Mode-Decomposition in DeepONets

Generalization and Coupling Analysis

Julius Johannes Taraz

Mode-Decomposition in DeepONets

Generalization and Coupling Analysis

Julius Johannes Taraz

to obtain the degree of Master of Science in Applied Mathematics
at the Delft University of Technology,
to be defended publicly on August 27, 2025 at 09:00.

Student Number:	6065279	
Project Duration:	November 13, 2024 – August 27, 2025	
Thesis committee:	Dr. A. Heinlein	TU Delft, supervisor
	Dr. ir. H. M. Schuttelaars	TU Delft, co-supervisor

An electronic version of this thesis is available at
http://repository.tudelft.nl/.

ABSTRACT

Operator learning promises to revolutionize scientific computing by learning solution operators for differential equations directly from data, potentially accelerating tasks like design optimization and uncertainty quantification by orders of magnitude. The deep operator network (DeepONet), the first practical architecture for operator learning, consists of a trunk and branch network. Its output is given by a linear combination of basis functions, where the functions are learned by the trunk network and the coefficients are learned by the branch network. However, despite their theoretical promise, DeepONets suffer from poor accuracy compared to classical numerical solvers, limiting their practical adoption. Understanding and addressing these accuracy limitations is crucial for advancing the field.

In this thesis, we first analyze the performance limitations of the classical DeepONet. We demonstrate that for many classical examples, the trunk network's error is much smaller than the total approximation error. Thus, the space spanned by the basis functions contains functions which approximate the true solutions well. The total approximation error is dominated by the branch network's error, i.e., the error of the coefficients.

To investigate this further, we construct a modified DeepONet. In this modification we replace the learnable trunk network with optimal basis vectors (modes) derived from a singular value decomposition (SVD). This modification is called SVD-based operator network (SVDONet). This simplification enables us to decompose the total error into mode-specific contributions, revealing how the coefficients of different spatial modes are approximated. Our mode decomposition analysis yields several key insights.

First, we discover that for some modes a low training error does not necessarily correspond to a low test error, i.e., the coefficients learned for these modes do not generalize well.

Second, we show that architectural choices profoundly impact generalization: the standard "unstacked" DeepONet architecture, where all modes share hidden neurons, significantly improves generalization for modes corresponding to small singular values at the cost of the modes with large singular values.

Third, we study how improving the coefficients of one mode impacts the coefficients of other modes. Here, we show fundamental differences between different optimization algorithms.

These findings establish mode decomposition as a powerful lens for analyzing neural operators, revealing that the success of operator learning hinges not on learning all modes equally well, but on the delicate balance between mode prioritization, architectural coupling, and optimization dynamics.

ACKNOWLEDGMENTS

I would like to thank Dr. Alexander Heinlein for his excellent supervision and valuable ideas, as well as for giving me the freedom to pursue my research interests. Furthermore, I am thankful that he endured my many explanations of which matrix I computed the SVD for this time and the (surely) spectacular insights it would give us.
I would also like to thank Dr. Henk Schuttelaars for his insightful remarks and suggestions regarding the research, and the last-minutes drafts he patiently read and helped improve.
I am very thankful for everyone I met in Delft who made this a very enjoyable year altogether. I am especially thankful for the people with whom I've spent so much time in the COSSE office:
I want to thank Alessandro for the excellent infotainment he provided and his shared love for the SVD. I want to thank Ezra and Guillaume for the *gezelligheid*. I want to thank Hugo for his expert judgment on questions of all matters and his initiative in suggesting we study together. I want to thank Javier for his brief, but intense, guidance in infinite dimensions. I want to thank Sinan for his company, making the summer much nicer.
Furthermore, I thank my friends and family for continuing to listen to my problems. I especially thank my parents for their ongoing support.
Special thanks go to my brother Martin and my girlfriend Katerina for the hours they have invested in making this a readable (and hopefully even understandable) thesis. Thank you!

Finally, I thank Eugenio Beltrami and Camille Jordan for discovering the SVD! What would we do without it?

I acknowledge using the Claude large language model to assist with the proofreading and language revision of this thesis. However, all ideas, analyses, and conclusions are my own work.

CONTENTS

1 INTRODUCTION 1
- 1.1 Background 1
- 1.2 Research Objectives 2
- 1.3 Structure 2

2 BACKGROUND 5
- 2.1 Problem Statement 5
- 2.2 Universal Approximation Theorem for Functions 6
- 2.3 Foundations of Neural Networks and Machine Learning 7
- 2.4 DeepONet 13
- 2.5 Setup for Training and Testing 16
- 2.6 Deriving the Trunk-Branch Error Decomposition 17

3 EXAMPLE PROBLEMS 21
- 3.1 Overview 21
- 3.2 Details 21
- 3.3 Spectral Properties of the Data 23

4 APPLYING THE TRUNK-BRANCH ERROR DECOMPOSITION 29
- 4.1 Error of the Trunk Network 29
- 4.2 Error of the Branch Network 30

5 SVDONET 33
- 5.1 Defining the SVDONet 33
- 5.2 Deriving the Mode-Based Error Decomposition 35
- 5.3 Choice of Trunk Matrix 37
- 5.4 Related Work 37

6 APPLYING THE MODE-BASED ERROR DECOMPOSITION 43
- 6.1 SVDONet Mode Loss Distribution 43
- 6.2 Coupling Between Different Modes 49

7 DISCUSSION 63
- 7.1 Summary 63
- 7.2 Future Work 64
- 7.3 Conclusion 65

BIBLIOGRAPHY 67

A APPENDIX 73
- A.1 Hyperparameters 73
- A.2 Characterizing the Spectrum of Multivariate Functions 75
- A.3 Total Variation Norm 75
- A.4 Laplacian Energy 75
- A.5 Fourier Transform of Data Projected onto Low-Dimensional Subspaces 76
- A.6 Comparison of SVDONet and POD-DeepONet 78
- A.7 SVDONet's Trunk Error for Test Data 80
- A.8 Comparison of Stacked and Unstacked SVDONets 81

Table 1: Notation frequently used in this thesis

Symbol	Name	Note
$[k]$	Set of indices	$[k] = \{1, 2, ..., k\}$
D	Spatial domain of interest	$D \subset \mathbb{R}^d$
$\mathcal{C}(D)$	Set of continuous functions with domain D	
p	Input function (e.g. initial condition of time-dependent PDE)	$p \in \mathcal{C}(D)$
G_*	True solution operator	
u_p^*	True solution $G_*(p)$ for input function p	$u_p^* \in \mathcal{C}(D)$
G_θ	Model / approximation of G_*	
u_p	Approximate solution $G_\theta(p)$ for input function p	$u_p \in \mathcal{C}(D)$
r	Coordinate at which u_p and u_p^* are evaluated	$r \in D$
n	Number of coordinates to evaluate u_p and u_p^*	
m	Number of input functions in a data set	
M	Number of sampling points of p	
$\bar{r}_j$	j-th sampling point of p with $j \in [M]$	$\bar{r}_j \in D$
$\hat{p}$	Discretized input function (sampled at $\bar{r}_j$)	e.g., Eq. 7
N	Inner dimension of DeepONet / number of output neurons of branch and trunk net	
A	Target data matrix	$\in \mathbb{R}^{n \times m}$
$\tilde{A}$	Approximation of A by the DeepONet	$\in \mathbb{R}^{n \times m}$
X	Either training or test variant (of any variable X)	
X_{tr}	Training variant of X	
X_{te}	Test variant of X	
$\mathcal{L}$	Loss / mean-squared error of DeepONet approximation	
L_i	Loss of mode i	
ε	Absolute squared error of DeepONet approximation	
δ	Relative error of DeepONet approximation	
Z^+	Moore-Penrose inverse of matrix Z	
$\|Z\|_F^2$	Frobenius norm	$\sum_{ij} Z_{ij}^2$

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)

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

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 $\bar{r}_1, ..., \bar{r}_M$:

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

This involves a loss of information, which is addressed in Section 2.5. We then build a parametric model $G_\theta : \hat{p} \mapsto u_p$, 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.

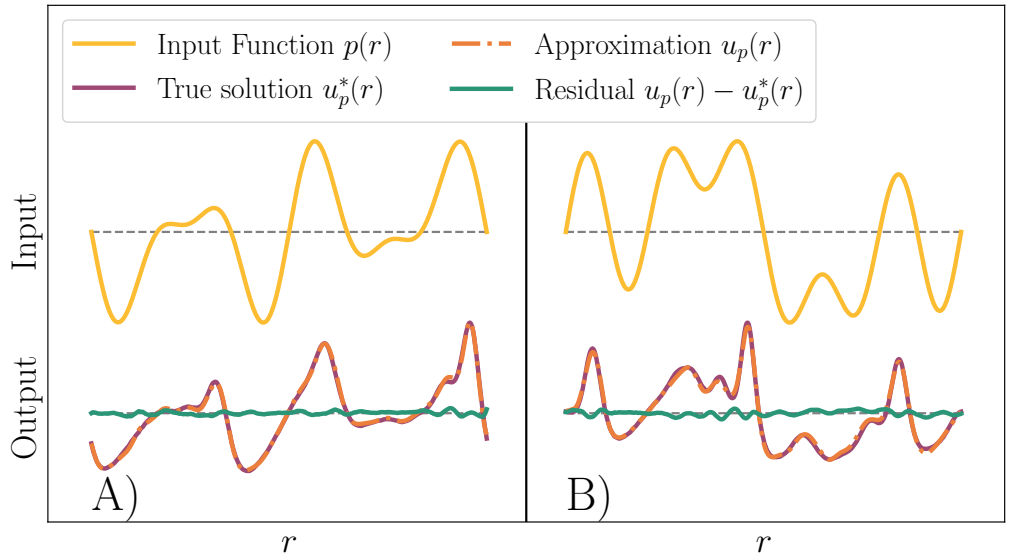


Figure 1: **Input and Output of DeepONet for the KdV Equation with $\tau = 0.2$. Left (A):** The left input function is part of the training data set. **Right (B):** The right input function is not part of the training data set. The yellow line shows the input function $p(r)$. The purple line shows the 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

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

In neural network literature, the mean-squared error

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

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

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

as a model. The parameters of this model are called

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

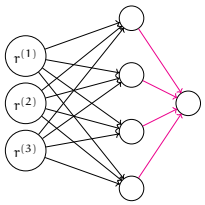


Figure 2: **One-Layer Perceptron.** The input is $r = \left(r^{(1)}\ r^{(2)}\ r^{(3)}\right)^T \in \mathbb{R}^3$. It has $N = 4$ hidden neurons. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the non-polynomial activation function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

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

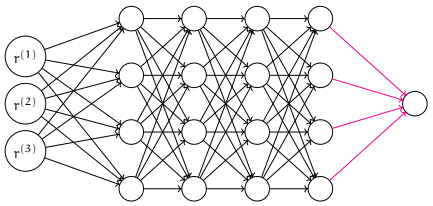


Figure 3: **Multi-Layer Perceptron.** It has input $r = \left(r^{(1)}\ r^{(2)}\ r^{(3)}\right)^T \in \mathbb{R}^3$, $D = 4$ hidden layers and $w = 4$ neurons per layer. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the non-polynomial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

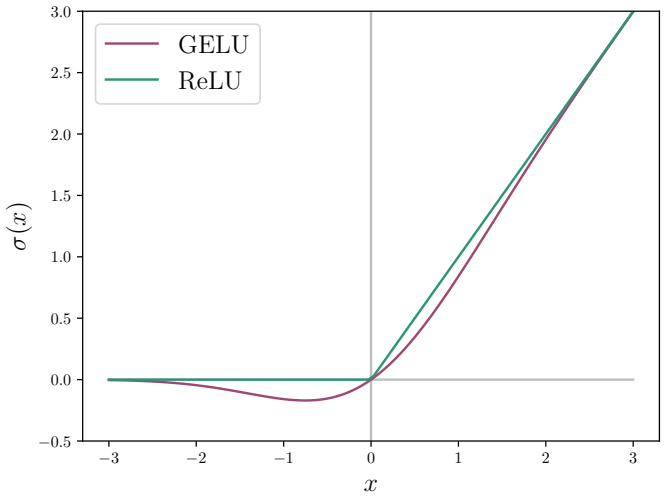


Figure 4: **GELU (purple) and ReLU (green) Activation Functions.** Reproduced and adapted from [18].

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

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

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 $v_t \in \mathbb{R}^{|\theta|}$ through its entries $(v_t)^{(i)}$:

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

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

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

This can also be written as

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

with the square and division applied component-wise. For brevity of notation we hereafter do not distinguish between θ and Θ. The normalization with v_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.,

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

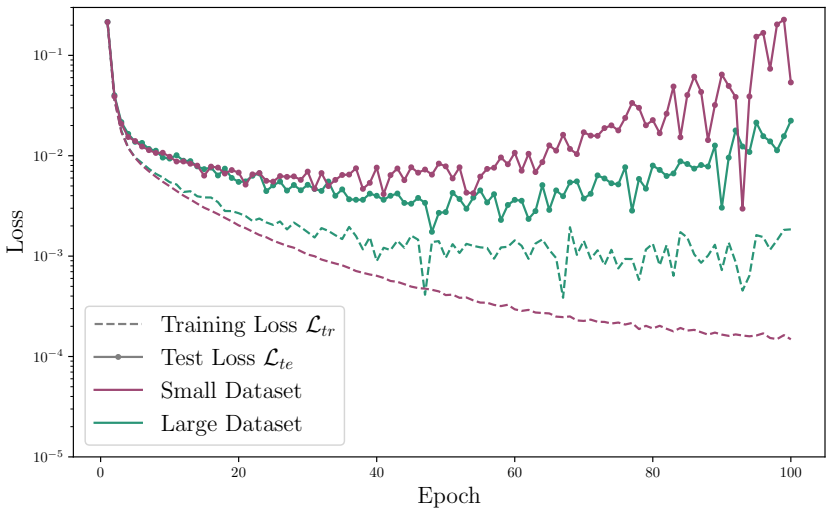


Figure 5: **Training and Test Loss of Stereotypical Neural Networks over the training course.** The two neural networks have the same architecture/size and different data sets. Adapted from [15].

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

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

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

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

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

$$b_i(g) = \sqrt{2} \int_0^1 g(r) \sin(2\pi i r) dr.$$

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 DEEPONET

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

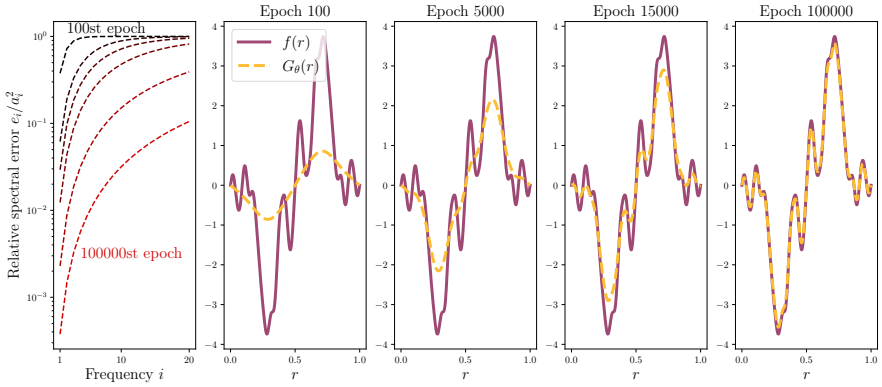


Figure 6: **Spectral Bias in Neural Networks for Function Approximation. Left panel:** Relative spectral errors e_i/a_i^2 for multiple epochs. Color indicates training progress, from black (early) to red (late epochs). **Remaining panels (2-5):** True solution function f (purple) and neural network approximation 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 $X \subset C(K_1)$, a compact set $K_2 \subset \mathbb{R}^d$ and a continuous operator $G_* : X \to C(K_2)$. 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*

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

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 $b_k([p(\bar{r}_1),\ \ldots,\ p(\bar{r}_M)])$. 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

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

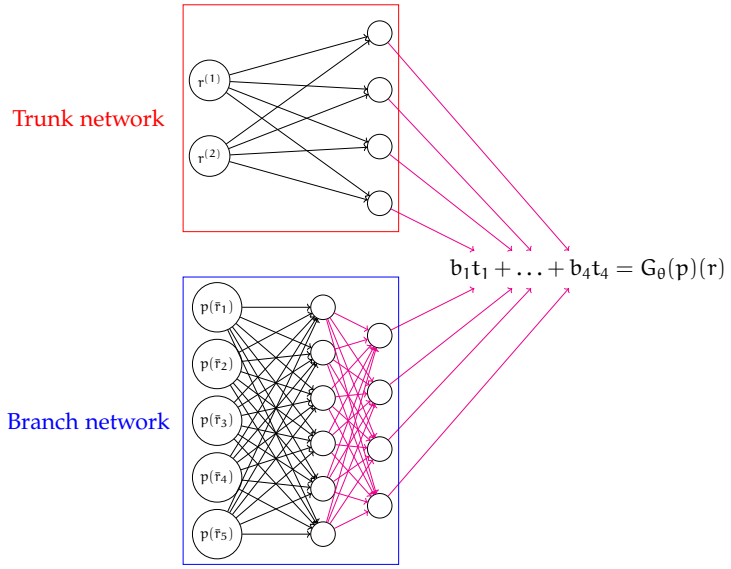


Figure 7: **DeepONet Architecture, as described in the Universal Approximation Theorem for Operators [5].** In this example the evaluation is done at position $r = (r^{(1)}\ r^{(2)})^T \in \mathbb{R}^2$, the input function p is sampled at $M = 5$ points $\bar{r}_j$, the branch network's first hidden 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.

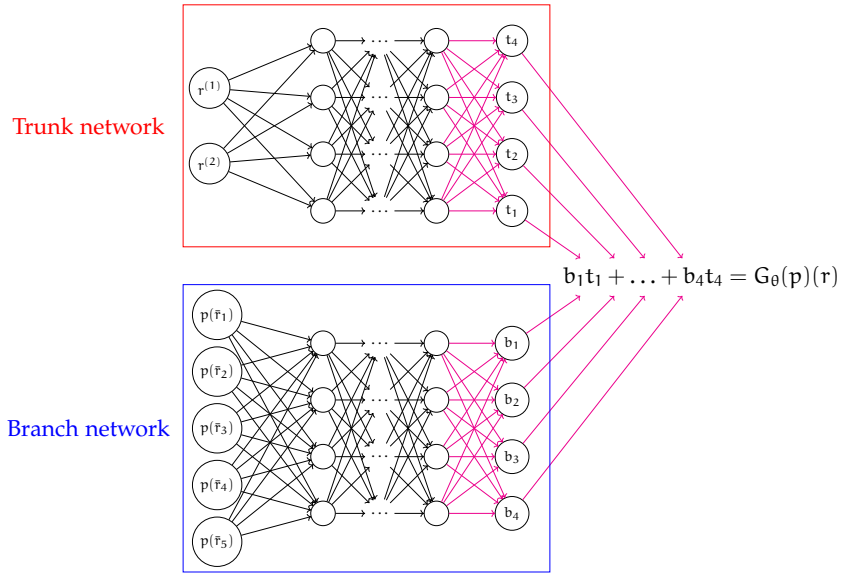


Figure 8: **General DeepONet Architecture.** In this example the evaluation is done at position $r = (r^{(1)}\ r^{(2)})^T \in \mathbb{R}^2$ and the input function p is sampled at $M = 5$ points $\bar{r}_j$. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping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

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

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

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

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 mesh-points $\bar{r}_j$ 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

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

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:

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

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 $\Sigma = diag(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r\times r}$, with $\sigma_1 \geqslant ... \geqslant \sigma_r \geqslant 0$, 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*

$$A = [\Phi_1\ \Phi_2]\begin{bmatrix}\Sigma_1 & 0\\ 0 & \Sigma_2\end{bmatrix}\begin{bmatrix}V_1^T\\ V_2^T\end{bmatrix} = \Phi_1\Sigma_1V_1^T + \Phi_2\Sigma_2V_2^T, \tag{10}$$

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,

$$\Phi_1\Phi_1^TA = \Phi_1\Sigma_1V_1^T \in \underset{\substack{X\in\mathbb{R}^{n\times m},\\ rank(X)\leqslant N}}{\arg\min}\ \|A - X\|_F^2.$$

The minimum rank N approximation error is then

$$\begin{aligned}\min_{\substack{X\in\mathbb{R}^{n\times m},\\ rank(X)\leqslant N}} \|A - X\|_F^2 &= \|A - \Phi_1\Sigma_1V_1^T\|_F^2\\ &= \|\Phi_1\Sigma_1V_1^T + \Phi_2\Sigma_2V_2^T - \Phi_1\Sigma_1V_1^T\|_F^2\\ &= \|\Phi_2\Sigma_2V_2^T\|_F^2 = \|\Sigma_2\|_F^2.\end{aligned}$$

Note that $\Phi_1\Phi_1^TA$ 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

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

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.

3 EXAMPLE PROBLEMS

In this chapter, we first describe the example problems considered in this thesis and how test and training data are generated. We then investigate the spectral properties of this data, as this will play a central role in the analysis presented in later chapters.

3.1 OVERVIEW

In this thesis, we train DeepONets (and SVDONets) to approximate the time evolution operators of the advection-diffusion equation, the Korteweg-de Vries (KdV) equation and Burgers' equation, which are given below. The time evolution operator's input function $p(r)$ is the PDE's initial condition $u(r, t = 0)$. The initial condition is mapped to the PDE's solution at the *evolution time* τ, i.e.,

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

In the following, an *example problem* denotes a time-dependent PDE with certain boundary and initial conditions together with a fixed evolution time τ. Thus,

> *the KdV equation (Equation 16) with the given boundary conditions (Equations 17–20), the given distribution of initial conditions (Equation 21) and evolution time* $\tau = 0.2$

describes one example problem. This is, in fact, the standard problem considered in this thesis. I.e., unless stated otherwise, all results shown in this thesis are computed for this standard example problem. However, unless stated otherwise, the observations reported apply qualitatively to all example problems described in this chapter.
For each example problem, the training and test data set contain 900 and 100 input functions, respectively. As discussed, both test and training input functions are drawn from the same distribution.

3.2 DETAILS

3.2.1 *Advection-Diffusion Equation*

We consider the advection-diffusion equation

$$\begin{aligned}
0 &= \frac{\partial u}{\partial t} + 4\frac{\partial u}{\partial r} - 0.01\frac{\partial^2 u}{\partial r^2} && \text{for all } r \in (0, 2\pi), t > 0,\\
u(0, t) &= u(2\pi, t) && \text{for all } t \geqslant 0,\\
\frac{\partial u}{\partial r}(0, t) &= \frac{\partial u}{\partial r}(2\pi, t) && \text{for all } t \geqslant 0,\\
u(r, t = 0) &= p(r) && \text{for all } r \in (0, 2\pi),
\end{aligned}$$

with evolution times $\tau = 0.5$ and $\tau = 1.0$. Thus, we consider two example problems based on the advection-diffusion equation. The input functions $p(r)$ are generated as

$$p(r) = \sum_{i=1}^{20} a_i \sin(ir).$$

All coefficients a_i are drawn from a uniform distribution $a_i \sim \mathcal{U}([-1, 1])$. Test and training data are obtained using the finite-difference method combined with a Runge-Kutta solver [59]. Due to the linearity of the advection-diffusion equation, the discretized solution operator can be written as a matrix $\hat{G}_* \in \mathbb{R}^{n \times M}$, such that $u^*_{\hat{p}} = \hat{G}_* \hat{p}$. Since (a) the considered input space is 20-dimensional, and (b) the 20 discretized trigonometric functions $[\sin(i\bar{r}_1) \ \dots \ \sin(i\bar{r}_M)]^T \in \mathbb{R}^M$ are mapped to 20 linearly independent vectors by $\hat{G}_*$, the output space is 20-dimensional as well. This furthermore implies that both the training and test data matrix A_{tr} and A_{te} have the same 20-dimensional column spaces. Thus, we use DeepONets of at most inner dimension $N = 20$ to approximate the advection-diffusion equation's solution operator.
Note that, for the basis of the input functions described here, we have to sample the input functions at $M \geqslant 2L$ interior meshpoints, where L is the number of basis functions (i.e., $L = 20$). We choose $n = M = 200$.

3.2.2 *Korteweg–de Vries Equation*

We consider the KdV equation

$$0 = \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} + 0.01\frac{\partial^3 u}{\partial r^3} \qquad \text{for all } r \in (0, 2\pi), t > 0, \tag{16}$$

$$u(0, t) = u(2\pi, t) \qquad \text{for all } t \geqslant 0, \tag{17}$$

$$\frac{\partial u}{\partial r}(0, t) = \frac{\partial u}{\partial r}(2\pi, t) \qquad \text{for all } t \geqslant 0, \tag{18}$$

$$\frac{\partial^2 u}{\partial r^2}(0, t) = \frac{\partial^2 u}{\partial r^2}(2\pi, t) \qquad \text{for all } t \geqslant 0, \tag{19}$$

$$u(r, t = 0) = p(r) \qquad \text{for all } r \in (0, 2\pi), \tag{20}$$

with evolution times $\tau = 0.2, \tau = 0.6$ and $\tau = 1.0$. Thus, we consider three example problems based on the KdV equation. The input functions $p(r)$ are generated as

$$p(r) = \sum_{i=1}^{5} a_i \sin(ir). \tag{21}$$

All coefficients a_i are drawn from a uniform distribution $a_i \sim \mathcal{U}([-1, 1])$. Test and training data are obtained using the finite-difference method combined with a Runge-Kutta solver [59]. We use a mesh with $n = 400$ points, and observe $\operatorname{rank}(A_{tr}) = 400$, for all considered values of τ. In Chapter 4 we investigate the approximation errors of DeepONets with inner dimensions up to $N = 100$ and observe the diminishing effect of growing N. To reduce computational complexity, we thereafter consider DeepONets of inner dimension $N = 50$ to approximate the KdV equation. As this will become relevant in a later chapter, we highlight that the standard basis vectors of $\mathbb{R}^{400}$ are a basis of A_{tr}'s column space $\operatorname{Col}(A_{tr})$. This implies $\operatorname{Col}(C) \subset \operatorname{Col}(A_{tr})$, for any matrix $C \in \mathbb{R}^{400 \times l}$ with any $l \in \mathbb{N}$. Thus, in particular $\operatorname{Col}(A_{te}) \subset \operatorname{Col}(A_{tr})$ for the test data matrix A_{te}.

Note that we use the same mesh for the solution and input functions, i.e., $n = M = 400$.

3.2.3 *Burgers' Equation*

We consider Burgers' equation

$$\begin{aligned} 0 &= \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} - 0.01\frac{\partial^2 u}{\partial r^2} && \text{for all } r \in (0,1), t > 0, \\ u(0,t) &= 0 && \text{for all } t \geqslant 0, \\ u(1,t) &= 0 && \text{for all } t \geqslant 0, \\ u(r, t=0) &= p(r) && \text{for all } r \in (0,1), \end{aligned}$$

with evolution times $\tau = 0.1$ and $\tau = 1.0$. Thus, we consider two example problems based on Burgers' equation. The input functions $p(r)$ are generated as

$$p(r) = \sum_{i=1}^{5} a_i \sin(\pi i r).$$

All coefficients a_i are drawn from a uniform distribution $a_i \sim \mathcal{U}([-1,1])$. Test and training data are obtained using a spectral solver with 100 basis functions combined with an explicit Euler scheme. Due to the spectral method with 100 basis functions, the output matrix A for Burgers' equation, for all considered values of τ, has rank 100. In Chapter 4 we investigate the approximation errors of DeepONets with inner dimensions up to $N = 100$ and observe the diminishing effect of growing N. To reduce computational complexity, we thereafter consider DeepONets of inner dimension $N = 50$ to approximate Burgers' equation. As this will become relevant in a later chapter, we highlight that the span of the discretized basis functions is equal to the column space of both the test and training data matrices. Note that we choose $n = 200$ and $M = 50$.

3.3 SPECTRAL PROPERTIES OF THE DATA

To later discuss the spectral bias in DeepONets we now examine the spectral properties of the SVD of the data.
Recall that the training data matrix A admits the SVD $A = \Phi\Sigma V^T$. Furthermore, recall that (a) the i-th rows of A and Φ correspond to the i-th evaluation coordinate $r_i \in \mathbb{R}$ and (b) the j-th columns of A and V^T correspond to the j-th discretized input function $p_j \in \mathbb{R}^M$. The k-th columns of Φ and V correspond to the k-th singular triplet (σ_k, ϕ_k, v_k).
We can then define the *left- and right-singular functions* λ_k and ρ_k, which correspond to the k-th left- and right-singular vectors, respectively. Or, more formally; functions λ_k, ρ_k which satisfy

$$\begin{aligned} \lambda_k &: r_i \in \mathbb{R} \mapsto \Phi_{ik} \in \mathbb{R} \\ \rho_k &: p_j \in \mathbb{R}^M \mapsto V_{jk} \in \mathbb{R} \end{aligned}$$

are called left- and right-singular functions, respectively. We now investigate the spectral properties of these functions. To define the frequency of a function, we first introduce the Fourier transform $\mathcal{F}_c$. The subscript c distinguishes the continuous (standard) Fourier transform from the discrete Fourier transform used later.

Consider a function $g \in L^2(D)$ with $D \subset \mathbb{R}^d$, we then define its Fourier transform at a wave-vector $\xi \in \mathbb{R}^d$ as

$$\mathcal{F}_c(g)(\xi) = \int_D f(x) \exp(-2\pi i \xi^T x) dx.$$

The (mean) *frequency* $f \in \mathbb{R}$ of a function g is then defined as

$$f(g) = \frac{\int_{\mathbb{R}^d} |\mathcal{F}_c(f)(\xi)|^2 \|\xi\|_2 d\xi}{\int_{\mathbb{R}^d} |\mathcal{F}_c(f)(\xi)|^2 d\xi}.$$

For the left- and right-singular functions we are interested in comparisons such as

- λ_i *has a higher mean frequency than* λ_j, i.e., $f(\lambda_i) > f(\lambda_j)$, or
- ρ_k *has a lower mean frequency than* ρ_l, i.e., $f(\rho_k) < f(\rho_l)$.

We are neither interested in

- in the entire spectrum $\mathcal{F}_c(\cdot)(\xi)$ of λ_k and ρ_k nor
- in absolute statements like λ_k's mean frequency.

Of course, if they can be computed in practice, the mean frequencies of λ_i and λ_j immediately provide the comparison $f(\lambda_i) \lessgtr f(\lambda_j)$. Note that since λ_k takes the evaluation coordinate as input, its oscillations are spatial oscillations. The oscillations of ρ_k correspond to oscillations in $\mathbb{R}^M$, which is the discretized space of input functions. These oscillations should not be confused with spatial oscillations in the input function $p(r)$.
Moreover, it is important to note that the spectral observations reported here are specific to the example problems under study and may not generalize to other settings. In particular, for linear PDEs with well-studied discretizations the relationship between the frequency of λ_k and the index k can be prescribed almost arbitrarily through the choice of the distribution from which the initial conditions p_i are drawn; see Section 5.4.4.1. As a consequence, example problems arising from different initial conditions or from different PDEs may exhibit markedly different spectral structures. Nevertheless, to investigate the impact of the herein studied dataset's spectral characteristics on the approximation behavior of DeepONets, it is relevant to analyze their spectral characteristics in detail.
Note that the k-th singular triplet is also referred to as the k-th mode.

3.3.1 *Left-Singular Function*

ϕ_j contains the evaluations of λ_j on equidistant coordinates, since we evaluate the true solution functions and their approximations on equidistant coordinates. Therefore, the discrete Fourier transform can be applied to ϕ_j to estimate the spectrum of λ_j. Furthermore, we thus identify the left-singular value ϕ_j with the function λ_j, similar to the identification between the input function p_i and its discretization $\hat{p}_i$. However, recall that p_i can be reconstructed from $\hat{p}_i$, since the space from which p_i is drawn is known. This is not the case for λ_j and ϕ_j, since λ_j is only defined via ϕ_j. We thus compute the mean frequency of λ_j

$$f_j = \frac{\sum_f |\mathcal{F}(\phi_j)(f)|^2 f}{\sum_f |\mathcal{F}(\phi_j)(f)|^2}.$$

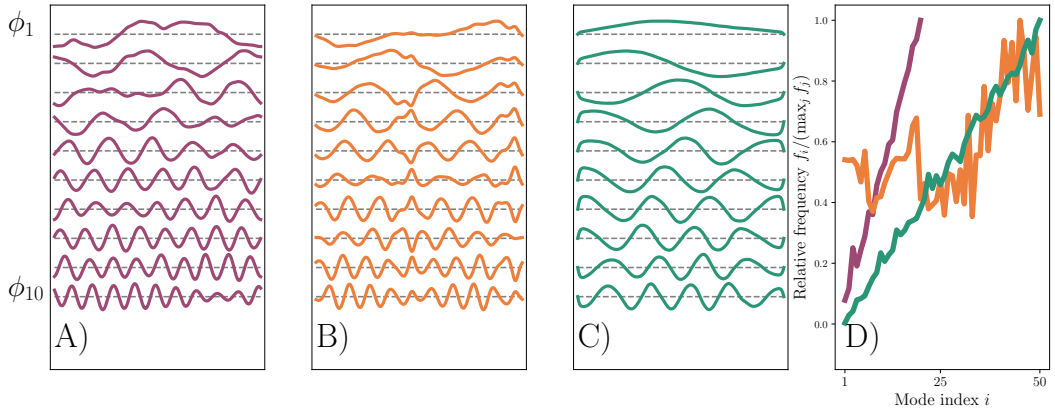


Figure 9: **Spectral properties of ϕ_i, and equivalently of the left-singular functions λ_i. A, B, C):** Left-singular vectors ϕ_i. **D)** Relative mean frequency $f_i/(\max_j f_j)$ of ϕ_i. In all plots purple corresponds to the advection-diffusion equation with $\tau = 0.5$, orange corresponds to the KdV equation with $\tau = 0.2$ and green corresponds to Burgers' equation with $\tau = 0.1$.

Here $\mathcal{F}(\phi_j)(f)$ is the discrete Fourier transform of the signal ϕ_j evaluated at frequency f.
Figure 9 shows both the mean relative frequencies f_j and the left-singular vectors ϕ_j for three different example problems, based on three different PDEs. For the advection-diffusion equation and Burgers' equation the growth of the mean frequency f_j is almost monotonic as j increases. The mean frequency of the left-singular vectors of the KdV equation fluctuates much more. However, the frequencies for $j \leqslant 30$ are on average significantly lower than the ones for $j > 30$. Note that the dominant frequency

$$f_{j,dom} := \arg\max_f |\mathcal{F}(\phi_j)(f)|^2$$

and the median frequency

$$f_{j,med} := \arg\min_{f_*} \left| \sum_{f<f_*} |\mathcal{F}(\phi_j)(f)|^2 - \frac{1}{2} \sum_f |\mathcal{F}(\phi_j)(f)|^2 \right|$$

show the same trends as the mean frequency. We thus conclude that for our example problems the frequency of ϕ_j tends to increase as j increases.

3.3.2 *Right-Singular Function*

We now discuss the spectral properties of $\rho_k : p_j \in \mathbb{R}^M \mapsto V_{jk} \in \mathbb{R}$. Since (a) ρ_k has multi-dimensional input and (b) is sampled at irregular locations, computing its Fourier transform presents significant challenges [58]. Thus, we explore different methods to characterize ρ_k's spectrum. We employ the total variation (TV) norm, a technique from image denoising [14, 45], the Laplacian energy (LE), a technique from spectral graph theory [47], and the Fourier transform of the data projected onto low-dimensional subspaces (Projection) [58]. All three methods are described in detail in Appendix A.2. As discussed there, the computations of both the TV norm and the LE rely on a k-nearest neighbors algorithm. k thus denotes the number of considered neighbors in this section. For ease of notation, we denote

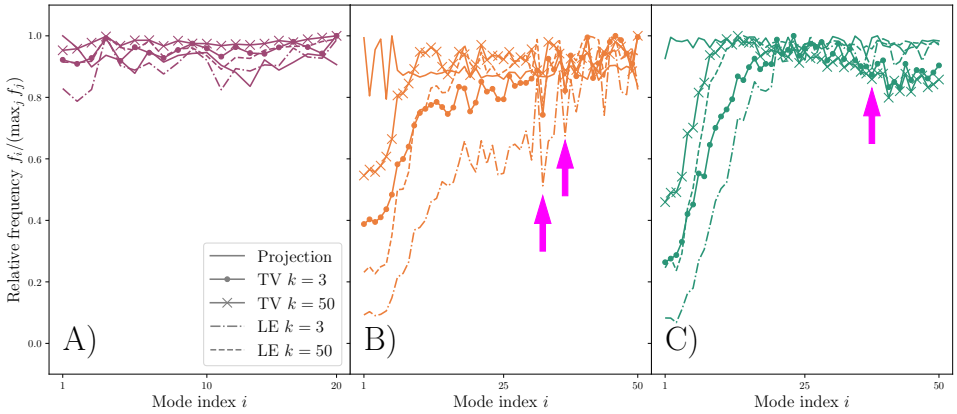


Figure 10: **Spectral properties of ρ_i, and likewise the right-singular vectors v_i. Left (A):** advection-diffusion equation with $\tau = 0.5$. **Center (B):** KdV equation with $\tau = 0.2$. **Right (C):** Burgers' equation with $\tau = 0.1$. The solid line shows the frequency estimated by projecting the data on the left-singular vectors of the input matrix (Projection). The solid lines with circle and cross markers show the frequency estimated via the total variation (TV) norm, with $k = 3$ and $k = 50$, respectively. The dash-dotted and dashed lines show the frequency estimated via the Laplacian energy (LE), with $k = 3$ and $k = 50$, respectively. The pink arrows show some modes for which the TV norm and the LE methods all identify frequency dips.

the outputs of all methods as frequencies, despite the fact that the TV norm and the LE technically do not compute the frequency, but rather a related quantity. To compare the estimations of these different methods, we use the relative frequency $f_i/(\max_j f_j)$. I.e., for a given estimation method we compute the frequencies f_j for all ρ_j and then normalize them by the maximum frequency $\max_j f_j$.

Figure 10 shows the relative frequencies estimated via the three different methods for three different example problems, based on three different PDEs. For the advection-diffusion equation, all methods agree that the frequencies of the different ρ_i are all similar. Thus, the frequency of ρ_i does not strongly vary with i. However, for the KdV and Burgers' equation, the TV norm and LE both compute a non-monotonic increase of f_i, as i increases. For indices i between 30 and 50, the growth of f_i slows down, until a plateau with fluctuations is reached. This holds for a wide range of considered neighbors k. The projection method however computes, similar to the advection-diffusion equation, approximately equal frequencies for all ρ_i.

There are several reasons for this disagreement between the TV norm and the LE on one side, and the projection method on the other. The TV norm and the LE mainly measure local oscillations of a function, whereas the Fourier transform used in the projection method attempts to measure the global oscillations. However, since the projection method relies on the division into different functions, whose inputs are projections of the true input p onto specific vectors, significant information is lost. This is described in more detail in Appendix A.2. Lastly, the non-uniform discrete Fourier transform employed in the projection method relies on polynomial interpolation to compute the discrete Fourier transform, which introduces more errors.

Based on the agreement between the TV norm and LE, and the identified limitations of the projection method, we consider the results obtained with the TV norm

and LE to be the most reliable for the present analysis. Thus, we conclude that for the KdV and Burgers' equation the frequency of ρ_i tends to increase as i increases. Lastly, as this will become relevant later, we define a *frequency dip*. Mode i exhibits a frequency dip (according to a frequency computation method) if

$$f_i < \min(f_{i-1}, f_{i+1}),$$

where f_j denotes the frequency of ρ_j. In other words, mode i has a frequency dip when its right-singular function has a lower frequency than the modes with indices $i-1$ and $i+1$. Modes for which both the TV norm and the LE methods consistently detect frequency dips are highlighted by pink arrows in Fig. 10.

4 APPLYING THE TRUNK-BRANCH ERROR DECOMPOSITION

In this chapter, we apply the decomposition of the error into (a) the error of the basis functions, i.e., the trunk error, and (b) the error of the coefficients, i.e., the branch error. We thus identify the bottleneck in the DeepONet performance for our example problems. The results of this chapter are foundational for the construction of the SVDONet in the next chapter.

4.1 ERROR OF THE TRUNK NETWORK

We compare the approximation errors of DeepONets that all have the same depth and width, but varying inner dimensions N.
In Fig. 11 (A and B) the total error δ, the trunk error δ_T, the branch error δ_B and the SVD truncation error δ_{SVD} are shown for DeepONets of increasing inner dimension N. We see that by increasing the inner dimension N, the trunk error δ_T is continuously lowered. Note that for small N, the trunk error is near optimal with $\delta_T \approx \delta_{SVD}$. For larger N, there is an emerging gap between the trunk error and the SVD truncation error. This is seen for both the training data (A) and the test data (B).
We now dissect the training trunk error, using the projection error $\Delta(i, N)$, i.e., the error of approximating the i-th left-singular vector ϕ_i by projecting it onto the trunk space of a DeepONet with inner dimension N. Recall the fundamental idea of the projection error: the trunk space approximates the SVD space.
Furthermore, recall two important properties of the SVD $A = \Phi\Sigma V^T$.

1. The singular values are ordered to be non-increasing. In practice they are descending, i.e., $\sigma_i > \sigma_{i+1}$.

2. The singular value σ_i indicates ϕ_i's relevance in approximating A.

Thus, the first left-singular vector ϕ_1 is the most relevant vector for the approximation of A. In contrast, ϕ_N is the least relevant vector for the trunk space to approximate, while still being in $S(N)$.
Figure 11 (C) shows the projection error in three different curves.

1. The projection error is shown for the N first left-singular vectors and one DeepONet with inner dimension $N = 50$ (purple).

2. The projection error of ϕ_1, i.e., the most relevant direction, and DeepONets with varying inner dimension N is shown (orange).

3. The projection error of ϕ_N, i.e., the least relevant direction, and DeepONets with varying inner dimension N is shown (green).

All three curves together show the following. The first few left-singular vectors tend to be approximated the best. In fact, the first left-singular vector's projection error $\Delta(i = 1, N)$ decreases, as N increases. However, as the index i increases, the projection error $\Delta(i, N)$ increases. This shows that even as N increases, the trunk spaces $\mathcal{T}(N)$ all approximate the SVD space $S(N')$ for some small N'. I.e., for the

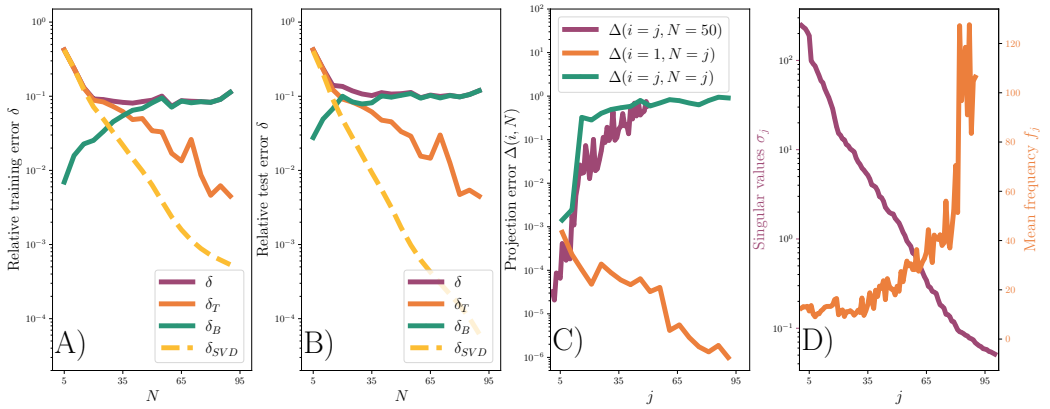


Figure 11: **Training, Test and Projection Errors of DeepONets with varying** N. **A)** Training errors and **B)** test errors: The total error δ (purple), the trunk error δ_T (orange), the branch error δ_B (green) and the SVD truncation error δ_{SVD} (yellow) are shown as functions of N, the inner dimension of the DeepONet. **C)** Projection errors $\Delta(i, N)$ for different modes ϕ_i and DeepONet inner dimensions N are shown. *Purple:* Projection errors $\Delta(i = j, N = 50)$ of varying modes ϕ_j and one DeepONet with $N = 50$ (error vs. i). *Orange:* Projection errors $\Delta(i = 1, N = j)$ of the first mode ϕ_1 and DeepONets with increasing inner dimension $N = j$ (error vs. N). *Green:* Projection errors $\Delta(i = j, N = j)$ of each DeepONet's last mode ϕ_j (error vs. $i = N$). **D)** Singular values σ_j of the training data matrix (purple, left scale) and mean frequencies f_j of the left-singular functions λ_j (orange, right scale) are shown.

standard example (Kdv equation with $\tau = 0.2$) and $N = 50$, shown in Fig. 11, only the first $N' = 20$ left-singular vectors are approximated with a projection error $\Delta(i, N) < 10^{-3}$. To summarize, the fact that the trunk error δ_T decreases, as N increases, can be attributed to (a) the continuous lowering of the approximation error of the most relevant, i.e., first few, left-singular vectors, and (b) the continuous inclusion of new relevant directions in the trunk space, especially for small N. However, since the projection error of the left-singular vectors grows with their index, the gap between δ_T and δ_{SVD} emerges.

As discussed in Section 3.3, the spatial frequency of the left-singular vectors can be computed. Figure 11 (D) shows both the singular value σ_i, indicating the relevance of ϕ_i, and its spatial frequency f_i. We observe that while the frequencies f_i of the first $i \leqslant 30$ vectors ϕ_i remain relatively consistent, the singular values σ_i drastically decrease, and the projection errors $\Delta(i, N)$ increase significantly. Furthermore, even for other example problems, which display a more consistent increase in spatial frequencies f_i, as i increases, there is no significantly stronger increase in $\Delta(i, N)$ observed. Therefore, we argue that the primary reason for the increasing projection errors is not the frequency, i.e. spectral bias, but the prioritization of the more relevant vectors. Note that the decrease of δ_T for the test and the training case is very similar. This shows that the DeepONet learns generalizing basis functions.

4.2 ERROR OF THE BRANCH NETWORK

As N increases, the branch error δ_B increases and reaches a plateau. Since δ_T decreases, as N increases, and since $\delta^2 = \delta_T^2 + \delta_B^2$, the total error δ is dominated by the branch error, $\delta \approx \delta_B$, for large N. This is seen for both the training data and the test data. The intuition behind this observation is that a branch network

with fixed width and depth cannot approximate the targets for $N = 100$ neurons as accurately as for, e.g., $N = 20$ neurons.
To conclude, both for test and training input functions, if the hidden dimension N is chosen sufficiently large, the error is dominated by the branch error. Thus, for our example problems and large N, the branch network is the bottleneck of the DeepONet's accuracy. Since the entire dissection of the trunk error relies on the comparison to the optimal trunk matrix, the next chapter introduces an adaptation of the DeepONet, the SVDONet, for which the optimal branch matrix is constant over the training course. The SVDONet thus enables us to dissect the branch error in a similar manner.

5 SVDONET

In the previous chapter, we observed that for a large enough N, the branch error dominates the total error. We now construct the SVD-based operator network (SVDONet), an adaption of the DeepONet. In the SVDONet the trunk network is removed and replaced with a fixed trunk matrix obtained through SVD of the training data matrix. We introduce the SVDONet for two main reasons.
Firstly, removing the trunk network significantly simplifies the analysis since all parameters are now branch parameters.
Secondly, the SVDONet enables us to decompose the branch error into different errors, the mode losses.

5.1 DEFINING THE SVDONET

To see the intuition behind the SVDONet, we restate the SVD of the training data matrix A_{tr} (Equation 10):

$$\begin{aligned} A_{tr} = \Phi\Sigma V^T &= [\Phi_1\ \Phi_2] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} \\ &= \Phi_1\Sigma_1 V_1^T + \Phi_2\Sigma_2 V_2^T = \sum_{j=1}^{\min(n,m)} \sigma_j \phi_j v_j^T. \end{aligned}$$

Recall that $\Phi_1 \in \mathbb{R}^{n\times N}$ and $V_1 \in \mathbb{R}^{m\times N}$ contain the first N left- and right-singular vectors of A_{tr}, respectively, and $\Sigma_1 \in \mathbb{R}^{N\times N}$ contains the first N singular values.
Furthermore, we restate the matrix formulation of the DeepONet (see Equation 8):

$$\begin{aligned} \tilde{A} = TB^T &= \sum_{j=1}^{N} t_j b_j^T \qquad (22) \\ \text{with } T = [t_1\ \dots\ t_N] &\in \mathbb{R}^{n\times N}, \\ B = [b_1\ \dots\ b_N] &\in \mathbb{R}^{m\times N}. \end{aligned}$$

As discussed in Section 2.6.1, for any matrix $C \in \mathbb{R}^{N\times N}$ with full rank N, the matrix $\Phi_1 C$ is an optimal trunk matrix. We choose $C = \Sigma_1$, i.e., each left-singular vector is scaled with its corresponding singular value, and replace the trunk network's output T by the fixed trunk matrix $\Phi_1\Sigma_1$. This specific scaling, $C = \Sigma_1$, is motivated in Section 5.3. Thus, we arrive at the SVDONet's output by inserting $T = \Phi_1\Sigma_1$ in Equation 22:

$$\tilde{A}_{SVDONet} = \Phi_1\Sigma_1 B^T = \sum_{j=1}^{N} \sigma_j \phi_j b_j^T. \qquad (23)$$

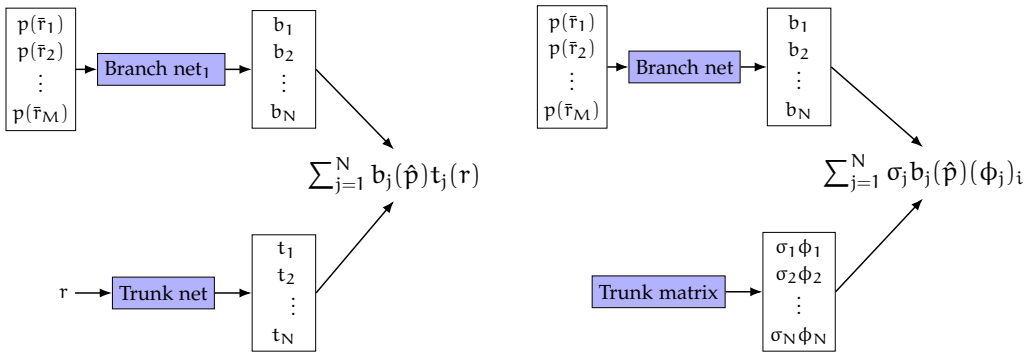
For a clear overview over the true solution (top row), the SVDONet (center row) and the DeepONet (bottom row), we denote their matrix formulations (left column), and the output for individual coordinates and individual input functions (right column) in the following Equations.

	Matrix Formulation		Individual Evaluation	
True Solution	$\Phi_1\Sigma_1 V_1^T + \Phi_2\Sigma_2 V_2^T$	(24)	$\sum_{j=1}^{r} \sigma_j(\phi_j)_i(v_j)_k$	(26)
SVDONet	$\Phi_1\Sigma_1 B^T$	(25)	$\sum_{j=1}^{N} \sigma_j(\phi_j)_i b_j(p_k)$	(27)
DeepONet	TB^T		$\sum_{j=1}^{N} t_j(r_i)b_j(p_k)$	

The comparison for individual coordinates and individual input functions (right column) again shows that the SVDONet replaces the trunk network's output neurons t_i with the scaled left-singular vectors ϕ_i. The left-singular vectors ϕ_i are from now on called *modes*. A brief interpretation of the modes in the PDE context is given in Section 5.4.4. The replacement of the trunk network by a fixed trunk matrix implies that the SVDONet can only be evaluated at the training coordinates r_j. The architectures of the neural networks behind the standard DeepONet and the SVDONet are both visualized in Fig. 12.
By comparing the matrix formulations for the true solution and the SVDONet (see Equations 24 and 25) one sees that for a well-trained SVDONet, $B \approx V_1$ is necessary. This can also be seen by using Equation 13 with $T = \Phi_1\Sigma_1$:

$$\begin{aligned} B_* &= (T^+ A_{tr})^T = (\Sigma_1^{-1}\Phi_1^T A_{tr})^T = \left(\Sigma_1^{-1}\Phi_1^T(\Phi_1\Sigma_1 V_1^T + \Phi_2\Sigma_2 V_2^T)\right)^T \\ &= (\Sigma_1^{-1}\Phi_1^T\Phi_1\Sigma_1 V_1^T)^T = V_1 \end{aligned}. \tag{28}$$

By comparing the value of the true solution and the SVDONet for individual evaluations (see Equations 26 and 27) we see that $b_j(p_k)$ should approximate $(v_j)_k = V_{jk}$. Hence, b_j should approximate the right-singular function ρ_j, introduced in Section 3.3.
Furthermore, note that the SVDONet always uses the trunk matrix $T = \Phi_1\Sigma_1$ based on the SVD of the training data matrix, also when evaluating the SVDONet on test data.



(a) DeepONet (b) SVDONet

Figure 12: **Comparison of the Architectures behind the standard DeepONet and the SVDONet:** In the SVDONet, the trunk network is replaced by the trunk matrix $\Phi_1\Sigma_1$. The DeepONet can be evaluated at any coordinate r, while the SVDONet can only be evaluated at r_i, one of the evaluation points in the training data.

As described, for the SVDONet we choose $T = \Phi_1\Sigma_1$. However, we could also construct an alternative SVDONet in which $T = \Phi_1$. This would imply $B \approx \Sigma_1 V_1$. Numerical examples show that both approaches perform similarly. A study comparing the performance impact of Σ_1 can be found in Appendix A.6. Note that the SVDONet with $T = \Phi_1$ has been proposed in [35] and was termed POD-DeepONet. However, the decomposition into, first, trunk and branch error, and second, mode errors, was not investigated in [35]. Our reasons to use $T = \Phi_1\Sigma_1$ are discussed in Section 5.3.

5.2 DERIVING THE MODE-BASED ERROR DECOMPOSITION

We now drop the subscript SVDONet for the approximation matrix $\tilde{A}$, since, unless noted otherwise, the approximation matrix from this point on corresponds to the SVDONet. We decompose the SVDONet's training error, by applying Equation 9. To this end, we use the SVD of the training data matrix A_{tr}.

$$\begin{aligned} A_{tr} &= \Phi_1\Sigma_1 V_1^T + \Phi_2\Sigma_2 V_2^T \\ \varepsilon_{tr} &= \|\tilde{A}_{tr} - A_{tr}\|_F^2 = \|\Phi_1\Sigma_1 B_{tr}^T - \Phi_1\Sigma_1 V_1^T - \Phi_2\Sigma_2 V_2^T\|_F^2 \\ &= \|\Sigma_1 B_{tr}^T - \Sigma_1 V_1^T\|_F^2 + \|\Sigma_2\|_F^2 = \sum_{i=1}^{N} \sigma_i^2 \underbrace{\|b_{i,tr} - v_i\|_2^2}_{=:L_{i,tr}} + \|\Sigma_2\|_F^2 \end{aligned} \tag{29}$$

By design, the trunk space is spanned by the first N left-singular vectors of A_{tr}, thus $\varepsilon_T = \varepsilon_{SVD} = \|\Sigma_2\|_F^2$, i.e., the SVDONet achieves the optimal trunk error for the training data. Since $\|\Sigma_2\|_F^2$ is the trunk error, $\|\Sigma_1 B_{tr}^T - \Sigma_1 V_1^T\|_F^2$ is the branch error. Equation 29 then displays the SVDONet's main advantage for analysis: the branch error can be decomposed into the errors $\sigma_i^2 L_i$ corresponding to the different modes. The *unweighted mode loss* L_i is the difference between v_i and b_i, i.e., the difference between the true coefficients of mode i and the approximate coefficients b_i predicted by the SVDONet, for all input functions. Since every mode is scaled with the corresponding singular value σ_i, the *weighted mode loss* is $\sigma_i^2 L_i$.
For test data, the error decomposition is similar, but slightly more involved. As discussed in Section 3.2 the numerical methods used to generate the training and test data imply that

$$\mathrm{Col}(A_{te}) \subset \mathrm{Col}(A_{tr}),$$

where $\mathrm{Col}(A)$ is the column space of a matrix A. Thus, the matrix $W = (\Sigma^+\Phi^T A_{te})^T$ satisfies

$$A_{te} = \Phi\Sigma W^T.$$

The matrix W thus contains the coefficients to build A_{te} using the scaled left-singular vectors of A_{tr}. As we divided V into V_1, V_2, we can divide $W = [W_1\ W_2]$, with $W_1 \in \mathbb{R}^{m_{tr}\times N}$, such that

$$A_{te} = \Phi_1\Sigma_1 W_1^T + \Phi_2\Sigma_2 W_2^T.$$

Thus, as V_1 contains the optimal coefficients to approximate A_{tr} using $\Phi_1\Sigma_1$, the matrix

$$W_1 = (\Sigma_1^{-1}\Phi_1^T A_{te})^T$$

contains the optimal coefficients to approximate the test data matrix A_{te} using $\Phi_1\Sigma_1$. Using the definitions of W_1 and W_2, we can compute the test error

$$\begin{aligned}\varepsilon_{te} &= \|\tilde{A}_{te} - A_{te}\|_F^2 = \|\Phi_1\Sigma_1 B_{te}^T - \Phi_1\Sigma_1 W_1^T - \Phi_2\Sigma_2 W_2^T\|_F^2 \\ &= \|\Phi_1\Sigma_1 B_{te}^T - \Phi_1\Sigma_1 W_1^T\|F^2 + \|\Phi_2\Sigma_2 W_2^T\|_F^2 \\ &= \|\Sigma_1 B_{te}^T - \Sigma_1 W_1^T\|_F^2 + \|\Sigma_2 W_2^T\|_F^2 \\ &= \sum_{i=1}^{N} \sigma_i^2 \underbrace{\|b_{i,te} - w_i\|_2^2}_{=:l_i} + \|\Sigma_2 W_2^T\|_F^2.\end{aligned}$$

Here w_i is the i-th column of W_1. Note that since we use the left-singular vectors of the training matrix, the true coefficient matrices V_1, V_2 for the training data are semi-orthogonal, unlike the true coefficient matrices W_1, W_2 for the test data. Thus the norm of W_2 is not bounded in any sense, and hence the relative test trunk error $\|\Sigma_2 W_2^T\|_F / \|A_{te}\|_F$ might be much larger than the relative training trunk error $\|\Sigma_2 V_2^T\|_F / \|A_{tr}\|_F = \|\Sigma_2\|_F^2 / \|A_{tr}\|_F$. However, this is not the case for our example problems; see Appendix A.7.
Furthermore, since the trunk error is determined by the choice of N, it can be omitted from the loss used in the neural network training. Thus, we use

$$\begin{aligned}\mathcal{L}_{tr} &:= \frac{1}{n_{tr} m_{tr}} \|\Sigma_1 B_{tr}^T - \Sigma_1 V_1^T\|_F^2 = \frac{1}{n_{tr} m_{tr}} \varepsilon_{B,tr}, \\ \mathcal{L}_{te} &:= \frac{1}{n_{te} m_{te}} \|\Sigma_1 B_{te}^T - \Sigma_1 W_1^T\|_F^2 = \frac{1}{n_{te} m_{te}} \varepsilon_{B,te}\end{aligned}$$

as training and test loss from now on. This corresponds to only measuring the error in the coefficients of the first N modes, not the full approximation error of the matrix A. This implies

$$\begin{aligned}\mathcal{L}_{tr} &= \frac{1}{n m_{tr}} \sum_{i=1}^{N} \sigma_i^2 L_{i,tr}, \\ \mathcal{L}_{te} &= \frac{1}{n m_{te}} \sum_{i=1}^{N} \sigma_i^2 l_i.\end{aligned}$$

To facilitate the comparison between the unweighted mode losses for test and training data we define the unweighted test loss of mode i as $L_{i,te} = l_i \frac{m_{tr}}{m_{te}}$. This implies

$$\mathcal{L}_{te} = \frac{1}{n m_{tr}} \sum_{i=1}^{N} \sigma_i^2 L_{i,te}. \tag{30}$$

We introduce the term *base loss* for modes as a practical reference when interpreting mode losses. The base loss of mode i is defined as the loss obtained when $b_i = 0$, i.e., when the SVDONet predicts a zero coefficient for mode i. The unweighted base loss on the training data is thus $\|v_i\|_2^2 = 1$, as V_1 is semi-orthogonal. Thus, the weighted base loss on the training data is σ_i^2. For the test data the unweighted base loss is $\frac{m_{tr}}{m_{te}} \|w_i\|_2^2$ and thus, the weighted base loss is $\frac{m_{tr}}{m_{te}} \sigma_i^2 \|w_i\|_2^2$. Observing a mode loss L_i which exceeds its base loss suggests that the optimizer is effectively neglecting mode i, since the trivial prediction $b_i = 0$ would yield a lower loss.

5.3 CHOICE OF TRUNK MATRIX

Our choice for $T = \Phi_1 \Sigma_1$ is motivated by three reasons.

1. As discussed, the first N left singular vectors of the training data matrix A_{tr} yield a basis for the best rank N approximation of the training data A_{tr} as seen from $\varepsilon_T = \varepsilon_{SVD}$. Thus, we choose $T = \Phi_1 C$, for a full-rank matrix $C \in \mathbb{R}^{N \times N}$.

2. Through the use of $C = \Sigma_1$, the branch network's training target is V_1, which is a semi-orthogonal matrix. Hence the sum of squares of the targets are normalized for each branch neuron, i.e., $\|v_i\|_2^2 = 1$. This type of target normalization is common in machine learning [2] and more specifically in multi-task learning [17, 20, 21]. This is not given in the POD-DeepONet, which uses $T = \Phi_1$.

3. Furthermore, the insertion of Σ_1 somewhat simplifies the analysis leading to a straight-forward definition of the unweighted mode loss L_i.

As we have seen, for the standard DeepONet the gap between the trunk error and the SVD truncation error is negligible compared to the branch error. Hence we do not expect a large error reduction by replacing the trunk network with the left-singular vectors. We mainly introduce the SVDONet to facilitate our analysis.

5.4 RELATED WORK

To contextualize this thesis within the broader field of OL research, we will discuss related work. Section 5.4.1 compares the SVDONet to OL architectures whose outputs also lie in the span of a predetermined basis. Section 5.4.2 includes OL architectures, such as the FNO, which utilize basis representations, but whose outputs are not in the span of a predetermined basis. Section 5.4.3 discusses the applicability of the derived mode-loss decomposition to other architectures and discusses the spectral bias in OL. Section 5.4.4 relates the SVDONet's modes to the discretized operator it is approximating. Section 5.4.5 introduces multi-task learning, another framework in which the SVDONet can be viewed.

5.4.1 *Closely Related Architectures*

The SVDONet's closest relative is the POD-DeepONet, suggested in [35]. The only difference between SVDONet and POD-DeepONet is the scaling of the fixed trunk matrix. Their relation is described in more detail in Sections 5.1 and 5.3.
There are multiple OL architectures that share some elements with the SVDONet, e.g., the spectral neural operator (SNO) [12] and the *principal component analysis neural operator* (PCANO) [1]. We now examine the differences and similarities between the SVDONet on the one side, and the SNO and PCANO on the other side. In line with the example problems considered in this work, we only discuss the approximation of time-evolution operators, for ease of exposure. I.e., the SVDONet's input function is the initial condition. To avoid any misunderstandings, we divide each OL architecture into its *learnable* and its *fixed* part. For the SVDONet, the branch network is the learnable part and the trunk matrix $T = \Phi_1 \Sigma_1$ is the fixed part. The DeepONet's trunk and branch network are both learnable, and hence it

does not have any fixed parts. I.e., we do not consider the classical neural network hyperparameters and the initial condition's sample locations to belong to the fixed part.
Thus, the SVDONet's learnable part takes the initial condition as input. The SNO's and PCANO's learnable parts however do not take the initial condition as input. They both use a fixed input basis, such that their learnable parts take the initial condition's coefficients with respect to this fixed input basis as input. Hereafter, SVDONet, SNO and PCANO behave similarly. The learnable part computes the output coefficients for the fixed output basis. The final output is then given as the linear combination of the computed coefficients and the fixed output basis vectors. Thus, for both SNO and PCANO, the fixed part contains two sets of vectors, the input and output basis vectors. The SNO uses either Chebyshev polynomials or trigonometric functions for both the input and output bases. The PCANO uses the left-singular vectors of the input and output matrices as input and output bases, respectively. The SNO's structure with trigonometric functions as basis functions is given by:

$$p \in \mathbb{R}^M \to \mathcal{F}(\cdot)(F) \to \mathcal{N}_\theta(\cdot) \to \mathcal{F}^{-1}(\cdot),$$

where $F = (f_1, \ldots, f_{\lfloor M/2 \rfloor})$ are all accessible frequencies, $\mathcal{F}, \mathcal{F}^{-1}$ denote the discrete Fourier and discrete inverse Fourier transform, respectively, and $\mathcal{N}_\theta$ is a neural network. Thus, $\mathcal{F}(p)(F)$ are the initial condition's coefficients with respect to the trigonometric basis functions, and $\mathcal{N}_\theta(\mathcal{F}(p)(F))$ are the final output's coefficients with respect to the trigonometric basis functions. For comparison, the SVDONet's structure is given by:

$$p \in \mathbb{R}^M \to \mathcal{N}_\theta(\cdot) \to \Phi_1 \Sigma_1(\cdot),$$

where $\Phi_1 \Sigma_1(\cdot)$ denotes multiplication of $\Phi_1 \Sigma_1$ with a vector.
Since the SVDONet and these related architectures map to finite dimensional vector spaces, they are technically not approximating the solution operator mapping to an infinite dimensional vector space anymore. However, this is often the more realistic situation. E.g., when using numerical simulations to generate training and test data, these simulation results are often only available on a fixed mesh anyways. We thus believe that this simpler setting still provides a realistic and insightful scenario. Furthermore, by adding a simple interpolator these architectures can be modified to map to function space.

5.4.2 *Broader Context*

Neural operators such as the Fourier neural operator (FNO) [33], the convolutional neural operator (CNO) [42] and the wavelet neural operator (WNO) [49] can be seen as relatives of the SNO and the PCANO. Thus, they can also be seen as more distant relatives of the SVDONet. For ease of exposition, we only compare the FNO to the SNO with trigonometric functions and we ignore the local linear transform of each Fourier layer due to its purely local influence. The FNOs structure can be written as

$$p \in \mathbb{R}^M \to \mathcal{F}(\cdot)(F') \to \mathcal{N}_\theta(\cdot) \to \mathcal{F}^{-1}(\cdot) \to \bar{\mathcal{N}}_{\bar{\theta}}(\cdot),$$

where $F' = (f_1, \ldots, f_{K<\lfloor M/2 \rfloor})$ contains the first K frequencies and $\mathcal{N}_\theta, \bar{\mathcal{N}}_{\bar{\theta}}$ are two neural networks. Thus, the FNO and SNO both begin by transforming the initial

condition into the input coefficients through a Fourier transform. However, then the FNO only uses the first K frequencies, introducing aliasing errors. Next, both architectures apply a neural network. Then, both architectures compute the discrete inverse Fourier transform. For the SNO, this is the final output. Thus, the SNO's final output is a linear combination of the Fourier basis with the computed output coefficients, as discussed. However, the FNO uses the inverse Fourier transform as input to a second neural network $\tilde{\mathcal{N}}_{\tilde{\theta}}$. This second neural network then produces the FNO's final output. Thus, the FNO's final output is not in a specific subspace spanned by a set of fixed basis functions. Thus, the FNO, and related architectures, are generally more expressive [33], while their output is less structured.
Furthermore, note that in practice the neural network $\mathcal{N}_\theta$ used in the FNO is a convolutional neural network, while the SNO uses a multi-layer perceptron.

5.4.3 *Error Decomposition and Spectral Bias*

A detailed comparison of the SVDONet and the DeepONet is given in Section 5.1. However, whether decompositions into losses corresponding to different spatial basis functions/vectors - similar to the one derived for the SVDONet in Section 5.2 - can also be constructed for other OL architectures has not yet been discussed. To demonstrate that this is indeed possible, we briefly discuss a similar error decomposition, which could, for example, be applied to the DeepONet.
For simplicity, consider a training data matrix $A \in \mathbb{R}^{n\times m}$ with $n \leqslant m$. This implies that Φ, the left-singular matrix of A, has $\operatorname{rank}(\Phi) = n$. Furthermore, we use (a) the left-singular vectors ϕ_i as modes, as done in previous sections, and (b) all $N = n$ modes ϕ_i, unlike in previous sections. We can then compute the following error decomposition:

$$\varepsilon = \|\tilde{A} - A\|_F^2 = \|\Phi^T(\tilde{A} - A)\|_F^2 = \sum_{i=1}^{n} \underbrace{\|\phi_i^T(\tilde{A} - A)\|_F^2}_{=:e_i}.$$

Thus, e_i is the norm of the residual's projection onto the i-th mode. The main difference in this decomposition for the DeepONet and the previous decomposition for the SVDONet is the following.

1. For a DeepONet, e_i is influenced both by the trunk space's approximation of ϕ_i, and by the coefficient the branch network learned for this mode.
2. For the SVDONet, the i-th mode is either perfectly represented ($i \leqslant N$), or not given at all in the trunk matrix. Thus, for $i \leqslant N$ the mode loss only contains the error in the coefficient.

Furthermore, note that the decomposition described in this section can be seen as an adaptation of the spectral error defined for the investigation of the spectral bias; see Section 2.3.4. Consequently, research in this direction is typically classified under the overarching theme of *spectral bias in OL*. Recently, spectral bias as a relevant obstruction to DeepONets [50, 56] and other OL architectures [22, 24] has become a prevalent research direction. However, to the best of our knowledge, an error decomposition into the mode losses has not been reported in the literature yet. The closest work we have identified to our mode loss decomposition is the decomposition of the error into frequency bands for the FNO [22]. The decomposition into frequency bands can be derived using the Plancherel theorem.

5.4.3.1 *Spectral Bias in Operator Learning*

The SVDONet framework offers two different and insightful viewpoints in the discussion surrounding spectral bias.
Firstly, we discuss the spectral bias regarding the left-singular vectors, i.e., the spatial dependency and the trunk matrix. The SVDONet's output is spanned by $\{\phi_i\}_{i=1}^{N}$ the first N left-singular vectors of A_{tr}. As discussed in Section 3.3, we can compute the frequency f_i of the left-singular vectors ϕ_i, or equivalently the left-singular functions λ_i. We then observe that f_i generally increases with i, though its growth is not monotone. This association of ϕ_i and growing frequencies f_i, as i increases, yields two different insights.

(a) Since (i) the SVDONet is designed to only use the first N modes to approximate the solution, and since (ii) these are the modes with the lowest frequencies, the SVDONet necessarily omits the high frequency parts of the solution. Hence the SVDONet has output-wise spectral bias. Note that the latter reason is not necessarily true for other data sets.

(b) The loss of mode i can also be associated with the frequency f_i. I.e., the decomposition of the loss into the mode losses equivalently is a decomposition into frequency losses. Hence the output-wise spectral bias of SVDONets can naturally be studied using the mode loss decomposition. Thus, the investigation of mode losses, where modes are sorted by relevance, can also be viewed through the lens of frequency dependent errors. The question of *How large is the error associated with the i-th most relevant direction?* thus turns into *How large is the error associated with a given spatial frequency?*.

Secondly, the input function dependence, i.e., the branch network, can be considered. Recall that to investigate the spectral bias, the Fourier transform of the approximated function and the Fourier transform of the target function are compared. Furthermore, a given trunk matrix T allows for the computation of target coefficients $(T^+A)^T$. However, since the trunk network, and hence the trunk matrix, of DeepONets are continually optimized over the training course, constant targets are not given for normal DeepONets. For SVDONets however the fixed trunk matrix $T = \Phi_1\Sigma_1$ yields constant target coefficients V_1. As discussed, the i-th branch output neuron b_i is trained to approximate the i-th right-singular function ρ_i, with $\rho_i(p_j) = V_{ji}$. Recall that we observed that for the KdV and the Burgers equation, the frequency of ρ_i increases as i increases; see Section 3.3. Thus, if we observe that the approximations of the right-singular functions ρ_i worsen as i increases, this might be attributed to the spectral bias. A growing frequency of ρ_i as i increases is not found for the advection-diffusion equation.
Both viewpoints can also be summarized in a comparison between the potential spectral bias in DeepONet and SVDONet. In the DeepONet spectral bias could manifest in two different ways.

(a) Due to spectral bias in the trunk network, the learned basis functions might not contain the high frequency components necessary to accurately approximate the solution. However, recall that Chapter 4 shows that spectral bias is not the dominating factor determining the approximation error of different basis functions.

(b) Due to spectral bias in the branch network, coefficients b_i whose target functions $p_j \mapsto (T^+A)_{ji}$ have high frequencies are not accurately approximated.

Note that the target function is not constant over the course of training for normal DeepONets, since the trunk matrix T changes.

In contrast, for the SVDONet reason (a) can be eliminated by choosing a large enough N, and thus only reason (b) remains.

5.4.4 *Interpretation of Modes*

In this section, we briefly discuss the connection between the modes ϕ_i and the discretized time-evolution operator. For simplicity, we only discuss linear PDEs, thus the following derivation does not apply to the KdV and Burgers' equation. However, we still believe that the interpretation yields a valuable intuition. Consider a time-dependent PDE of the form

$$0 = \frac{\partial u}{\partial t} + \mathcal{D}u,$$

where $\mathcal{D}$ is a linear spatial differential operator. Together with the boundary conditions this forms an initial value problem.
Using appropriate spatial discretizations $\hat{p} \in \mathbb{R}^n$ and $\hat{G}_* \in \mathbb{R}^{n\times n}$ of the initial condition $p(\cdot) = u(\cdot, t = 0)$ and the infinite-dimensional time-evolution operator $G_* : p(\cdot) \mapsto u(\cdot, t = \tau)$, respectively, we can compute the discretized solution $u^*_{\hat{p}} = \hat{G}_*\hat{p} \in \mathbb{R}^n$. The matrix $\hat{G}_*$ admits an SVD $\hat{G}_* = \Psi S Q^T$, with the left-singular vectors $\Psi = [\psi_1 \ \dots \ \psi_n] \in \mathbb{R}^{n\times n}$ and the singular values $S = \operatorname{diag}(s_1, \dots, s_n)$. We assume that $\hat{G}_*$ has full rank n, i.e., $s_i > 0, \forall i \in [n]$. Furthermore, consider the input matrix

$$P = [\hat{p}_1 \ \dots \ \hat{p}_m] \in \mathbb{R}^{n\times m},$$

with $n \leqslant m$ and assume $\operatorname{rank}(P) = n$. Then, the (output) data matrix is

$$A := [u^*_{\hat{p}_1} \ \dots \ u^*_{\hat{p}_m}] = \hat{G}_*P = \Psi S Q^T P.$$

Recall that the modes ϕ_i, which are used in the SVDONet, are the left-singular vectors of A. We thus see that the set of all modes ϕ_i and the set of all left-singular vectors ψ_i of $\hat{G}_*$ span the same space. In linear algebra terms this is equivalent to the following simple statement.
For any matrix $G \in \mathbb{R}^{n\times n}$ *and any matrix* $P \in \mathbb{R}^{n\times m}$ *for which* $m \geqslant n$ *and rank*$(P) = n$, *the column spaces of* G *and* GP *are identical.*

We furthermore consider the special case, in which P is given as

$$P = QCY \in \mathbb{R}^{n\times n},$$

with C being a full-rank diagonal $n \times n$ matrix and $Y \in \mathbb{R}^{n\times n}$ being an orthogonal matrix. Then the data matrix is

$$A = \Psi S Q^T Q C Y = \Psi (SC) Y.$$

This is, up to a potential reordering of Ψ's columns, Y's rows and SC's diagonal entries, and potential sign switches, an SVD of A. Thus, the modes ϕ_i and the left-singular vectors ψ_j of $\hat{G}_*$ are, up to the potential reordering and potential sign switches, the same. Thus, for all $i \in [n]$, there exists $j \in [n]$, such that $\phi_i = \pm\psi_j$.

5.4.4.1 *Relationship to Spectral Properties*

Section 3.3 discusses the spectral properties of the SVD of the data matrix. It states that the relationship between the mode index i and f_i, the spatial frequency of ϕ_i, can be arbitrarily prescribed through the choice of the initial conditions. This is demonstrated by the following example.
Consider a discretized time-evolution operator $\hat{G}_*$, whose left-singular vectors ψ_j have a spatial frequency f_j, which increases with j. This spatial frequency is computed by the discrete Fourier transform. Furthermore, by choosing $C = S^{a-1}$, i.e., $C_{ii} = s_i^{a-1}$, in the aforementioned special case $P = QCY$, the data matrix becomes

$$A = \Psi(SC)Y = \Psi S^a Y.$$

Thus, for $a > 0$, neither reordering nor sign switches are necessary; $\psi_i = \phi_i$. Thus, the spatial frequency of ϕ_i also increases with i. However, for $a < 0$, the order of the singular triplets has to be reversed, i.e., $\phi_i = \psi_{n-i}$. Thus, the spatial frequency of ϕ_i decreases with i.

5.4.5 *Multi-Task Learning*

Recall that the SVDONet's branch network takes one input $\hat{p} \in \mathbb{R}^M$ and produces N outputs b_i, which are referred to as coefficients of the modes ϕ_i.
Unrelated to the field of operator learning, there is a subfield of machine learning, called multi-task learning (MTL). As the name suggests, in MTL there is one model which is supposed to learn multiple tasks. An example for MTL in practice is the software FaceID to unlock iPhones, which concurrently locates the face and identifies the user [54]. We now provide a formal definition of a specific variant of MTL [54]. Consider $K \in \mathbb{N}$ different tasks and the set of K models $\{G_\theta^{(k)}\}_{k=1}^K$, which are all parametrized by the same parameters θ. Note that the different models can all be parametrized differently. A simple example of the different parametrizations using the same parameter $\theta \in \mathbb{R}$ is $G_\theta^{(k)}(x) = \theta^k x$ with $x \in \mathbb{R}$. Then the different models $G_\theta^{(k)}$ are all based on the same parameter(s) θ, but their dependency on θ differs. However, not all task-specific models have to use all of θ. For instance, consider the parameters $\theta = (\theta_1 \ \dots \ \theta_K \ \theta_{K+1})^T \in \mathbb{R}^{K+1}$ and the task-specific models $G_\theta^{(k)}(x) = \theta_k x + \theta_{K+1}$. Then only θ_{K+1} is present in all models.
Consider furthermore the dataset $\{x_i^{(k)}, y_i^{(k)}\}_{i=1}^{m_k}$ for the k-th task. For simplicity we consider the inputs $x_i^{(k)}$, the targets $y_i^{(k)} \in \mathbb{R}$, and the task-specific loss function

$$\tilde{L}_k(\theta) = \frac{1}{m_k} \sum_{i=1}^{m_k} \left| y_i^{(k)} - G_\theta^{(k)}\left(x_i^{(k)}\right) \right|^2. \tag{31}$$

Then the total loss function is $\mathcal{L}(\theta) = \sum_{k=1}^K \tilde{L}_k(\theta)$. As for other machine learning problems, good parameters θ can be found using gradient descent of the loss function $\mathcal{L}$.
The advantages of MTL, in comparison to training a separate model for each task, are described later, together with a more in-depth comparison to SVDONets; see Sections 6.2.1.3 and 6.2.2.4.
To give some motivation here, we note that the SVDONet can be seen as a special case of MTL, for which the inputs for all tasks are the same, i.e., $x_i = x_i^{(k)}$. Approximating the coefficient of the i-th mode is thus the i-th task.

6 APPLYING THE MODE-BASED ERROR DECOMPOSITION

To showcase the usefulness of the SVDONet, we present three different insights gained from the SVDONet and the mode decomposition.
In Section 6.1, we decompose the error into the mode losses, to understand which modes contribute to the test and training errors. We investigate how well the learned coefficients of the different modes generalize. In Section 6.2, we investigate the interaction between different modes in two ways. In Section 6.2.1, we examine the stacked SVDONet, an architecture in which the coefficients for the different modes are learned separately. In Section 6.2.2, we analyze the influence different modes have on each other in the standard SVDONet architecture.
Throughout this chapter, we use the following terms. Since mode i is associated with the singular value σ_i, and since the singular values are ordered descendingly, we refer to modes with low indices $i \lessapprox 10$ as *large modes* and to modes with high indices $i \gtrapprox 30$ we refer to as *small modes*. Additionally, the term *intermediate modes* refers to modes in between. The terms large, intermediate and small are meant for orientation, not for assigning specific numbers.

6.1 SVDONET MODE LOSS DISTRIBUTION

In this section, we investigate how training and test branch errors are distributed over the individual modes. We first consider SVDONets trained using gradient descent (GD). We then propose a modified training algorithm, through re-weighting the different terms in the loss function. This leads to a lower loss in general, and specifically to more evenly distribution of the mode losses. Something similar is observed, when considering the Adam optimizer, instead of GD. Lastly, we discuss the impact of spectral bias on the mode loss distribution.

6.1.1 *Gradient Descent Training*

We start by looking at SVDONets trained using GD. In Fig. 13 (center column) the weighted mode losses for training (top) and test data (bottom), $\sigma_i^2 L_i/m$ are shown over the course of the training. They are normalized with m, the number of input functions in the respective data set. The figures also show the base losses σ_i^2/m_{train} and s_i^2/m_{test}, which are achieved by $b_i = 0$ (see Section 5.2). Note that test and training mode losses are very similar here. We observe that only the loss of the largest ≈ 10 modes is significantly lowered below the base loss, over the course of the training process.
The distribution of the weighted mode losses can be summarized like this: On the one hand, we have the well-approximated large modes, which contribute a low loss. On the other hand, we have the small modes, whose contribution is very small due to the small singular values. This leaves the intermediate modes, who thus have the highest weighted mode losses. In the standard example problem, $12 \leqslant i \leqslant 17$ are the intermediate modes. To understand why only the largest

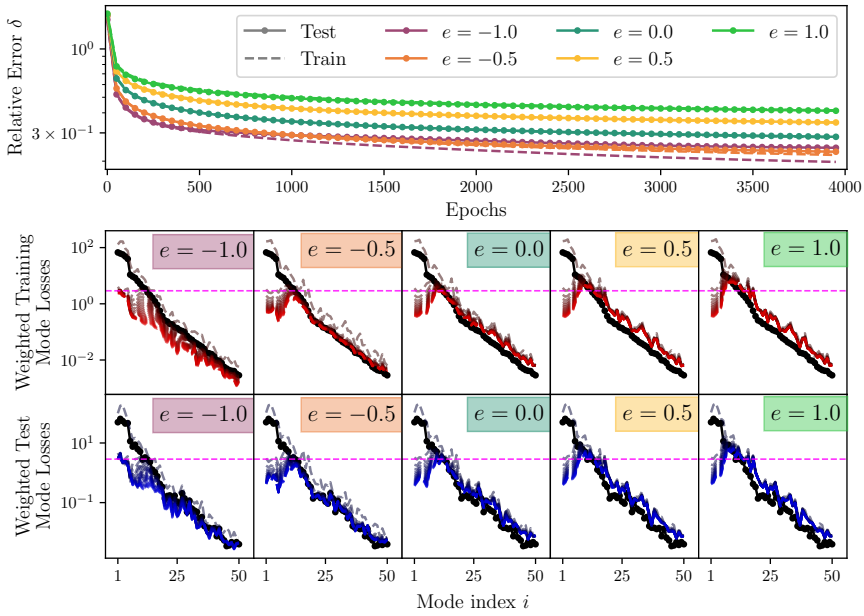


Figure 13: **Model Performance across different Exponents e and training epochs for SVDONets trained using GD. Top panel:** Relative error $\delta = \|A - \tilde{A}\|_F / \|A\|_F$ for both training (dashed lines) and test (dot-solid lines) data across different exponents ($e = -1.0, -0.5, 0.0, 0.5, 1.0$) over 4000 epochs. **Center and bottom row:** Weighted training (center row) and test (bottom row) mode losses at different training steps, colored from gray (initial) to red/blue (final). Each column corresponds to a different exponent e. **The third column shows the SVDONet trained using the standard loss ($e = 0$).** The center and bottom row plots also contain the respective base losses in black, and a pink dashed horizontal line marking the maximum mode loss in the last training epoch of $e = 0$, facilitating comparison between different exponents e.

modes are approximated well, we recall that the training loss can be decomposed into the mode losses

$$\mathcal{L}_{tr}(\theta) = \frac{1}{nm_{tr}} \varepsilon_B = \frac{1}{nm_{tr}} \sum_{i=1}^{N} \sigma_i^2 L_{i,tr}(\theta),$$

where θ are the parameters, such as weights and biases. Hence, the gradient with respect to the parameters θ of the loss can also be decomposed

$$\nabla_\theta \mathcal{L}_{tr}(\theta) = \frac{1}{nm_{tr}} \sum_{i=1}^{N} \sigma_i^2 \nabla_\theta L_{i,tr}(\theta).$$

Since the singular values σ_i vary hugely in size, the different modes are very differently represented in the gradient. Thus, the coefficients of the modes with smaller singular values (*smaller modes*) only change negligibly - the gradient with respect to them is multiplied with a very small number (compared to the *large modes*, i.e., modes with large singular values). For more training epochs, the loss of the first few modes continues to decrease, whereas the losses of smaller modes in practice is not lowered significantly.

6.1.2 *Gradient Descent Training with Modified Loss Weighting*

What happens, if we increase the smaller modes' contribution in the gradient? To emphasize different modes, we introduce a modified loss, by re-weighting the mode losses using an exponent $e \in \mathbb{R}$.

$$\mathcal{L}_{e,tr} := \frac{1}{nm_{tr}} \sum_{i=1}^{N} \sigma_i^{2+2e} L_{i,tr}$$

Note that we use $\mathcal{L}_{e,tr}$ only to compute the gradients, and hence the parameter updates, not as a performance metric. Furthermore, for GD we divide the learning rate by σ_1^{2e}. Thus, the contribution of $\nabla L_{1,tr}$ in $\nabla \mathcal{L}_{e,tr}$ is invariant of e, and hence the updates are neither exploding for $e > 0$, nor vanishing for $e < 0$.
To recall, the idea behind this re-weighting is that small modes are not acted on by gradient descent, since the gradients with respect to the large modes dominate the total gradient, and hence the parameter update direction. Re-weighting with $e < 0.0$ emphasizes the small modes, while $e > 0$ emphasizes the large modes even more and for $e = 0$, the standard loss is recovered. Specifically, for $e = -1$ all modes, independent of their singular value, are equally weighted. This is equivalent to using ε_C (Equation 14), instead of ε_B in the loss function. It should, however, not be confused with the POD-DeepONet in which the trunk matrix is chosen as $T = \Phi_1$; see Section 5.4. For the POD-DeepONet the singular values are moved to the branch network, i.e., the optimal coefficient for mode i is $\sigma_i v_i$, in contrast to v_i for the SVDONet. Hence the POD-DeepONet and the SVDONet for $e = 0$ use the same mode-weighting.
Obviously, the large modes are more relevant for the approximation, yet the smaller modes retain some influence. As discussed, a branch network of fixed architecture achieves higher approximation errors when learning coefficients for $N = 100$ basis functions compared to $N = 20$ basis functions. Similarly, we expect the large modes' coefficients to get worse, when forcing it to approximate the coefficients of more modes through $e < 0.0$. However, it is not clear exactly how strong the impact on the large modes' coefficients will be. Moreover, will the large modes' higher mode loss be compensated by a significantly lower loss on the smaller modes?
We first discuss the re-weightings effect on the mode losses for the training data, see Fig. 13 (center row). For $e = -1.0$, a loss significantly below the base loss is reached for all modes. Compared to $e = 0$, a significantly lower loss for all but the ≈ 10 largest modes is reached. For $e = -0.5$, the largest ≈ 10 modes display a low loss, while all others are reduced to the base loss. For $e \geqslant 0$, only the first 10, 8 and 5 modes, respectively, display low mode losses, with the first modes loss L_1 being similar for $e = 0.0, 0.5$ and 1.0. Furthermore, for $e \geqslant 0$, the mode losses of the small modes are larger than the base loss, i.e., they are entirely neglected by the optimizer. Even though the loss of the first few modes is significantly larger for $e = -1.0$ compared to the other values of e, the loss reduction through the small modes compensates this and the total training loss is lowest for $e = -1.0$ (top panel).
We now discuss the re-weighting's effect on the mode losses for the test data, see Fig. 13 (bottom row). For $e > -1.0$, the mode losses for test and training data are similar. For $e = -1.0$, however, only the test and training losses of large modes are similar. For small modes and $e = -1.0$, the training loss is significantly smaller than the test loss. In fact, the test loss is similar to the base loss for the small modes. This overfitting of the small modes means that the low training loss for $e = -1.0$ is

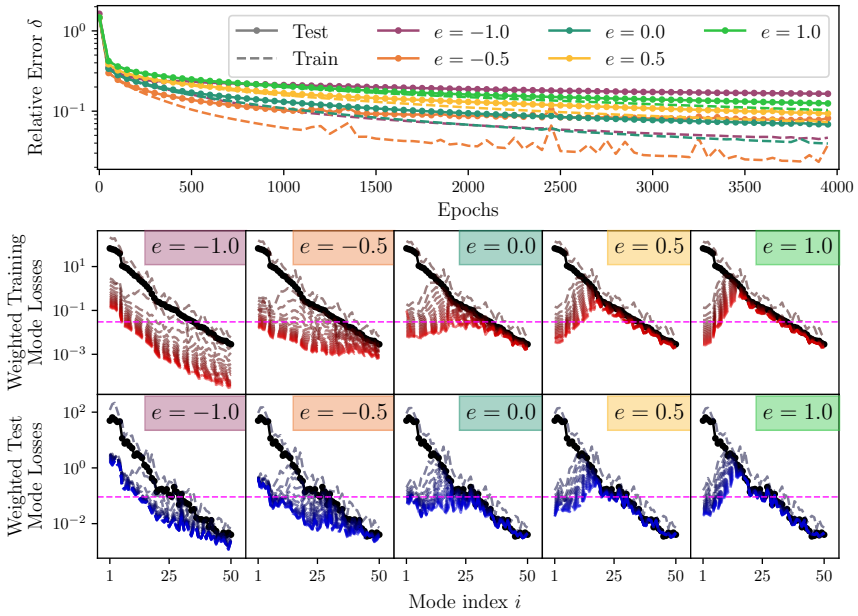


Figure 14: **Model Performance across different Exponents e and training epochs for SVDONets trained using Adam.** Layout and parameters identical to Fig. 13, with Adam replacing GD. **Top panel:** Relative error $\delta = \|A - \tilde{A}\|_F / \|A\|_F$ for both training (dashed lines) and test (dot-solid lines) data across different exponents ($e = -1.0, -0.5, 0.0, 0.5, 1.0$) over 4000 epochs. **Center and bottom row:** Weighted training (center row) and test (bottom row) mode losses at different training steps, colored from gray (initial) to red/blue (final). Each column corresponds to a different exponent e. **The third column shows the SVDONet trained using the standard loss ($e = 0$).** The center and bottom row plots also contain the respective base losses in black, and a pink dashed horizontal line marking the maximum mode loss in the last training epoch of $e = 0$, facilitating comparison between different exponents e.

not directly translated to a low test loss. In fact, $e = -0.5$ achieves the lowest test loss, due to (a) a lower loss on the largest modes than $e = -1.0$ and (b) a lower loss on the intermediate / smaller modes than $e \geqslant 0$.

6.1.3 *Adam Training*

We now examine the mode loss distribution for a re-weighted loss for SVDONets trained using Adam, since Adam adapts learning rates per parameter and may mitigate the poor performance on small modes; see Section 2.3.2.
We first discuss the standard loss ($e = 0$). When comparing the total loss for Adam, Fig. 14, and GD, Fig. 13, one observes that Adam achieves significantly lower test and training losses than GD. Furthermore, in contrast to GD which is underfitting, the SVDONet trained with Adam and $e = 0$ is overfitting, i.e., a widening gap between training and test loss can be seen. We observe that, while at very different speeds, finally all training mode losses significantly decrease below the base loss (not visible for all modes in the first 4000 epochs). Furthermore, we observe a significant loss reduction on the first 30 modes, on both test and training data, for Adam. Judging from these observations, Adam seems to be more similar to

GD with $e = -1$, than to GD with $e = 0$. Thus, we argue that the large difference in magnitude between the singular values of different modes, and hence their contributions to the gradient, appears to be one of the reasons why adaptive gradient optimization schemes perform much better than GD. Numerical examples show that adaptive gradient schemes without momentum, such as AdaGrad also perform significantly better than GD.
For Adam, the distribution of the weighted mode losses can be summarized like this: On the one hand, the largest ≈ 20 modes are well-approximated. On the other hand, we have the small modes, whose contribution is very small due to the small singular values. This leaves the intermediate modes, whose loss is significantly lower than the base loss, but due to the size of the singular values, they nevertheless have the highest weighted mode losses.

6.1.4 *Adam Training with Modified Loss Weighting*

When considering $e \neq 0$, see Fig. 14, we observe the re-weighting's strong effect on the mode level. We first discuss the training loss (center row). For $e > 0.0$, the loss for the first few modes is slightly lower than for $e = 0$, while the loss for all other modes is increased. Note that Adam, for all e, reduces every mode loss at least to its base loss, unlike GD with $e \geqslant 0$. For $e < 0.0$, the loss for the first few modes is larger than for $e = 0$, while the losses for all other modes are significantly lower. In particular, $e = -1.0$ achieves an approximately homogeneous unweighted mode loss, i.e., $L_{i,tr} \approx$ constant. For the training loss, the optimal balance between (a) prioritizing the large modes and (b) not completely ignoring the small modes is achieved by $e = -0.5$.
We now discuss the test loss (bottom row). For an SVDONet trained with $e > 0.0$, the test and training mode loss distributions are qualitatively similar. For $e \leqslant 0$, however, the low training mode losses on small modes do not transfer to the test losses. Recall that we observed the same overfitting on the small modes for GD with $e = -1.0$. Thus, for $e < 0$ and the test data, the higher losses on the large modes are no longer compensated by the loss reduction on the small modes. In fact, the total test error is thus lowest for $e = 0.0$ (top panel).
However, it is important to note that Adam for all considered values of e overfits on every mode, not exclusively on the small modes.
Interestingly, large modes generalize so much better that there are some example problems (e.g., the KdV equation with $\tau = 1$) for which $e = 1.0$ achieves a slightly lower test error than $e = 0.0$.

6.1.5 *Spectral Bias*

We now investigate the impact of spectral bias in the branch network on the mode loss distribution.
The (inverse) singular values, the unweighted mode losses, and the frequencies f_i of the right-singular functions ρ_i are shown in Fig. 15 for the example problems: advection-diffusion equation with $\tau = 0.5$, KdV equation with $\tau = 0.2$, KdV equation with $\tau = 0.6$ and Burgers equation with $\tau = 0.1$. To show the trend of the frequencies of ρ_i, we arbitrarily choose two estimation methods: TV norm with $k = 3$ and the LE norm with $k = 50$. Note that each plot in Fig. 15 contains three different y-axis scales, all of which are logarithmic. One for the inverse singular

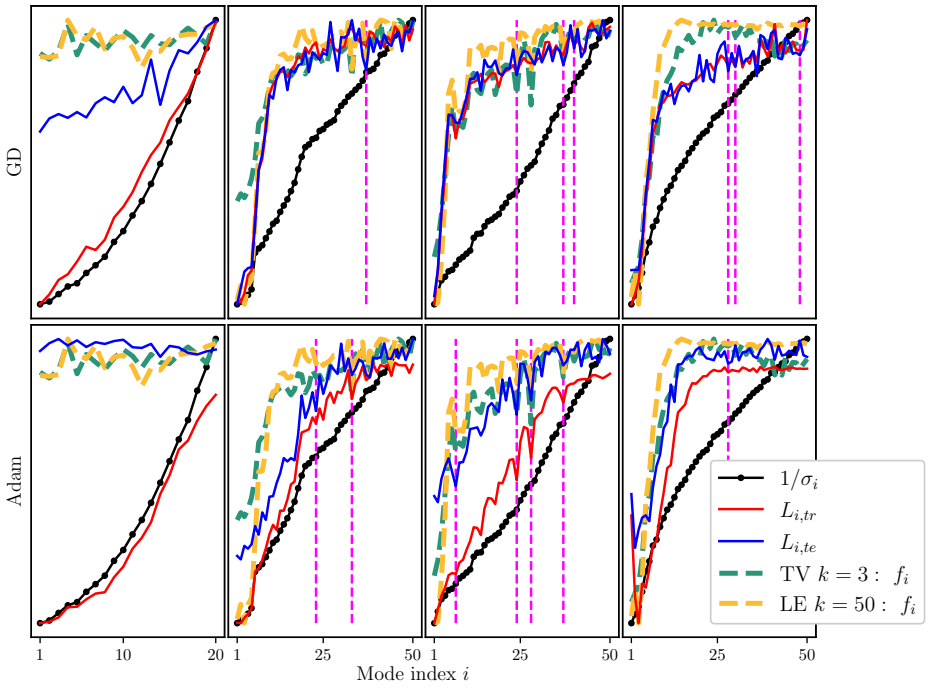


Figure 15: **Spectral Bias in SVDONets for various Example Problems.** Inverse singular values $1/\sigma_i$ (black), unweighted training (red) and test (blue) mode losses, and the ρ_i's frequency estimated via the TV norm with $k = 3$ (green) and LE $k = 50$ (yellow) are shown. The inverse singular values, the mode losses and the frequencies all have individual (logarithmic) scales. A mode is marked with a pink dashed line, if it exhibits both a frequency and a loss dip. **Top row:** SVDONets trained using GD. **Bottom row:** SVDONets trained using Adam. **Example problems (by column):** advection-diffusion equation with $\tau = 0.5$, KdV equation with $\tau = 0.2$, KdV equation with $\tau = 0.6$ and Burgers' equation with $\tau = 0.1$.

values (black), one for the mode losses (red and blue), and one for the frequencies (green and yellow). This is done to facilitate the direct comparison between the different quantities. This specific scaling implies that if two quantities x and y from different scales appear to be equal in the plot, then there exist $\alpha, \beta \in \mathbb{R}$, such that $\alpha x^\beta = y$. We then say x and y exhibit a power law dependence. Furthermore, $1/x$ and y then also exhibit a power law dependence.

We first discuss the results for the KdV and the Burgers equation (second-fourth column in Fig. 15). For SVDONets approximating the solution operator of the KdV or Burgers' equation, a clear correlation between f_i, the frequency of ρ_i, and the mode losses, for test and training data, is observed. For some example problems mode loss and frequency exhibit an approximate power law dependency. Recall that mode i has a frequency dip, if $f_i < \min(f_{i-1}, f_{i+1})$; see Section 3.3. The correlation can be seen very clearly for some of the dips in the spectrum for example problems based on the KdV equation; we observe, especially for Adam, that modes with frequency dips commonly have significantly smaller mode losses than their neighbors, i.e., they have *loss dips*. Note that, if the mode loss was purely influenced by the singular value, i.e., the optimizer prioritizing the more relevant modes, there should not be any loss dips. Modes which exhibit both frequency dips and loss dips are highlighted in Fig. 15 with pink dashed lines. Since both test and training mode losses show a correlation with the frequency, the branch

network is learning a generalizing approximation for low-frequencies, while it is underfitting for high-frequencies.
We now discuss the results for the advection-diffusion equation (first column). As discussed in Section 3.3, the right-singular functions ρ_i have constant frequencies. Furthermore, we observe a power law dependency between the singular values and the training mode losses. In this case this corresponds to very low training mode losses for the large modes. However, we observe a constant test mode loss $L_{i,te}$. Thus, we observe significant overfitting. Due to the different input dimensions M for the different example problems, a reliable method to compare the frequencies of the ρ_i between different example problems is not available. Thus, a spectral bias based explanation of why SVDONet overfit more for the advection-diffusion equation than for other equations cannot be given.

6.1.6 *Conclusion*

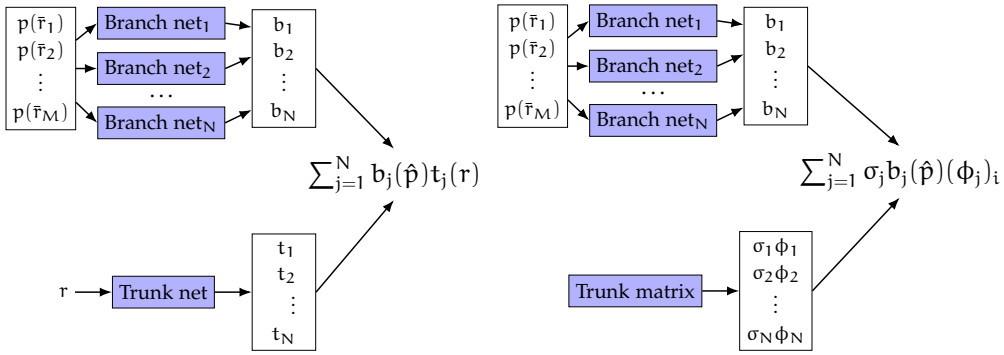
This section has three main observations.

- For SVDONets trained with GD only the coefficients of the first few modes are approximated well, since the gradients of the smaller mode losses are underrepresented. Thus the modes that have the highest weighted loss are the intermediate ones.
- By explicitly modifying the loss function or using adaptive gradient schemes, more modes can be approximated well.
- For both Adam and GD, small modes don't generalize well: even if their training loss is lowered, their test loss is not.

Furthermore, we show that for example problems based on the KdV or Burgers' equation there is a clear correlation between the mode losses and the frequencies of the right-singular functions, indicating the existence of strong spectral bias. This is highlighted by the coinciding frequency and loss dips. However, not all frequency dips show loss dips, furthermore, since the frequency f_i, for most examples, generally increases with i, we cannot conclude the true relevance of the frequency, compared to the singular value, for the approximation error. To study the impact of spectral bias in more depth, example problems with various spectral properties and better frequency estimation methods are necessary.
Furthermore, it is important to note that it is not clear, to which extent the poor generalizability of the small modes is a shortcoming of SVDONets, potentially caused by spectral bias, and to which extent these modes contain noise introduced by the numerical methods used to generate the training and test data.

6.2 COUPLING BETWEEN DIFFERENT MODES

In the previous section we analyzed the losses corresponding to different modes. While decomposing the error into the different mode losses yields insights into how well the individual modes are approximated, the coefficients of the different modes are not computed independently in the standard SVDONet. In Section 6.2.1 this is investigated by comparing the standard SVDONet to a modified SVDONet, which computes the coefficient of each mode separately. In Section 6.2.2 we return to the standard SVDONet and examine how the coefficients of the different modes are coupled in parameter space.



(a) Stacked DeepONet (b) Stacked SVDONet

Figure 16: **Comparison of the Architectures behind the stacked DeepONet and the stacked SVDONet.** Fig. 12 shows the unstacked DeepONet and the unstacked SVDONet for reference. Adapted and reproduced from [34].

6.2.1 *Architectural Mode Coupling*

In this section, the performance of the standard SVDONets and stacked SVDONets is compared. This is done to study how the approximation of the different coefficients is impacted by isolating them from each other in the neural network.

6.2.1.1 *Stacked DeepONet*

Together with the standard (or unstacked) DeepONet, Lu et al. proposed the so-called stacked DeepONet [34]. While the branch network in the unstacked DeepONet consists of one multi-layer perceptron with N output neurons, the branch network in the stacked DeepONet consists of N multi-layer perceptrons with one output neuron each. The stacked DeepONet can be seen in Fig. 16a, the corresponding unstacked DeepONet is shown in Fig. 12a.
The different basis functions' coefficients b_j are thus separated from each other in the stacked DeepONet. In [34] it was reported that stacked DeepONets tend to reach lower training losses and higher test losses, compared to unstacked DeepONets. It is intuitively not clear, why this *neuron-sharing* of the coefficients of different basis functions would be beneficial for generalization. Note that [34] compared DeepONets in which the width of the hidden layers of the unstacked branch network was equal to the width of the hidden layers in each of the stacked branch sub-networks. Thus, the stacked DeepONet in their comparison has significantly more parameters.
The concept of stacked DeepONets can readily be transferred to the SVDONet. The SVDONet's one branch network is replaced with N multi-layer perceptrons, each approximating the coefficient corresponding to one mode; see Fig. 16b.

6.2.1.2 *Comparison of Stacked and Unstacked SVDONets*

How does the performance of stacked and unstacked SVDONets differ? To answer this, we compare a stacked to an unstacked SVDONet. Since increasing the number of parameters in a neural network can lead to overfitting [15], we compare networks that have the same number of parameters. In our case the unstacked SVDONet's branch network has depth D and width w_{unst}. The individual branch networks of the stacked SVDONet have the same depth D and different widths

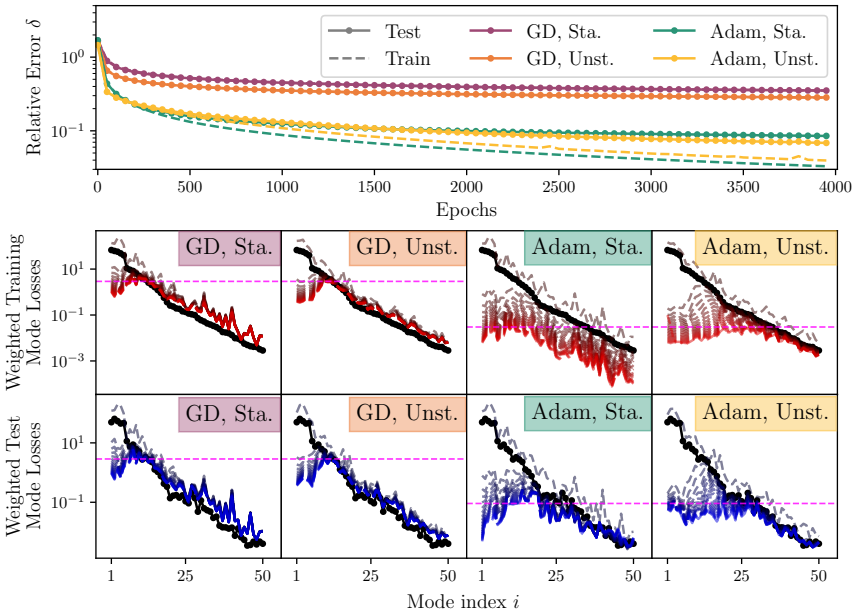


Figure 17: **Model Performance of stacked and unstacked SVDONets trained using GD and Adam. Top panel:** Relative error $\delta = \|A - \tilde{A}\|_F / \|A\|_F$ for both training (dashed lines) and test (dot-solid lines) data over 4000 epochs. **Center and bottom row:** Weighted training (center row) and test (bottom row) mode losses at different training steps, colored from gray (initial) to red/blue (final). Columns correspond to optimizer (GD or Adam) and architecture (stacked (Sta.) or unstacked (Unst.)) as indicated by the labels. The center and bottom row plots also contain the respective base losses in black, and a pink dashed horizontal line marking the maximum mode loss of the unstacked SVDONet with the respective optimizer.

$w_{sta} < w_{unst}$; see Appendix A.8. However, as described in more detail in Appendix A.8, we also conduct experiments comparing stacked and unstacked SVDONets with $w_{sta} \approx w_{unst}$. These experiments demonstrate that, as was found for DeepONets [34], stacked SVDONets tend to overfit more than unstacked SVDONets with same individual widths,

We now discuss the results for $w_{sta} < w_{unsta}$. In Fig. 17, the training and test loss curves of the stacked and unstacked SVDONet and the training and test mode losses of both are shown.

For SVDONets trained using GD, the unstacked SVDONet achieves lower test and training errors than the stacked SVDONet trained with GD. The mode loss distributions are qualitatively very similar for stacked and unstacked SVDONets.

In the following, we discuss stacked and unstacked SVDONets trained with Adam. We observe that the unstacked SVDONet always achieves a lower test error. For the training error, it depends on the example problem. This shows that, for our example problems and the Adam optimizer, even for the same number of parameters with fixed depth, the neuron-sharing helps generalization.

When investigating the mode errors of the stacked and unstacked SVDONets, we observe that the stacked SVDONet achieves much lower training losses on most modes. However, for some intermediate modes (e.g., $10 \leqslant i \leqslant 15$ in Fig. 17) the unstacked SVDONet achieves lower training losses.

It is important to note that even for modes where comparable training losses are achieved, the test losses of the stacked and the unstacked SVDONet might still differ. This means that the generalization abilities of the two architectures cannot be solely explained by (a) which modes to approximate how well and (b) how well do these modes generalize, since there is no universal generalizability of modes.
For all considered examples, there are some large modes whose training loss is lower for the unstacked SVDONet, whereas their test loss is lower for the stacked SVDONet (in Fig. 17 these are modes $2-7$). The opposite can be observed for intermediate and small modes. There the stacked SVDONet significantly outperforms the unstacked SVDONet on the training data, but the unstacked SVDONet achieves a lower test loss. This holds for all considered example problems; in Fig. 17 these are modes $17-21$ and $35-49$. Thus, the neuron-sharing, in form of the unstacked SVDONet, seems to lead to more generalizable coefficients for intermediate and small modes, and worse generalization on large modes.

6.2.1.3 *Conclusion*

The first key observation from the previous section was that small modes don't seem to generalize well. The experiments with the stacked SVDONet strongly corroborate this hypothesis. Furthermore, we observe that the unstacking, or neuron-sharing, helps the total generalizability, but not for all modes. It seems to hurt the training loss and generalizability of the large modes, while improving the generalizability of smaller modes.
Note that the unstacked SVDONet with the mode loss decomposition can be seen as a specific multi-task learning (MTL) problem; see Section 5.4.5. I.e., the SVDONet is one model which has to solve N separate tasks; it has to approximate N right-singular functions ρ_i, for $i \in [N]$. In contrast, the j-th branch sub-network of the stacked SVDONet just has one task; approximate ρ_j. In MTL, the question of *(how) should the parameters associated with the different tasks be shared?* is very common [44]. There are many different arguments for so-called *hard parameter sharing*, i.e., all tasks should share most of (or all) of the parameters, which corresponds to the unstacked SVDONet. One of them is the following: Sharing the parameters can lead the model to learn features which work well for all tasks. If the tasks to learn are similar enough, the features who work well for all tasks are usually the ones that generalize well [44].
Importantly, our results suggest that even when practitioners are only interested in specific modes of system dynamics, such as particular resonance frequencies, the unstacked SVDONet architecture may still be preferable. The parameter sharing across modes appears to facilitate learning of generalizable features that improve approximation quality for the target mode, even if other output neurons are not directly used in the application.

6.2.2 *Update Based Mode Coupling*

For the comparison between the stacked and the unstacked SVDONet, we asked the following question: *If the branch network is rearranged such that the modes have separated parameters, while keeping the number of parameters constant, how does this affect generalization abilities?* In this section, we consider unstacked SVDONets and ask: *Which contribution to the loss comes from the coupling of the modes?* We are concerned with the coupling of the modes in the parameter space. Changing the value of

the i-th branch output neuron does not affect the SVDONet output in any other mode due to the modes' orthogonality ($\phi_i^T \phi_j = 0$ for $i \neq j$). However, updating the parameters to achieve a lower loss in mode i can significantly change the values of all branch output neurons due to the neuron-sharing discussed in the previous section.
In Section 6.2.2.1, we will give a definition for *mode coupling in parameter space*. This definition relies on computing the parameter updates, and we thus also refer to it as *update based mode coupling*. Sections 6.2.2.2 and 6.2.2.3 will examine the strength of the mode coupling for SVDONets trained with GD and Adam, respectively. In the entire Section 6.2.2, mode coupling always refers to update based mode coupling, not architectural mode coupling.

6.2.2.1 *Defining Update Based Mode Coupling*

Unless stated otherwise, any loss function in this section is evaluated at the current parameters θ, and the gradients are computed with respect to θ. Thus we write ∇L_i instead of $\nabla_\theta L_i(\theta)$ and $\nabla \mathcal{L}$ instead of $\nabla_\theta \mathcal{L}(\theta)$. Consider a GD parameter update

$$\delta\theta = -\alpha \nabla \mathcal{L}_{tr} = -\frac{\alpha}{n m_{tr}} \sum_{j=1}^{N} \sigma_j^2 \nabla L_{j,tr}.$$

Choosing a sufficiently small learning rate α allows us to approximate the change in the training and test loss function using a first order Taylor expansion. As we will see, the Taylor expansion allows us to partition the loss change into coupling and non-coupling parts.
Recall that we use the symbols L_i and $\mathcal{L}$ when an equation applies to both the test and training losses. If necessary, the test and training losses are specified using the subscripts te and tr, respectively.
For the i-th mode loss the first order Taylor expansion of the loss change ΔL_i is

$$\begin{aligned}\Delta L_i &= L_i(\theta + \delta\theta) - L_i(\theta) \\ &\approx (\nabla L_i)^T \delta\theta = -\frac{\alpha}{n m_{tr}} \sum_{j=1}^{N} \sigma_j^2 (\nabla L_i)^T \nabla L_{j,tr}.\end{aligned}$$

Then, the first order Taylor expansion of the total loss change $\Delta \mathcal{L}$ is

$$\begin{aligned}\Delta \mathcal{L} &= \mathcal{L}(\theta + \delta\theta) - \mathcal{L}(\theta) = \frac{1}{n m_{tr}} \sum_{i=1}^{N} \sigma_i^2 \Delta L_i \\ &\approx -\frac{\alpha}{n^2 m_{tr}^2} \sum_{i=1}^{N} \sigma_i^2 \sum_{j=1}^{N} \sigma_j^2 (\nabla L_i)^T \nabla L_{j,tr}.\end{aligned}$$

We denote $\eta = \frac{\alpha}{n^2 m_{tr}^2}$ to simplify the notation. By introducing the matrices D and S this can be rewritten as

$$\begin{aligned} D &:= \left[\sigma_1^2 \nabla L_1 \ \dots \ \sigma_N^2 \nabla L_N\right] \in \mathbb{R}^{|\theta| \times N} \\ S &:= -\eta D^T D_{tr}, \text{ and} \\ \Rightarrow \Delta \mathcal{L} &\approx \sum_{i=1}^{N} \sum_{j=1}^{N} S_{ij}. \end{aligned} \tag{32}$$

Here $|\theta|$ is the number of parameters; see Appendix A.8.1 for a precise definition of $|\theta|$. The sum of the matrix entries can be split into the *diagonal term*

$$d := \sum_{i=1}^{N} S_{ii} = -\eta \sum_{i=1}^{N} \sigma_i^4 (\nabla L_i)^T \nabla L_{i,tr},$$

and the *off-diagonal term*

$$\Omega := \sum_{i=1}^{N} \sum_{j \neq i} S_{ij} = -\eta \sum_{i=1}^{N} \sigma_i^2 \sum_{j \neq i} \sigma_j^2 (\nabla L_i)^T \nabla L_{j,tr},$$

such that $\Delta\mathcal{L} \approx \sum_{i=1}^{N} \sum_{j=1}^{N} S_{ij} = d + \Omega$. Thus, $d + \Omega$ is the first order Taylor expansion of the loss change. For the training loss, the diagonal term is given as

$$d_{tr} = -\eta \sum_{i=1}^{N} \sigma_i^4 (\nabla L_{i,tr})^T \nabla L_{i,tr} = -\eta \sum_{i=1}^{N} \sigma_i^4 \|\nabla L_{i,tr}\|_2^2 \leqslant 0.$$

This is not given for the test loss.
Furthermore note that $S_{ij} = -\eta \sigma_i^2 \sigma_j^2 (\nabla L_i)^T \nabla L_{j,tr}$ is the first order Taylor expansion of the weighted loss change of mode i given the parameter update $-\eta \sigma_j^2 \nabla L_{j,tr}$. This parameter update is the GD update to minimize the j-th mode loss, scaled with σ_j^2. Thus, S_{ij} contains the information how the i-th mode loss changes through a parameter update improving the j-th mode loss. Note that $S_{ij,tr} = S_{ji,tr}$. Thus, for the training loss, coupling is symmetric, and $S_{tr} = S_{tr}^T$. Thus, Ω contains the coupling between the different modes in parameter space.
In the following we consider the case $d + \Omega < 0$. This is given, if the first order Taylor expansion is a good approximation, i.e., $d + \Omega \approx \Delta\mathcal{L}$, and the training is successful (e.g., no overfitting), i.e., $\Delta\mathcal{L} < 0$. Then, since (a) Ω contains the coupling and (b) $d + \Omega$ approximates the loss change, the overarching question of Section 6.2.2 (*Which contribution to the loss comes from the coupling of the modes?*) is reduced to the ratio of Ω and $d + \Omega$. We thus define

$$\gamma := \frac{\Omega}{d + \Omega}$$

as the (update based) relative mode coupling strength among all modes, or in short *relative coupling strength*. Note that for $\gamma < 0, \Omega > 0$, the coupling is detrimental to the training, while for $\gamma > 0, \Omega < 0$ the coupling is beneficial for training.

6.2.2.2 *Gradient Descent Training*

We now examine d and Ω for SVDONets trained with GD. In Fig. 18 the test and training loss over the epochs (left), the different contributions d, Ω, the first order Taylor expansion $d + \Omega$ and the total loss change $\mathcal{L}$ (center), and the relative coupling strength γ (right) are shown over the training course. As established in the previous sections, GD is underfitting. Thus the behavior of d and Ω is very similar for test and training data. We thus discuss the results for the training data. We start by noting that $\Delta\mathcal{L} \approx d + \Omega$ is in fact observed, i.e., the learning rate α is chosen to be sufficiently small. In this example we observe $\Omega > 0$. Similar to the loss change, the diagonal term d and the off-diagonal term Ω decay over the epochs. The relative coupling strength decreases from -1 to -7 over the first 4000 epochs. Thus, the relative contribution of the coupling grows over the course of

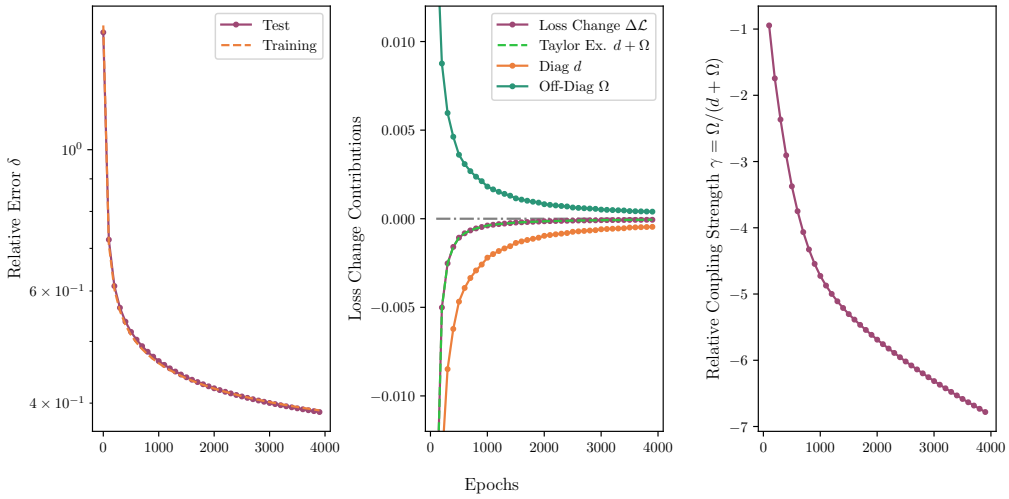


Figure 18: **Loss, Loss Change per epoch and Relative Coupling Strength over the training epochs for an SVDONet trained with GD. Left:** Training (dashed) and test (dot-solid) loss over 4000 epochs. **Center:** Loss change $\mathcal{L}$ (purple), first order Taylor expansion of loss change $d + \Omega$ (dashed, lightgreen), diagonal term d (orange) and off-diagonal term Ω (darkgreen) on the training data over 4000 epochs. **Right:** Relative coupling strength γ over 4000 epochs.

the training. Note that there are other example problems, for which Ω is initially negative. However, even for these examples, Ω then increases significantly over the epochs, becoming positive and showing detrimental coupling with $\gamma \approx -1$ after 4000 epochs.

Furthermore, we can inspect the contributions to the diagonal and off-diagonal parts through the entries of $S = -\eta D^T D_{tr}$; see Fig. 19. Since we focus our discussion on the training loss, we consider S_{tr}. Recall that S_{tr} is always symmetric and has non-positive diagonal entries. Note however that since we observe no overfitting for SVDONets trained using GD, we also observe $S_{tr} \approx S_{te}$. Hence our observations also hold for S_{te}.

For later epochs, we observe that there are certain regions in the matrix, in which most matrix entries are positive. More specifically, most entries in the first 10 rows and columns are positive (red). This means that the coupling between any two modes i, j is likely to be detrimental, if $i \leqslant 10$. We furthermore note that only the first 10 modes' loss is significantly lower than their base loss; we thus refer to them (a bit euphemistically) as *well-approximated*. Thus, the coupling between a well-approximated mode and any other mode is likely to be detrimental. However, the coupling between two poorly approximated modes ($i, j > 10$) is likely to be beneficial; $S_{ij} < 0$ (blue).

It is important to point out that this pattern is not a strict observation; there are some negative, off-diagonal matrix entries S_{ij} with $i, j \leqslant 10$.

However, what happens if we consider larger networks? We increase the width of the hidden layers, while keeping the depth of the networks and the inner dimension N constant.

Fig. 20 shows the negative relative coupling strength $-\gamma$ over the loss reduction $-\Delta\mathcal{L}$ for SVDONets of multiple widths w. We find strong decoupling with increased width. For the examples in which Ω is initially negative, decoupling is clearly observed for later epochs, when $\Omega > 0$. Note that the wider, decoupled SV-

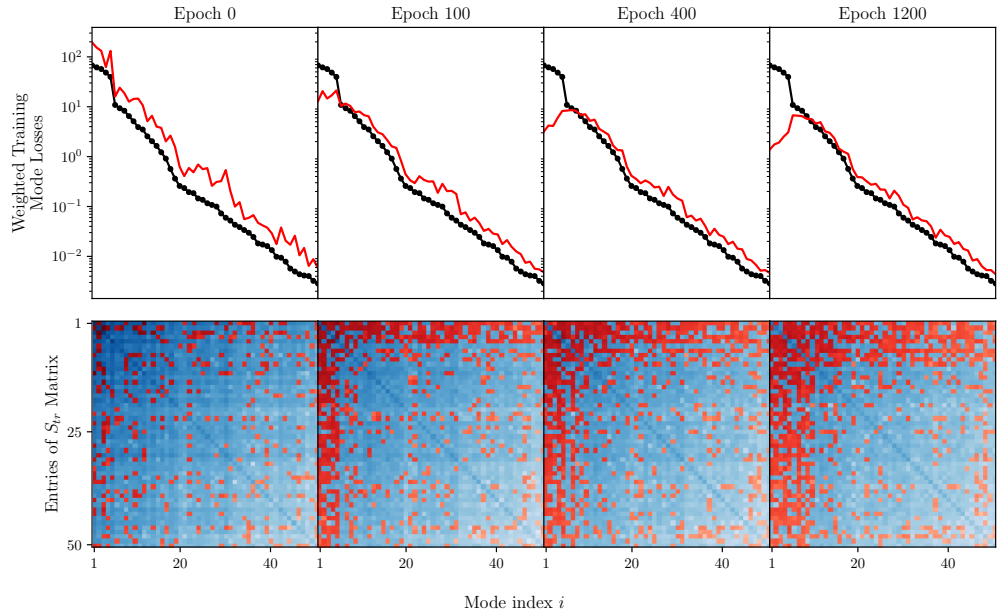


Figure 19: **Mode Losses and Entries of S_{tr} for an SVDONet trained with GD at 4 different epochs. Top row:** Weighted mode losses (red) and singular values (black) for each epoch. **Bottom row:** Entries of S_{tr} matrix. Red entries indicates a positive entry and blue indicates a negative entry (same colorscale for all epochs). The SVDONet shown here is the same as in Fig. 18.

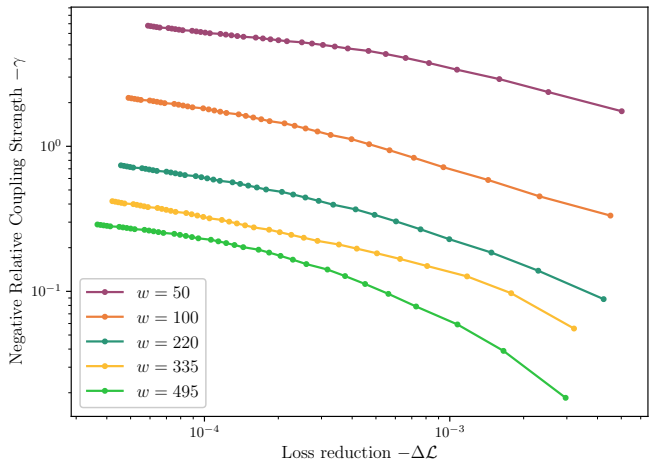


Figure 20: **Negative Relative Coupling Strength $-\gamma$ plotted over Loss Reduction $-\Delta\mathcal{L}$ for SVDONets of different Hidden-Layer Widths w.** The figure shows models with hidden layer widths 50 (purple), 100 (orange), 220 (darkgreen), 335 (yellow) and 495 (lightgreen).

DONets achieve lower test and training losses. Furthermore, we note that the pattern described for S_{tr} for the SVDONet with $w = 50$ weakens, as w increases, but the tendency for large positive matrix entries to come from the rows and columns associated with well-approximated modes stays.

We can now compare the unstacked SVDONet to the stacked net. For the stacked SVDONet we have that $\Omega = 0$, per definition, since we can divide the stacked network's parameters θ into N mode-specific parameters θ_i and then $\nabla_\theta L_i = \nabla_{\theta_i} L_i$.

Thus, the question whether mode coupling causes the difference in performance between stacked and unstacked SVDONets arises. In Section 6.2.1 it was found that for SVDONets trained using GD, an unstacked SVDONet consistently performs better than a stacked SVDONet on both test and training data. Since a detrimental coupling, $\Omega > 0$, is observed for many examples, for both training and test data, the mode coupling cannot be the main cause of performance difference between stacked and unstacked SVDONets.
An expanded discussion on reasons for these observations (strong detrimental coupling and decoupling for wider networks) and comparisons to similar phenomena are given in Section 6.2.2.4.

6.2.2.3 *Adam Training*

We can also apply this mode coupling analysis to SVDONets trained using Adam. We first investigate the difference between the actual loss change $\Delta\mathcal{L}$, the first order Taylor expansion of the loss change with the Adam update $(\nabla\mathcal{L})^T\delta\theta$ and the first order Taylor expansion of the loss change with the gradient descent update $(\nabla\mathcal{L})^T(-\eta\nabla\mathcal{L}_{tr}) = d + \Omega$, since the Taylor expansion is the foundation of our coupling definition. We compare two SVDONets. While both have the same depth and inner dimension N, the width of their hidden layers differ. The results are shown in Fig. 21. For the narrower branch network (width $= 50$), the first order Taylor expansion is a good approximation of the loss change, i.e., $\Delta\mathcal{L} \approx (\nabla\mathcal{L})^T\delta\theta$. Furthermore, the first order Taylor expansion of the loss change with gradient descent shows qualitative agreement with $\Delta\mathcal{L}$. For the wider branch network (width $= 335$) this is only true for early epochs. The actual loss reduction decays smoothly to 0 over all 4000 epochs. However, there are some epochs for which both Taylor expansions show spikes with strong loss reduction. For visual clarity, the figure only shows the loss change for every 100-th epoch, thus we expect there to be many more spikes, when considering all epochs. The loss reduction through the Taylor expansions of the GD update is larger than the reduction through the Adam update, and is usually a consequence of the diagonal term $d \approx (\nabla\mathcal{L})^T(-\alpha\nabla\mathcal{L}_{tr})$, while $\Omega \approx 0$ in these epochs. This shows (a) the difference between the Adam and GD update direction and (b) the relevance of higher derivatives for wider nets. We continue to consider the entries of S (see Equation 32) who sum up to the loss change according to the first order Taylor expansion of the loss with a gradient descent update, not an Adam update. However, the following should not be seen as an analysis of the optimization procedure, but as an analysis of the state (produced by the optimization procedure).

Training Data

Figure 22 shows the entries of S_{tr}. Recall that for GD, we observed most positive entries in the first few rows and columns of S_{tr}.
In later epochs, we observe a notably different but related pattern in wide SVDONets trained with Adam. This pattern consists of three parts.

(a) We observe negative off-diagonal entries in the top-left 10×10 block, i.e., $S_{ij} < 0$ with $i, j \leqslant 10$ (blue). This means that the well-approximated modes are coupled beneficially among each other.

(b) However, the coupling between a well-approximated mode and a poorly-approximated mode is detrimental (red).

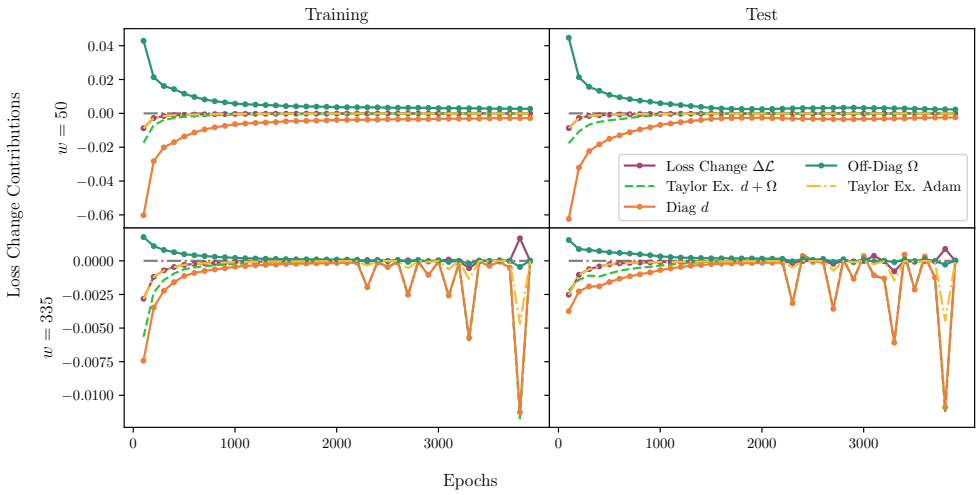


Figure 21: **Different Loss Contributions and Loss Change Approximations for SVDONets trained using Adam.** Loss change $\mathcal{L}$ (purple), first order Taylor expansion of the loss change with the gradient descent update $(\nabla\mathcal{L})^T(-\alpha\nabla\mathcal{L}_{tr}) = d + \Omega$ (dashed, lightgreen), diagonal term d (orange), off-diagonal term Ω (darkgreen) and first order Taylor expansion with the Adam update $(\nabla\mathcal{L})^T\delta\theta$ over 4000 epochs. **Left column:** Training losses. **Right column:** Test losses. **Top row:** SVDONet with hidden layer width $w = 50$. **Bottom row:** SVDONet with hidden layer width $w = 335$.

(c) The coupling among poorly approximated modes is beneficial.

Thus, observation (a) is the key difference between the pattern for GD, and the pattern for Adam, while observations (b) and (c) are present for both optimization algorithms. This pattern, beneficial coupling between well-approximated modes, strengthens over the epochs, leading to clearly visible region boundaries as the training progresses. The beneficial coupling between the well-approximated modes might be another key reason why Adam achieves lower losses than GD, next to the consideration of more modes.
Note that the visibility of this pattern varies strongly from example to example. For examples where the weighted loss for the first few modes is less uniform, the visibility strongly decreases.
An expanded discussion on possible reasons for this coupling pattern are given in Section 6.2.2.4.

Test Data

In the test case, S_{te}, such a pattern cannot be identified.
However, the inspection of S_{te}'s entries gives insight into another phenomenon that we call mode-internal overfitting (MIO). Here MIO means: *Ignoring all parameter updates, besides the one specifically for mode* i*, does mode* i *still overfit?*
We quantify MIO using two mathematical definitions. The intuitive definition for mode i is the following. If a parameter update $\delta\theta_i$ decreases the training loss of mode i, i.e., $L_{i,tr}(\theta + \delta\theta_i) < L_{i,tr}(\theta)$, but increases the test loss of mode i, i.e., $L_{i,te}(\theta + \delta\theta_i) > L_{i,te}(\theta)$, mode i is internally overfitting.
However, this definition is permissive to noise. We thus give a second stricter, noise robust criterion. Mode i is internally overfitting, if (a) the training loss is reduced, $\Delta L_{i,tr} < 0$, and (b) the test loss reductions is less than half the size of the training

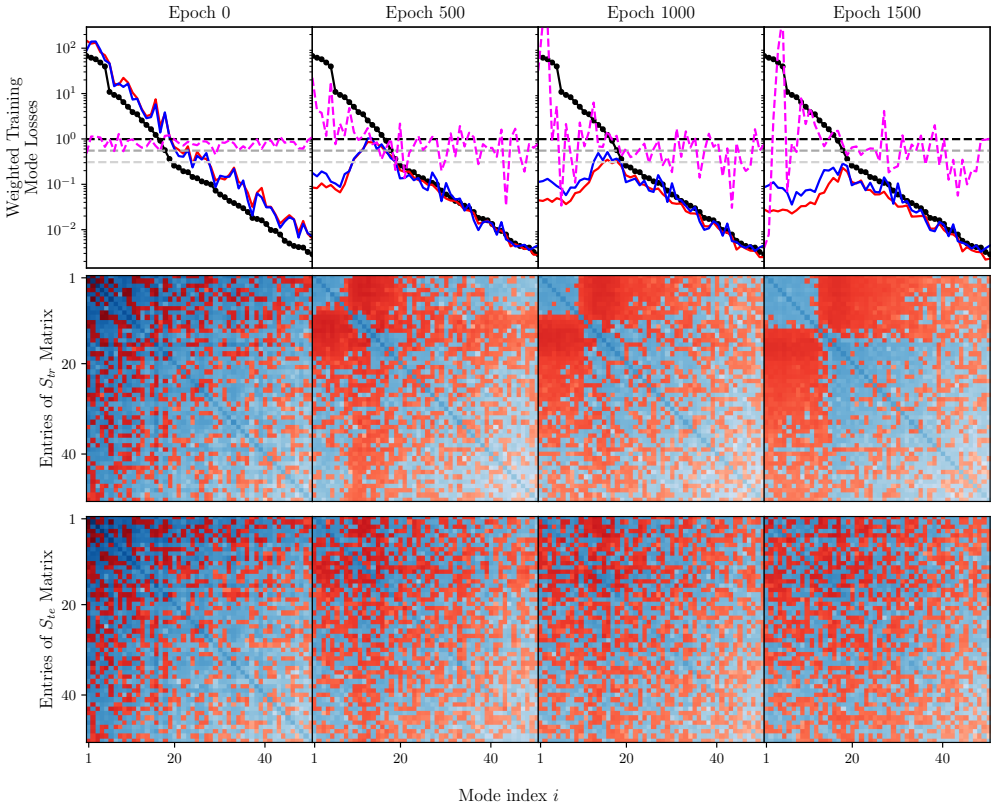


Figure 22: **Weighted Mode Losses and Entries of S Matrix.** *Top row:* Singular values (black), training mode losses (red), test mode losses (blue) and MIO indicator μ_i (pink) at epochs 0, 400, 800, 1200, and 1600. Note that the MIO indicator uses a different scale than the mode losses. This second scale is indicated by horizontal lines at 0 (light gray), 0.5 (dark gray) and 1 (black). *Center and bottom rows:* Training (center) and test (bottom) results. Heatmaps of matrix S entries at corresponding epochs, with red/blue indicating positive/negative values. Note that there are two separate colorscales, one for the training matrices S_{tr} for different epochs and one for the test matrices S_{te} for different epochs.

loss reduction, $-\Delta L_{i,te} < -\frac{1}{2}\Delta L_{i,tr}$. Thus the strict definition applies to a subset of the modes captured by the permissive definition.

We consider $\delta\theta_i = -\alpha\nabla_\theta L_{i,tr}(\theta)$ as the update to minimize the training loss of mode i, and the first order Taylor expansion of L_i. Thus, the question whether MIO takes place is reduced to the comparison of the diagonals of S_{tr} and S_{te}. We thus define the MIO indicator $\mu_i = S_{ii,te}/S_{ii,tr}$. Since $S_{ii,tr} < 0$ is given, a positive μ_i implies MIO according to the permissive criterion, and $\mu_i > \frac{1}{2}$ is equivalent to the stricter criterion. Figure 22 shows clear MIO for some modes. This is seen particularly in wider networks, which tend to overfit more. Of the 50 modes, 10-20 display MIO (depending on the criterion and epoch), demonstrating that while MIO is common, not all overfitting modes exhibit it.

Note that neither MIO nor the structured coupling are necessarily unique to Adam. However, the SVDONets trained with GD underfit. This might change for more training epochs, or a different learning rate, in which case MIO might be observed in, e.g., GD too.

6.2.2.4 *Conclusion*

In this section, we defined the update-based coupling strength γ. For SVDONets trained using GD, γ represents the ratio of Ω—the part of the loss change due to interactions between modes i and $j \neq i$ in parameter space—to the total loss change $d + \Omega$. For SVDONets with small hidden layer widths, we observe strongly detrimental coupling, i.e., $\Omega > 0$. Specifically, we find strong detrimental coupling between well-approximated modes, while poorly-approximated modes exhibit beneficial coupling among themselves. As the hidden layer width increases, γ decreases substantially, indicating that the modes progressively decouple with increasing network capacity.
When analyzing SVDONets trained using Adam, we observe distinct coupling patterns for certain example problems. Here, well-approximated modes exhibit beneficial coupling in the training loss, contrasting sharply with the GD case. However, examination of the test loss reveals that a substantial number of modes exhibit mode-internal overfitting: their individual gradient updates increase the test mode loss even without influence from other modes.

Similar to the discussion of the stacked and unstacked SVDONets (Section 6.2.1.3), we again relate our results to MTL. What we term detrimental coupling corresponds to *gradient conflicts* in the MTL literature [55]. Previous work has established that gradient conflicts intensify as training progresses [57], which aligns with our observation of γ evolving from -1 to -7 during training.
The specific coupling pattern – strong detrimental coupling between well-approximated modes – can be understood through task heterogeneity. While these modes possess the highest singular values and therefore dominate gradient updates, they attempt to approximate different right-singular functions. Consequently, their approximations require different parameter configurations, which generates gradient conflicts. In contrast, poorly-approximated modes do not receive sufficient gradient signals to develop strong internal representations. This limitation paradoxically gives them flexibility: they can adapt to parameter changes induced by other modes without generating significant conflicts. However, this flexibility is never used, since the singular values corresponding to these modes vanish in the gradient compared to the larger singular values.
The decoupling phenomenon observed in wider networks admits several complementary explanations.

- From a geometric perspective, the expected squared inner product between random vectors in $\mathbb{R}^k$ decreases as the dimension k increases [48]. Since the coupling contribution Ω scales with these inner products, wider networks naturally exhibit reduced coupling even under random parameter updates.

- From a mechanistic interpretability perspective, wider networks have been shown to develop more modular, disentangled representations [11, 38], with distinct features emerging in separate parameter subspaces. These features are thus decoupled in parameter space.

- Third, the so-called lottery ticket hypothesis [13] offers another perspective: wider networks contain more sparse subnetworks that can achieve comparable performance to the full network. In our context, this suggests that wider SVDONets may contain multiple *winning tickets*, i.e., independent subnetworks capable of handling different modes with minimal interference. As

width increases, the probability of finding non-overlapping winning tickets for different modes grows, naturally leading to the observed decoupling.

The coupling patterns under Adam optimization reveal additional subtleties. The beneficial coupling between well-approximated modes manifests only in the training loss, suggesting that this apparent cooperation may be a form of collective overfitting. The presence of mode-internal overfitting (MIO) is, to some extent, expected: as in most machine learning tasks, finite training samples eventually become exhausted, leading to overfitting after sufficient epochs. In our framework, this manifests as MIO. Our analysis thus suggests that some modes in the SVDONets are already close to their generalizability limits. However, note, that these limits can be surpassed using, e.g., explicit regularization techniques.

7 DISCUSSION

7.1 SUMMARY

In this thesis, we investigated the sources of the approximation error in DeepONets by systematically decomposing the total error and analyzing the learning dynamics of different solution components.

RQ I: Error Distribution Between Basis Functions and Coefficients

We first applied the error decomposition framework of Lanthaler et al. to partition the total approximation error into trunk error (from learned basis functions) and branch error (from their coefficients). Our analysis across multiple PDE examples revealed that while DeepONets successfully learn linearly independent basis functions that reduce the trunk error as the inner dimension grows, the total approximation error for large inner dimensions is dominated by the branch error. This finding established that the primary bottleneck in DeepONet performance lies in learning accurate coefficients rather than basis functions.

RQ II: Coefficient Approximation and Mode Analysis

To investigate which coefficients are poorly approximated and why, we constructed the SVDONet, which replaces the learnable trunk network with optimal basis functions (modes) obtained through SVD of the training data. This architectural modification enabled us to decompose the branch error into mode-specific contributions, providing direct insight into how the coefficients of different spatial modes are approximated.

RQ II.1: Coefficient Error Distribution

Our mode-decomposition analysis revealed fundamental limitations in how optimization algorithms handle coefficients of different modes:

(a) Optimization scheme: For our examples, gradient descent only accurately approximates coefficients of the leading ≈ 10 modes due to the dominance of large singular values in gradient updates. The contributions of smaller modes are effectively drowned out during training. Adaptive gradient schemes like Adam and explicit loss re-weighting can improve approximation of more modes by addressing this gradient imbalance.

(b) Generalization patterns: While coefficients of leading modes generalize well from training to test data, smaller modes exhibit poor generalization even when their training losses are reduced. This suggests a fundamental difference in how different modes can be learned from finite data.

Furthermore, we compare the i-th mode error to the frequency corresponding to the i-th right singular function. For some example problems, we observe a clear

positive correlation. This suggests spectral bias in the branch network as one of the reasons for poor accuracy on the intermediate and small modes.

RQ II.2: Mode Interactions

We investigated coupling between different modes through two complementary approaches:

(a) Architectural coupling: Comparing stacked (separate networks per mode) versus unstacked (shared hidden layers) SVDONets revealed that parameter sharing improves overall generalization, particularly for intermediate and small modes, despite sometimes hurting individual mode performance. This finding connects SVDONets to multi-task learning, where parameter sharing enables the learning of generalizable parameter configurations that benefit multiple modes.

(b) Update-based coupling: Through our coupling strength analysis, we discovered that parameter updates create complex interactions between modes. For gradient descent, we observed detrimental coupling between well-approximated modes, while Adam exhibited beneficial coupling among well-approximated modes during training. Wider networks showed reduced coupling, suggesting that mode interactions diminish with increased network capacity.

7.2 FUTURE WORK

While we observed the trunk error to be very small compared to the total approximation error for our example problems, a key question is under what circumstances this observation holds. We expect the trunk error to increase for e.g., Burgers' equation with smaller viscosities, or problems with discontinuous initial conditions. Similarly, for these problems, investigating the resolution dependence might also reveal larger trunk errors than in our simplified setup.
Especially through the lens of spectral bias, the SVDONet is a very useful tool. One the one hand, if a high-frequency residual is observed in the prediction of a standard DeepONet it is not clear whether this residual comes from (a) the fact that the trunk network does not contain high-frequent basis functions or (b) that the coefficients, learned by the branch network, are poorly approximated for the high-frequent basis functions. For the SVDONet this is clear. On the other hand, the SVDONet's fixed trunk matrix allows us to study the prevalence of spectral bias in the branch network. To get a more robust understanding of the impact of spectral bias in future work, more example problems with various spectral properties and better frequency estimation methods have to be employed.
Furthermore, to better understand the limits of generalizability of different modes, it is important to investigate how much of the small modes represents genuine solution structure versus numerical noise introduced during data generation.
The SVDONet framework establishes a valuable connection between operator learning (OL) and multi-task learning (MTL), enabling knowledge transfer in both directions. From the MTL perspective, each mode coefficient in the SVDONet represents a distinct learning task, making concepts like hard/soft parameter sharing directly applicable, as demonstrated by our comparison of unstacked/stacked SVDONet architectures. Similarly, established methods for mitigating gradient conflicts in

MTL can be readily adapted to address mode coupling issues in OL. Our investigation also contributes insights back to the MTL research. While gradient conflicts have been extensively documented, the quantitative relationship between network width and conflict intensity remains largely unexplored. Our systematic measurements show that increasing network width consistently reduces mode coupling and gradient conflicts across all example problems. This width-dependent decoupling likely extends beyond SVDONets to other neural network architectures, highlighting that network width may be a fundamental factor in MTL performance. Future work should investigate whether this decoupling behavior represents a general principle for wide neural networks, potentially informing architectural design across both OLg and MTL domains.
Additionally, the observation that unstacked SVDONets demonstrate better generalization for small and intermediate modes, through parameter sharing, suggests that even when the focus is on specific modes (e.g., for resonance phenomena), unstacked SVDONets may still be preferable. However, for applications where only large modes are relevant, stacked SVDONets tend to generalize better.
A natural next step, building on this work, would be to quantify the extent to which mode-internal overfitting contributes to overfitting.

In a broader sense, the present thesis demonstrates that computing the SVD of the training data matrix prior to training can inform the hyperparameter choice for a standard DeepONet. For instance, the number of singular values larger than some threshold, i.e., the matrix's rank, informs the choice of N, and inspecting the left-singular vectors of the data matrix shows the optimal trunk space, informing the choice of the trunk network's width and depth.

7.3 CONCLUSION

This thesis investigates sources of the approximation error in DeepONets by decomposing the total error into trunk and branch components. Building on the theoretical framework of Lanthaler et al. [28], our analysis across several PDE examples suggests that for the problems studied, the branch network – responsible for learning coefficients – often dominates the approximation error, while the trunk network learns adequate basis functions.
The SVDONet framework facilitates mode-specific error analysis by replacing the learnable trunk with optimal SVD-derived modes. This reveals that for our datasets, small modes exhibit poor generalization even when training losses are reduced, and different modes interact through complex coupling patterns during optimization. Our update-based coupling analysis shows that network width can reduce detrimental mode interactions.
These findings have immediate practical implications for practitioners. Network width appears crucial for reducing mode coupling, adaptive optimizers like Adam help learn beyond the first ≈ 10 modes, and parameter sharing through unstacked architectures can improve generalization.
Computing the SVD of training data may also inform hyperparameter choices including inner dimension and network architecture.
Our work establishes connections between operator learning and MTL, where each mode coefficient represents a distinct learning task.
Important limitations include our focus on relatively smooth PDE solutions – the trunk error may become more significant for problems with discontinuous initial

conditions or shock formations. Future work should investigate these scenarios, apply MTL techniques to address mode coupling, and explore the impact of spectral bias for other example problems.

The SVDONet framework and mode-decomposition analysis provide useful tools for understanding DeepONet behavior on similar problems, potentially enabling more targeted architectural improvements in operator learning applications.

BIBLIOGRAPHY

[1] K. Bhattacharya, B. Hosseini, N. B. Kovachki, and A. M. Stuart. "Model Reduction And Neural Networks For Parametric PDEs." en. In: *The SMAI Journal of computational mathematics* 7 (2021), pp. 121–157. DOI: 10.5802/smai-jcm.74. URL: https://smai-jcm.centre-mersenne.org/articles/10.5802/smai-jcm.74/.

[2] C. Bishop. "Neural Networks for Pattern Recognition." Oxford University Press, 1995.

[3] K. P. Burnham and D. R. Anderson. "Model Selection and Multimodel Inference." Springer, 2002. DOI: 10.1007/b97636.

[4] Q. Cao, S. Goswami, and G. E. Karniadakis. "Laplace neural operator for solving differential equations." en. In: *Nature Machine Intelligence* 6.6 (June 2024). Publisher: Nature Publishing Group, pp. 631–640. ISSN: 2522-5839. DOI: 10.1038/s42256-024-00844-4. URL: https://www.nature.com/articles/s42256-024-00844-4 (visited on 06/29/2024).

[5] T. Chen and H. Chen. "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems." In: *IEEE Transactions on Neural Networks* 6.4 (1995), pp. 911–917. DOI: 10.1109/72.392253.

[6] T. Cover and P. Hart. "Nearest neighbor pattern classification." In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: 10.1109/TIT.1967.1053964.

[7] G. Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of Control, Signals and Systems* (1989). URL: https://doi.org/10.1007/BF02551274.

[8] DeepMind et al. "The DeepMind JAX Ecosystem." 2020. URL: http://github.com/google-deepmind.

[9] J. Denker, W. Gardner, H. Graf, D. Henderson, R. Howard, W. Hubbard, L. D. Jackel, H. Baird, and I. Guyon. "Neural Network Recognizer for Hand-Written Zip Code Digits." In: *Advances in Neural Information Processing Systems*. Vol. 1. Morgan-Kaufmann, 1988. URL: https://proceedings.neurips.cc/paper_files/paper/1988/file/a97da629b098b75c294dffdc3e463904-Paper.pdf.

[10] J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[11] N. Elhage, T. Hume, C. Olsson, N. Schiefer, T. Henighan, S. Kravec, Z. Hatfield Dodds, R. Lasenby, D. Drain, C. Chen, R. Grosse, S. McCandlish, J. Kaplan, D. Amodei, M. Wattenberg, and C. Olah. "Toy Models of Superposition." In: *Transformer Circuits Thread* (2022). URL: https://transformer-circuits.pub/2022/toy_model/index.html.

[12] V. Fanaskov and I. Oseledets. "Spectral Neural Operators." 2024. arXiv: 2205.10573 [math.NA]. URL: https://arxiv.org/abs/2205.10573.

[13] J. Frankle and M. Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks." 2019. arXiv: 1803.03635. URL: https://arxiv.org/abs/1803.03635.

[14] P. Getreuer. "Rudin-Osher-Fatemi Total Variation Denoising using Split Bregman." In: *Image Processing On Line* 2 (2012). https://doi.org/10.5201/ipol.2012.g-tvd, pp. 74–95.

[15] I. Goodfellow, Y. Bengio, and A. Courville. "Deep Learning." http://www.deeplearningbook.org. MIT Press, 2016.

[16] S. Goswami, A. Bora, Y. Yu, and G. E. Karniadakis. "Physics-Informed Deep Neural Operator Networks." 2022. arXiv: 2207.05748. URL: https://arxiv.org/abs/2207.05748.

[17] T. Hastie, R. Tibshirani, and J. Friedman. "The Elements of Statistical Learning." Springer, 2009.

[18] D. Hendrycks and K. Gimpel. "Gaussian Error Linear Units (GELUs)." 2023. arXiv: 1606.08415. URL: https://arxiv.org/abs/1606.08415.

[19] G. Hinton. "Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude." Accessed 21.07.2025. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[20] Y. Hu, R. Xian, Q. Wu, Q. Fan, L. Yin, and H. Zhao. "Revisiting Scalarization in Multi-Task Learning: A Theoretical Perspective." 2023. arXiv: 2308.13985. URL: https://arxiv.org/abs/2308.13985.

[21] L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu, and L. Shao. "Normalization Techniques in Training DNNs: Methodology, Analysis and Application." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.8 (2023), pp. 10173–10196. DOI: 10.1109/TPAMI.2023.3250241.

[22] S. Khodakarami, V. Oommen, A. Bora, and G. E. Karniadakis. "Mitigating Spectral Bias in Neural Operators via High-Frequency Scaling for Physical Systems." 2025. arXiv: 2503.13695. URL: https://arxiv.org/abs/2503.13695.

[23] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization." 2017. arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

[24] Q. Kong, C. Zou, Y. Choi, E. M. Matzel, K. Azizzadenesheli, Z. E. Ross, A. J. Rodgers, and R. W. Clayton. "Reducing Frequency Bias of Fourier Neural Operators in 3D Seismic Wavefield Simulations Through Multi-Stage Training." 2025. arXiv: 2503.02023. URL: https://arxiv.org/abs/2503.02023.

[25] D. J. Korteweg and G. de Vries. "On the change of form of long waves advancing in a rectangular canal, and on a new type of long stationary waves." In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 39.240 (1895), pp. 422–443. DOI: 10.1080/14786449508620739. URL: https://doi.org/10.1080/14786449508620739.

[26] S. K. Kumar. "On weight initialization in deep neural networks." 2017. arXiv: 1704.08863. URL: https://arxiv.org/abs/1704.08863.

[27] I. Lagaris, A. Likas, and D. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178.

[28] S. Lanthaler, S. Mishra, and G. E. Karniadakis. "Error estimates for DeepONets: a deep learning framework in infinite dimensions." In: *Transactions of Mathematics and Its Applications* 6.1 (Mar. 2022), tnac001. ISSN: 2398-4945. DOI: 10.1093/imatrm/tnac001. eprint: https://academic.oup.com/imatrm/article-pdf/6/1/tnac001/42785544/tnac001.pdf. URL: https://doi.org/10.1093/imatrm/tnac001.

[29] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition." In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[30] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning." In: *Nature* (2015). DOI: 10.1038/nature14539.

[31] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller. In: *Neural Networks: Tricks of the Trade*. Springer, Berlin Heidelberg, 1998. URL: https://doi.org/10.1007/3-540-49430-8_2.

[32] S. Lee and Y. Shin. "On the training and generalization of deep operator networks." 2023. arXiv: 2023.01020. URL: https://arxiv.org/abs/2309.01020.

[33] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. "Fourier Neural Operator for Parametric Partial Differential Equations." arXiv:2010.08895 [cs, math]. May 2021. DOI: 10.48550/arXiv.2010.08895. URL: http://arxiv.org/abs/2010.08895 (visited on 12/23/2022).

[34] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis. "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators." In: *Nature Machine Intelligence* 3.3 (Mar. 2021), 218–229. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00302-5. URL: http://dx.doi.org/10.1038/s42256-021-00302-5.

[35] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, and G. E. Karniadakis. "A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data." In: *Computer Methods in Applied Mechanics and Engineering* 393 (Apr. 2022), p. 114778. ISSN: 0045-7825. DOI: 10.1016/j.cma.2022.114778.

[36] M. Minsky and S. A. Papert. "Perceptrons: An Introduction to Computational Geometry." MIT Press, 1969.

[37] K. Nakamura, B. Derbel, K.-J. Won, and B.-W. Hong. "Learning-Rate Annealing Methods for Deep Neural Networks." In: *Electronics* 10.16 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10162029. URL: https://www.mdpi.com/2079-9292/10/16/2029.

[38] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. "Zoom In: An Introduction to Circuits." In: *Distill* (2020). DOI: 10.23915/distill.00024.001. URL: https://distill.pub/2020/circuits/zoom-in.

[39] A. Pinkus. "Approximation theory of the MLP model in neural networks." In: *Acta Numerica* 8 (1999), 143–195. DOI: 10.1017/S0962492900002919.

[40] L. Prechelt. "Automatic early stopping using cross validation: quantifying the criteria." In: *Neural Networks* 11.4 (1998), pp. 761–767. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00010-0. URL: https://www.sciencedirect.com/science/article/pii/S0893608098000100.

[41] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. Hamprecht, Y. Bengio, and A. Courville. "On the Spectral Bias of Neural Networks." In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 5301–5310. URL: https://proceedings.mlr.press/v97/rahaman19a.html.

[42] B. Raonić, R. Molinaro, T. De Ryck, T. Rohner, F. Bartolucci, R. Alaifari, S. Mishra, and E. de Bézenac. "Convolutional Neural Operators for robust and accurate learning of PDEs." May 2023. DOI: 10.48550/arXiv.2302.01178. (Visited on 12/01/2023).

[43] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* (1958). DOI: 10.1037/h0042519.

[44] S. Ruder. "An Overview of Multi-Task Learning in Deep Neural Networks." 2017. arXiv: 1706.05098. URL: https://arxiv.org/abs/1706.05098.

[45] L. I. Rudin, S. Osher, and E. Fatemi. "Nonlinear total variation based noise removal algorithms." In: *Physica D: Nonlinear Phenomena* 60.1 (1992), pp. 259–268. ISSN: 0167-2789. DOI: https://doi.org/10.1016/0167-2789(92)90242-F. URL: https://www.sciencedirect.com/science/article/pii/016727899290242F.

[46] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning internal representations by error propagation." 1986. URL: https://api.semanticscholar.org/CorpusID:62245742.

[47] D. Spielman. "Spectral and Algebraic Graph Theory." Yale Lecture Notes, 2025. URL: http://cs-www.cs.yale.edu/homes/spielman/sagt/.

[48] T. Tao. "Topics in random matrix theory." American Mathematical Society, 2012.

[49] T. Tripura and S. Chakraborty. "Wavelet neural operator: a neural operator for parametric partial differential equations." 2022. arXiv: 2205.02191. URL: https://arxiv.org/abs/2205.02191.

[50] B. Wang, L. Liu, and W. Cai. "Multi-scale DeepOnet (Mscale-DeepOnet) for Mitigating Spectral Bias in Learning High Frequency Operators of Oscillatory Functions." 2025. arXiv: 2504.10932. URL: https://arxiv.org/abs/2504.10932.

[51] S. Wang, H. Wang, and P. Perdikaris. "Improved Architectures and Training Algorithms for Deep Operator Networks." In: *Journal of Scientific Computing* (2022). DOI: 10.1007/s10915-022-01881-0.

[52] E. Williams, A. Howard, B. Meuris, and P. Stinis. "What do physics-informed DeepONets learn? Understanding and improving training for scientific computing applications." 2024. arXiv: 2411.18459. URL: https://arxiv.org/abs/2411.18459.

[53] Z.-Q. J. Xu, Y. Zhang, and Y. Xiao. "Training behavior of deep neural network in frequency domain." 2019. arXiv: 1807.01251. URL: https://arxiv.org/abs/1807.01251.

[54] J. Yu, Y. Dai, X. Liu, J. Huang, Y. Shen, K. Zhang, R. Zhou, E. Adhikarla, W. Ye, Y. Liu, Z. Kong, K. Zhang, Y. Yin, V. Namboodiri, B. D. Davison, J. H. Moore, and Y. Chen. "Unleashing the Power of Multi-Task Learning: A Comprehensive Survey Spanning Traditional, Deep, and Pretrained Foundation Model Eras." 2024. arXiv: 2404.18961. URL: https://arxiv.org/abs/2404.18961.

[55] T. Yu, S. Kumar, A. Gupta, S. Levine, K. Hausman, and C. Finn. "Gradient Surgery for Multi-Task Learning." In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 5824–5836. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/3fe78a8acf5fda99de95303940a2420c-Paper.pdf.

[56] E. Zhang, A. Kahana, A. Kopaničáková, E. Turkel, R. Ranade, J. Pathak, and G. E. Karniadakis. "Blending Neural Operators and Relaxation Methods in PDE Numerical Solvers." 2024. arXiv: 2208.13273. URL: https://arxiv.org/abs/2208.13273.

[57] Z. Zhang, J. Shen, C. Cao, G. Dai, S. Zhou, Q. Zhang, S. Zhang, and E. Shutova. "Proactive Gradient Conflict Mitigation in Multi-Task Learning: A Sparse Training Perspective." 2024. arXiv: 2411.18615. URL: https://arxiv.org/abs/2411.18615.

[58] Z.-Q. J. X. Zhi-Qin John Xu, Y. Z. Yaoyu Zhang, T. L. Tao Luo, Y. X. Yanyang Xiao, and Z. M. Zheng Ma. "Frequency Principle: Fourier Analysis Sheds Light on Deep Neural Networks." In: *Communications in Computational Physics* 28.5 (Jan. 2020), 1746–1767. ISSN: 1815-2406. DOI: 10.4208/cicp.oa-2020-0085. URL: http://dx.doi.org/10.4208/cicp.OA-2020-0085.

[59] D. Zwicker. "py-pde: A Python package for solving partial differential equations." In: *Journal of Open Source Software* 5.48 (2020), p. 2158. DOI: 10.21105/joss.02158. URL: https://doi.org/10.21105/joss.02158.

A

APPENDIX

A.1 HYPERPARAMETERS

The hyperparameters used in this thesis are similar to those commonly employed in related works on DeepONets [34, 35, 52]. They were selected to balance accuracy with computational cost. To maintain generality and avoid overfitting to the specific benchmark problems, we restricted the hyperparameter search to a coarse set of candidate values rather than performing an extensive fine-tuning. Moreover, since the objective of this work is not to demonstrate that one method outperforms another, but rather to analyze the behavior of DeepONets themselves, identifying the precise optimal hyperparameters is of secondary importance. Furthermore, the initial learning rate $\alpha_1 = 10^{-4}$, used throughout Chapter 6, is chosen such that the first order Taylor expansion of the loss change is a good approximation of the true loss change $\Delta\mathcal{L}$; see Section 6.2.2.1.

A.1.1 *Specific Hyperparameter Configurations for Reported Results*

For every optimizer and every initial learning rate α_1, the learning rate after t epochs, α_t, is given as

$$\alpha_t = 0.95^{\lfloor t/500 \rfloor} \cdot \alpha_1.$$

I.e., every 500 epochs the learning rate is reduced by 5%. This is a very common learning rate schedule in machine learning [37].
Furthermore, for Adam we always use the following parameters [8]:

$$\begin{aligned} \beta_1 &= 0.9, \\ \beta_2 &= 0.999, \\ \epsilon &= 10^{-8}, \\ \bar{\epsilon} &= 0. \end{aligned}$$

Table 2 contains the hyperparameters and the example problems used for all DeepONets and SVDONets whose results are shown in figures in this thesis. Note that Fig. 1 and 11 show DeepONets, the other figures show SVDONets. For DeepONets the trunk and branch network are chosen to both be MLPs of width w and depth D.
In Table 2, *var1* refers to the fact that the initial learning rate is scaled with the exponent e, as described in Section 6.1.2. In this case $\alpha_1 = 10^{-4}\sigma_1^{-2e}$, where σ_1 denotes the first singular value of the training data matrix.
The terms *var2*, *var3* are both caused by the fact that Fig. 15 shows multiple example problems. For the advection-diffusion (AD) equation with $\tau = 0.5$, we use $N = 20$ and $w = 332$. For the KdV equation with $\tau = 0.2$ and $\tau = 0.6$, we use $N = 50$ and $w = 335$. For Burgers' equation with $\tau = 0.1$, we use $N = 20$ and $w = 337$.
The term *var4* refers to the fact that Fig. 17 shows a stacked SVDONet with $w_{\mathtt{sta}} = 24$ and an unstacked SVDONet with $w_{\mathtt{unst}} = 335$.

Table 2: **Hyperparameters of DeepONets and SVDONets used in each figure.** A cell is left empty, if there are multiple DeepONets or SVDONets shown in the figure and it is apparent which hyperparameter values correspond to which DeepONet or SVDONet. The term *varx* denotes a varying value for the different DeepONets or SVDONets which is not apparent from the figure. Each *varx* is explained in the text of this section.

Figure	Example Prob.	Optimizer	α_1	Depth	Width	N	Number of Epochs
1	KdV, $\tau = 0.2$	Adam	2×10^{-3}	5	200	50	10000
11	KdV, $\tau = 0.2$	Adam	2×10^{-3}	5	200		10000
13	KdV, $\tau = 0.2$	GD	*var1*	5	335	50	4000
14	KdV, $\tau = 0.2$	Adam	10^{-4}	5	335	50	4000
15			10^{-4}	5	*var2*	*var3*	4000
17	KdV, $\tau = 0.2$		10^{-4}	5	*var4*	50	4000
18	KdV, $\tau = 0.2$	GD	10^{-4}	5	50	50	4000
19	KdV, $\tau = 0.2$	GD	10^{-4}	5	50	50	4000
20	KdV, $\tau = 0.2$	GD	10^{-4}	5		50	4000
21	KdV, $\tau = 0.2$	Adam	10^{-4}	5		50	4000
22	KdV, $\tau = 0.2$	Adam	10^{-4}	5	335	50	4000
23	KdV, $\tau = 0.2$	Adam	2×10^{-3}			50	10000
24a	AD, $\tau = 0.5$	Adam		5	200	20	10000
24b	KdV, $\tau = 0.2$	Adam		5	200	50	10000
24c	Burgers, $\tau = 0.1$	Adam		5	200	50	10000
25	Burgers, $\tau = 0.1$		10^{-4}	5		50	

A.2 CHARACTERIZING THE SPECTRUM OF MULTIVARIATE FUNCTIONS

A.3 TOTAL VARIATION NORM

The total variation (TV) norm of a function $g : D \subset \mathbb{R}^d \to \mathbb{R}$ is defined as

$$\|g\|_{TV} = \int_D \|\nabla g\|_2 \, dx.$$

The TV norm is commonly used in image denoising to penalize oscillations in the denoised image [14, 45]. In practice, the function g is only available through samples $\{x_i, y_i\}_{i=1}^m$ with $x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$ and $y_i = g(x_i)$. We thus use a k-nearest neighbors algorithm to estimate the TV norm [6].

$$\|g\|_{TV} \approx \frac{1}{mk} \sum_{i=1}^{m} \sum_{j \in N_k(i)} \frac{|y_i - y_j|}{\|x_i - x_j\|_2}$$

$N_k(i)$ denotes the set of the k nearest neighbors of x_i. We use the TV norm as a proxy for the frequency f:

$$f(g) = \|g\|_{TV}.$$

A.4 LAPLACIAN ENERGY

To define the Laplacian energy, we first introduce some concepts from spectral graph theory [47]. Consider a weighted graph $G = (V, E, w)$ where V are the vertices, E are the edges and $w : E \to \mathbb{R}_{>0}$ are the weights. Let v be the number of vertices, then we define the Laplacian matrix $L \in \mathbb{R}^{v \times v}$, as

$$L_{ij} = \begin{cases} -w((i,j)) \text{ if } (i,j) \in E, \\ \sum_{j \in N(i)} w((i,j)) \text{ if } i = j, \\ 0 \text{ else.} \end{cases}$$

Here $N(i)$ is the set of vertices adjacent to vertex i. It can then be shown that for undirected graphs

$$y^T L y = \sum_{(i,j) \in E} w((i,j))(y_i - y_j)^2 \geqslant 0.$$

Furthermore, since $L\mathbb{1} = 0$, with $\mathbb{1} = (1 \ \dots \ 1)^T \in \mathbb{R}^v$, the smallest eigenvalue of L is $\mu_1 = 0$. The Laplacian matrix's eigenpairs (μ_i, u_i) with small indices i are generally associated with low frequencies, while the eigenpairs with large indices i are associated with high frequencies. This correspondence can be understood intuitively in the context of graph clustering.
Recall that we seek to characterize the spectrum of a function $g : D \subset \mathbb{R}^d \to \mathbb{R}$ which is only available through samples $\{x_i, y_i\}_{i=1}^m$ with $x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$ and $y_i = g(x_i)$. To apply the Laplacian matrix, we construct a graph, where every sample x_i corresponds to a vertex; thus $v = m$. The edges are defined via a k-nearest neighbors algorithm. The weights $w((i,j))$ are computed via a Gaussian kernel based on the Euclidean distance between the inputs x_i and x_j

$$w((i,j)) = \exp\left(-\frac{k}{2} \frac{\|x_i - x_j\|_2^2}{\sum_{l \in N(i)} \|x_i - x_l\|_2^2}\right).$$

This defines a Laplacian matrix L_0. Since the hereby constructed graph is directed, we symmetrize the matrix $L = \frac{1}{2}(L_0 + L_0^T)$. This then corresponds to the Laplacian of an undirected graph.
Furthermore, consider the eigenvectors u_i of the Laplacian matrix L, which can be chosen to be orthonormal. Any vector $w \in \mathbb{R}^m$ can be decomposed as

$$w = \sum_{i=1}^{m} a_i u_i,$$

with $a_i \in \mathbb{R}$. Then the Rayleigh quotient for L and a non-zero vector w is given as

$$\frac{w^T L w}{w^T w} = \frac{\sum_{i=1}^{m} a_i^2 \mu_i}{\sum_{i=1}^{m} a_i^2}.$$

Since the eigenvalues μ_i correspond to frequencies on the graph, the Rayleigh quotient thus computes the mean frequency of w, as a signal on the graph. Thus, the Rayleigh quotient of the vector $y = (y_1 \ \dots \ y_m)^T \in \mathbb{R}^M$ containing the values of the function g can be seen as the mean frequency of g:

$$f(g) = \frac{y^T L y}{y^T y}.$$

A.5 FOURIER TRANSFORM OF DATA PROJECTED ONTO LOW-DIMENSIONAL SUBSPACES

In [58] the spectral bias for neural networks whose input are images is studied. One of the methods used to characterize the spectrum of the target function $g : D \subset \mathbb{R}^d \to \mathbb{R}$ is the following.
We cannot directly compute the Fourier transform of g due to its multi-dimensional input x. Thus, we seek to move to a one-dimensional setting. In the method used in [58] this is done by considering the projection of the input x onto multiple vectors $u_j \in \mathbb{R}^d$, i.e., $a_{ji} = u_j^T x \in \mathbb{R}$. Then, for each j, we consider the pseudo function $h_j : \mathbb{R} \to \mathbb{R}$ such that $h_j(a_{ij}) = y_i$. We call this a pseudo function, since there might not be a well-defined mapping such that $h_j(a_{ij}) = y_i$, e.g., if $a_{ij} = a_{kj}$ and $y_i \neq y_k$. However, we can still consider the signal $\{a_{ij}, y_i\}_{i=1}^M$ and compute its discrete Fourier transformation. Note that we use a non-uniform discrete Fourier transform, since the a_{ij} are irregularly distributed.
Given a set of Z vectors $\{v_j\}_{j=1}^Z$, the samples $\{x_i, y_i\}_{i=1}^m$, containing the function g, are thus transformed into a set of Fourier transforms $\{\mathcal{F}(h_j)(F)\}_{j=1}^Z$ at the frequencies $F = (f_1, \dots, f_{\lfloor m/2 \rfloor})$. We then take the average of the Z Fourier transforms

$$q(f) = \frac{1}{Z} \sum_{j=1}^{Z} \mathcal{F}(h_j)(f)$$

and compute the mean frequency of the averaged Fourier transform:

$$f(g) = \frac{\sum_f |q(f)|^2 f}{\sum_f |q(f)|^2}.$$

We consider the first Z left-singular vectors of the input data matrix

$$X = [x_1 \ \dots \ x_m] \in \mathbb{R}^{d \times m}.$$

Note that since the input x_j for the DeepONet is the input function p_j, which is given as a sum of L trigonometric functions (see Section 3.2), we choose $L = Z$. Thus the input data X can be fully reconstructed using $L = Z$ vectors, i.e., $X = [v_1 \ \dots \ v_Z][v_1 \ \dots \ v_Z]^T X$. However, this method nonetheless computes the spectrum of g via the spectra of pseudo functions h_j. Consequently, their spectra generally do not determine the full multivariate spectrum of g, and some information is inevitably lost. This can also be seen through the following fact. Consider two different sets of vectors $\{v_j\}_{j=1}^Z$ and $\{w_j\}_{j=1}^Z$. Let both of them fully reconstruct X, i.e., $X = [v_1 \ \dots \ v_Z][v_1 \ \dots \ v_Z]^T X = [w_1 \ \dots \ w_Z][w_1 \ \dots \ w_Z]^T X$. Then the spectrum of g estimated through the projection onto the different sets of vectors is in general not the same.

A.6 COMPARISON OF SVDONET AND POD-DEEPONET

As discussed, for any full-rank matrix $C \in \mathbb{R}^{N\times N}$, the matrix $T_* = \Phi_1 C$ is an optimal trunk matrix. The SVDONet, as defined in this work, uses $C = \Sigma_1$. The POD-DeepONet [35] uses $C = I \in \mathbb{R}^{N\times N}$. For further comparison we also consider $C = \Sigma_1/\sigma_1$, i.e., the first mode is multiplied with 1, but the singular values retain their different influence on the different modes. Additionally, we consider $C = \sigma_1 I \in \mathbb{R}^{N\times N}$. Lastly, we also consider standard DeepONets, i.e., a learned trunk matrix.

Figures 23 - 24c show the relative errors for these five architectures when applied to various example problems with different learning rates, widths, and depths. As can be seen, architectures with a fixed trunk matrix typically converge much faster. Additionally, the architectures with $C = I$ and $C = \Sigma_1/\sigma_1$ perform best for the advection-diffusion equation and Burgers' equation, while the architectures with $C = \Sigma_1$ and $C = \sigma_1 I$ perform best for the KdV equation. These results suggest that the main difference in performance between SVDONet and POD-DeepONet stems from how the largest mode is scaled. Additionally, it indicates that none of the four candidates considered for C is optimal for all cases.

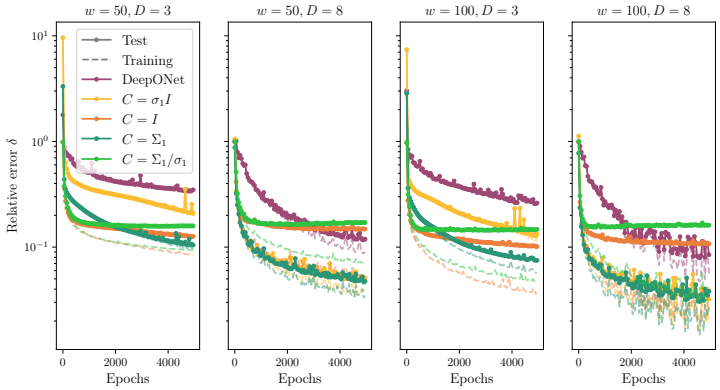
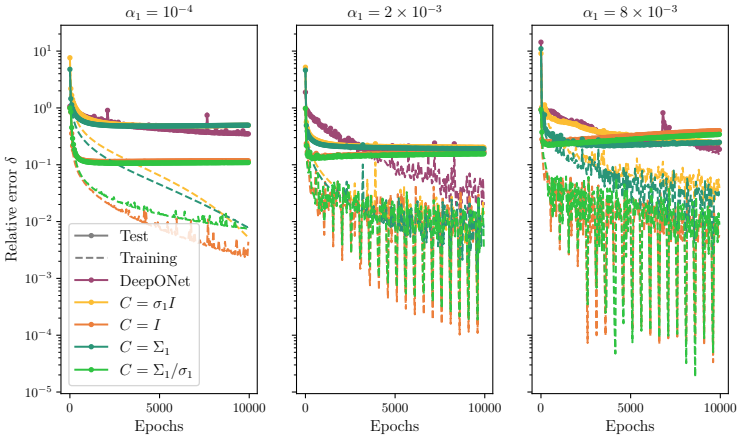
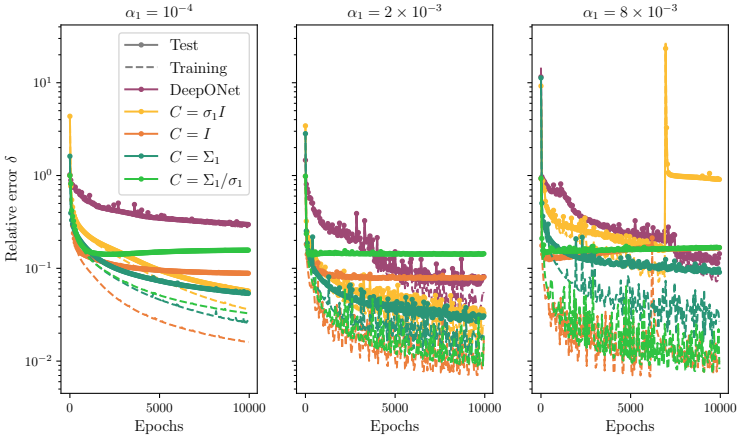


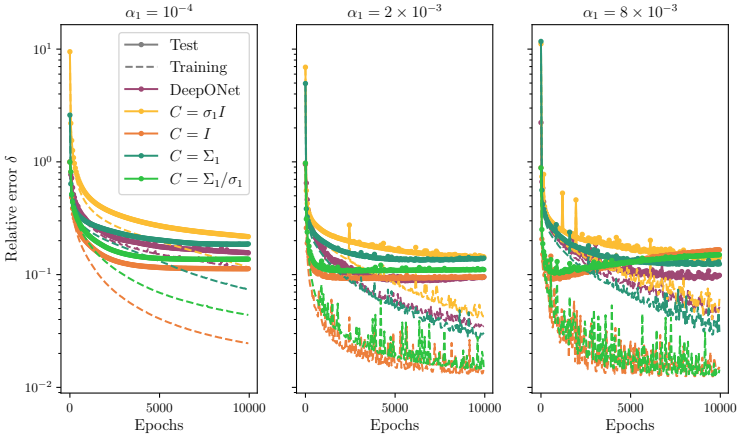
Figure 23: **Comparison of various OL architectures and sizes. KdV Equation with $\tau = 0.2$.** Every column corresponds to a network size $((w = 50, D = 3), (w = 50, D = 8), (w = 100, D = 3), (w = 100, D = 8))$. The DeepONet (purple), 'rescaled' POD-DeepONet ($C = \sigma_1 I$, yellow), POD-DeepONet ($C = I$, orange), SVDONet ($C = \Sigma_1$, darkgreen), 'rescaled' SVDONet ($C = \Sigma_1/\sigma_1$, lightgreen) are shown.



(a) Advection-Diffusion Equation with $\tau = 0.5$



(b) KdV Equation with $\tau = 0.2$.



(c) Burgers' Equation with $\tau = 0.1$.

Figure 24: **Comparison of various OL architectures.** Every column corresponds to a different initial learning rate ($\alpha_0 = 10^{-4}, 2 \times 10^{-3}, 8 \times 10^{-3}$). The DeepONet (purple), 'rescaled' POD-DeepONet ($C = \sigma_1 I$, yellow), POD-DeepONet ($C = I$, orange), SVDONet ($C = \Sigma_1$, darkgreen), 'rescaled' SVDONet ($C = \Sigma_1/\sigma_1$, lightgreen) are shown.

A.7 SVDONET'S TRUNK ERROR FOR TEST DATA

Table 3: Relative training and test trunk errors δ_T for all example problems and various N.

	$\log_{10}\delta_{T,tr}$	$\log_{10}\delta_{T,te}$	$\log_{10}\delta_{T,tr}$	$\log_{10}\delta_{T,te}$	$\log_{10}\delta_{T,tr}$	$\log_{10}\delta_{T,te}$
	$N = 10$	$N = 10$	$N = 15$	$N = 15$	$N = 20$	$N = 20$
AD, $\tau = 0.5$	-0.41	-0.41	-0.78	-0.78	-12.6	-12.6
AD, $\tau = 1.0$	-0.69	-0.69	-1.32	-1.32	-13.1	-13.1
	$N = 30$	$N = 30$	$N = 50$	$N = 50$	$N = 70$	$N = 70$
KdV, $\tau = 0.2$	-1.48	-1.46	-2.19	-2.10	-2.94	-2.84
KdV, $\tau = 0.6$	-1.04	-1.03	-1.90	-1.88	-2.71	-2.59
KdV, $\tau = 1.0$	-1.00	-0.99	-1.91	-1.93	-2.70	-2.70
Burgers, $\tau = 0.1$	-1.36	-1.29	-1.90	-1.75	-2.42	-2.20
Burgers, $\tau = 1.0$	-3.56	-3.35	-5.53	-5.10	-7.54	-6.75

A.8 COMPARISON OF STACKED AND UNSTACKED SVDONETS

A.8.1 *Number of Parameters in an MLP*

As described in Section 2.3, θ denotes the parameters of a neural network. Consider a one-layer perceptron with w neurons in the hidden layer, M-dimensional input and one output neuron. As discussed in Section 2.3.1, this has parameters

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

where $\theta^{\mathrm{out}} \in \mathbb{R}^{1\times w}$ are the outer weights and $\theta^{(1)} = \left(\vartheta^{(W,1)}, \vartheta^{(B,1)}\right)$ are the hidden layer's weights. Note that $\vartheta^{(W,1)} \in \mathbb{R}^{w\times M}$ and $\vartheta^{(B,1)} \in \mathbb{R}^{w}$. Thus, vectorizing the matrix $\vartheta^{(W,1)}$, and concatenating the resulting vector with $\vartheta^{(B,1)}$ and the vectorization of θ^{out}, yields a vector Θ of length $Mw + 2w$:

$$\Theta = \left(\underbrace{\vartheta_{1,1}^{(W,1)} \ \dots \ \vartheta_{1,M}^{(W,1)} \ \vartheta_{2,1}^{(W,1)} \ \dots \ \vartheta_{w,M}^{(W,1)}}_{w\times M} \ \underbrace{\vartheta_{1}^{(B,1)} \ \dots \ \vartheta_{w}^{(B,1)}}_{w} \ \underbrace{\theta_{1,1}^{\mathrm{out}} \ \dots \ \theta_{1,w}^{\mathrm{out}}}_{w} \right)^{T}.$$

Thus, a one-layer perceptron with width w, M-dimensional input and one output neuron has $= Mw + 2w$ parameters. The number of parameters of a neural network is denoted as $|\theta|$.
Similarly, we now consider an MLP with M input neurons, D hidden layers with w neurons each and K output neurons. The number of parameters of such an MLP is denoted as $Q(M, D, w, K) = |\theta|$. Then,

$$Q(M, D, w, K) = \underbrace{Mw + w}_{\text{input}\to\text{first hidden layer}} + (D-1)\underbrace{(w + w^2)}_{\text{between hidden layers}} + \underbrace{wK.}_{\text{last hidden layer}\to\text{output}}$$

Note that $Q(M, 1, w, 1) = Mw + w + 0 + w = Mw + 2w$, since an MLP with $D = 1$ is a one-layer perceptron.

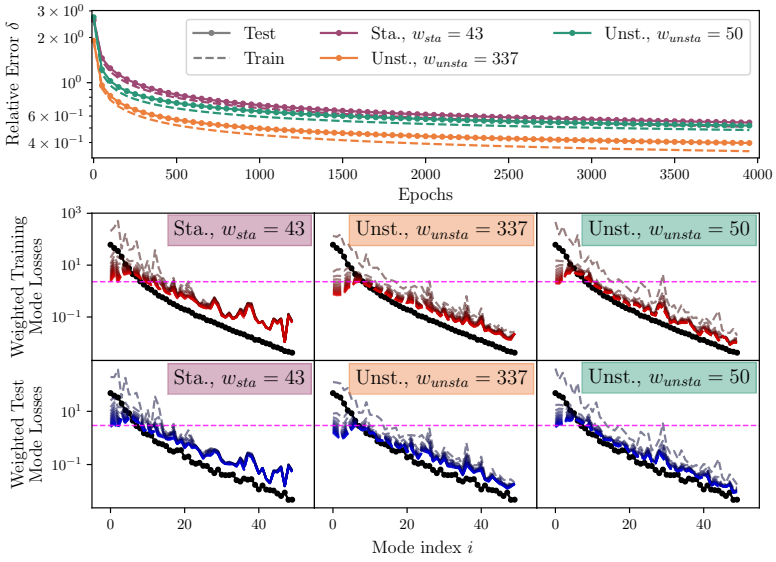
A.8.2 *Number of Parameters in Stacked and Unstacked SVDONets*

Since the branch network of the unstacked SVDONet is just an MLP with $K = N$ output neurons, the number of parameters for the unstacked SVDONet is $|\theta|_{\mathrm{unst}} = Q(M, D, w_{\mathrm{unst}}, N)$. Since the branch network of the stacked SVDONet consists of N separate MLPs with $K = 1$ output neurons each, the number of parameters for the stacked SVDONet is $|\theta|_{\mathrm{sta}} = NQ(M, D, w_{\mathrm{sta}}, 1)$.
Thus, for the KdV equation ($M = 400$), the stacked SVDONet with $w_{\mathrm{sta}} = 24$ and the unstacked SVDONet with $w_{\mathrm{unst}} = 335$ both have $\approx 6 \times 10^5$ parameters. These SVDONets are compared in Fig. 17.

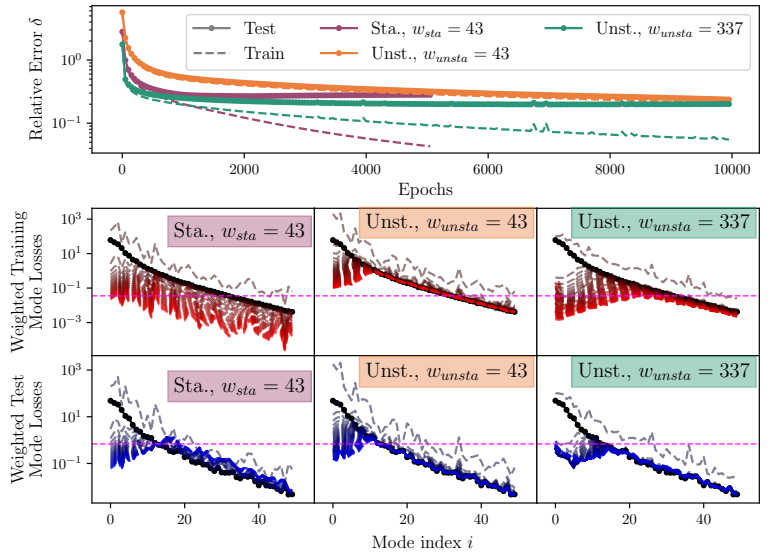
A.8.3 *Comparing Stacked and Unstacked SVDONets with Equal Widths*

In this section, we compare three SVDONets. First, we consider a stacked SVDONet whose branch subnetworks all have width $w_{\mathrm{sta}} = 43$. Thus, the considered stacked SVDONet has $|\theta| \approx 5 \times 10^5$ parameters. Second, we consider an unstacked SVDONet whose branch network has width $w_{\mathrm{unst}} = 43$. Thus, this unstacked SVDONet has $|\theta| \approx 10^4$ parameters. Third, for further comparison, we consider an

unstacked SVDONet which has $|\theta| \approx 5 \times 10^5$ parameters, matching the stacked case; this implies a branch width of $w_{\mathtt{unst}} = 337$.
The relative errors and the weighted mode losses for the three SVDONets trained using both GD and Adam are shown in Fig. 25. For GD, the unstacked SVDONet ($w = 43$) performs slightly better than the stacked SVDONet. However, their behavior is very similar. The wider unstacked SVDONet ($w = 337$) achieves a significantly lower (test and training) error than both. For Adam, the stacked and unstacked SVDONets of same width behave significantly different from each other. While the stacked SVDONet starts overfitting significantly after ≈ 1000 epochs, the unstacked SVDONet ($w = 43$) slowly, but steadily, lowers both test and training error over the 10000 epochs. The unstacked SVDONet ($w = 43$) takes ≈ 7000 epochs to reach a test loss comparable to the minimum test loss reached by the stacked SVDONet. The wider unstacked SVDONet ($w = 337$) starts overfitting after ≈ 7000 epochs and reaches a lower test error than the other two SVDONets. The stacked SVDONet achieves the lowest training error of the three.



(a) SVDONets trained with GD



(b) SVDONets trained with Adam

Figure 25: **Relative Errors and Weighted Mode Loss for stacked and unstacked SVDONets. Top panel:** Relative error $\delta = \|A - \tilde{A}\|_F / \|A\|_F$ for both training (dashed lines) and test (dot-solid lines) data over 4000 (a) and 10000 (b) epochs. **Center and bottom row:** Weighted training (center row) and test (bottom row) mode losses at different training steps, colored from gray (initial) to red/blue (final). Columns correspond to different architectures and widths. From left to right: Stacked with $w_{sta} = 43$, unstacked with $w_{uns} = 43$ and unstacked with $w_{uns} = 337$, as indicated by the labels. The center and bottom row plots also contain the respective base losses in black, and a pink dashed horizontal line marking the maximum mode loss of the wieder unstacked SVDONet. The SVDONets shown here approximate the solution operator of Burgers' equation with $\tau = 0.1$.