The following is an excerpt from Johannes Taraz' master's thesis ("Mode-Decomposition in DeepONets: Generalization and Coupling Analysis" supervised by Dr. Alexander Heinlein at TU Delft, 2025). The whole thesis can be found at
`https://repository.tudelft.nl/record/uuid:e8a0439c-ecfa-4adc-8ea7-2679847995eb`.

1 INTRODUCTION

1.1 BACKGROUND

Many models in science and engineering can be formulated as differential equations. Hence, a large body of research exists on solving these equations efficiently and accurately. For ordinary differential equations (ODEs) numerical integration methods such as the (implicit) Euler method and Runge Kutta method are widely used. For partial differential equations (PDEs) methods such as the finite volume method, finite element method, finite difference method, and spectral methods dominate the field.

In this work, we are concerned with the approximation of the solution operator; more specifically, we focus on the time evolution operator of time dependent PDEs, mapping from the initial condition to the solution at a later time. The aforementioned methods can be used to efficiently compute the time evolution for one initial condition via numerical integration. This corresponds to evaluating the solution operator for one initial condition. However, these methods alone offer no efficient way of approximating the entire operator, as each initial condition requires an independent numerical integration. For many tasks, such as PDE-constrained optimization (e.g. design or optimal control) or uncertainty quantification, the approximation of the entire solution operator is necessary. Thus methods that approximate the entire solution operator accurately promise very significant performance improvements [1].

Next to methods like reduced order modeling (ROM), which project the dynamics onto a lower dimensional system built from previous solutions, operator learning (OL) has gained traction in recent years. The fundamental idea of OL is to learn to approximate an operator mapping between two function spaces, given some evaluations of this operator. In the case of time evolution, the operator maps the initial condition onto the solution at a later time. Building on theoretical results for operator approximation [5], Lu et al. introduced the deep operator network (DeepONet) in 2020 [34], which has since become the standard architecture for operator learning.

Despite the prospects of approximating non-linear operators with a theoretically arbitrary accuracy, DeepONets still face severe practical limitations, such as poor accuracy compared to classical numerical solvers, or conversely a high data demand for good accuracy. Another issue is the resolution dependence of DeepONets, specifically for training and testing on different meshes.

To address the poor accuracy and generalizability several modified training methods and architectural modifications to the DeepONet have been suggested in the literature [1, 12, 32, 35, 51]. Additionally, entirely new architectures for OL have been proposed, such as the Fourier neural operator, the convolutional neural operator and the Laplace neural operator [4, 33, 42]. These architectures explicitly address the resolution dependence. Moreover, the combination of OL and physics-informed neural networks (PINNs), i.e., neural networks solving a PDE by minimizing the residual of the PDE [27], has sparked significant interest [16, 52].

In recent years, addressing spectral bias – i.e., the tendency of neural networks to learn low-frequency components faster than high-frequency ones – in operator learning architectures has emerged as a prominent research direction [22, 24, 50, 56]. While these works have proposed valuable architectural modifications that improve performance, the focus on empirical solutions over solutions based on understanding is symptomatic of a broader phenomenon in the field. Theoretical frameworks, such as [28], are emerging but applying them to dissect existing architectures – understanding their inner workings and fundamental limitations – remains underexplored.
For instance, fundamental questions about error sources in DeepONets are insufficiently studied. To our knowledge, the relative contributions of learned basis functions (trunk network) versus their coefficients (branch network) to the total approximation error have not been systematically studied in practice. Such insights could guide more principled architecture design.

1.2 RESEARCH OBJECTIVES

In this work, we focus on examples where DeepONets have poor accuracy even when trained and tested on the same mesh and on test data drawn from the same distribution as training data. To gain an understanding of the origin of the DeepONets errors, we first investigate the following research question (RQ).

RQ (I) *How is the total approximation error distributed between the error of learned basis functions and the error of their coefficients?*

To answer this we make use of analytical work by Lanthaler et al. [28]. We show that in all of our examples, the coefficients' errors dominate. We thus focus on the coefficients' errors and investigate the following question:

RQ (II) *The coefficients of which basis functions are not accurately approximated, and why?*

This can be split into different subquestions.

1. How are the coefficient errors distributed over the different basis functions?
   (a) How does the optimization scheme influence the error distribution?
   (b) How well do the approximations of different coefficients generalize?
2. How do the coefficients of different basis functions interact with each other?
   (a) Should the different coefficients be learned in separate neural networks, or should they all share the same hidden neurons?
   (b) How does reducing the error of coefficient i impact the error of a different coefficient $j \neq i$?

The second subquestion is one approach to answer the 'why' in this research question.

1.3 STRUCTURE

In Chapter 2, we provide the background with frequently used notation, formalism, a recap on neural networks, and introduce the DeepONet architecture.

In Chapter 3, we describe how the training and test data used throughout this thesis are generated. Furthermore, we highlight key properties of the data that will be relevant in later discussions.
In Chapter 4, we investigate RQ (I) and outline the shortcomings of the DeepONet architecture. We perform the Lanthaler error decomposition, which shows that the error mainly originates in the branch network.
In Chapter 5, we then introduce the SVD-based operator network (SVDONet), which replaces the trunk net with a trunk matrix and thereby reduces the DeepONet to the branch network. The trunk matrix is determined through SVD of the training data matrix. Note that the SVDONet can be considered as a POD-DeepONet [35] with a rescaled trunk matrix. For the SVDONet we propose a novel decomposition of the error; we partition the branch error into the errors of the coefficients of the different basis functions modes, yielding deep insights into the shortcomings of the branch network.
In Chapter 6, we then use the error decomposition and investigate the error distribution, i.e., which coefficients are poorly approximated, for different optimizers and training and test data. Thus, Section 6.1 addresses RQ (II).1 (a) and (b). In Section 6.2, we furthermore investigate architectural coupling of the coefficients, through the comparison to the stacked SVDONet, and update based coupling of the coefficients, through the effect of improving one coefficient on other coefficients. Thus, Sections 6.2.1 and 6.2.2 address RQ (II).2 (a) and (b) respectively.
In Chapter 7, we summarize our findings and conclude with their discussion.

2 BACKGROUND

This chapter provides the necessary background for the thesis, starting with a short but formal problem statement, an introduction to neural networks, the DeepONet, the setup for testing and training used in this thesis, and lastly the derivation of an error decomposition into trunk and branch error.

2.1 PROBLEM STATEMENT

In the general setting of OL, any operator mapping from function space to function space can be approximated. In this work, we seek to approximate the solution operator $G_* : \mathcal{C}(D) \to \mathcal{C}(D)$ for time-dependent PDEs, where G_* maps the initial condition to the solution at a fixed time $\tau > 0$. The PDE has spatial domain D. For ease of exposition, we consider the Korteweg-de Vries (KdV) equation as an example. The KdV equation can be used to model shallow water waves [25]. It is a time-dependent PDE of the form

$$\begin{aligned}
0 &= \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} + 0.01\frac{\partial^3 u}{\partial r^3} && \text{for all } r \in (0, 2\pi), t > 0, \\
u(0,t) &= u(2\pi, t) && \text{for all } t \geqslant 0, \\
\frac{\partial u}{\partial r}(0,t) &= \frac{\partial u}{\partial r}(2\pi, t) && \text{for all } t \geqslant 0, \\
\frac{\partial^2 u}{\partial r^2}(0,t) &= \frac{\partial^2 u}{\partial r^2}(2\pi, t) && \text{for all } t \geqslant 0, \\
u(r, t=0) &= p(r) && \text{for all } r \in (0, 2\pi),
\end{aligned}$$

where $p(r)$ is the initial condition. Since we are concerned with the time evolution of the state u up to a fixed time τ, we define the solution operator (or time evolution operator)

$$G_* : p(\cdot) \to u(\cdot, t = \tau),$$

mapping the initial condition to the solution at time τ. To make the problem computationally accessible, we *encode* p in a finite dimensional vector space as $\hat{p} \in \mathbb{R}^M$. This is done by sampling p at M different locations $\bar{r}_1, ..., \bar{r}_M$:

$$\hat{p} = \left(p(\bar{r}_1) \ \ldots \ p(\bar{r}_M)\right)^T .$$

This involves a loss of information, which is addressed in Section 2.5. We then build a parametric model $G_\theta : \hat{p} \to u_p$, parametrized by θ, to approximate G_*. This model thus maps the finite dimensional encoding to an infinite dimensional vector space, $G_\theta : \mathbb{R}^M \to \mathcal{C}(D)$. The model's output $u_p(\cdot)$ is hence a function of r, meant to approximate the true solution function $u(\cdot, t = \tau)$ for all $r \in D$. Section 2.3 gives an overview over the types of models G_θ we consider, namely neural networks, how good parameters θ are found, and what it means for G_θ to approximate G_*. Section 2.4 explains how neural networks can be used for OL, by introducing the DeepONet.

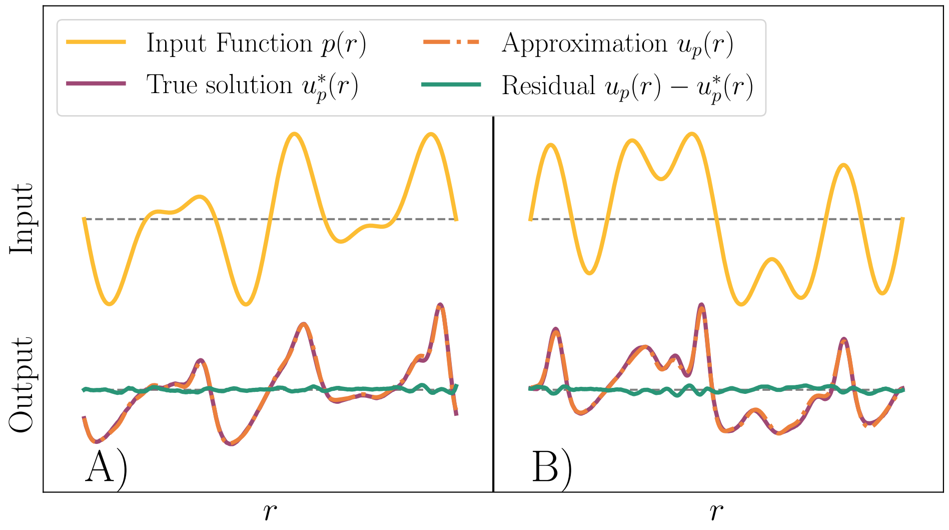


Figure 1: **Input and Output of DeepONet for the KdV Equation with $\tau = 0.2$. Left (A):** The left input function is part of the training data set. **Right (B):** The right input function is not part of the training data set. The yellow line shows the input function $p(r)$. The purple line shows the true solution $u_p^*(r)$. The orange (dash-dotted) line shows the DeepONet approximation $u_p(r)$. The green line shows the residual $u_p - u_p^*$. Dashed gray lines indicate the zero reference level to aid visual interpretation.

To denote the dependency on the initial condition, we henceforth write $u_p^*(\cdot)$ for the true solution $u(\cdot, t = \tau)$ corresponding to initial condition $u(\cdot, t = 0) = p(\cdot)$. We omit the dependency on τ since τ is fixed for every considered example problem; see Section 3.1. As an example, Fig. 1 shows the true solution u_p^*, the DeepONet approximation u_p and the residual $u_p - u_p^*$ for one training and one test initial condition, i.e., input function $p(r)$, of the KdV equation with $\tau = 0.2$.

2.2 UNIVERSAL APPROXIMATION THEOREM FOR FUNCTIONS

In this section, we briefly introduce the universal approximation theorem (UAT) for function approximations. This section serves as both a segue to the topic of neural networks (see Section 2.3) and as a basis for the UAT for operators, which is the theoretical foundation for DeepONets (see Section 2.4).
One of the foundational results for function approximation using neural networks is the UAT, published in 1989 by Cybenko [7], which applies to sigmoid activation functions. Here, we use a more general formulation by Pinkus [39], which extends the result to any non-polynomial activation function.

Theorem 1 (UAT for Functions) *For any non-polynomial* $\sigma \in \mathcal{C}(\mathbb{R})$, *any function* $f \in \mathcal{C}(\mathbb{R}^d)$ *and any* $\varepsilon > 0$, *there exist* $N \in \mathbb{N}$, $a_k, d_k \in \mathbb{R}$ *and* $c_k \in \mathbb{R}^d$ *for* $k \in [N]$, *such that*

$$\left| f(r) - \sum_{k=1}^{N} a_k \sigma(c_k^T r + d_k) \right| < \varepsilon,$$

for all $r \in [0, 1]^d$.

This means that for any continuous function f there is a linear combination of N terms which approximates f arbitrarily well. Each of the N terms first applies an individual scaling and shifting to the same input and then applies the same non-polynomial function σ. Note that the number of terms N is not bounded, meaning this linear combination might be completely impractical to approximate some f.

2.3 FOUNDATIONS OF NEURAL NETWORKS AND MACHINE LEARNING

This section provides a self-contained overview of foundational concepts in neural networks and machine learning, covering (1) neural networks, based on the UAT for functions, (2) different optimization algorithms to find parameters for neural networks, (3) overfitting and generalization, and (4) spectral bias. Readers already familiar with these topics may wish to skip this section.

We now discuss function approximation using neural networks, a common problem in machine learning. Consider some inputs $\{r_1, ..., r_m\}$ and corresponding target outputs $\{y_1, ..., y_m\}$. These inputs could be generated by a function f, i.e., $y_i = f(r_i)$. We then use a parametric model G_θ parametrized by θ, i.e., we seek to find parameters θ such that G_θ approximates f well. Or, more formally, training the neural network is equivalent to finding some not necessarily unique optimal parameters

$$\theta_{opt} \in \arg\min_{\theta} \mathop{}\limits_{i=1}^{m} |y_i - G_\theta(r_i)|^2.$$

In neural network literature, the mean-squared error

$$\mathcal{L}_{tr}(\theta) = \frac{1}{m} \mathop{}\limits_{i=1}^{m} |y_i - G_\theta(r_i)|^2,$$

is often termed as training loss function. The subscript tr distinguishes the training loss from the test loss introduced later. We now discuss neural networks, one type of parametric model G_θ.

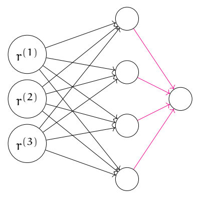
2.3.1 *Neural Networks*

Since the UAT shows that for any continuous function f there is a linear combination of N terms, where each term applies an individual scaling and shifting to the same input and then the same non-polynomial function σ, which approximates f arbitrarily well, we use this construction

$$G_\theta(r) = \mathop{}\limits_{k=1}^{N} \theta_k^{(out)} \sigma\left((\vartheta_k^{(W,1)})^T r + \vartheta_k^{(B,1)} \right)$$

as a model. The parameters of this model are called

$$\begin{aligned} \textit{outer weights } \theta^{(out)} &= \;\; \theta_1^{(out)}\ \theta_2^{(out)}\ \ldots\ \theta_N^{(out)} \;\; \in \mathbb{R}^{1\times N}, \\ \textit{inner weights } \vartheta^{(W,1)} &= \;\; \vartheta_1^{(W,1)}\ \vartheta_2^{(W,1)}\ \ldots\ \vartheta_N^{(W,1)} \;\; \in \mathbb{R}^{d\times N}, \text{ and} \\ \textit{biases } \vartheta^{(B,1)} &= \left(\vartheta_1^{(B,1)}\ \vartheta_2^{(B,1)}\ \ldots\ \vartheta_N^{(B,1)} \right)^T \in \mathbb{R}^N. \end{aligned}$$



$$G_\theta(r) = \overset{N}{\underset{k=1}{}} \theta_k^{(out)} \sigma\left((\vartheta_k^{(W,1)})^T r + \vartheta_k^{(B,1)}\right)$$

Figure 2: **One-Layer Perceptron.** The input is$r = \left(r^{(1)}\ r^{(2)}\ r^{(3)}\right)^T \in \mathbb{R}^3$. It has $N = 4$ hidden neurons. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the non-polynomial activation function in the receiving neuron.

This can be rewritten, by introducing the hidden layer $H_{\theta^{(1)}}$ with parameters $\theta^{(1)} = (\vartheta^{(W,1)}, \vartheta^{(B,1)})$, such that

$$G_\theta(r) = \theta^{(out)} H_{\theta^{(1)}}(r). \tag{1}$$

The model is called a perceptron with one hidden layer and a linear output layer [36, 43]. The outer weights θ^{out} together with the weights of the hidden layer $\theta^{(1)} = (\vartheta^{(W,1)}, \vartheta^{(B,1)})$ make up the parameters of the one-layer perceptron

$$\theta = \left(\theta^{(1)}, \theta^{out}\right).$$

While θ consists of a $1 \times N$ matrix and a tuple of a $d \times N$ matrix and an N dimensional vector, it is often practical to work with the vector Θ, such that Θ's entries are given by the entries of $\theta^{(1)}, \vartheta^{(W,1)}$ and $\vartheta^{(B,1)}$. Thus, for a one-layer perceptron, $\Theta \in \mathbb{R}^{dN+2N}$. The number of parameters, or dimension of the vector, is denoted by $|\theta|$. This is described in more detail in Section A.8.1. The one-layer perceptron is visualized in Fig. 2.

In practice, the hidden layers are often applied in succession, yielding a multi-layer perceptron (MLP): The input r is processed in the first layer, yielding $z_1 = H_{\theta^{(1)}}(r)$ with some parameters $\theta^{(1)}$. The first layer's output is processed in the second hidden layer, as $z_2 = H_{\theta^{(2)}}(z_1)$ with different parameters $\theta^{(2)}$, etc. If the network consists of D successively applied hidden layers, the output of the D-th layer $z_D = H_{\theta^{(D)}}(z_{D-1})$ is then processed by a linear layer, such that $G_\theta(r) = \theta^{(out)} z_D$. The parameters of each hidden layer and the linear layer together make up the parameters of the MLP:

$$\theta = (\theta^{(1)},\ \theta^{(2)},\ \ldots,\ \theta^{(D)},\ \theta^{(out)}). \tag{2}$$

It is visualized in Fig. 3. This composition is done because, in practice, it reduces the number of necessary parameters for a given accuracy [30]. The number of hidden layers D is called depth, hence the term *deep learning*. The number of neurons in one layer is often called the *width* of this layer. If all layers have the same width, this is referred to as the *width of the neural network*. Modern neural networks are often not MLPs. They modify the classical MLP architecture by restricting the

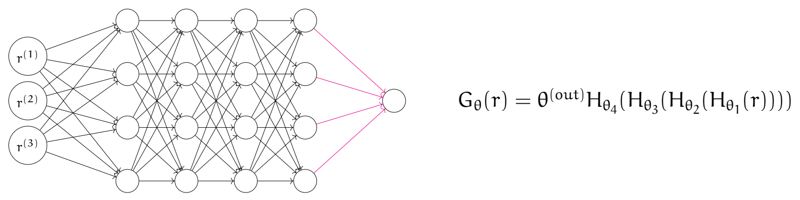

$r^{(1)}$
$r^{(2)}$
$r^{(3)}$
$G_\theta(r) = \theta^{(out)} H_{\theta_4}(H_{\theta_3}(H_{\theta_2}(H_{\theta_1}(r))))$

Figure 3: **Multi-Layer Perceptron.** It has input $r = \left(r^{(1)}\ r^{(2)}\ r^{(3)}\right)^T \in \mathbb{R}^3$, $D = 4$ hidden layers and $w = 4$ neurons per layer. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the non-polynomial activation function in the receiving neuron.

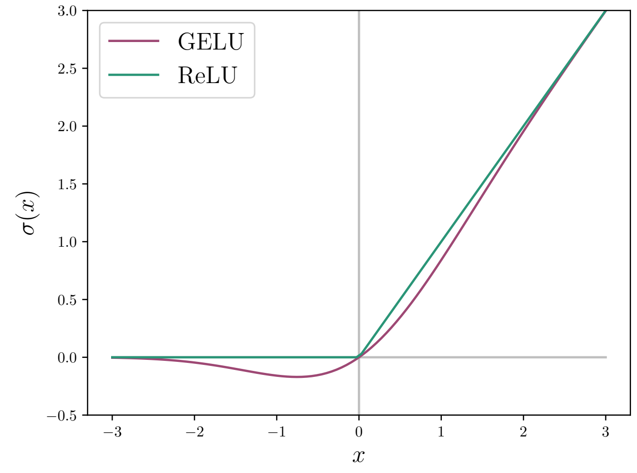


Figure 4: **GELU (purple) and ReLU (green) Activation Functions.** Reproduced and adapted from [18].

parameter space to a structured subspace. For example, instead of optimizing general weight vectors $\vartheta_k^{(W,1)} \in \mathbb{R}^d$, one may constrain $\vartheta_k^{(W,1)} \in V \subset \mathbb{R}^d$ to enforce architectural properties such as convolutions [9, 29].

One of the key components of neural networks is the non-polynomial function σ, the so-called activation function. In this thesis we use the Gaussian error linear unit (GELU) as an activation function [18]. GELU is defined as

$$\mathrm{GELU}(x) = \frac{x}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \approx \frac{x}{2} \quad 1 + \tanh \quad \overline{\frac{2}{\pi}}(x + 0.044715x^3) \quad ,$$

where $\mathrm{erf}(x)$ is the Gauss error function. Fig. 4 shows GELU and the classical rectified linear unit ($\mathrm{ReLU}(x) = \max(0, x)$) for comparison.

A remaining question is: how do we find network parameters θ (Equation 2) that yield a low approximation error, and thus a low loss $\mathcal{L}_{tr}(\theta)$?

2.3.2 *Optimization Schemes*

One standard approach is gradient descent (GD). Consider some initial parameters θ_0 (Equation 2), which are chosen randomly according to a specified distribution [26, 31]. The parameters are then updated with a step in the direction opposite to the gradient, $\theta_1 = \theta_0 - \alpha_1 \nabla\mathcal{L}_{tr}(\theta_0)$, where α_1 is the so-called learning rate for the first parameter update. The gradient $\nabla\mathcal{L}_{tr}$ points in the direction of steepest increase of the loss, thus moving in the opposite direction decreases the loss most rapidly for small steps. This procedure is repeated until a satisfactory loss is reached. Iterations of training algorithms are typically called epochs. However, GD often converges slowly or the parameters get trapped in local minima of the loss function, with poor performance.
One very popular variant of classical GD, which often leads to faster convergence, is *momentum based GD*. Momentum means that the update in the t-th step $\delta\theta_t$ is not just given by the the gradient of the loss function at that step $\nabla\mathcal{L}_{tr}(\theta_t)$, but it includes the update of the last step

$$\delta\theta_t = -\alpha_t \nabla\mathcal{L}_{tr}(\theta_t) + \beta\ \delta\theta_{t-1},$$

for some momentum factor $\beta \in \mathbb{R}$. The idea behind this is to average out high-frequency oscillations in the gradient [46].
Another popular variant is the method of *adaptive gradients*. In the adaptive gradient scheme AdaGrad [10], the parameter update is defined component-wise. As described in detail in Appendix A.8.1, the parameters θ of a neural network can be written as a vector $\Theta \in \mathbb{R}^{|\theta|}$, where $|\theta|$ denotes the number of parameters and equivalently the dimension of the vector. The loss function's gradient can then also be written as a vector $\nabla\mathcal{L}_{tr}(\Theta) \in \mathbb{R}^{|\theta|}$. For every training epoch t, we define the vector $\nu_t \in \mathbb{R}^{|\theta|}$ through its entries $(\nu_t)^{(i)}$:

$$(\nu_t)^{(i)} = \mathop{}\limits_{j=1}^{t} \left(\nabla\mathcal{L}_{tr}(\Theta_j))^{(i)}\right)^2 .$$

The parameter update vector $\delta\Theta_t \in \mathbb{R}^{|\theta|}$ in AdaGrad is then defined as:

$$(\delta\Theta_t)^{(i)} = -\alpha_t \frac{(\nabla\mathcal{L}_{tr}(\Theta_t))^{(i)}}{\overline{(\nu_t)^{(i)}} + \epsilon}.$$

This can also be written as

$$\delta\Theta_t = -\alpha_t \frac{\nabla\mathcal{L}_{tr}(\Theta_t)}{\sqrt{\nu_t} + \epsilon} \text{ with } \nu_t = \mathop{}\limits_{j=1}^{t} (\nabla\mathcal{L}_{tr}(\Theta_j))^2,$$

with the square and division applied component-wise. For brevity of notation we hereafter do not distinguish between θ and Θ. The normalization with ν_t emphasizes parameter components with persistently small gradients, allowing AdaGrad to capture rare but informative patterns in the data.
RMSprop, another adaptive gradient scheme, introduces a decay mechanism in the normalization term to gradually discount earlier gradient contributions [19], i.e.,

$$\nu_t = \beta_2 \nu_{t-1} + (1 - \beta_2)(\nabla\mathcal{L}_{tr}(\theta_i))^2.$$

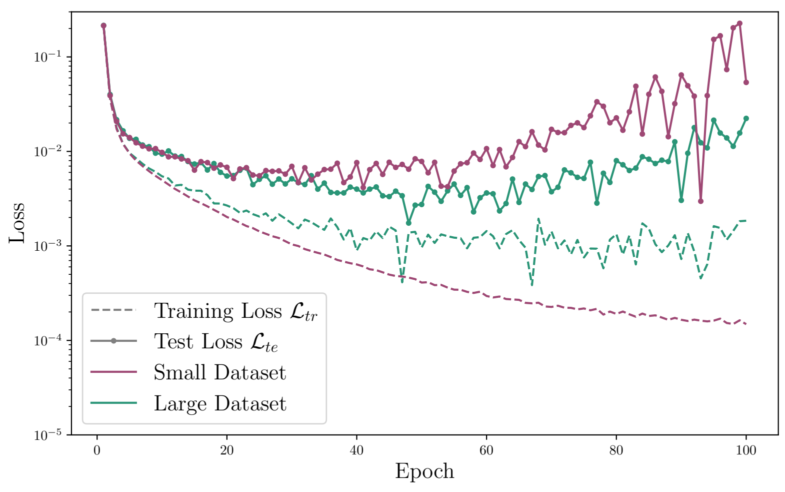


Figure 5: **Training and Test Loss of Stereotypical Neural Networks over the training course.** The two neural networks have the same architecture/size and different data sets. Adapted from [15].

The adaptivity of RMSprop and the momentum are combined in a method called Adam [23]. Here the update is computed as follows:

$$\begin{aligned}
m_t &= \beta m_{t-1} + (1-\beta_1)\nabla\mathcal{L}_{tr}(\theta_t), \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla\mathcal{L}_{tr}(\theta_t))^2, \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, && (3) \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t}, && (4) \\
\delta\theta_t &= -\alpha_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \bar{\epsilon}} + \epsilon}.
\end{aligned}$$

The steps in Equations 3 and 4 are done to normalize the size of m_t and v_t in early epochs. Note that for all optimizers α_t is the learning rate following some trajectory, e.g., a constant learning rate $\alpha_t = \alpha$ or exponential decay $\alpha_t = 0.95^{t/500}\alpha_1$. This trajectory is often called learning rate schedule. Unless noted otherwise, the optimization scheme Adam is used in this work. A table containing all neural network hyperparameters, such as width, depth, optimization scheme, and learning rate schedule, used in each numerical experiment in this thesis is given in Appendix A.1.

2.3.3 *Overfitting and Generalization*

So far, we only considered a low loss approximation error of the training data as desirable. However, in most applications, the trained model should be able to accurately predict the approximated function f also for inputs not in the training data set. Those new inputs, and their targets, are termed test data. A regression model is *overfitting*, if it can approximate its training data significantly better than the test data. Conversely, *generalization* refers to the ability of accurately approximating the test data, after only having seen training data.

We denote the training inputs and targets as R_{tr}, Y_{tr} and the test inputs and targets as R_{te}, Y_{te}, with $Y_{tr} \in \mathbb{R}^{m_{tr}}, Y_{te} \in \mathbb{R}^{m_{te}}$, and define the training and test loss

$$\mathcal{L}_{tr} = \frac{1}{m_{tr}} ||Y_{tr} - G(R_{tr})||_2^2,$$
$$\mathcal{L}_{te} = \frac{1}{m_{te}} ||Y_{te} - G(R_{te})||_2^2.$$

The test data can either be drawn from the same distribution as the training data, or drawn from an entirely new one. In this thesis we only consider test data from the same distribution as the training data. Fig. 5 shows the stereotypical training and test loss for two neural networks, of the same architecture, but with training data sets of different sizes. For the small data set, we see that the training error is lowered significantly, and continuously over the epochs. The test error, on the other hand, only decreases in the first 40 epochs. Hereafter, the test error increases significantly. The growing gap, between training and test error, and especially the increasing test error, show that the neural network trained on the small data set is overfitting.

For the larger data set, the training error is not lowered as much. The test error decreases in the first 70 epochs, and only starts increasing hereafter. Thus, this shows a later onset of overfitting in the neural network trained on the large data set. Additionally, the smaller gap between training and test error indicates that the neural network trained on the large data set is overfitting to a lesser extent.

This can be attributed to two reasons; (a) the model relies on structures specific to the training set which do not generalize to other data sets, and (b) the model fails to capture structures that may appear in unseen data [3].

Besides using a larger dataset, methods to prevent overfitting are known as regularization. As seen in Fig. 5, for the small dataset, the test loss starts increasing after some point. Thus, if only this small dataset is available and a low test loss is desired, the parameters obtained after 100 epochs are not the best parameters. The parameters at the minimal test loss would obviously be better. This is the motivation behind early stopping [40]. For early stopping, the original training data set is split into a new training set and a validation set. For the training, the gradient of the loss is computed using only the new training data set. The validation dataset is only used to compute the validation loss. Then this validation loss is used as a proxy for the test loss. If the validation loss is observed to increase over some number of epochs, the training is stopped and the parameters achieving the lowest validation loss are used.

Other very common regularization techniques are l^1 and l^2 regularization, where the l^1 or l^2 norm of the weights is added to the loss function. This either encourages sparse (l^1) or small, evenly distributed (l^2) weights [2].

Note that in the very first epochs, the training and test losses are very similar, yet both remain high; see Fig. 5. This behavior is known as *underfitting* and highlights an important point: a model that is not overfitting is not necessarily performing well. Even though the losses match closely, the model has not yet captured the underlying structure of the data, resulting in poor predictive performance.

In this thesis, the employment of regularization techniques is omitted in order to study the properties of pure DeepONets. In practice, the employment of regularization techniques, including early stopping and l^2 regularization, is recommended.

2.3.4 *Spectral Bias*

This section describes the *spectral bias*, or alternatively the *frequency principle* [41, 53], a commonly observed phenomenon in neural network training. During the training, neural networks tend to capture the low frequency features of the solution before capturing the features with higher frequencies - if the latter are captured at all.
To illustrate this, consider a function f with a Fourier sum representation

$$f(r) = \sqrt{2} \mathop{}\limits_{i=1}^{F} a_i \sin(2\pi i r) \text{ for } r \in [0, 1],$$

with arbitrary coefficients a_i. Furthermore, consider a neural network G_θ to approximate the target function f. We define the projection of a function g onto the i-th basis function $\sin(2\pi i r)$ as

$$b_i(g) = \sqrt{2} \mathop{}\limits_{0}^{1} g(r) \sin(2\pi i r) dr.$$

Note that for $g = f$, the projections and coefficients are identical, i.e., $a_i = b_i(f)$. Thus a_i is the target value of the projection $b_i(G_\theta)$ of the neural network's output G_θ. The normalization constant $\sqrt{2}$ ensures orthonormality of the basis functions $\sin(2\pi i r)$.
Empirically, it is observed that the projections of the neural network output $b_i(G_\theta)$ approach their target values a_i much faster for small i than for large i. This is observed regardless of the coefficients a_i. Fig. 6 visualizes the spectral bias via e_i, the i-th spectral error, $e_i = |a_i - b_i(G_\theta)|^2$ and the neural network's output G_θ for different training epochs.
As noted, the term spectral bias is usually used to describe the difference in convergence speed of features with different frequencies. However, the term could also be used to describe the behavior of any approximation model for which the spectral errors for low frequencies are smaller than the ones for high frequencies. We term this *output-wise spectral bias*.
Furthermore, similar to the discussion here, the spectral bias is typically applied to the general approximation error, i.e., both the test and training error. However, the spectral bias can also be interpreted as a theory on generalization [58]. The key argument is summarized as follows:
When trying to learn a high-frequency target function, the neural network learns a low-frequency function if (a) it only receives a low-frequency signal (due to insufficient training data), or if (b) it cannot represent the target function (because the number of parameters is insufficient). In the case of (a) the neural network is overfitting; in the case of (b) the neural network is underfitting. If there are sufficient training samples and the neural network has sufficient parameters, it will learn an approximation that achieves low training and test error (*fitting*).

2.4 DEEP O NET

As the UAT for functions is the theoretical foundation for function approximation with neural networks, the UAT for operators is the corresponding cornerstone of OL. It was published by Chen and Chen in 1995 [5]. The DeepONet, which introduced one of the first practical architectures for OL and has since become

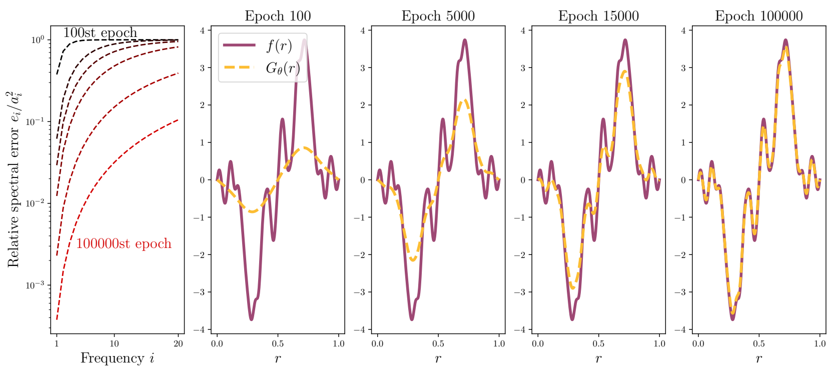


Figure 6: **Spectral Bias in Neural Networks for Function Approximation. Left panel:** Relative spectral errors e_i/a_i^2 for multiple epochs. Color indicates training progress, from black (early) to red (late epochs). **Remaining panels (2-5):** True solution function f (purple) and neural network approximation G_θ (yellow) for epochs 100, 5000, 15000 and 100000. Reproduced and adapted from [41].

a widely used baseline, can be seen as a direct implementation of the UAT for operators [34].

Theorem 2 (UAT for Operators) *Consider a continuous non-polynomial function* σ*, a Banach space* V*, a compact set* $K_1 \subset V$*, a compact set* $X \subset \mathcal{C}(K_1)$*, a compact set* $K_2 \subset \mathbb{R}^d$ *and a continuous operator* $G_* : X \to \mathcal{C}(K_2)$*. Then, for any* $\varepsilon > 0$*, there exist constants* $I, M, K \in \mathbb{N}$*,* $d_k, a_i^k, \beta_{ij}^k, \gamma_i^k \in \mathbb{R}$*,* $\bar{r}_j \in K_1$*, and* $c_k \in \mathbb{R}^d$ *such that*

$$\left| G_*(p)(r) - \sum_{k=1}^{N} \underbrace{\sigma(c_k^T r + d_k)}_{=:t_k(r)} \underbrace{\sum_{i=1}^{I} a_i^k \sigma\left(\sum_{j=1}^{M} \beta_{ij}^k p(\bar{r}_j) + \gamma_i^k \right)}_{=:b_k([p(\bar{r}_1),\ \dots,\ p(\bar{r}_M)])} \right| < \varepsilon \tag{5}$$

for all functions $p \in X$ *and coordinates* $r \in K_2$*.*

A crucial part of the DeepONet architecture - already visible in Equation 5 - is the division of the model into two sub-networks. The trunk network with the coordinates r as input has output neurons $t_k(r)$. As described in the UAT for operators, the trunk network consists of one hidden layer, which applies the non-polynomial activation function σ. The branch network with the sampled input function p as input has output neurons $b_k([p(\bar{r}_1),\ \dots,\ p(\bar{r}_M)])$. As described in the UAT for operators, the branch network consists of two hidden layers. The first hidden layer applies the non-polynomial activation function σ. The second hidden layer is linear. This architecture can be seen in Fig. 7.
In the DeepONet both trunk and branch networks are generalized to MLPs. As in the UAT for operators, both sub-networks of a DeepONet have the same number of output neurons N and the final output is given as

$$G_\theta(\hat{p})(r) = \sum_{j=1}^{N} b_j(\hat{p}) t_j(r) = (t_1(r) \dots t_N(r)) \begin{pmatrix} b_1(\hat{p}) \\ \vdots \\ b_N(\hat{p}) \end{pmatrix},$$

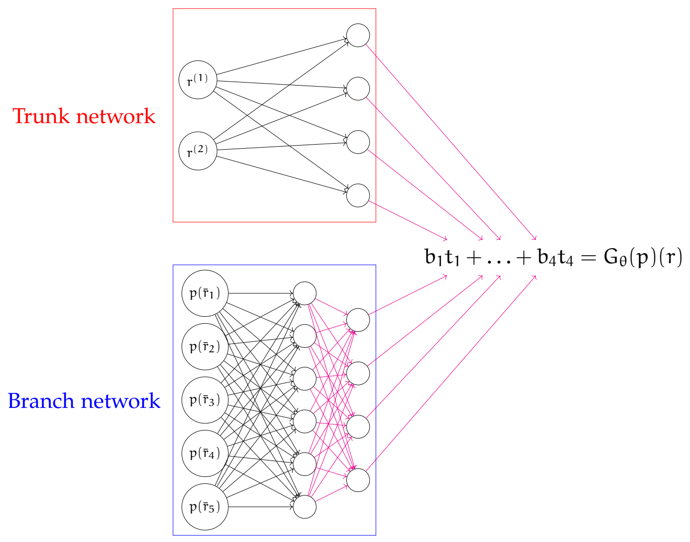


Figure 7: **DeepONet Architecture, as described in the Universal Approximation Theorem for Operators [5].** In this example the evaluation is done at position $r = (r^{(1)} \; r^{(2)})^T \in \mathbb{R}^2$, the input function p is sampled at $M = 5$ points $\bar{r}_j$, the branch network's first hidden layer has $I = 6$ neurons, and both trunk and branch network have $N = 4$ output neurons. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the non-polynomial activation function in the receiving neuron.

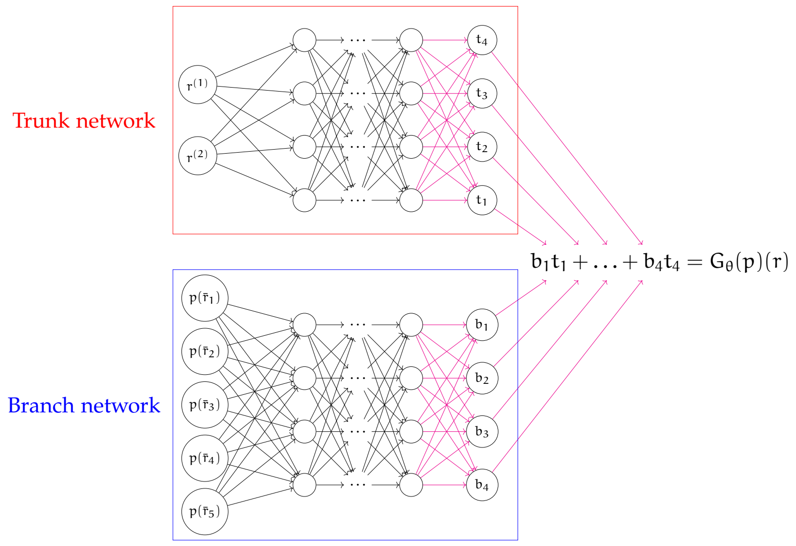


Figure 8: **General DeepONet Architecture.** In this example the evaluation is done at position $r = (r^{(1)} \; r^{(2)})^T \in \mathbb{R}^2$ and the input function p is sampled at $M = 5$ points $\bar{r}_j$. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the non-polynomial activation function in the receiving neuron.

where θ is the set of weights of the DeepONet and $\hat{p} = (p(\bar{r}_1) \; \ldots \; p(\bar{r}_M))^T$ is the input function sampled at $\bar{r}_1, \ldots, \bar{r}_M$. The general DeepONet architecture can be seen in Fig. 8.

When evaluating the DeepONet for one discretized input function $\hat{p}$ and n different evaluation coordinates $R = \{r_1, ..., r_n\}$, the output is

$$G_\theta(\hat{p})(R) = \sum_{j=1}^{N} b_j(\hat{p}) t_j(R),$$

where t_j is applied to R element-wise. Thus, the DeepONet's outputs for all discretized input functions $\hat{p}$ lie in a subspace spanned by the trunk neurons $t_j(R)$, hence the t_j's are also called basis functions, or when evaluated on fixed coordinates, basis vectors. Note that N is thus the maximum dimension of the trunk space

$$\mathcal{T}(N) := \text{span}\{t_1, \ldots, t_N\} \subset \mathcal{C}(D). \tag{6}$$

For simplicity, we term N the *dimension of the DeepONet*. Furthermore, the output neurons of the branch network $b_j(\hat{p})$ can be interpreted as the coefficients of the j-th basis function for a given discretized input function $\hat{p}$.
To summarize, the trunk network takes the evaluation coordinate r as input. The j-th trunk output neuron, as a function of r is the j-th basis function spanning the approximate solution.
The branch network takes the discretized input function $\hat{p}$, e.g., the initial condition of the time-dependent PDE, as input. The j-th branch output neuron, as a function of $\hat{p}$, is the coefficient for the j-th basis function for the given input function.
The two networks are then combined into the output of the DeepONet through an inner product, i.e., the output is given as the sum over all basis functions, each multiplied with the respective coefficient.

2.5 SETUP FOR TRAINING AND TESTING

For ease of notation and faster training we consider the following setup for our training and test data.

1. We consider PDEs with real-valued solutions, i.e., $u_p^*(r) \in \mathbb{R}$. The example problems are described in Section 3.2.

2. We choose $L \in \mathbb{N}$ and restrict the input functions p to an L-dimensional subspace of $\mathcal{C}(D)$. We now discuss this for the example

$$p(r) = \sum_{i=1}^{L} a_i \sin(i\pi r) \in \text{span}\{\sin(i\pi r)\}_{i=1}^{L} \subset \mathcal{C}([0,1]).$$

   We then sample p on a uniform mesh with M interior points $\bar{r}_j$, such that $\bar{r}_j = \frac{j}{M+1}$ for $j \in [M]$,

$$\hat{p} = (p(\bar{r}_1) \ \ldots \ p(\bar{r}_M))^T = \Psi a \in \mathbb{R}^M. \tag{7}$$

   Here $\Psi \in \mathbb{R}^{M \times L}$ contains the sine functions sampled on the same meshpoints $\bar{r}_j$ as columns, i.e., $\Psi_{ji} = \sin(i\pi\bar{r}_j)$, and $a = (a_1 \ \ldots \ a_L)^T \in \mathbb{R}^L$ is the coefficient vector. Then Ψ has full column rank whenever $M \geqslant L$. Hence the coefficients $\{a_i\}_{i=1}^{L}$ can be recovered exactly from $\hat{p}$, as $a = \Psi^+\hat{p}$ with $^+$ denoting the Moore-Penrose inverse. This yields 0 encoding error [28]. For simplicity, we thus stop distinguishing between p and $\hat{p}$.

3. We use m different discretized input functions p_j with $j \in [m]$, and for each input function we evaluate both the approximations and the true solutions on the same $n = n_{tr} = n_{te}$ equidistant coordinates r_i. Thus, we have nm data points.

We then arrange the targets in a matrix $A_{ij} = u^*_{p_j}(r_i)$ such that $A \in \mathbb{R}^{n \times m}$. Then, the DeepONet's output for all coordinates and all input functions is

$$\begin{aligned}
G_\theta(\{p_1, ..., p_m\})(\{r_1, ..., r_n\}) &= \mathop{}\limits_{j=1}^{N} t_j(\{r_1, ..., r_n\}) b_j(\{p_1, ..., p_m\}) \\
&= \mathop{}\limits_{j=1}^{N} \begin{pmatrix} t_j(r_1) \\ \vdots \\ t_j(r_n) \end{pmatrix} \begin{pmatrix} b_j(p_1) & \dots & b_j(p_m) \end{pmatrix} \\
&= \begin{bmatrix} t_1(r_1) & \dots & t_N(r_1) \\ \vdots & & \vdots \\ t_1(r_n) & \dots & t_N(r_n) \end{bmatrix} \begin{bmatrix} b_1(p_1) & \dots & b_1(p_m) \\ \vdots & & \vdots \\ b_N(p_1) & \dots & b_N(p_m) \end{bmatrix} \\
&= TB^T =: \tilde{A}, \qquad (8)
\end{aligned}$$

such that $\tilde{A}_{ij} = G_\theta(p_j)(r_i)$. Here, $T_{ij} = t_j(r_i)$ and $B_{ij} = b_j(p_i)$ are the output matrices of the trunk and branch network, respectively. Note that because of the DeepONet's evaluation at n fixed coordinates, the trunk space $\mathcal{T}(N)$, originally contained in $\mathcal{C}(D)$, can now be treated a subspace of $\mathbb{R}^n$. This setup (a) enables faster training, since the evaluation of trunk and branch network is independent – all input functions use the same evaluation of the trunk network and vice versa – and (b) the matrix notation significantly simplifies the analysis.

2.6 deriving the trunk-branch error decomposition

Since the DeepONet's output is given as a linear combination of the trunk basis functions with the branch coefficients, one might ask how the approximation error is distributed between the basis functions and the coefficients. This question is formalized and can be answered using the following error decomposition into trunk and branch error. This decomposition is a simplification and adaption of the work done by Lanthaler et al. [28] for the setup described in the previous section. We consider the difference between the DeepONet's output $\tilde{A}$ and the target data matrix A:

$$\begin{aligned}
\tilde{A} - A = TB^T - A &= TB^T - TT^+A + TT^+A - A \\
&= T(B^T - T^+A) + (TT^+ - I)A \\
\varepsilon := \|\tilde{A} - A\|_F^2 &= \underbrace{\|T(B^T - T^+A)\|_F^2}_{=:\varepsilon_B} + \underbrace{\|(TT^+ - I)A\|_F^2}_{=:\varepsilon_T}. \qquad (9)
\end{aligned}$$

Here T^+ is the Moore-Penrose inverse of T, and thus $I - TT^+$ is the projection onto the subspace orthogonal to the trunk space. Equation 9 thus defines the trunk error ε_T as the error of projecting the target data matrix onto the trunk space. The branch error ε_B is defined for a given trunk matrix as the appropriately scaled (see below) difference between the branch matrix and the, for this given trunk matrix, optimal branch matrix. In Chapter 4 we will use this decomposition to identify the bottleneck in current DeepONet performance. Hence we are interested in bounds on the error parts. Furthermore, we compute the optimal trunk and branch matrices based on this error decomposition.

2.6.1 *Trunk Error*

For a given N and a target matrix A, a trunk matrix $T_* \in \mathbb{R}^{n \times N}$ is said to be optimal, if it minimizes the trunk error ε_T. To compute an optimal trunk matrix T_*, we first introduce the singular value decomposition (SVD). Note that we state the so-called thin SVD, which will be used throughout. The full SVD is obtained by extending Φ and V to square matrices. For the thin SVD, we get *semi-orthogonal matrices*, i.e., non-square matrices that have either orthogonal rows or columns.

Theorem 3 (Thin SVD) *Consider any matrix* $A \in \mathbb{R}^{n \times m}$. *Let* $r = \min(n, m)$. *Then there exist semi-orthogonal matrices* $\Phi \in \mathbb{R}^{n \times r}, V \in \mathbb{R}^{m \times r}$, *and the diagonal matrix* $\Sigma = \mathit{diag}(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r \times r}$, *with* $\sigma_1 \geqslant \ldots \geqslant \sigma_r \geqslant 0$, *such that*

$$A = \Phi \Sigma V^T.$$

Note that the diagonal entries σ_i *of* Σ *are called singular values of* A. *Furthermore,* Φ *and* V *contain the so-called left- and right-singular vectors as columns, respectively.*

Note that the SVD of a matrix is not unique. Thus we usually consider any SVD of A, when writing $A = \Phi \Sigma V^T$. An SVD of A can be used to compute a best rank N approximation of A.

Theorem 4 (Rank N approximation) *Let* $A = \Phi \Sigma V^T$ *be an SVD of* A. *Then for any* $N < r$, *we can split* Φ, Σ *and* V *such that*

$$A = [\Phi_1 \ \Phi_2] \begin{matrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{matrix} \begin{matrix} V_1^T \\ V_2^T \end{matrix} = \Phi_1 \Sigma_1 V_1^T + \Phi_2 \Sigma_2 V_2^T, \tag{10}$$

with $\Phi_1 \in \mathbb{R}^{n \times N}, \Sigma_1 \in \mathbb{R}^{N \times N}$ *and* $V_1 \in \mathbb{R}^{m \times N}$. *Then,*

$$\Phi_1 \Phi_1^T A = \Phi_1 \Sigma_1 V_1^T \in \underset{\substack{X \in \mathbb{R}^{n \times m}, \\ \mathit{rank}(X) \leqslant N}}{\arg\min} \ ||A - X||_F^2.$$

The minimum rank N *approximation error is then*

$$\begin{aligned} \min_{\substack{X \in \mathbb{R}^{n \times m}, \\ \mathit{rank}(X) \leqslant N}} ||A - X||_F^2 &= ||A - \Phi_1 \Sigma_1 V_1^T||_F^2 \\ &= ||\Phi_1 \Sigma_1 V_1^T + \Phi_2 \Sigma_2 V_2^T - \Phi_1 \Sigma_1 V_1^T||_F^2 \\ &= ||\Phi_2 \Sigma_2 V_2^T||_F^2 = ||\Sigma_2||_F^2. \end{aligned}$$

Note that $\Phi_1 \Phi_1^T A$ *is the unique minimizer of* $||A - X||_F^2$, *if* A*'s singular values are pairwise distinct.*

Recall that the trunk error is defined as $\varepsilon_T = ||(TT^+ - I)A||_F^2$. Thus, Φ_1 is, for any SVD of A, an optimal trunk matrix. Hence, $T_* = \Phi_1$ yields a lower bound on the trunk error, i.e., for a given N, the trunk error ε_T of any DeepONet with inner dimension N is bounded from below by the SVD truncation error $\varepsilon_{SVD} = \|(\Phi_1\Phi_1^T - I)A\|_F^2$:

$$\varepsilon_T \geqslant \varepsilon_{SVD}.$$

Note that the optimal trunk matrix is not uniquely determined by A and N. For an SVD of A, we denote the space spanned by the first N left-singular vectors of A as *SVD space*

$$S(N) = \text{span}\{\phi_1, \ \ldots, \ \phi_N\}.$$

If the singular values of A are all pairwise distinct, then for any N, all SVDs of a matrix A yield the same SVD space $S(N)$. This is always the case in our example problems. Since the trunk matrix T only enters the trunk error via the matrix TT^+, which is projecting A onto T's column space, any matrix whose column space is $S(N)$ is an optimal trunk matrix, for this N. Thus, for any full-rank matrix $C \in \mathbb{R}^{N\times N}$, the matrix $\Phi_1 C$ is an optimal trunk matrix.

2.6.1.1 *Projection Error*

As described, ε_T and ε_{SVD} measure the capability of approximating the data matrix A through a projection onto the trunk space $\mathcal{T}(N)$ and the SVD space $S(N)$, respectively. Since the SVD space is the optimal space to approximate the data matrix through projections, we can also discuss the trunk space as approximating the SVD space, instead of comparing the (projection-based) approximation capabilities of both spaces. To later investigate which parts of the SVD space are well approximated by the trunk space, we define the *projection error* for a trunk matrix $T \in \mathbb{R}^{n\times N}$ as

$$\Delta(i, N) := \|\phi_i - TT^+\phi_i\|_2^2.$$

The projection error $\Delta(i, N)$ thus computes the error of approximating ϕ_i by projecting it onto the trunk space of a DeepONet with inner dimension N.

2.6.2 *Branch Error*

For a given trunk matrix T, the matrix B_* containing the target coefficients, i.e., the branch matrix such that the data matrix A is approximated best in the Frobenius norm by TB_*^T, is

$$\begin{aligned} B_* &= \underset{B\in\mathbb{R}^{m\times N}}{\arg\min} \|A - TB^T\|_F^2 = \underset{B\in\mathbb{R}^{m\times N}}{\arg\min} \|(I - TT^+)A + TT^+A - TB^T\|_F^2 && (11) \\ &= \underset{B\in\mathbb{R}^{m\times N}}{\arg\min} \left(\|(I - TT^+)A\|_F^2 + \|T(T^+A - B^T)\|_F^2\right) && (12) \\ &= \underset{B\in\mathbb{R}^{m\times N}}{\arg\min} \|T(T^+A - B^T)\|_F^2 = (T^+A)^T. && (13) \end{aligned}$$

Thus, when investigating the branch network, one might intuitively define the branch error as

$$\varepsilon_C := \|B - (T^+A)^T\|_F^2 = \|B^T - T^+A\|_F^2, \qquad (14)$$

i.e., the difference between the actual branch matrix B and the optimal branch matrix B_*. Using ε_C, one can derive the inequality

$$\varepsilon \leqslant ||T||_2^2 \underbrace{||B^T - T^+A||_F^2}_{\varepsilon_C} + ||(TT^+ - I)A||_F^2 \tag{15}$$

as a bound on ε. While ε_C puts the same weight on the error of each branch neuron, $\varepsilon_B = ||T(B^T - T^+A)||_F^2$ directly includes the trunk matrix T, which weights the approximation error of each branch neuron (column of B) with the norm of the corresponding trunk neuron. In Equation 15 the trunk matrix's spectral norm $||T||_2^2$, which is an upper bound on the maximum trunk neuron norm, is used. Thus Equation 15 yields an inequality. The weighting with the true corresponding neuron norms in Equation 9 yields an equality. Hence we use ε_B. An obvious lower bound on the branch error cannot be derived.
Note that we derived B_* as the branch matrix minimizing the approximation error $\varepsilon = ||A - TB^T||_F^2$ for a given trunk matrix T. However, since $\varepsilon = \varepsilon_T + \varepsilon_B$, where ε_T is independent of B, a branch matrix minimizing ε is equivalent to a branch matrix minimizing ε_B. This is formally described in Equations 11 - 13.

2.6.3 *General Remarks*

Note the relation between the (absolute squared) error $\varepsilon = ||A - \tilde{A}||_F^2$ used in derivations, the relative error $\delta = \frac{\sqrt{\varepsilon}}{||A||_F} = \frac{||A - \tilde{A}||_F}{||A||_F}$ used as an intuitively meaningful performance indicator, and the loss $\mathcal{L} = \frac{1}{nm}\varepsilon = \frac{1}{nm}||A - \tilde{A}||_F^2$ used in the training of neural networks. Additionally, we also use the relative SVD truncation error $\delta_{SVD} = \frac{\sqrt{\varepsilon_{SVD}}}{||A||_F}$ and the relative partial errors $\delta_T = \frac{\sqrt{\varepsilon_T}}{||A||_F}$ and $\delta_B = \frac{\sqrt{\varepsilon_B}}{||A||_F}$, such that $\delta^2 = \delta_T^2 + \delta_B^2$.
For readers interested in error decomposition of DeepONets in a more general setting, we recommend the work by Lanthaler et al. [28]. They consider the more general case in which p cannot necessarily be fully reconstructed from $\hat{p}$, which adds an encoding error. Moreover, the error in their work is defined with respect to a probability measure over the sample space, rather than solely over the finite set of sampled points.