_0 \in C^k(D)$, $k \geq 2$, is the initial data, $u \in C^k([0,T] \times D)$ is the classical solution and $f : \mathbb{R} \times \mathbb{R}$ is the non-linear source (reaction) term. We assume that the non-linearity is globally Lipschitz i.e, there exists a constant $C_f > 0$ such that

$$|f(v) - f(w)| \leq C_f |v - w|, \quad \forall v, w \in \mathbb{R}. \tag{4.28}$$

In particular, the homogeneous linear heat equation with $f(u) \equiv 0$ and the linear source term $f(u) = \alpha u$ are examples of (4.27). Semilinear heat equations with globally Lipschitz nonlinearities arise in several models in biology and finance [5]. The existence, uniqueness and regularity of the semi-linear parabolic equations with Lipschitz non-linearities such as (4.27) can be found in classical textbooks such as [20].

4.3.1 Training set

Let $D = D \times (0,T)$ be the space-time domain. We will choose the training set $\mathcal{S} \subset [0,T] \times \overline{D}$, based on suitable quadrature points. We have to divide the training set into the following three parts,

- Interior training points $\mathcal{S}_{int} = \{y_n\}_n$ for $1 \leq n \leq N_{int}$, with each $y_n = (x_n, t_n) \in D$. Depending on the dimension, these points are quadrature points, low-discrepancy points or random points.
- Spatial boundary training points $\mathcal{S}_{sb} = \{z_n\}_n$ for $1 \leq n \leq N_{sb}$ with each $z_n = (x,t)_n$ and each $x_n \in \partial D$. Again the points can be chosen from a grid-based quadrature rule on the boundary, as low-discrepancy sequences or randomly.
- Temporal boundary training points $\mathcal{S}_{tb} = \{x_n\}_n$, with $1 \leq n \leq N_{tb}$ and each $x_n \in D$, chosen either as grid points, low-discrepancy sequences or randomly chosen in D.

The full training set is $\mathcal{S} = \mathcal{S}_{int} \cup \mathcal{S}_{sb} \cup \mathcal{S}_{tb}$. An example for the full training set is shown in Figure 4.1.

4.3.2 Residuals

Next, we need to define appropriate residuals. For the neural network $u_\theta \in C^k([0,T] \times \overline{D})$, $\theta \in \Theta$, with continuous extensions of the derivatives to the boundaries, with a smooth activation function such as $\sigma = \tanh$ and Θ as the set of tuning parameters, we define the following residuals,

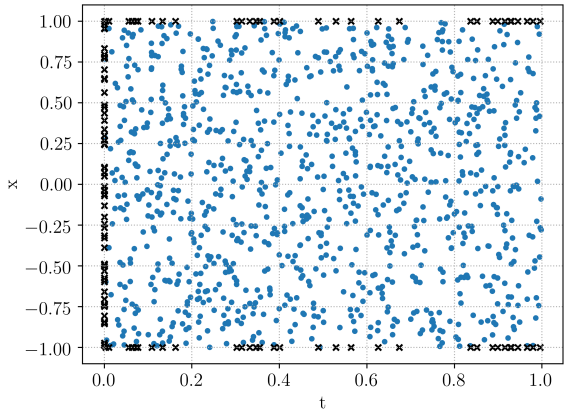


Fig. 4.1: An illustration of the training set $\mathcal{S}$ for the one-dimensional heat equation (4.27) with randomly chosen training points. Points in $\mathcal{S}_{int}$ are depicted with blue dots and those in $\mathcal{S}_{tb} \cup \mathcal{S}_{sb}$ are depicted with black crosses.

- Interior residual given by,

$$\mathfrak{R}_{int,\theta}(x,t) := \partial_t u_\theta(x,t) - \Delta u_\theta(x,t) - f(u_\theta(x,t)). \tag{4.29}$$

  Here $\Delta = \Delta_x$ is the spatial Laplacian. Note that the residual is well-defined and $\mathfrak{R}_{int,\theta} \in C^{k-2}([0,T] \times \overline{D})$ for every $\theta \in \Theta$.
- Spatial boundary residual given by,

$$\mathfrak{R}_{sb,\theta}(x,t) := u_\theta(x,t), \quad \forall x \in \partial D,\ t \in (0,T]. \tag{4.30}$$

  Given the fact that the neural network is smooth, this residual is well-defined.
- Temporal boundary residual given by,

$$\mathfrak{R}_{tb,\theta}(x) := u_\theta(x,0) - \overline{u}(x), \quad \forall x \in D. \tag{4.31}$$

  Again this quantity is well-defined and $\mathfrak{R}_{tb,\theta} \in C^k(D)$ as both the initial data and the neural network are smooth.

4.3.3 Loss function

Finally, we need a loss function to train the PINN. To this end, we generalize (4.6) to,

$$\begin{aligned}\mathcal{J}(\theta, \mathcal{S}) = &\sum_{n=1}^{N_{tb}} w_n^{tb} |\mathfrak{R}_{tb,\theta}(x_n)|^2 + \sum_{n=1}^{N_{sb}} w_n^{sb} |\mathfrak{R}_{sb,\theta}(x_n, t_n)|^2 \\ &+ \lambda \sum_{n=1}^{N_{int}} w_n^{int} |\mathfrak{R}_{int,\theta}(x_n, t_n)|^2 + \lambda_{reg} \|\theta\|_q^q.\end{aligned} \tag{4.32}$$

Here the residuals are defined by (4.31), (4.30), (4.29), the w_n^{tb} are the N_{tb} quadrature weights corresponding to the temporal boundary training points $\mathcal{S}_{tb}$, the w_n^{sb} are the N_{sb} quadrature weights corresponding to the spatial boundary training points $\mathcal{S}_{sb}$ and the w_n^{int} are the N_{int} quadrature weights corresponding to the interior training points $\mathcal{S}_{int}$. Furthermore, $\lambda > 0$ is a hyperparameter for balancing the residuals, on account of the PDE and the initial and boundary data, respectively.

The algorithm for training a PINN to approximate the semilinear parabolic equation (4.27) is now completely specified and can be run to generate the required PINN, which we denote as $u^* = u_{\theta^*}$, where $\theta^* \in \Theta$ is the (approximate) minimizer of the optimization problem, corresponding to the loss function (4.32).

4.3.4 Theoretical guarantees

We will now answer the fundamental questions Q1-Q3 to affirm that the use of PINNs for semilinear heat equations is justified. In this case, $X = L^2(D \times (0,T))$ and total error (4.11) is concretely defined as,

$$\mathcal{E} := \left(\int_0^T \int_D |u(x,t) - u^*(x,t)|^2 dx dt \right)^{\frac{1}{2}}. \tag{4.33}$$

As for the abstract PDE (4.1), we are going to estimate this error in terms of the *training error* (4.12) that we define as,

$$\mathcal{E}_T^2 := \underbrace{\sum_{n=1}^{N_{tb}} w_n^{tb} |\mathfrak{R}_{tb,\theta^*}(x_n)|^2}_{(\mathcal{E}_T^{tb})^2} + \underbrace{\sum_{n=1}^{N_{sb}} w_n^{sb} |\mathfrak{R}_{sb,\theta^*}(x_n, t_n)|^2}_{(\mathcal{E}_T^{sb})^2} + \lambda \underbrace{\sum_{n=1}^{N_{int}} w_n^{int} |\mathfrak{R}_{int,\theta^*}(x_n, t_n)|^2}_{(\mathcal{E}_T^{int})^2}. \tag{4.34}$$

Note that the training error can be readily computed *a posteriori* from the loss function (4.32).

First, we need to ensure the **existence** (Q1) of PINNs that approximate the solution u of the semilinear heat equation (4.27) well. From Theorem 4.1 we find immediately the following result. It proves that there exists a neural network that makes the PINN (interior) residual, as well as the spatial boundary and temporal boundary residual, arbitrarily small.

Corollary 4.1 *Let $u \in C^k([0,T] \times D)$ be the solution of* (4.27)*. For every $\epsilon, \gamma > 0$ there exists a tanh neural network $\widehat{u}$ with 2 hidden layers of width $O(\epsilon^{-\frac{d}{k-2-\gamma}})$ such that*

$$\begin{aligned} \mathcal{E}_G^{int} &:= \|\widehat{u}_t - \Delta\widehat{u} - f(\widehat{u})\|_{L^2(D\times[0,T])} < \epsilon, \\ \mathcal{E}_G^{sb} &:= \|u - \widehat{u}\|_{L^2(\partial D\times[0,T])} < \epsilon \quad \textit{and} \quad \mathcal{E}_G^{tb} := \|u_0 - \widehat{u}\|_{L^2(D)} < \epsilon. \end{aligned} \tag{4.35}$$

Proof Theorem 4.1 guarantees the existence of a tanh neural network $\widehat{u}$ with 2 hidden layers of width $O(\epsilon^{-\frac{d}{k-2-\gamma}})$ such that

$$\|u - \widehat{u}\|_{C^2(D\times[0,T])} < \epsilon. \tag{4.36}$$

From this and (4.28), one can calculate

$$\begin{aligned} \|\widehat{u}_t - \Delta\widehat{u} - f(\widehat{u})\|_{C^0(D\times[0,T])} &= \|\widehat{u}_t - \Delta\widehat{u} - f(\widehat{u}) - (u_t - \Delta u - f(u))\|_{C^0} \\ &\le \|\widehat{u}_t - u_t\|_{C^0} + \|\Delta\widehat{u} - \Delta u\|_{C^0} + \|f(\widehat{u}) - f(u)\|_{C^0} \\ &\le \|\widehat{u} - u\|_{C^1} + d\|\widehat{u} - u\|_{C^2} + C_f\|\widehat{u} - u\|_{C^0} \\ &\le (1 + d + C_f)\|\widehat{u} - u\|_{C^2} \\ &< (1 + d + C_f)\epsilon. \end{aligned} \tag{4.37}$$

Moreover, it follows from a multiplicative trace inequality that there exists a constant $c > 0$ independent of ϵ such that

$$\max\{\|u - \widehat{u}\|_{C^0(\partial D\times[0,T])}, \|u_0 - \widehat{u}\|_{C^0(D)}\} \le c\|\widehat{u} - u\|_{C^1(D\times[0,T])} < c\epsilon. \tag{4.38}$$

This then proves the statement. □

Second, we need to address the **stability** (Q2) of the semilinear heat equation i.e., whether a small generalization error $\mathcal{E}_G$ (as in Corollary 4.1) implies a small total error $\mathcal{E}$ (4.33). The following theorem [52, Theorem 3.1] ensures that this is indeed the case. A generalization to linear Kolmogorov equations (2.74) (including the Black-Scholes model) can be found in [16, Theorem 3.7].

Theorem 4.3 *Let $u \in C^k(\overline{D} \times [0,T])$ be the unique classical solution of the semilinear parabolic equation* (4.27) *with the source f satisfying* (4.28)*. Let $u^* = u_{\theta^*}$ be a PINN generated using loss function* (4.32)*. Then the total error* (4.33) *can be estimated as,*

$$\mathcal{E} \le C_1\left(\mathcal{E}_G^{tb} + \mathcal{E}_G^{int} + C_2(\mathcal{E}_G^{sb})^{\frac{1}{2}}\right), \tag{4.39}$$

with constants given by,

$$\begin{aligned} C_1 &= \sqrt{T + (1 + 2C_f)T^2 e^{(1+2C_f)T}}, \quad C_2 = \sqrt{C_{\partial D}(u, u^*)T^{\frac{1}{2}}}, \\ C_{\partial D} &= |\partial D|^{\frac{1}{2}}\left(\|u\|_{C^1([0,T]\times\partial D)} + \|u^*\|_{C^1([0,T]\times\partial D)}\right). \end{aligned} \tag{4.40}$$

Proof By the definitions of the residuals (4.29), (4.30), (4.31) and the underlying PDE (4.27), we can readily verify that the error $\widehat{u} : u^* - u$ satisfies the following (forced) parabolic equation,

$$\begin{aligned} \widehat{u}_t &= \Delta\widehat{u} + f(u^*) - f(u) + \mathfrak{R}_{int}, \quad \forall x \in D \; t \in (0,T), \\ \widehat{u}(x,0) &= \mathfrak{R}_{tb}(x), \quad \forall x \in D, \\ u(x,t) &= \mathfrak{R}_{sb}(x,t), \quad \forall x \in \partial D, \; t \in (0,T). \end{aligned} \tag{4.41}$$

Here, we have denoted $\mathfrak{R}_{int} = \mathfrak{R}_{int,\theta^*}$ for notational convenience and analogously for the residuals $\mathfrak{R}_{tb}, \mathfrak{R}_{sb}$.

Multiplying both sides of the PDE (4.41) with $\widehat{u}$, integrating over the domain and integrating by parts, denoting $\widehat{n}$ as the unit outward normal, yields,

$$\begin{aligned} \frac{1}{2}\frac{d}{dt}\int_D |\widehat{u}(x,t)|^2 dx &= -\int_D |\nabla\widehat{u}|^2 dx + \int_{\partial D} \mathfrak{R}_{sb}(x,t)(\nabla\widehat{u}\cdot\widehat{n}) ds(x) \\ &\quad + \int_D \widehat{u}(f(u^*) - f(u)) dx + \int_D \mathfrak{R}_{int}\widehat{u} dx. \\ &\le \int_D |\widehat{u}||f(u^*) - f(u)| dx + \frac{1}{2}\int_D \widehat{u}(x,t)^2 dx + \frac{1}{2}\int_D |\mathfrak{R}_{int}|^2 dx \\ &\quad + \underbrace{|\partial D|^{\frac{1}{2}}\left(\|u\|_{C^1([0,T]\times\partial D)} + \|u^*\|_{C^1([0,T]\times\partial D)}\right)}_{C_{\partial D}(u,u^*)} \left(\int_{\partial D} |\mathfrak{R}_{sb}(x,t)|^2 ds(x)\right)^{\frac{1}{2}} \\ &\le (C_f + \frac{1}{2})\int_D |\widehat{u}(x,t)|^2 dx \\ &\quad + \frac{1}{2}\int_D |\mathfrak{R}_{int}|^2 dx + C_{\partial D}(u,u^*)\left(\int_{\partial D} |\mathfrak{R}_{sb}(x,t)|^2 ds(x)\right)^{\frac{1}{2}}, \end{aligned}$$

where the last inequality holds by (4.28). Integrating the above inequality over $[0,\overline{T}]$ for any $\overline{T} \le T$ and the definition (4.31) together with Cauchy-Schwarz inequality, we obtain,

$$\begin{aligned} \int_D |\widehat{u}(x,\overline{T})|^2 dx \le \int_D |\mathfrak{R}_{tb}(x)|^2 dx + (1 + C_f)\int_0^{\overline{T}}\int_D |\widehat{u}(x,t)|^2 dx dt + \int_0^T \int_D |\mathfrak{R}_{int}|^2 dx dt \\ + C_{\partial D}(u,u^*) T^{\frac{1}{2}} \left(\int_0^T \int_{\partial D} |\mathfrak{R}_{sb}(x,t)|^2 ds(x) dt\right)^{\frac{1}{2}}. \end{aligned}$$

Applying the integral form of the Grönwall's inequality to the above, we obtain,

$$\int_D |\widehat{u}(x,\overline{T})|^2 dx \leq \left(1 + (1+2C_f)Te^{(1+2C_f)T}\right) \times \ldots$$
$$\ldots \left(\int_D |\mathfrak{R}_{tb}(x)|^2 dx + \int_0^T \int_D |\mathfrak{R}_{int}|^2 dxdt + C_{\partial D}(u,u^*)T^{\frac{1}{2}} \left(\int_0^T \int_{\partial D} |\mathfrak{R}_{sb}(x,t)|^2 ds(x)dt \right)^{\frac{1}{2}} \right).$$

Integrating over $\overline{T} \in [0,T]$ yields,

$$\mathcal{E}^2 = \int_0^T \int_D |\widehat{u}(x,\overline{T})|^2 dxdt \leq \left(T + (1+2C_f)T^2e^{(1+2C_f)T}\right) \times \ldots$$
$$\ldots \left(\int_D |\mathfrak{R}_{tb}(x)|^2 dx + \int_0^T \int_D |\mathfrak{R}_{int}|^2 dxdt + C_{\partial D}(u,u^*)T^{\frac{1}{2}} \left(\int_0^T \int_{\partial D} |\mathfrak{R}_{sb}(x,t)|^2 ds(x)dt \right)^{\frac{1}{2}} \right). \tag{4.42}$$

By the definitions of different components of the generalization error (4.35) yields the desired inequality (4.39). □

Finally, we need to address whether a PINN that performs well on a training set will **generalize** (Q3) well to the full domain. In other words, we are interested in finding an upper bound on the generalization error in terms of the training error (4.34). Using the stability result from Theorem 4.3 this will also imply an upper bound on the total error. For simplicity, we only discuss the case where the spatial dimension d is low. In this setting, deterministic quadrature points can be used to create the training set.

and applying the estimates (4.43), (4.44), (4.45) on the quadrature error

We will make the following assumptions on the quadrature error, analogous to (4.15). For any function $g \in C^k(D)$, the quadrature rule corresponding to quadrature weights w_n^{tb} at points $x_n \in \mathcal{S}_{tb}$, with $1 \leq n \leq N_{tb}$, satisfies

$$\left| \int_D g(x)dx - \sum_{n=1}^{N_{tb}} w_n^{tb} g(x_n) \right| \leq C_{quad}^{tb}(\|g\|_{C^k}) N_{tb}^{-\alpha_{tb}}. \tag{4.43}$$

For any function $g \in C^k(\partial D \times [0,T])$, the quadrature rule corresponding to quadrature weights w_n^{sb} at points $(x_n,t_n) \in \mathcal{S}_{sb}$, with $1 \leq n \leq N_{sb}$, satisfies

$$\left| \int_0^T \int_{\partial D} g(x,t)ds(x)dt - \sum_{n=1}^{N_{sb}} w_n^{sb} g(x_n,t_n) \right| \leq C_{quad}^{sb}(\|g\|_{C^k}) N_{sb}^{-\alpha_{sb}}. \tag{4.44}$$

Finally, for any function $g \in C^\ell(D \times [0,T])$, the quadrature rule corresponding to quadrature weights w_n^{int} at points $(x_n,t_n) \in \mathcal{S}_{int}$, with $1 \leq n \leq N_{int}$, satisfies

$$\left| \int_0^T \int_D g(x,t)dxdt - \sum_{n=1}^{N_{int}} w_n^{int} g(x_n,t_n) \right| \leq C_{quad}^{int}(\|g\|_{C^\ell}) N_{int}^{-\alpha_{int}}. \tag{4.45}$$

Note that in the above $\alpha_{int}, \alpha_{sb}, \alpha_{tb} > 0$ are the convergence rates of quadrature rules that can be of different order.

Combining these results on the accuracy of the chosen quadrature rules with the stability result of Theorem 4.3 readily provides the following upper bound on the total error of the PINN (4.33) in terms of the computable training errors and the sizes of the training sets.

Corollary 4.2 *Let $u \in C^k(\overline{D} \times [0,T])$ be the unique classical solution of the semi-linear parabolic equation* (4.27) *with the source f satisfying* (4.28)*. Let $u^* = u_{\theta^*}$ be a PINN generated using loss function* (4.8)*,* (4.32)*. Then the PINN error* (4.33) *can be estimated as,*

$$\mathcal{E} \leq C_1 \left(\mathcal{E}_T^{tb} + \mathcal{E}_T^{int} + C_2 (\mathcal{E}_T^{sb})^{\frac{1}{2}} + (C_{quad}^{tb})^{\frac{1}{2}} N_{tb}^{-\frac{\alpha_{tb}}{2}} + (C_{quad}^{int})^{\frac{1}{2}} N_{int}^{-\frac{\alpha_{int}}{2}} + C_2 (C_{quad}^{sb})^{\frac{1}{4}} N_{sb}^{-\frac{\alpha_{sb}}{4}} \right), \tag{4.46}$$

where C_1 and C_2 are defined as in (4.40) *and $C_{quad}^{tb} = C_{quad}^{tb}(\|\mathfrak{R}_{tb,\theta^*}\|_{C^k})$, $C_{quad}^{sb} = C_{quad}^{tb}(\|\mathfrak{R}_{sb,\theta^*}\|_{C^k})$ and $C_{quad}^{int} = C_{quad}^{int}(\|\mathfrak{R}_{int,\theta^*}\|_{C^{k-2}})$ are the constants defined by the quadrature errors, as in* (4.43)*,* (4.44) *and* (4.45)*, respectively.*

The estimate (4.46) bounds the error in terms of each component of the training error and the quadrature errors. Clearly, each component of the training error can be computed from the loss function (4.32) once the training has been completed. As long as the PINN is well-trained, i.e. each component of the training error is small, the bound (4.46) implies that the error will be small for large enough number of training points. The estimate is not by any means sharp as triangle inequalities and the Grönwall's inequality are used. Nevertheless, it provides interesting information. For instance, the error due to the boundary residual has a bigger weight in (4.46), relative to the interior and initial residuals. This is consistent with the observations of [38] and can also be seen in the recent papers such as [67] and suggests that the loss function (4.32) could be modified such that the boundary residual is penalized more.

In addition to the training errors, which could depend on the underlying PDE solution, the estimate (4.46) shows explicit dependence on the underlying solution through the constant $C_{\partial D}$ (4.40), which is based only on the value of the underlying solution on the boundary. Similarly, the dependence on dimension is only seen through the quadrature error.

4.4 PINNs for forward problems

We close this chapter by discussing some applications of PINNs, forward and inverse problems in particular. We give examples where PINNs outperform classical numerical methods, but also highlight challenging settings where the performance of PINNs is not satisfactory yet. All of the mentioned results are based on the very recent papers mentioned in the beginning of the chapter.

In scientific computing, one is often interested in calculating the solution to or an observable of a *known* (partial) differential equation. As all the parameters are known, this is called a *forward* problem, as opposed to the inverse problems that we will discuss later on.

4.4.1 High-dimensional problems

A key drawback of many classical numerical methods is that they use dimension-dependent grids, where the number of grid points grow exponentially with the dimension. As a consequence, such methods are not suitable for high-dimensional problems. The PINN framework, however, does not rely on a grid if random training points are used. In principle, PINNs might therefore overcome the curse of dimensionality. We present some experimental examples that this is indeed the case.

For the **high-dimensional heat equation**, experiments show that a PINN of depth 4 and width 20 can achieve a (generalization) error of 2.6% for dimension $d = 100$ [52]. Moreover, the generalization error only increases sublinearly as a function of the input dimension, as can be seen in Table 4.1. Hence, there is no curse of dimensionality in this setting.

Dimension	1	5	10	20	50	100
Training error	0.00003	0.0002	0.0003	0.006	0.006	0.004
Generalization error	0.0035%	0.016%	0.03%	0.79%	1.5%	2.6%

Table 4.1: Training error and generalization error of PINNs for the high-dimensional heat equation. The number of training points and the network architecture is the same for all dimensions. Source: [52, Table 2].

Many mathematical models in finance are high-dimensional as well, well-known examples being the **Black-Scholes model** and the **Heston model**. These models are used in option pricing and the input dimension is equal to the number of assets. Experiments show that PINNs overcome the curse of dimensionality also in this setting [55].

For **linear Kolmogorov equations**, a class of PDEs that include the heat equation and the Black-Scholes equation, it can also be rigourously proven that the curse of dimensionality is indeed overcome [16]. More specifically, the network size needed to reach a certain approximation error does not increase exponentially with the input dimension, nor does the number of needed training samples to obtain a certain generalization error.

Another important example is given by the **radiative transfer equation**, which is used in climate models, astrophysics, models for nuclear processes and also electrical engineering. The radiative transfer equation is $(2d + 1)$-dimensional integro-differential PDE for the radiative intensity u, which is a function of time t, space x (d dimensions), angle ω ($d - 1$ dimensions) and frequency ν of the radiation,

$$\frac{1}{c}u_t + \omega \cdot \nabla_x u + (k(x,\nu) + \sigma(x,\nu)u \tag{4.47}$$

$$- \frac{\sigma(x,\nu)}{s_d} \int_\Lambda \int_S \Phi(\omega,\omega',\nu,\nu')u d\omega' d\nu' = f(t,x,\omega,\nu). \tag{4.48}$$

From left to right, the time evolution of u is controlled by a transport term, a source term that models local absorption and scattering, and a global scattering term in the form of the integral of the product of u and a scattering kernel Φ. One the right-hand side there is a source term that models emission. In an opaque medium, diffusion will dominate, whereas in a transparent medium the transport term will dominate. As the radiative-transfer equation is at most 7-dimensional, it seems less challenging that the previous example. However, the non-locality, hyperbolicity and multiphysics nature complicate the problem considerably. Nevertheless, experiments show that PINNs can also perform well in this context [53], amongst others by replacing the integral with a numerical quadrature formula. As the integral is merely three-dimensional, convential numerical quadrature rules can still be used. Moreover, theoretical bounds on the generalization (cf. those in Section 4.3) were also developed.

4.4.2 PDEs with higher-order derivatives

PDEs with higher-order derivatives are often challenging to solve using classical numerical methods. On the one hand, CFL conditions will require that the time step is very small, making it very expensive to compute the numerical solution. On the other hand, finite difference approaches will quickly lead to large numerical errors, making the results less accurate. Fortunately, PINNs can solve both of these issues. As automatic differentation, and backpropagation in particular, is not based on finite differences, it is less prone to rounding errors. Moreover, PINNs are grid-independent and CFL conditions are thus not relevant.

Examples of PDEs with higher-order derivatives are **non-linear dispersive equations** that model different aspects of shallow water waves. These include the well-known Camassa-Holm type equations, the Benjamin-Ono equations, the famous Korteweg-De Vries (KdV) equation and its high-order extension, the so-called Kawahara equation, which is given by

$$u_t + uu_x + \alpha u_{xxx} - \beta u_{xxxxx} = 0, \tag{4.49}$$

for some $\alpha, \beta > 0$. All these equations are integrable and contain interesting structures such as interacting solitons in their solutions. Moreover, classical solutions and their stability have been extensively investigated for these equations, such that one can easily investigate the accuracy of deep learning approximations. Numerical experiments show that PINNs perform well in this setting [2]. This good performance is also backed up by rigourous bounds on the total error.

4.4.3 Remaining challenges

In this section, we give example of PDEs where PINNs only perform well in certain regimes and might not succeed to yield a suitable approximation in other regimes.

Our first example is given by the **Navier-Stokes equations**,

$$\begin{cases} u_t + u \cdot \nabla u + \nabla p = \nu \Delta u & \text{in } [0,T] \times \Omega, \\ \operatorname{div}(u) = 0 & \text{in } [0,T] \times \Omega, \\ u(t=0) = u_0 & \text{in } \Omega, \end{cases} \tag{4.50}$$

which model the motion of an incompressible Newtonian fluid. Here, $u : [0,T] \times \Omega \to \mathbb{R}^d$ is the fluid velocity, $p : \Omega \to \mathbb{R}$ is the pressure and $u_0 : \Omega \to \mathbb{R}^d$ is the initial fluid velocity. The viscosity is denoted by $\nu \geq 0$. When the viscosity vanishes, $\nu = 0$, the system of equations above reduces to the Euler equations. The existence and regularity of the solution to (4.50) depends on the regularity of u_0 and the dimension d [48]. First note that this is not a scalar PDE but a system of PDEs. Therefore, the PINN framework presented in Section 4.1 now has to be applied for each equation and the total loss function is obtained by summing the contributions from each equation. In this case, one distinguishes velocity residuals and the divergence residual. Similar to as for the heat equation (Theorem 4.3), one can prove an upper bound on the PINN generalization error [52],

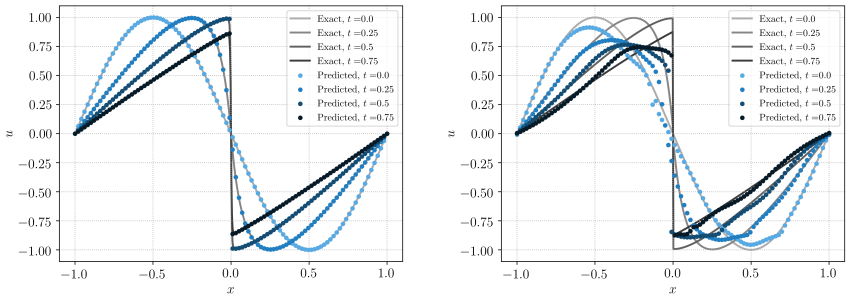
$$\mathcal{E}_G^2 \leq C e^{CT} \left((\mathcal{E}_T^{tb})^2 + (\mathcal{E}_T^{int})^2 + \mathcal{E}_T^{sb} + \mathcal{E}_T^{div} + C_{quad} (N_{tb}^{-\alpha_{tb}} + N_{int}^{-\frac{\alpha_{int}}{2}} N_{sb}^{-\frac{\alpha_{sb}}{2}}) \right), \tag{4.51}$$

where notation is similar to that in earlier sections and C is a constant that depends on $\|\nabla u\|_{L^\infty}$. For the two-dimensional Navier-Stokes (or Euler) equations with smooth initial data, it certainly holds that $C < \infty$. However, no such guarantee exists in three space dimensions, meaning that a well-trained PINN for the 3D Navier-Stokes equation need not generalize well.

A second example is the class fo **viscous scalar conservation laws**. These non-linear hyperbolic-parabolic PDEs are of the form

$$\begin{cases} u_t + \operatorname{div} f(u) = \nu \Delta u & \text{in } [0,T] \times \Omega, \\ u(t=0) = u_0 & \text{in } \Omega, \end{cases} \tag{4.52}$$

where $f(u)$ is a flux functions. For $f(u) = u^2/2$, one retrieves the viscous Burgers' equation. In this setting, a rigourous bound on the generalization error is again available [52], which is quite similar to (4.51). In particular, the constant C still depends on $\|\nabla u\|_{L^\infty}$. However, it is known that $\|\nabla u\|_{L^\infty} \sim \frac{1}{\sqrt{\nu}}$. As a result, the error might blow up near shocks, which appear when $\nu \to 0$. Experiments with the viscous Burgers' equation show that this might indeed happen, as demonstrated in Figure 4.2, leading to a generalization error of 23% for $\nu = 0$. Therefore, PINNs might not work well when the solution to the PDE has jump discontinuities.



(a) $\nu = 0.0016$, $\mathcal{E}_G = 0.012$ (b) $\nu = 0$, $\mathcal{E}_G = 0.23$

Fig. 4.2: Burgers' equation with discontinuous solution for different values of ν

4.5 PINNs for inverse problems

So far, we have already presented many examples that demonstrate that PINNs are an attractive framework for approximating solutions of PDEs. In this section, we will show that PINNs are particularly successful for approximating the so-called **inverse problems** involving PDEs, see [63, 64, 49, 9] and references therein. For such inverse problems, one does not necessarily have complete information on the inputs to the PDEs, such as initial data, boundary conditions and coefficients. Hence, the so-called *forward problem* cannot be solved uniquely. However, we have access to (noisy) data for (observables) of the underlying solution. The aim is to use this data in order to determine the unknown inputs of the PDEs and consequently its solution. PINNs are very promising in solving inverse problems efficiently on account of their ability to integrate data and PDEs. This is seen in the robustness and accuracy of the results (see [63, 64, 49] and references therein). Another very attractive feature of PINNs is simplicity of their implementation. In particular, essentially the same code (with very minor modifications) can be used to approximate both forward and inverse problems.

4.5.1 PINNs for data assimilation

One kind of inverse problem is the **unique continuation problem**, for time-dependent problems also known as **data assimilation**. We first introduce some notation. We recall the abstract PDE $\mathcal{D}(u) = f$ in $\mathbb{D}$ (4.1), where we now denote by $\mathbb{D}$ the space-time domain. Moreover, we adopt the assumptions (4.2) of the introduction of this chapter. The difference with the forward problem is that we now have incomplete information on the boundary conditions. As a result, a unique solution

to the PDE does not exist, such that training a PINN does not make sense at all. Instead, we assume that we have access to (possibly noisy) measurements of a certain observable

$$\Psi(u) = g \qquad \text{in } \mathbb{D}', \tag{4.53}$$

where $\mathbb{D}' \subset \mathbb{D}$ is the observation domain, g is the measured data and $\Psi : X^* \to Z^*$, where Z^* is some Banach space. We make the assumption that

$$\begin{aligned} &(H3): \quad \|\Psi(u)\|_{Z^*} < +\infty, \quad \forall\, u \in X^*, \text{ with } \|u\|_{X^*} < +\infty. \\ &(H4): \quad \|g\|_{Z^*} < +\infty. \end{aligned} \tag{4.54}$$

If $\mathbb{D}'$ includes the boundary of $\mathbb{D}$, meaning that one has measurements on the boundary, then it is again feasible to just use the standard PINN framework for forward problems (Section 4.1). However, this is often not possible. In heat transfer problems, for example, the maximum temperature will be reached at the boundary. As a result, it might be too hot near the boundary to place a sensor there and one will only have measurements that are far enough away from the boundary. Fortunately, one can adapt the PINN framework to retrieve an approximation to u based on f and g.

When using PINNs for a data assimilation problem, one uses slightly different residual than for forward problems. The PDE residual $\mathfrak{R}_\theta(y) = \mathcal{D}(u_\theta(y)) - f(y)$, where $y \in \mathbb{D}$, is still the same, but in addition one also defines a *data residual* $\mathfrak{R}_\theta^{\text{data}}(y) = \Psi(u_\theta(z)) - g(z)$, where $z \in \mathbb{D}'$. Using this notation, minimizing

$$\|\mathfrak{R}_\theta\|_Y^{p_Y} + \left\|\mathfrak{R}_\theta^{\text{data}}\right\|_Z \sim \int_{\mathbb{D}} |\mathfrak{R}_\theta(y)|^{p_Y}\, dy + \int_{\mathbb{D}'} \left|\mathfrak{R}_\theta^{\text{data}}(z)\right|^{p_Z} dz \tag{4.55}$$

yields a physics-informed neural network that approximates u in the entire domain $\mathbb{D}$. In practice, one of course needs to approximate the integrals in (4.55) using a suitable numerical quadrature rule.

We further assume that solutions to the inverse problem (4.1), (4.53), satisfy the following *conditional stability estimate*. Let $\widehat{X} \subset X^* \subset X = L^{p_x}(D)$ be a Banach space. For any $u, v \in \widehat{X}$, the differential operator $\mathcal{D}$ and restriction operator Ψ satisfy,

$$(H5): \quad \|u - v\|_{L^{p_x}(E)} \leq C_{pd}\left(\|u\|_{\widehat{X}}, \|v\|_{\widehat{X}}\right)\left(\|\mathcal{D}(u) - \mathcal{D}(v)\|_Y^{\tau_p} + \|\Psi(u) - \Psi(v)\|_Z^{\tau_d}\right), \tag{4.56}$$

for some $0 < \tau_p, \tau_d \leq 1$ and for any subset $D' \subset E \subset D$. This bound (4.56) is termed a *conditional stability estimate* as it presupposes that the underlying solutions have sufficiently regularity as $\widehat{X} \subset X^* \subset X$.

Remark 4.3 We can extend the hypothesis for the inverse problem in the following ways,

- Allow the measurement set D' to intersect the boundary i.e, $\partial D' \cap \mathbb{D} \neq \emptyset$.
- Replace the bound (4.56) with the weaker bound,

$$(H6): \quad \|u-v\|_{L^{px}(E)} \leq C_{pd}\left(\|u\|_{\widehat{X}}, \|v\|_{\widehat{X}}\right)\omega\left(\|\mathcal{D}(u)-\mathcal{D}(v)\|_Y + \|\Psi(u)-\Psi(v)\|_Z\right), \tag{4.57}$$

with $\omega : \mathbb{R} \mapsto \mathbb{R}_+$ being a modulus of continuity.

In this section, we will estimate the error due to the PINN in approximating the solution u of the inverse problem for PDE (4.1) with data (4.53). Following [52] we set $D' \subset E \subset D$ and define the corresponding generalization error as,

$$\mathcal{E}_G(E) = \mathcal{E}_G(E; \theta^*(\mathcal{S}_{int}, \mathcal{S}_d)) := \|u - u^*\|_{L^{px}(E)}. \tag{4.58}$$

As usual, we will bound the generalizatoin error in terms of the training errors for both the data residual as the PDE residual, which we here define as,

$$\mathcal{E}_{d,T} := \left(\sum_{j=1}^{N_d} w_j^d |\mathfrak{R}_{d,\theta}(z_j)|^{p_z}\right)^{\frac{1}{p_z}}, \quad \mathcal{E}_{p,T} := \left(\sum_{i=1}^{N_{int}} w_i |\mathfrak{R}_\theta(y_i)|^{p_y}\right)^{\frac{1}{p_y}}. \tag{4.59}$$

Note that, after the training has concluded, the training error $\mathcal{E}_T$ can be readily computed from the loss function. The bound on generalization error in terms of training error is given by the following estimate.

Theorem 4.4 *Let $u \in \widehat{X} \subset X^* \subset X$ be the solution of the inverse problem associated with PDE* (4.1) *and data* (4.53)*. Assume that the stability hypothesis* (4.56) *holds for any $D' \subset E \subset D$. Let $u^* \in \widehat{X} \subset X^*$ be a PINN generated based on the training sets $\mathcal{S}_{int}$ (of N_{int} quadrature points corresponding to the quadrature rule with convergence rate α) and $\mathcal{S}_d$ (of N_d quadrature points corresponding to the quadrature rule with convergence rate α_d). Further assume that the residuals $\mathfrak{R}_{\theta^*}$ and $\mathfrak{R}_{d,\theta^*}$ are such that $|\mathfrak{R}_{\theta^*}|^{p_y} \in \underline{Y} \subset Y^*$, $|\mathfrak{R}_{d,\theta^*}|^{p_z} \in \underline{Z} \subset Z^*$. Then the following estimate on the generalization error* (4.58) *holds,*

$$\|u - v\|_{L^{px}(E)} \leq C_{pd}\left(\mathcal{E}_{p,T}^{\tau_p} + \mathcal{E}_{d,T}^{\tau_d} + C_q^{\frac{\tau_p}{p_y}} N_{int}^{-\frac{\alpha\tau_p}{p_y}} + C_{qd}^{\frac{\tau_d}{p_z}} N_d^{-\frac{\alpha_d\tau_d}{p_z}}\right), \tag{4.60}$$

with constants $C_{pd} = C_{pd}\left(\|u\|_{\widehat{X}}, \|u^*\|_{\widehat{X}}\right)$ as from (4.56) *and $C_q = C_q\left(\||\mathfrak{R}_{\theta^*}|^{p_y}\|_{\underline{Y}}\right)$ and $C_{qd} = C_{qd}\left(\left\||\mathfrak{R}_{d,\theta^*}|^{p_z}\right\|_{\underline{Z}}\right)$ which stem from the numerical quadrature rule.*

Proof For notational simplicity, we write $\mathfrak{R} = \mathfrak{R}_{\theta^*}$ and $\mathfrak{R}_d = \mathfrak{R}_{d,\theta^*}$. One can observe that

$$\mathfrak{R} = \mathcal{D}(u^*) - \mathcal{D}(u), \qquad \mathfrak{R}_d = \Psi(u^*) - \Psi(u). \tag{4.61}$$

By our assumptions, PINN $u^* \in \widehat{X}$, hence, we can directly apply the *conditional stability estimate* (4.56) and use (4.61), (4.61) to obtain,

$$\begin{aligned}
\mathcal{E}_G(E) &= \|u - u^*\|_{L^{px}(E)} \quad \text{by (4.58)}, \\
&\leq C_{pd}\left(\|\mathcal{D}(u^*) - \mathcal{D}(u)\|_Y^{\tau_p} + \|\Psi(u) - \Psi(u^*)\|_Z^{\tau_d}\right) \quad \text{by (4.56)}, \\
&\leq C_{pd}\left(\|\mathfrak{R}\|_Y^{\tau_p} + \|\mathfrak{R}_d\|_Z^{\tau_d}\right) \quad \text{by (4.61)}.
\end{aligned} \tag{4.62}$$

Since the training error component $\mathcal{E}_{p,T}$ is a quadrature for the residual and $Y = L^{p_y}(D)$, we obtain that,

$$\|\mathfrak{R}\|_Y^{p_y} \leq \mathcal{E}_{p,T}^{p_y} + C_q N_{int}^{-\alpha},$$

and as $\tau_p \leq 1$

$$\|\mathfrak{R}\|_Y^{\tau_p} \leq \mathcal{E}_{p,T}^{\tau_p} + C_q^{\frac{\tau_p}{p_y}} N_{int}^{-\frac{\alpha\tau_p}{p_y}}. \tag{4.63}$$

Similarly as $\tau_d \leq 1$, we find that

$$\|\mathfrak{R}_d\|_Z^{\tau_d} \leq \mathcal{E}_{d,T}^{\tau_d} + C_{qd}^{\frac{\tau_d}{p_z}} N_d^{-\frac{\alpha_d\tau_d}{p_z}}. \tag{4.64}$$

Substituting (4.63) and (4.64) into (4.62) yields the desired bound (4.60). □

The estimate (4.60) indicates mechanisms that underpin possible efficient approximation of solutions of inverse (unique continuation, data assimilation) problems by PINNs as it breaks down the sources of error into the following parts,

- The PINN has to be well-trained i.e, the training error $\mathcal{E}_T$ has to be sufficiently small. Note that we have no a priori control on the training error but can compute it *a posteriori*.
- The class of approximating PINNs has to be sufficiently regular such that the residuals can be approximated to high accuracy by the quadrature rules. This regularity of PINNs can be enforced by choosing smooth activation functions such as the sigmoid and hyperbolic tangent.
- Finally, the whole estimate (4.60) relies on the conditional stability estimate (4.56) for the inverse problem (4.1), (4.53). Thus, the generalization error estimate leverages conditional stability for inverse problems of PDEs into efficient approximation by PINNs. This is very similar to the earlier estimate for forward problems, see Theorem 4.2.

Moreover, the estimate (4.60) on the error due to PINNs contains the constants C_{pd}, C_q, C_{qd}. These constants depend on the underlying solution but also on the trained neural network u^* and on the residuals. For any given number of training samples N_{int}, N_d, these constants are bounded as the underlying neural networks and residuals are smooth functions on bounded domains. However, as $N_{int}, N_d \to \infty$, these constants might blow-up. As long as these constants blow up at rates that are slower wrt N_{int}, N_d than the decay terms in (4.60), one can expect that the overall bound (4.60) still decays to zero as $N_{int}, N_d \to \infty$. In practice, one has a finite number of training samples and the bounds on the constants can be verified a posteriori from the computed residuals and trained neural networks.

Many concrete examples of PDEs where PINNs are used to find a unique continuation can be found in [54]. We briefly summarize their result for the **Poisson equation**, which is given by

$$-\Delta u = f, \quad \text{in } D \subset \mathbb{R}^d, \qquad u|_{D'} = g \quad \text{in } D' \subset D. \tag{4.65}$$

In this case, the conditional stability, cf. (4.56), is guaranteed by the three balls inequality [1]. Therefore, a result like Theorem 4.4 holds. However, note that this theorem only calculates the generalization error on $E \subset D$. It can however be guaranteed that the generalization error is small on the whole domain D and even in Sobolev norm, as follows from the following lemma [54, Lemma 3.3].

Lemma 4.1 *For $f \in C^{k-2}(D)$ and $g \in C^k(D')$, with continuous extensions of the functions and derivatives up to the boundaries of the underlying sets and with $k \geq 2$, Let $u \in H^1(D)$ be the weak solution of the inverse problem corresponding to the Poisson's equation* (4.65) *and let $u^* = u_{\theta^*} \in C^k(D)$ be a PINN. Then, the generalization error is bounded by,*

$$\|u-u^*\|_{H^1(D)} \leq C \left(\|u\|^{1-\tau}_{H^1(D)} + \|u^*\|^{1-\tau}_{H^1(D)} \right) \left| \log \left(\mathcal{E}_{p,T} + \mathcal{E}_{d,T} + C_q^{\frac{1}{2}} N_{int}^{-\frac{\alpha}{2}} + C_{qd}^{\frac{1}{2}} N_d^{-\frac{\alpha_d}{2}} \right) \right|^{-\tau}. \tag{4.66}$$

for some $\tau \in (0,1)$ and with constants $C_q = C_q \left(\|\mathfrak{R}\|_{C^{k-2}(D)} \right)$ and $C_{qd} = C_{qd} \left(\|\mathfrak{R}\|_{C^k(D')} \right)$ which stem from the numerical quadrature rules.

Similar results and numerical experiments for the heat equation, the wave equation and the Stokes equation can be found in [54].

4.5.2 PINNs for parameter identification

Another very common inverse problem is that of **parameter identification**. In this setting, the PDE of under consideration is characterized by an unknown parameter that one wishes to recover based on some data, i.e. measurements. For example for the radiative transfer equations (4.47), one could wish to retrieve the absorption coefficient k based on the incident radiation $G = \int_\Omega u(t,x,\omega,\nu) d\omega$. This was tried out experimentally with PINNs for the 5-dimensional radiative transfer equations in [53]. They found were able to retrieve k up to an error of 2.8% in only 104 minutes.

Appendix A
Preliminaries

A.1 Sobolev spaces

Let $d \in \mathbb{N}$, $k \in \mathbb{N}_0$, $1 \leq p \leq \infty$ and let $\Omega \subseteq \mathbb{R}^d$ be open. For a function $f : \Omega \to \mathbb{R}$ and a (multi-)index $\alpha \in \mathbb{N}_0^d$ we denote by

$$D^\alpha f = \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \cdots \partial x_d^{\alpha_d}} \tag{A.1}$$

the classical or distributional (i.e. weak) derivative of f. We denote by $L^p(\Omega)$ the usual Lebesgue space and for we define the Sobolev space $W^{k,p}(\Omega)$ as

$$W^{k,p}(\Omega) = \{f \in L^p(\Omega) : D^\alpha f \in L^p(\Omega) \text{ for all } \alpha \in \mathbb{N}_0^d \text{ with } |\alpha| \leq k\}. \tag{A.2}$$

For $p < \infty$, we define the following seminorms on $W^{k,p}(\Omega)$,

$$|f|_{W^{m,p}(\Omega)} = \left(\sum_{|\alpha|=m} \|D^\alpha f\|_{L^p(\Omega)}^p \right)^{1/p} \qquad \text{for } m = 0, \ldots, k, \tag{A.3}$$

and for $p = \infty$ we define

$$|f|_{W^{m,\infty}(\Omega)} = \max_{|\alpha|=m} \|D^\alpha f\|_{L^\infty(\Omega)} \qquad \text{for } m = 0, \ldots, k. \tag{A.4}$$

Based on these seminorms, we can define the following norm for $p < \infty$,

$$\|f\|_{W^{k,p}(\Omega)} = \left(\sum_{m=0}^{k} |f|_{W^{m,p}(\Omega)}^p \right)^{1/p}, \tag{A.5}$$

and for $p = \infty$ we define the norm

$$\|f\|_{W^{k,\infty}(\Omega)} = \max_{0 \leq m \leq k} |f|_{W^{m,\infty}(\Omega)}. \tag{A.6}$$

The space $W^{k,p}(\Omega)$ equipped with the norm $\|\cdot\|_{W^{k,p}(\Omega)}$ is a Banach space.

We denote by $C^k(\Omega)$ the space of functions that are k times continuously differentiable and equip this space with the norm $\|f\|_{C^k(\Omega)} = \|f\|_{W^{k,\infty}(\Omega)}$.

References

1. Alessandrini, G., Rondi, L., Rosset, E., Vessella, S.: The stability of the cauchy problem for elliptic equations. Inverse problems **25**(12), 47 (2009)
2. Bai, G., Koley, U., Mishra, S., Molinaro, R.: Physics informed neural networks (PINNs) for approximating nonlinear dispersive PDEs. arXiv preprint arXiv:2104.05584 (2021)
3. Barron, A.R.: Universal approximation bounds for superpositions of a sigmoidal function. IEEE Transactions on Information theory **39**(3), 930–945 (1993)
4. Basu, K., Owen, A.B.: Transformations and hardy-krause variation. SIAM Journal on Numerical Analysis **54**(3), 1946–1966 (2016)
5. Beck, C., Becker, S., Grohs, P., Jaafari, N., Jentzen, A.: Solving stochastic differential equations and kolmogorov equations by means of deep learning. Preprint, available as arXiv:1806.00421v1
6. Beck, C., Jentzen, A., Kuckuck, B.: Full error analysis for the training of deep neural networks (2020)
7. Belkin, M., Hsu, D., Ma, S., Mandal, S.: Reconciling modern machine-learning practice and the classical bias–variance trade-off. Proceedings of the National Academy of Sciences **116**(32), 15849–15854 (2019)
8. Caflisch, R.E.: Monte carlo and quasi-monte carlo methods. Acta Numerica **7**, 1–49 (1998)
9. Chen, Y., Lu, L., Karniadakis, G.E., Dal Negro, L.: Physics-informed neural networks for inverse problems in nano-optics and metamaterials. Optics express **28**(8), 11618–11633 (2020)
10. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
11. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems **2**(4), 303–314 (1989). DOI 10.1007/BF02551274. URL https://doi.org/10.1007/BF02551274
12. Dautray, R., Lions, J.L.: Mathematical analysis and numerical methods for science and technology vol 5 Evolution equations I. Springer Verlag (1992)
13. De Ryck, T., Jagtap, A.D., Mishra, S.: Error analysis for PINNs approximating the Navier-Stokes equations. In preparation (2021)
14. De Ryck, T., Lanthaler, S., Mishra, S.: On the approximation of functions by tanh neural networks. arXiv preprint arXiv:2104.08938 (2021)
15. De Ryck, T., Mishra, S.: Error analysis for deep neural network approximations of parametric hyperbolic conservation laws. In preparation (2021)
16. De Ryck, T., Mishra, S.: Error analysis for physics informed neural networks (pinns) approximating kolmogorov pdes. In preparation (2021)
17. Dick, J., Gantner, R.N., Le Gia, Q.T., Schwab, C.: Higher order quasi-Monte Carlo integration for Bayesian PDE inversion. Comput. Math. Appl. **77**(1), 144–172 (2019). DOI 10.1016/j.camwa.2018.09.019. URL https://doi.org/10.1016/j.camwa.2018.09.019
18. Dick, J., Le Gia, Q.T., Schwab, C.: Higher order quasi-Monte Carlo integration for holomorphic, parametric operator equations. SIAM/ASA J. Uncertain. Quantif. **4**(1), 48–79 (2016). DOI 10.1137/140985913. URL https://doi.org/10.1137/140985913
19. Fletcher, R.: Practical methods of optimization. John Wiley and sons (1987)
20. Friedman, A.: Partial differential equations of the parabolic type. prentice hall (1964)
21. Gantner, R.N., Schwab, C.: Computational higher order quasi-Monte Carlo integration. In: Monte Carlo and quasi-Monte Carlo methods, *Springer Proc. Math. Stat.*, vol. 163, pp. 271–288. Springer, [Cham] (2016). DOI 10.1007/978-3-319-33507-0_12. URL https://doi.org/10.1007/978-3-319-33507-0_12
22. Giles, M.B.: Multilevel monte carlo methods. Acta Numer. **24**, 259–328 (2015)
23. Glorot, X., Bengio, Y.: 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)
24. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)

25. Grohs, P., Hornung, F., Jentzen, A., Von Wurstemberger, P.: A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of black-scholes partial differential equations. arXiv preprint arXiv:1809.02362 (2018)
26. Gühring, I., Raslan, M.: Approximation rates for neural networks with encodable weights in smoothness spaces. Neural Networks **134**, 107–130 (2021)
27. Halton, J.H.: On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numerische Mathematik **2**(1), 84–90 (1960)
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778 (2016)
29. Heinrich, S.: Multilevel monte carlo methods. In: International Conference on Large-Scale Scientific Computing, pp. 58–67. Springer (2001)
30. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
31. Hornung, F., Jentzen, A., Salimova, D.: Space-time deep neural network approximations for high-dimensional partial differential equations. arXiv preprint arXiv:2006.02199 (2020)
32. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 4700–4708 (2017)
33. Jagtap, A.D., Karniadakis, G.E.: Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Communications in Computational Physics **28**(5), 2002–2041 (2020)
34. Jagtap, A.D., Kharazmi, E., Karniadakis, G.E.: Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. Computer Methods in Applied Mechanics and Engineering **365**, 113028 (2020)
35. Kharazmi, E., Zhang, Z., Karniadakis, G.E.: hp-vpinns: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering **374**, 113547 (2021)
36. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: 3rd International Conference on Learning Representations, ICLR 2015 (2015)
37. Lagaris, I.E., Likas, A., D., P.G.: Neural-network methods for bound- ary value problems with irregular boundaries. IEEE Transactions on Neural Networks **11**, 1041–1049 (2000)
38. Lagaris, I.E., Likas, A., Fotiadis, D.I.: Artificial neural networks for solving ordinary and partial differential equations. IEEE Transactions on Neural Networks **9**(5), 987–1000 (1998)
39. Leshno, M., Lin, V.Y., Pinkus, A., Schocken, S.: Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks **6**(6), 861 – 867 (1993). DOI https://doi.org/10.1016/S0893-6080(05)80131-5. URL `http://www.sciencedirect.com/science/article/pii/S0893608005801315`
40. Lin, H.W., Tegmark, M., Rolnick, D.: Why does deep and cheap learning work so well? Journal of Statistical Physics **168**(6), 1223–1247 (2017)
41. Liu, D.C., Nocedal, J.: On the limited memory bfgs method for large scale optimization. Mathematical programming **45**(1), 503–528 (1989)
42. Longo, M., Mishra, S., Rusch, T.K., Schwab, C.: Higher-order quasi-monte carlo training of deep neural networks. arXiv preprint arXiv:2009.02713 (2020)
43. Lu, Y., Chen, H., Lu, J., Ying, L., Blanchet, J.: Machine learning for elliptic pdes: Fast rate generalization bound, neural scaling law and minimax optimality. arXiv preprint arXiv:2110.06897 (2021)
44. Lye, K.O., Mishra, S., Molinaro, R.: A multi-level procedure for enhancing accuracy of machine learning algorithms. arXiv preprint arXiv:1909.09448 (2019)
45. Lye, K.O., Mishra, S., Ray, D.: Deep learning observables in computational fluid dynamics. Journal of Computational Physics **410**, 109339 (2020)
46. Lye, K.O., Mishra, S., Ray, D., Chandrashekar, P.: Iterative surrogate model optimization (ismo): An active learning algorithm for pde constrained optimization with deep neural networks. Computer Methods in Applied Mechanics and Engineering **374**, 113575 (2021)

47. M. Mohri, A.R., Talwalkar, A.: Foundations of machine learning. MIT press (2018)
48. Majda, A.J., Bertozzi, A.L., Ogawa, A.: Vorticity and incompressible flow. cambridge texts in applied mathematics. Appl. Mech. Rev. **55**(4), B77–B78 (2002)
49. Mao, Z., Jagtap, A.D., Karniadakis, G.E.: Physics-informed neural networks for high-speed flows. Computer Methods in Applied Mechanics and Engineering **360**, 112789 (2020)
50. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics **5**(4), 115–133 (1943)
51. Meng, X., Li, Z., Zhang, D., Karniadakis, G.E.: Ppinn: Parareal physics-informed neural network for time-dependent pdes. Computer Methods in Applied Mechanics and Engineering **370**, 113250 (2020)
52. Mishra, S., Molinaro, R.: Estimates on the generalization error of physics informed neural networks (PINNs) for approximating PDEs. arXiv preprint arXiv:2006.16144 (2020)
53. Mishra, S., Molinaro, R.: Physics informed neural networks for simulating radiative transfer. arXiv preprint arXiv:2009.13291 (2020)
54. Mishra, S., Molinaro, R.: Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for pdes. IMA Journal of Numerical Analysis (2021)
55. Mishra, S., Molinaro, R., Tanios, R.: Physics informed neural networks in computational finance: High dimensional forward & inverse option pricing. In preparation (2021)
56. Mishra, S., Rusch, T.K.: Enhancing accuracy of deep learning algorithms by training with low-discrepancy sequences. arXiv preprint arXiv:2005.12564 (2020)
57. Niederreiter, H.: Random number generation and quasi-Monte Carlo methods, vol. 63. SIAM (1992)
58. Owen, A.B.: Monte carlo variance of scrambled net quadrature. SIAM Journal on Numerical Analysis **34**(5), 1884–1910 (1997)
59. Pang, G., D'Elia, M., Parks, M., Karniadakis, G.E.: npinns: nonlocal physics-informed neural networks for a parametrized nonlocal universal laplacian operator. algorithms and applications. Journal of Computational Physics **422**, 109760 (2020)
60. Pang, G., Lu, L., Karniadakis, G.E.: fpinns: Fractional physics-informed neural networks. SIAM Journal on Scientific Computing **41**(4), A2603–A2626 (2019)
61. Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B., Liao, Q.: Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. International Journal of Automation and Computing **14**(5), 503–519 (2017)
62. Raissi, M., Karniadakis, G.E.: Hidden physics models: Machine learning of nonlinear partial differential equations. Journal of Computational Physics **357**, 125–141 (2018)
63. Raissi, M., Perdikaris, P., Karniadakis, G.E.: Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics **378**, 686–707 (2019)
64. Raissi, M., Yazdani, A., Karniadakis, G.E.: Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data. arXiv preprint arXiv:1808.04327 (2018)
65. Rosenblatt, F.: The perceptron–a perciving and recognizing automation. Cornell Aeronautical Laboratory (1957)
66. Schmidt-Hieber, J.: Deep relu network approximation of functions on a manifold. arXiv preprint arXiv:1908.00695 (2019)
67. Shin, Y., Darbon, J., Karniadakis, G.E.: On the convergence and generalization of physics informed neural networks (2020). Preprint, available from arXiv:2004.01806v1
68. Sobol', I.M.: On the distribution of points in a cube and the approximate evaluation of integrals. Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki **7**(4), 784–802 (1967)
69. Stoer, J., Bulirsch, R.: Introduction to numerical analysis. Springer Verlag (2002)
70. Telgarsky, M.: Representation benefits of deep feedforward networks. arXiv preprint arXiv:1509.08101 (2015)
71. Yang, L., Meng, X., Karniadakis, G.E.: B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with noisy data. Journal of Computational Physics **425**, 109913 (2021)

72. Yarotsky, D.: Error bounds for approximations with deep ReLU networks. Neural Networks **94**, 103–114 (2017)
73. Yu, B., et al.: The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. arXiv preprint arXiv:1710.00211 (2017)