Mode-Decomposition in DeepONets

Generalization and Coupling Analysis

Julius Johannes Taraz

27. August 2025

Contents

Chapter 1

Introduction

Most problems 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) methods such as the (implicit) Euler method and the 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 solution operator, specifically the time-evolution operator mapping from the initial condition to the solution at a later time. The aforementioned methods are excellent at computing one evaluation of the solution operator, i.e. computing the time-evolution for one initial condition. However, these methods alone offer no efficient way of approximating the whole operator, since each evaluation of the solution operator has to be computed from the start. For many tasks such as PDE constrained optimization (such as design or optimal control) or uncertainty quantification the approximation of the entire operator would be a very significant increase in performance [1]. Next to methods such as Reduced Order Modeling (ROM), which project the dynamics onto a lower dimensional system, built from previous solutions, in recent years methods such as operator learning (OL) have gained more traction. The fundamental idea of OL is using (discretizations of) solution functions for some initial conditions as training data for a neural network, such that it then learns the dependency between the initial conditions and the solution function. A classical (or the most widely varied on) architecture for OL is the Deep Operator Network (DeepONet) introduced by Lu et al. in 2020 [2], which is based on theoretical results regarding operator approximation [3].
Despite the prospects, DeepONets still face severe practical limitations. Namely poor accuracy, or conversely a high data demand for a good accuracy. Another issue, is the resolution dependence of DeepONets. Specifically about training and testing on different meshes. In this work, we focus on problems where DeepONets have poor accuracy even when trained and tested on the same mesh and on in-distribution test data.

Several improved training methods and architectural modifications to the DeepONet have been suggested in the literature [1, 4, 5, 6, 7]. Additionally, entirely new neural architectures for OL have been proposed, such as the fourier neural operator, the convolutional neural operator and the Laplace neural operator [8, 9, 10]. Furthermore, the combination of OL and physics-informed neural networks (PINNs), i.e. neural networks solving a PDE by minimizing the residual of the PDE [11] has sparked significant interest [12, 13]. Next to this practical progress, considerable efforts have been undertaken to achieve a theoretical understanding of DeepONets and similar architectures [4, 14, 15].
However, some fundamental questions about DeepONets and their inner workings in practice still remain open. We investigate the following.

1. Which parts of the solution are not captured accurately.
2. Why are these parts neglected by the optimizer.
3. How do the predictions of different parts interact with each other.
4. How well do certain parts of the prediction generalize.

To this end, we use Lanthaler's error decomposition [14] to locate the error in the branch net, not the trunk net. We thus construct a simpler operator learning architecture, the SVDONet, which reduces the problem to the branch net. Furthermore, we decompose the branch-error into different mode errors. In the following we use this framework of mode decomposition to shine light on several phenomena relevant for a deeper understanding of DeepONets.

We see that eventually, for a large enough network, a low training error for all modes is achieved. However, we find that the SVDONet's generalization abilities hugely depend on the mode. With the dominant modes (large singularvalue in the data matrix, low physical frequency) generalizing significantly better. As a result, less dominant modes in the training data disproportionately contribute to the test error. Additionally, by introducing new weightings for the different modes we can see that the generalizability of some modes can not be increased beyond some limit, even if their training error can be significantly lowered. This hints at the fact that it is inherently more difficult to generalize well on these modes. Furthermore, we see that while gradient descent only acts on the largest modes, adaptive gradient schemes, such as Adam, manage to act on a much larger set of modes, thus achieving significantly lower test and training errors.

Despite the usefulness of mode-wise decomposition, modes do not act independently. We explore this by comparing standard (unstacked) SVDONets to stacked SVDONets, where the coefficients for each mode are predicted separately. Surprisingly, the standard architecture achieves better test performance. This suggests that despite the limitations of normal SVDONets, their generalization remains surprisingly effective compared to their stacked counterparts. The comparison of the stacked and unstacked SVDONets shows that the mode interaction, through the shared branch net, has positive effects on generalization.

However, we also study the mode coupling in a different, complementary sense: update based mode coupling. How does updating our parameters to yield a lower loss on mode i, affect the loss on mode $j \neq i$? For SVDONets trained using gradient descent, we see the update based mode couplings detrimental effect on the loss reduction for narrow networks. However, we find this to weaken significantly for wider, better performing, networks, effectively decoupling the modes. Thus, we find differences in the performance of stacked and unstacked SVDONets, beyond the effect of update based mode coupling. Lastly, we use the update based mode coupling to investigate SVDONets trained with Adam. Here we find less clearly detrimental coupling, a structured coupling and mode-internal overfitting for many, but not all overfitted modes.

To conclude, our SVDONet with the novel mode-loss-decomposition

1. reveals which modes cause the training and test errors, depending on the used optimizer,
2. locates the improved generalizability abilities of the normal SVDONet compared to the stacked SVDONet in mode space,
3. lets us investigate the automatic mode decoupling with increasing network size, and
4. uncovers the complex coupling structure in SVDONets trained with Adam.

Chapter 2

Background

This chapter provides necessary background for the thesis, starting with frequently used notation, a short but formal problem statement, an introduction to neural networks, the DeepONet and lastly the framework for testing and training used in this thesis.

2.1 Notation

Table 2.1 contains commonly used symbols in this thesis as a guide to the reader.

2.2 Problem Statement

The general setting of operator learning considers any operator mapping from function space to function space. However, 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 $\tau > 0$. Specifically, for an initial condition $p \in \mathcal{C}(D)$, the operator yields $G_*(p) = u_p^*(\cdot, \tau)$, where $u_p^*(\cdot, t)$ denotes the solution of the PDE with initial condition $u_p^*(\cdot, 0) = p(\cdot)$ for the spatial domain D. For an example, see Section 3.1. To make the problem computationally accessible, we 'encode' p in a finite dimensional vectorspace $\hat{p} \in \mathbb{R}^M$. In this work we sample p on a uniform mesh, see Eq. 2.16 for more detail. We then build an operator $G_\theta : \hat{p} \mapsto u_p$ with some parameters θ, to approximate G_*. Section 2.4 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 2.5 explains how neural networks can be used for operator learning, by introducing the DeepONet.

Table 2.1: Notation frequently used in this thesis

Symbol	**Name**	**Not**
D	Spatial domain of interest	$D \subset$
$\mathcal{C}(D)$	set of continuous functions with domain D	
p	Input function (e.g. initial condition of time-dependent PDE)	$p \in \mathcal{C}($
$\hat{p}$	Discretized input function	e.g. Eq.
G_*	True solution operator	
u_p^*	Solution $G_*(p)$ for input function p	$u_p^* \in \mathcal{C}($
G_θ	Model / approximation of G_*	
r	Coordinate at which u_p^* is evaluated	$r \in$
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	
N	Dimension of DeepONet / number of output neurons of branch and trunk net	
A	Target data matrix	$\in \mathbb{R}^{n\times}$
$\tilde{A}$	Approximation of A by the DeepONet	$\in \mathbb{R}^{n\times}$
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	
$[n]$	Set of indices	$[n] = \{1, 2$
$Z^\dagger$	Moore-Penrose inverse of matrix Z	

2.3 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, Section 2.4, and as an introduction to the UAT for operators, which is the theoretical foundation for DeepONets, Section 2.5.
One of the foundational results for function approximation using neural networks is the UAT, published in 1989 by Cybenko [16]. Here we use a more general theorem by Pinkus [17]. The theorem states that for any non-polynomial continuous function σ, any function $f \in \mathcal{C}(\mathbb{R}^d)$ and any $\varepsilon > 0$, there exist $K \in \mathbb{N}$, $a_k, c_k \in \mathbb{R}$ and $b_k \in \mathbb{R}^d$ for $k \in [K]$, such that

$$\left| f(r) - \sum_{k=1}^{K} a_k \sigma(b_k^T r + c_k) \right| < \varepsilon \tag{2.1}$$

for all $r \in [0,1]^d$. This means that any continuous function f can be approximated arbitrarily well by a linear combination of K non-polynomial functions σ

with shifted and scaled input. Note that K is not bounded, meaning this might be a completely impractical construction to approximate some f.

2.4 Neural Networks

This section is a self-contained explanation of (1) neural networks, based on the UAT for functions, (2) different optimization algorithms to find parameters for neural networks and (3) overfitting.
Neural networks can be seen as solutions of an optimization problem of the following type. Given some inputs $\{r_1, ..., r_m\}$ and corresponding target outputs $\{y_1, ..., y_m\}$ generated by a function f, i.e. $y_i = f(r_i)$, and a model G_θ with parameters θ, we want to find parameters such that G_θ approximates f well. Or, more formally, training the neural network is equivalent to finding the optimal parameters

$$\theta_{opt} := \arg\min_\theta \sum_{i=1}^{m} |y_i - G_\theta(r_i)|^2. \tag{2.2}$$

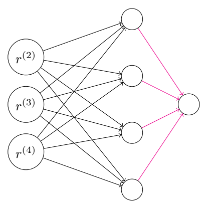
In neural network literature, the mean-squared error $\mathcal{L}_{tr}(\theta) = \frac{1}{m}\sum_{i=1}^{m} |y_i - G_\theta(r_i)|^2$ is often termed training loss function. We now discuss the model G_θ.

2.4.1 What is a Neural Network

Since the UAT shows that any continuous function f can be approximated arbitrarily well by a linear combination of K non-polynomial functions σ with shifted and scaled input, we use the construction

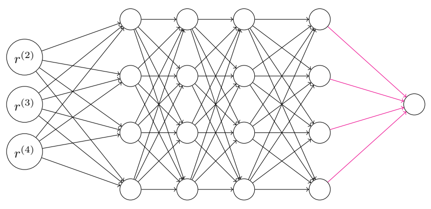
$$G_\theta(r) = \sum_{k=1}^{K} \theta_k^{(1)} \sigma\left(\theta_k^{(2)^T} r + \theta_k^{(3)}\right) =: \theta^{(1)} H_{\left(\theta^{(2)},\theta^{(3)}\right)}(r) \tag{2.3}$$

as a model. This model and its parameters can be split into the 'outer weights' $\theta^{(1)} = (\theta_1^{(1)}, \theta_2^{(1)}, ..., \theta_K^{(1)}) \in \mathbb{R}^{1\times K}$ and the hidden layer H parametrized by the 'inner weights' $\theta^{(2)}$ and the biases $\theta^{(3)}$. The model is called a perceptron with one hidden layer and a linear output layer, visualized in Figure 2.1.
In practice, the hidden layers are often composed, yielding a multi-layer perceptron: The input r is processed in the first layer, yielding $z_1 = H_{\theta_A}(r)$ with some parameters θ_A. The first layers output is processed in the second layer, as $z_2 = H_{\theta_B}(z_1)$, etc. The output of the last layer $z_L = H_{\theta_Z}(z_{L-1})$ is then processed by a linear layer, such that $G_\theta(r) = \theta^{(1)} z_L$. The parameters of each hidden layer and the linear layer together make up the parameters of the multi-layer perceptron. It is visualized in Figure 2.2. This composition is done because it, in practice, allows for faster training and reduces the number of necessary parameters for a given accuracy [18]. Modern neural networks often modify the classical multi-layer perceptron architecture by restricting the parameter space to a structured subspace. For example, instead of optimizing



$$G_\theta(r) = \sum_{k=1}^{K} \theta_k^{(1)} \sigma \left({\theta_k^{(2)}}^T r + \theta_k^{(3)} \right)$$

Figure 2.1: One-layer perceptron with an input $r \in \mathbb{R}^3$ and $K = 4$ neurons. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the nonlinear activation function in the receiving neuron.



$$G_\theta(r) = \theta^{(1)} H_{\theta_Z}(H_{\theta_C}(H_{\theta_B}(H_{\theta_A}(r))))$$

Figure 2.2: Multi-layer perceptron with an input $r \in \mathbb{R}^3$ with 4 hidden layers and 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 nonlinear activation function in the receiving neuron.

general weight vectors $\theta_k^{(2)} \in \mathbb{R}^d$, one may constrain $\theta_k^{(2)} \in V \subset \mathbb{R}^d$ to enforce architectural properties such as convolutional structure, weight sharing, or locality.
A remaining question is, how do we find parameters θ that yield a low approximation error, or loss $\mathcal{L}(\theta)$?

2.4.2 Optimization schemes

One standard approach is gradient descent (GD), i.e., consider some initial parameters θ_0, and then update them 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 in the first epoch. This procedure is repeated until a satisfactory loss is reached. However, this is often a slow process, or a satisfactory loss is never reached.
One very popular variation 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 time $\nabla \mathcal{L}_{tr}(\theta_t)$, but it includes the update of the last step

$$\delta\theta_t = -\alpha_t \nabla \mathcal{L}_{tr}(\theta_t) + \beta \ \delta\theta_{t-1}. \tag{2.4}$$

The idea behind this is to average out high-frequency oscillations in the gradient [19]. Another popular variation are 'adaptive gradients'. In the adaptive gradients method AdaGrad [20], the parameter update is given as

$$\delta\theta_t = -\alpha_t \frac{\nabla \mathcal{L}_{tr}(\theta_t)}{\sqrt{v_t} + \varepsilon} \text{ with } v_t = \sum_{i=1}^{t} (\nabla \mathcal{L}_{tr}(\theta_i))^2, \tag{2.5}$$

with the square and division applied component-wise. This emphasizes parameters with persistently small gradients, allowing AdaGrad to capture rare but informative patterns in the data. In contrast, RMSprop [21] uses an exponentially weighted average for the normalization term, i.e. $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla \mathcal{L}_{tr}(\theta_i))^2$, which introduces a decay mechanism to gradually discount earlier gradient contributions. The adaptivity of RMSprop and the momentum are combined in a method called 'Adam' [22]. Here the update is computed as follows.

$$\begin{aligned}
m_t &= \beta m_{t-1} + (1 - \beta_1) \nabla \mathcal{L}_{tr}(\theta_t), && (2.6)\\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2)(\nabla \mathcal{L}_{tr}(\theta_t))^2, && (2.7)\\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, && (2.8)\\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t}, && (2.9)\\
\delta\theta_t &= -\alpha_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \bar{\varepsilon}} + \varepsilon}. && (2.10)
\end{aligned}$$

The steps in Equations 2.8 and 2.9 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 schedule, e.g. exponential decay $\alpha_t = 0.95^{t/500}\alpha_1$. Unless noted otherwise, Adam is used in this work.

2.4.3 Overfitting

Sofar, 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 outside of the training data set. Those new inputs, and their reference solutions, are termed test data. A regression model is overfitting, if it can approximate it's 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

$$\mathcal{L}_{tr} = \frac{1}{m_{tr}} ||Y_{tr} - G(R_{tr})||_2^2, \tag{2.11}$$

$$\mathcal{L}_{te} = \frac{1}{m_{te}} ||Y_{te} - G(R_{te})||_2^2. \tag{2.12}$$

We now show results of the DeepONet, introduced in the following section, to visualize overfitting. Fig. 2.3 Left) shows the DeepONet approximations (dot-solid) and the true solutions (dashed) of the Burgers equation for three input functions from the training data set (black, red, blue) and three input functions from the test data set (green, yellow, orange). The approximations of the test data show significant oscillations, compared to both the true solutions and the training data oscillations. In Fig. 2.3 Right) the training and test loss $\mathcal{L}_{tr}$ and $\mathcal{L}_{te}$ are shown over epochs for three networks trained with data sets of different sizes ($m_{train} = 250$, $m_{train} = 900$ and $m_{train} = 3000$). We see a significantly smaller difference between $\mathcal{L}_{te}$ and $\mathcal{L}_{tr}$ for the larger data set. Furthermore, we see that the test loss of the network trained on the small dataset reaches a minimum after ≈ 1000 epochs and then increases.
Besides using a larger dataset, methods to prevent overfitting are known as regularization. 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 [23]. As seen in Fig. 2.3 Right), 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 after 20000 epochs is not the best network. The parameters at the minimal test loss would be better. This is the motivation behind early stopping [24]. 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 used to compute the loss, and the validation loss is used as a proxy

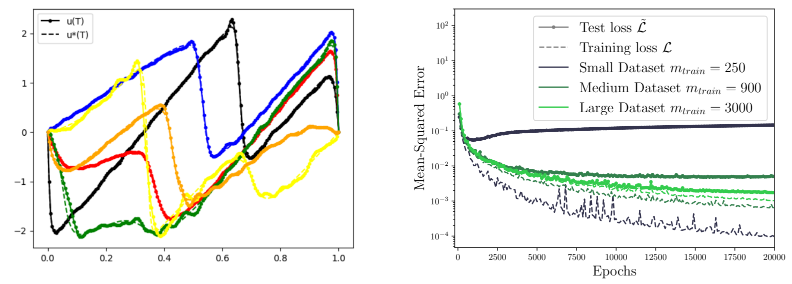


Figure 2.3: Left) Approximations (dot-solid) and true solutions (dashed) of Burgers equation for $\tau = 0.1$ for both training (black, red, blue) and test (green, yellow, orange) data. Right) Test and training loss over epochs for datasets of different size.

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.

2.5 DeepONet

As the UAT for functions is the theoretical foundation for function approximation with neural networks, the UAT for operators is the cornerstone of operator learning. It was published by Chen and Chen in 1995 [3]. Consider a continuous non-polynomial function σ, a Banach space V, a compact subset $K_1 \subset V$, a compact set $X \subset \mathcal{C}(K_1)$, a compact set $K_2 \subset \mathbb{R}^d$ and a continuous operator $G_* : X \to \mathcal{C}(K_2)$. Then for any $\varepsilon > 0$, there exist constants $I, J, K \in \mathbb{N}$, $c_k, a_i^k, \beta_{ij}^k, \gamma_i^k \in \mathbb{R}$, $x_j \in K_1$, and $b_k \in \mathbb{R}^d$ such that

$$\left| G_*(p)(r) - \sum_{k=1}^{K} \sigma(b_k^T r + c_k) \sum_{i=1}^{I} a_i^k \sigma\left(\sum_{j=1}^{J} \beta_{ij}^k p(x_j) + \gamma_i^k \right) \right| < \varepsilon \tag{2.13}$$

for all $p \in X$ and $r \in K_2$.

One could argue that the 2019 proposed Deep-Operator-Network (DeepONet), the first practically used architecture to learn operators [2], is the direct implementation and generalization of this double sum of activation functions. The DeepONet consists of two subnetworks, which are both fully connected neural networks, the so-called trunk and branch networks. The trunk network takes the coordinate r at which the operator output is evaluated as input, while the branch network takes a discretization $\hat{p}$ of the input function p as input. They both have the same number of output neurons N and the final output is given

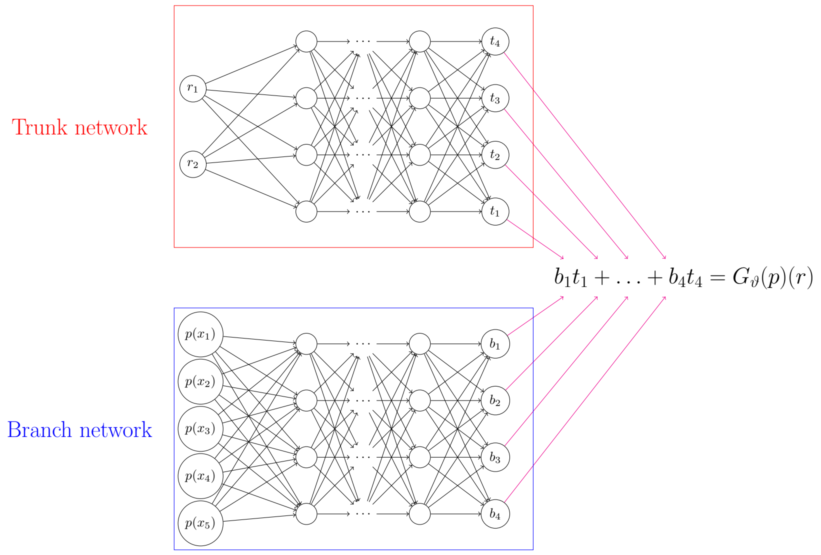


Figure 2.4: DeepONet. In this example the evaluation is done at position $(r_1, r_2) \in \mathbb{R}^2$ and the parameter function p is sampled at 5 points.

as

$$G_\theta(\hat{p})(r) = \sum_{j=1}^{N} b_j(\hat{p})t_j(r) = (t_1(r)\ \ldots\ t_N(r)) \begin{pmatrix} b_1(\hat{p}) \\ \vdots \\ b_N(\hat{p}), \end{pmatrix} \tag{2.14}$$

where b_j and t_j are the j-th output neuron of the respective subnetwork and θ is the set of weights of the DeepONet. When evaluating the DeepONet for a discretized input function $\hat{p}$ and n coordinates $R = (r_1, ..., r_n)^T$, the output is

$$G_\theta(\hat{p})(R) = \sum_{j=1}^{N} b_j(\hat{p})t_j(R), \tag{2.15}$$

where t_j is evaluated component-wise. Thus, all possible DeepONet outputs lay in a subspace spanned by the trunk neurons $t_j(R)$, hence they are also called basis functions, or when evaluated on coordinates basis vectors, and the number of output neurons N is 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 $\hat{p}$.
The general architecture can be seen in Figure 2.4. The double sum in the UAT for operators, Equation 2.13, is hence just a DeepONet, were the trunk network has one layer and the branch network has two layers, where only the hidden one has the activation function. This architecture can be seen in Figure 2.5.

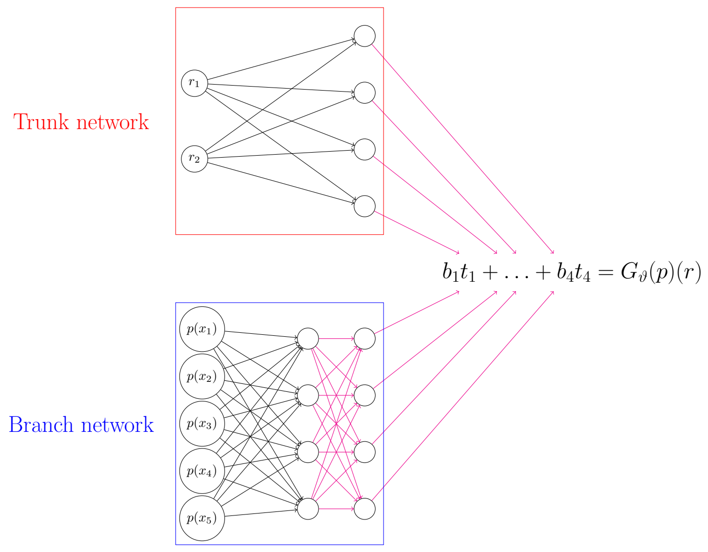


Figure 2.5: DeepONet for UAT. The magenta arrows indicate the purely linear mapping in contrast to the black arrows whose mapping also contains the use of the nonlinear activation function in the receiving neuron.

2.6 Framework and error decomposition

In this section we present the framework used for our training and derive an error decomposition, into trunk and branch error. This decomposition is a simplification and adaption for our framework of the work done by Lanthaler et al. [14].

2.6.1 Framework

For convenience and faster training we consider the following situation for our training and test data.

1. We use m different input function $\hat{p}_j$ with $j \in [m]$, and for each input function we evaluate the solution / approximation on the same n coordinates r_i. Thus, we have nm data points.

2. We consider scalar solutions: $u_p^*(r) \in \mathbb{R}$.

3. We choose input functions p from finite dimensional subspace of $\mathcal{C}(D)$. E.g. instead of $p \in \mathcal{C}([0,1])$ we choose $p(x) = \sum_{i=1}^F a_i \sin(i\pi x)$. Furthermore, we sample p on a uniform mesh with M points,

$$\hat{p} = \left(p\left(\frac{0}{M-1}\right), p\left(\frac{1}{M-1}\right),\ \ldots,\ p\left(\frac{M-1}{M-1}\right)\right)^T \in \mathbb{R}^M. \tag{2.16}$$

We choose M large enough such that p can be fully reconstructed. This yields 0 encoding error. We thus stop distinguishing between p and $\hat{p}$.

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 DeepONets output for all coordinates and all input functions is

$$\begin{aligned}
G_\theta(\{p_1, ..., p_m\})(\{r_1, ..., r_n\}) &= \sum_{j=1}^{N} t_j(\{r_1, ..., r_n\}) b_j(\{p_1, ..., p_m\}) && (2.17)\\
&= \sum_{j=1}^{N} \begin{pmatrix} t_j(r_1) \\ \vdots \\ t_j(r_n) \end{pmatrix} \begin{pmatrix} b_j(p_1) & \dots & b_j(p_m) \end{pmatrix} && (2.18)\\
&= \begin{pmatrix} t_1(r_1) & ... & t_N(r_1) \\ \vdots & & \vdots \\ t_1(r_n) & ... & t_N(r_n) \end{pmatrix} \begin{pmatrix} b_1(p_1) & ... & b_1(p_m) \\ \vdots & & \vdots \\ b_N(p_1) & ... & b_N(p_m) \end{pmatrix} && (2.19)\\
&= TB^T =: \tilde{A}, && (2.20)
\end{aligned}$$

such that $\tilde{A}_{ij} = G_\theta(p_j)(r_i)$. Here, $T_{ij} = t_j(r_i)$ and $B_{ij} = b_j(p_i)$ are the output matrices of trunk and branch network respectively. This framework (1) 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 (2) the matrix notation significantly simplifies the analysis.

2.6.2 Error decomposition

We can now derive a simple error decomposition into trunk and branch error given this framework.

$$\begin{aligned}
\tilde{A} - A &= TB^T - A = TB^T - TT^\dagger A + TT^\dagger A - A && (2.21)\\
&= T(B^T - T^\dagger A) + (TT^\dagger - I)A && (2.22)\\
\varepsilon := ||\tilde{A} - A||_F^2 &= \underbrace{||T(B^T - T^\dagger A)||_F^2}_{\varepsilon_B} + \underbrace{||(TT^\dagger - I)A||_F^2}_{\varepsilon_T} && (2.23)
\end{aligned}$$

Here $T^\dagger$ is the Moore-Penrose inverse of T, and thus $I - TT^\dagger$ is the orthogonal projection matrix of the trunk space. Thus, the trunk error ε_T measures the error of projecting the target data matrix onto the trunk space.
A lower bound on the trunk error is given by the optimal rank N trunk basis, which is given by the first N left-singular vectors of A. Thus, we perform SVD of A,

$$A = \Phi\Sigma V^T = [\Phi_1 \ \Phi_2] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} \qquad (2.24)$$

with $\Phi_1 \in \mathbb{R}^{n\times N}, \Sigma_1 \in \mathbb{R}^{N\times N}, V_1 \in \mathbb{R}^{m\times N}$. Then $\varepsilon_{SVD} = ||(\Phi_1\Phi_1^T - I)A||_F^2$ is the SVD truncation error and $\varepsilon_{SVD} \leq \varepsilon_T$ for a given N.

Given a trunk matrix T, the branch matrix B_* which approximates the target matrix A best in the Frobenius norm is

$$B_* = \underset{B}{\arg\min} \, ||A - TB^T||_F^2 = (T^\dagger A)^T. \tag{2.25}$$

Thus, $\varepsilon_C := ||B - (T^\dagger A)^T||_F^2 = ||B^T - T^\dagger A||_F^2$ is the difference between the optimal and the actual branch matrix. However, the branch error ε_B weights the error of the branch approximation through the trunk matrix. An obvious lower bound on the branch error can not be derived.
Note that the final error bound in [14] is more similar to the following, which weights all branch output neurons by the trunk matrix' spectral norm.

$$\varepsilon \leq ||T||_2^2 \underbrace{||B^T - T^\dagger A||_F^2}_{\varepsilon_C} + ||(TT^\dagger - I)A||_F^2 \tag{2.26}$$

Furthermore, Lanthaler et al. consider the more general case in which p can not necessarily be fully reconstructed from $\hat{p}$, which adds an encoding error. Also their error is defined via a measure over the sample space, not only over the sampled points.
Note the relation between the (absolute squared) error $\varepsilon = ||A - \tilde{A}||_F^2$ used in derivations, the relative error $\delta = \frac{||A-\tilde{A}||_F}{||A||_F}$ used as an intuitive performance indicator and the loss $\mathcal{L} = \frac{1}{nm}||A - \tilde{A}||_F^2$ used in the training of neural networks. Additionally, we also use the relative error for the partial errors $\delta_T = \frac{\sqrt{\varepsilon_T}}{||A||_F}$ and $\delta_B = \frac{\sqrt{\varepsilon_B}}{||A||_F}$, such that $\delta^2 = \delta_T^2 + \delta_B^2$.

Chapter 3

SVDONet framework

3.1 Example problems

In all following sections, we trained DeepONets (and SVDONets) to approximate the operators described in detail in the Appendix. We consider the time evolution operators for three different PDEs and various solution times. All examples shown in the main part of the thesis approximate the solution operator of the Korteweg-de Vries equation, unless noted otherwise.

$$0 = \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} + 0.01\frac{\partial^3 u}{\partial r^3} \text{ for } r \in (0, 2\pi), t > 0 \tag{3.1}$$

$$u(r, t = 0) = p(r) \tag{3.2}$$

$$u(0) = u(2\pi) \tag{3.3}$$

We approximate the operator $G_* : p(\cdot) \mapsto u(\cdot, t = \tau)$ for $\tau = 0.2$, with

$$p(r) = \sum_{i=1}^{5} a_i \sin(ir) \tag{3.4}$$

and all coefficients $a_i \sim \mathcal{U}([-1, 1])$. The reference solutions are obtained using the finite-difference and Runge-Kutta solver *py-pde* [25] with a uniform mesh with $n = M = 401$ spatial gridpoints and a timestep of $\Delta t = 10^{-4}$. The training and test data set contain 900 and 100 input functions respectively.

3.2 Applying Lanthaler

The DeepONet consists of the branch and the trunk network. Our simplified setup together with Lanthaler's error decomposition allows us to decompose the error into trunk and branch error, see Section 2.6. Computing these errors for our 7 example problems and DeepONets of increasing N shows a similar trend. In Figure 3.1 the different errors (trunk, branch, total and SVD truncation

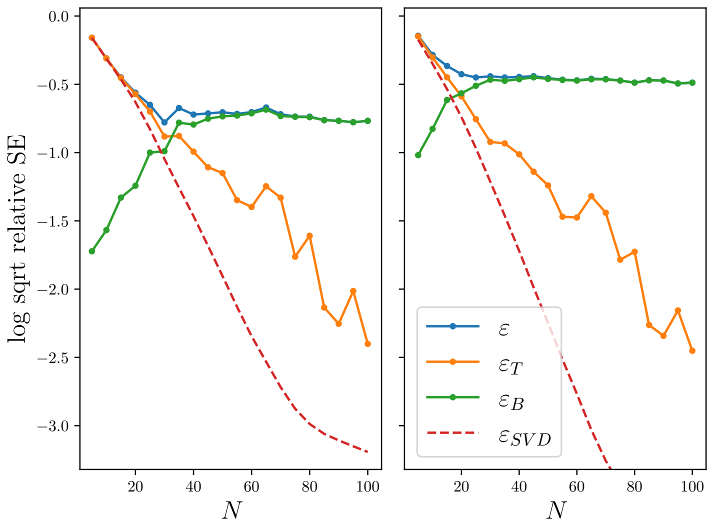


Figure 3.1: Training errors (left) and test errors (right). ε_{SVD} for some N gives the error done by truncating the true data matrix to it's best rank N approximation. KdV Equation $\tau = 0.6$ (width of hidden layers 200).

error) are shown. We see that by increasing the inner dimension N, the trunk error is continuously lowered, while the branch error increases and reaches a plateau until $\varepsilon \approx \varepsilon_B$. This observation is equivalent to the fact that new trunk neurons add a linearly independent direction to the trunk space, leading to a growing trunk space and hence a decreasing projection error. This fundamental effect has, to the best of our knowledge, not been reported in the literature yet. Conversely, it is intuitive that a branch network of fixed width and depth can less accurately learn the coefficients for $N = 100$ neurons than for $N = 20$ neurons.
Note that for small N, the trunk error is near optimal with $\varepsilon_T \approx \varepsilon_{SVD}$. Furthermore, we observe a similar decrease of ε_T for the test and the training case, showing that the DeepONet learns generalizing basis functions.

3.3 SVDONet

3.3.1 Gist

We've seen the error is in the branch net, for a large enough N. We now construct a DeepONet which removes the trunk net and replaces it with some fixed vectors. Firstly, this significantly simplifies the analysis since all parameters are now branch parameters. Secondly, we choose the, in some sense (described later), ideal trunk basis, highlighting the gravity of the branch error. However, the SVDONet is technically not approximating the solution operator mapping to an infinite dimensional vector space anymore, since we directly map to a finite dimensional one. Finally, the so-called SVDONet enables us to decompose the branch error into different errors, the mode losses which gives us deep insight into the inner workings of the branch net.

3.3.2 What exactly is SVDONet

To see the intuition behind the SVDONet, we consider (1) the matrix formulation of the DeepONet $\tilde{A} = TB^T$ with $T \in \mathbb{R}^{n\times N}, B \in \mathbb{R}^{m\times N}$ and (2) the SVD of the training data matrix A_{tr} (see Eq. 2.24)

$$A_{tr} = \Phi\Sigma V^T = [\Phi_1 \ \Phi_2] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} = \Phi_1\Sigma_1 V_1^T + \Phi_2\Sigma_2 V_2^T, \tag{3.5}$$

with $\Phi_1 \in \mathbb{R}^{n\times N}, \Sigma_1 \in \mathbb{R}^{N\times N}, V_1 \in \mathbb{R}^{m\times N}$. The SVDONet is set up such that the branch net approximates the coefficients corresponding to the first N left-singular vectors, i.e. for the training data, we want B to approximate the right-singular vectors V_1. Thus, the trunk matrix is replaced by $\Phi_1\Sigma_1$, the first N left-singular vectors of A, scaled with the corresponding singular value. Thus, the SVDONets output is $\tilde{A} = \Phi_1\Sigma_1 B^T$. For single coordinates r and input functions p, this yields

$$G(p)(r) = \sum_{i=1}^{N} b_i(p)t_i(r),$$

for the standard DeepONet and

$$G^{Eigen}(p)(r_j) = \sum_{i=1}^{N} \sigma_i b_i(p)(\phi_i)_j$$

for the SVDONet. We see here, that the trunk networks output neurons t_i are replaced by the left-singular vectors ϕ_i, which are from now on called (spatial) 'modes'. This entails that the SVDONet can only be evaluated at the training coordinates r_j. Also, note that without the factor σ_i in the sum, a similarly performing net can be constructed. However the analysis is slightly simplified with σ_i's. An extensive study comparing the performance impact of σ_i's can be found in the appendix.

Note, that when evaluating the SVDONet on the test data, the left-singular values Φ_1 of the training data matrix are used.

3.3.3 SVDONet's Error Decomposition

For the error of the SVDONet, we use $A_{tr} = \Phi_1 \Sigma_1 V_1^T + \Phi_2 \Sigma_2 V_2^T$ and thus get a decomposition of the training error

$$\varepsilon_{tr} = ||\tilde{A}_{tr} - A_{tr}||_F^2 = ||\Phi_1 \Sigma_1 B_{tr}^T - \Phi_1 \Sigma_1 V_1^T - \Phi_2 \Sigma_2 V_2^T||_F^2 \tag{3.6}$$

$$= ||\Sigma_1 B_{tr}^T - \Sigma_1 V_1^T||_F^2 + ||\Sigma_2||_F^2 = \sum_{i=1}^{N} \sigma_i^2 \underbrace{||v_i - b_{i,tr}||_2^2}_{L_{i,tr}} + ||\Sigma_2||_F^2. \tag{3.7}$$

By design, the trunk space is spanned by the first N left-singular vector of A_{tr}, thus $||\Sigma_2||_F^2 = \varepsilon_T = \varepsilon_{SVD}$, and hence 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. This displays the main advantage of the SVDONet: the branch error can be decomposed into the error of the different modes L_i.
For test data, the error decomposition is slightly less straight forward. We begin by computing the coefficients W_1 of the N first modes for the test input functions, i.e. the equivalent of V_1,

$$W_1 = (\Sigma_1^{-1} \Phi_1^T A_{te})^T. \tag{3.8}$$

The part of A_{te} orthogonal to Φ_1, i.e. not captured by the SVDONet is

$$(I - \Phi_1 \Phi_1^T) A_{te} =: \Phi_2 \Sigma_2 W_2^T. \tag{3.9}$$

Note that neither W_1, nor W_2 are orthogonal, in contrast to V_1 and V_2. Then

$$\varepsilon_{te} = ||\tilde{A}_{te} - A_{te}||_F^2 = ||\Phi_1 \Sigma_1 B_{te}^T - \Phi_1 \Sigma_1 W_1^T - \Phi_2 \Sigma_2 W_2^T||_F^2 \tag{3.10}$$

$$= ||\Sigma_1 B_{te}^T - \Sigma_1 W_1^T||_F^2 + ||\Sigma_2 W_2^T||_F^2 \tag{3.11}$$

$$= \sum_{i=1}^{N} \sigma_i^2 \underbrace{||w_i - b_{i,te}||_2^2}_{L_{i,te}} + ||\Sigma_2 W_2^T||_F^2. \tag{3.12}$$

Here w_i is the i-th column of W_1. Note that the norm of W_2 is not bounded in any sense, the test trunk error $||\Sigma_2 W_2^T||_F^2$ might be much larger than the training trunk error. However, this is not the case in practice, see Appendix.
Furthermore, since the choice of N determines the trunk error, it can be omitted from the loss. Thus, we use

$$\mathcal{L}_{tr} = \frac{1}{n_{tr} m_{tr}} ||\Sigma_1 B_{tr}^T - \Sigma_1 V_1^T||_F^2 = \frac{1}{n_{tr} m_{tr}} \varepsilon_{B,tr} \tag{3.13}$$

$$\mathcal{L}_{te} = \frac{1}{n_{te} m_{te}} ||\Sigma_1 B_{te}^T - \Sigma_1 W_1^T||_F^2 = \frac{1}{n_{te} m_{te}} \varepsilon_{B,te} \tag{3.14}$$

$$\tag{3.15}$$

as training and test loss from now on. This means we only measure the approximation error on the first N modes.

The mode loss L_i is also termed unweighted mode loss, while $\sigma_i^2 L_i$ is the weighted mode loss. We also introduce the term 'base loss' for modes here, as a useful reference for mode losses in practice. The base loss of mode i is the loss of mode i attained, if $b_i = 0$. Thus, the unweighted base loss on the training data is $||v_i||_2^2 = 1$, as V_1 is orthogonal, and thus the weighted base loss on the training data is σ_i^2. For the test data the unweighted base loss is $||w_i||_2^2$ and thus the weighted base loss is $s_i^2 := \sigma_i^2 ||w_i||_2^2$.
Note that a similar decomposition of the loss into different modes could be done for standard DeepONets as well, either with the left-singular vectors of A_{tr}, or the left-singular vectors of the learned trunk matrix T etc. However, fact that the trunk network is fixed over time and the coincidence of modes and output neurons makes the decomposition particularly elegant for SVDONets.

3.3.4 How is this ideal

The chosen trunk basis is ideal in the sense that

1. as discussed, the first N left singular vectors of the training data matrix A_{tr} obviously yield a basis for the best rank N approximation of A as seen from $\varepsilon_T = \varepsilon_{SVD}$, and

2. through the insertion of Σ_1, the branch networks training target V_1 is normalized. It is well known that it is desirable for neural network performance if the target output is somehow normalized [23].

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 accuracy increase from replacing the trunk net with the left-singular vectors. We mainly introduce the SVDONet to facilitate our analysis.

3.3.5 Related work

Note that a similar modification to the DeepONet is proposed in [7] as the POD-DeepONet. However, the POD-DeepONet does not scale the modes with the corresponding singularvalues, but uses a bias term. Most importantly, the decomposition into, first, trunk and branch error, and second, mode errors, is not done.
Recall that the SVDONet takes the input function and outputs the coefficients of some predetermined basis vectors. Taking this one step further, some proposed architectures do not take the true input function as input, but the coefficients of this true input function with respect to a predetermined basis of the input space. The spectral neural operator uses Chebyshev polynomials and trigonometric functions [6] as a basis, whereas the left-singular vectors of input and output matrices are used in [1].

Chapter 4

Methods and Results

To showcase the usefulness of the SVDONet, we present three very different insights gained from the SVDONet and the mode decomposition in this chapter. First, we analyze the error to understand which modes contribute to the test and training error, and investigate the different generalizability of the modes. To this end we investigate different optimization algorithms.
Second, we investigate the stacked SVDONet, an architecture in which the coefficients for the different modes are learned separately. We find this to lead to improvements for some, but not all modes.
Third, we analyze the influence different modes have on each other, in the standard architecture.

4.1 Mode loss distribution

We start by looking at SVDONets trained using gradient descent (GD). In Fig. 4.1 (3rd from the right, top and bottom), the mode losses train (top) and test (bottom) weighted normalized mode losses $\sigma_i^2 L_i/\bar{m}$, with $\bar{m}$ being the number of input functions. The figures also show the (squared) singular values σ_i^2/m_{train} and the (squared) norm of the projection of the test data matrix onto the modes of the training data matrix s_i^2/m_{test} as the 'base losses', which are achieved by $b_i = 0$. One sees that only the loss of the first ≈ 10 the mode loss is lowered below the 'base loss', over the course of the training process. Since the singular values vary hugely in size, the different modes are very differently represented in the gradient. Thus, the modes with smaller singular values ('smaller modes') are not optimized - the gradient with respect to them is multiplied by a very small number (compared to other modes' gradients). 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. This can be seen for both the test and training error. As seen in the left most plot, the total error for test and training data are very similar too.
What happens, if we increase the coefficient of these smaller modes in the gra-

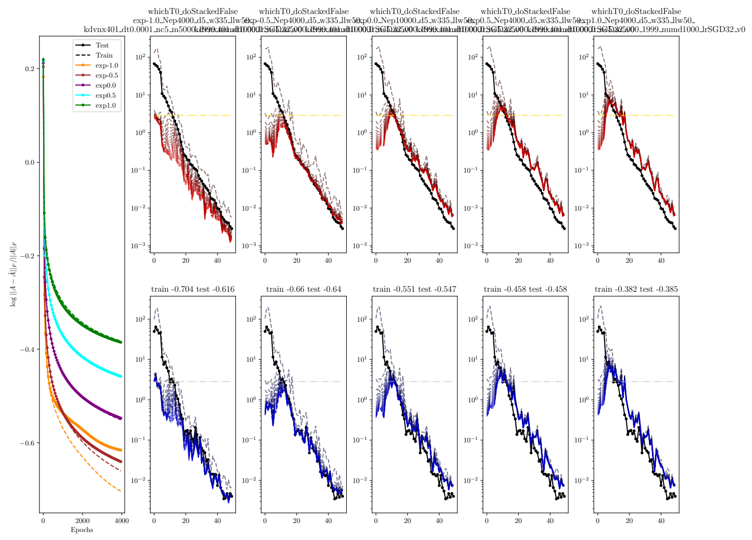


Figure 4.1: ...

dient? This can, e.g. be done by re-weighting: We go from the training loss for SVDONets (based on the branch error)

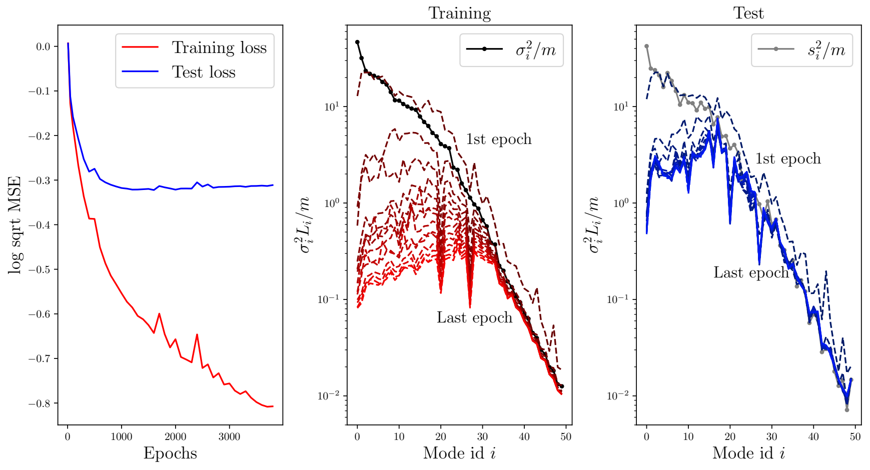
$$\mathcal{L}_{tr} = \frac{1}{n_{tr}m_{tr}} \sum_{i=1}^{N} \sigma_i^2 ||v_i - b_i||_2^2, \tag{4.1}$$

to the new loss

$$\mathcal{L}_{e,tr} = \frac{1}{n_{tr}m_{tr}} \sum_{i=1}^{N} \sigma_i^{2+2e} ||v_i - b_i||_2^2. \tag{4.2}$$

Note that we only use $\mathcal{L}_{e,tr}$ only to compute the gradients, and hence the parameter updates, not as a performance metric. For $e = 0$ the old loss is recovered, for $e > 0$ large modes are emphasized and for $e < 0$ the small modes are emphasized. For $e = -1$ all modes are equally important. The results for $e \in \{-1, -0.5, 0, 0.5, 1\}$ can be seen in Fig. 4.1.
We see that $e = -1$ achieves the lowest training loss. Notably in the training case, for $e = -1$ the loss of the first 10 modes is significantly higher than for larger e, however the loss for all other modes is significantly lower. For all shown values of the exponent, $e = -1$ is the only one that achieves a significant loss reduction for all modes below the base loss. For $e > 0$ the large modes are even more dominant in the gradient and thus the loss of even fewer modes is reduced, compared to $e = 0$. However, the loss of the first mode is not lower for $e = 1$, compared to $e = 0$, leading to a larger training loss for $e > 0$.

We also observe that $e = -0.5$ achieves the lowest test loss. This stems from the fact that some modes which achieve a low training loss for $e = -1$ do not achieve a low test loss. Thus, since $e = -0.5$ achieves a lower training loss on the first few modes compared to $e = -1$, which is transferred to a lower test loss, the overall test loss is lower.
This difference in magnitude of the base losses appears to be (at least) one of the reasons why adaptive gradient optimization schemes such as AdaGrad (or Adam) either cite papers or put data in appendix perform so much better. We will now look at the mode loss distribution for a weighted loss for Adam, an adaptive scheme with momentum. See Fig.4.2 A) ($e = 0$).
We see much lower losses are achieved compared to GD. Furthermore, in contrast to GD, the model is overfitting not for all examples in the first 4k epochs, the gap btw test and train is visible always though; the training error is continuously lowered, while the test error reaches a local minimum. For the training error we observe that, while at very different speeds, finally all mode losses significantly decrease (not for all visible in this plot). For Adam we observe a significant loss reduction on the first 30 modes, for both test and training data. However, even for these modes the test loss is significantly higher than the training loss. For the 20 smallest modes both the test and the training error are slightly below the respective base loss.
When considering $e \neq 0$, Fig. 4.2 B), we see that the re-weighting has a very strong effect on the mode level. For $e = 1$ the loss for the first few modes is lower than for $e = 0$, while the loss for all other modes is increased. For $e = -1$ a, for some examples, approximately homogeneous unweighted loss $L_{i,tr}$ can be achieved. However, these low mode losses are, yet again, not transferred to the test losses. Furthermore, since for $e = -1$ the training error is increased for the large modes, and since these are dominant in the loss, the overall training loss for $e = -1$ is larger than for $e = 0$. Interestingly, since the large modes generalize much better, there are some examples (e.g. Bid5, KdV, $\tau = 1$) for which $e = 1$ achieves a slightly lower test error than both $e = 0$ and $e = -1$.
We have seen for both Adam and GD, that small modes don't seem to generalize well: even if their training loss is lowered, their test loss is not. Furthermore, GD fails, since the gradients of the smaller mode losses are underrepresented. This problem can be approached either by explicitly re-weighting the mode losses or using adaptive gradient schemes.



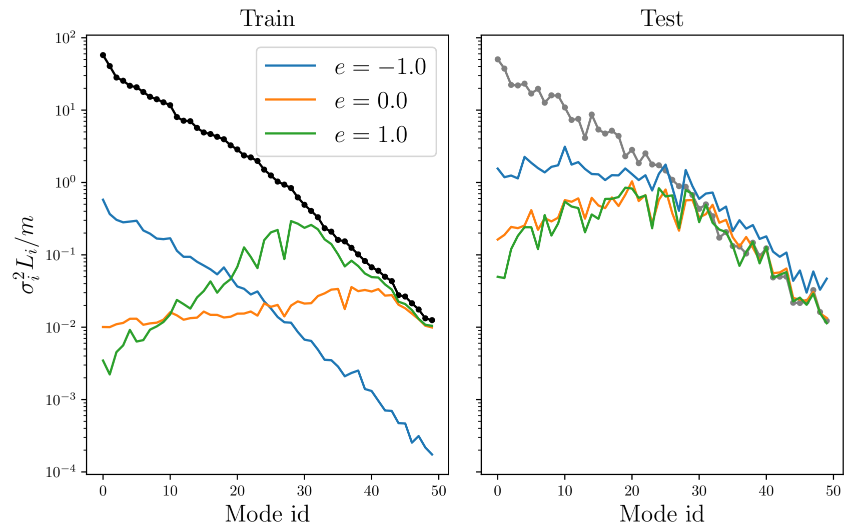


Figure 4.2: Total training and test loss over epochs (left), training mode losses (center) and test mode losses (right). AND Training mode losses (left) and test mode losses (right). Both for SVDONet and KdV equation $\tau = 0.6$.

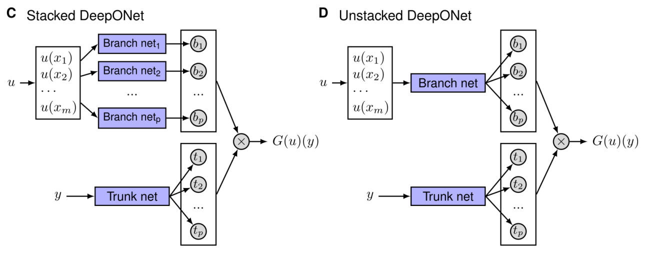


Figure 4.3: Stacked and Unstacked DeepONet, from [2].

4.2 Stacked DeepONet

In the branch net of the SVDONet, the i-th output neuron corresponds to the map from the input function p to $b_i(p)$ the coefficient of the i-th mode. In the SVDONet, and correspondingly in the standard DeepONet, the branch output neurons share the hidden layers. It is intuitively not clear, why this 'neuron sharing' of the coefficients of the different modes would be beneficial. Alternatively, one could build a separate feedforward network for each of the N coefficients. This transfers the idea behind the so-called stacked DeepONet, proposed by Lu et al. [2] (see Figure 4.3), to the SVDONet. The stacked DeepONet contains one feedforward network for the trunk network, the stacked SVDONet of course does not.

How does the performance of stacked and unstacked SVDONets differ? For this we need to compare a stacked to an unstacked net. We compare nets with the same number of parameters, see Appendix. In Fig. 4.4 the training and test loss curves of the stacked and unstacked SVDONet and the training and test mode losses of both are shown. We find that, the unstacked net always achieves a better test error. For the training error, it depends on the example. Thus, in some sense, 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. 10-15 in Fig. 4.4) the unstacked SVDONet achieves lower 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 go beyond (1) which modes to approximate how well and (2) 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 net, whereas their test loss is lower for the stacked net (in Fig. 4.4 these are modes 2-7). The opposite can be observed for intermediate and small modes. There the stacked net significantly outperforms the unstacked net on the training data, but the unstacked net achieves a lower test loss. This again holds for all considered examples, in Fig. 4.4 these are modes 17-21 and

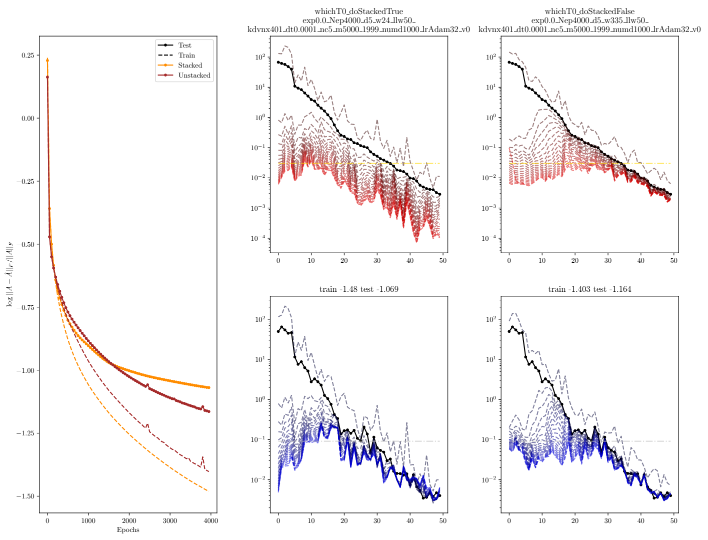


Figure 4.4: ...

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.

4.3 Mode coupling

For the comparison between the stacked and the unstacked net we asked the question, "If you rearrange your network so that the modes have separated parameters, but keep the number of parameters constant, how does this affect generalization abilities?". In this section we take unstacked SVDONets and ask "Which contribution to the loss comes from the coupling of the modes?". In this section we are concerned with the coupling of the modes in the parameter space. Whereas changing the value of the i-th branch output neuron does not effect the output of the SVDONet in any other mode, due to their orthogonality, updating the parameters to achieve a lower loss on mode i can significantly change the values of all branch output neurons. Note that this is not the case for the stacked SVDONet. Hence, we define mode coupling in the following way.

4.3.1 GD training

Consider a GD parameter update $\delta\theta = -\alpha \nabla_\theta \mathcal{L}_{tr} = -\alpha \sum_{j=1}^{N} \sigma_j^2 \nabla_\theta L_{j,tr}$. If we choose α small enough, we can approximate the change in the training and test loss function using a 1st order Taylor expansion. First, for the i-th mode

$$L_i(\theta + \delta\theta) - L_i(\theta) \approx \nabla_\theta L_i^T \delta\theta = -\alpha \sum_{j=1}^{N} \sigma_j^2 \nabla_\theta L_i^T \nabla_\theta L_{j,tr} \tag{4.3}$$

$$= -\alpha\sigma_i^2 \nabla_\theta L_i^T \nabla_\theta L_{i,tr} - \alpha \sum_{j\neq i} \sigma_j^2 \nabla_\theta L_i^T \nabla_\theta L_{j,tr}. \tag{4.4}$$

Then, for the overall loss $\mathcal{L}$

$$\mathcal{L}(\theta + \delta\theta) - \mathcal{L}(\theta) = \sum_{i=1}^{N} \sigma_i^2 (L_i(\theta + \delta\theta) - L_i(\theta)) \tag{4.5}$$

$$\approx \underbrace{-\alpha \sum_{i=1}^{N} \sigma_i^4 \nabla_\theta L_i^T \nabla_\theta L_{i,tr}}_{=:d} \underbrace{-\alpha \sum_{i=1}^{N} \sigma_i^2 \sum_{j\neq i} \sigma_j^2 \nabla_\theta L_i^T \nabla_\theta L_{j,tr}.}_{=:\omega} \tag{4.6}$$

Note, that $-\alpha\sigma_i^2 \nabla_\theta L_i^T \nabla_\theta L_{j,tr}$ is the 1st order approximation of the weighted training or test loss change of mode i due to the parameter update $-\alpha\nabla_\theta L_{j,tr}$, i.e. the update done to minimize the j-th mode loss.
We call the first term in Eq. 4.6 the 'diagonal term' d, and the second 'off-diagonal term' ω. The names diagonal and off-diagonal refer to the matrix

$$S = -\alpha D^T D_{tr}, \text{ where } D = [\sigma_1^2 \nabla L_1, \ ..., \ \sigma_N^2 \nabla L_N]. \tag{4.7}$$

We define the mode coupling strength as ω, since it contains the weighted inner products of the gradients of the different mode losses.

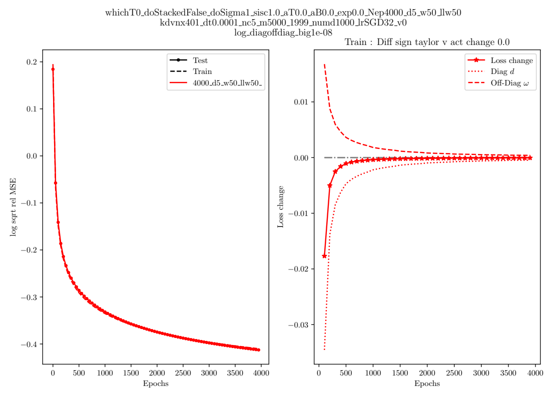


Figure 4.5: ...

We now investigate the questions: How large are d and ω? What are their signs? Is one of them negligible?

Note, that negative $d + \omega$ leads to a loss reduction and is hence beneficial for the performance.

In Fig. 4.5 the test and training loss over the epochs (left) and the different contributions d, ω and the total loss change (right) are shown. Since the behavior of the d and ω in the considered cases is very similar for test and training data we focus on the training data. In this example we observe $d < 0$ and $\omega > 0$. More specifically, as the loss change, the diagonal term d and the off-diagonal term ω decay over the epochs, the ratio ω/d decreases from -0.5 to -0.85 over the first 4k epochs. For other examples, such as the advection diffusion equation or the burgers equation, ω is initially negative too, and then increases significantly over the epochs, leading to $\omega/d \approx -0.5$ after 4k epochs. Furthermore, we can inspect the contributions to the diagonal and off-diagonal parts through the entries of $S = -\alpha D^T D$, see Fig. 4.6. We see that while the magnitude of the maximum entries decreases over the epochs, the number of positive entries increases. Hence for specific mode pairs there is less strong detrimental coupling, but there are more mode pairs with detrimental coupling in total. Note also, that the majority of the positive contribution of ω comes from the first 10 modes, which are also the only modes which are also the only modes reached by GD, see Fig. 4.1. Hence detrimental coupling between the relevant modes seems to be a crucial part of SVDONets trained with GD.

However, what happens if we consider larger networks? We increase the width of the hidden layers, not the inner dimension N.

Fig. 4.7 shows the off-diagonal term ω over the loss reduction for SVDONets of multiple widths w. While $\omega \geq -(d + \omega)$ for $w = 50$, for $w = 220$ we find $\omega < -0.1(d + \omega))$, i.e., a strong decoupling with increased width. For the examples in which ω is initially negative, decoupling is only clearly observed for later epochs, when $\omega > 0$. Note, that the wider SVDONets achieve significantly

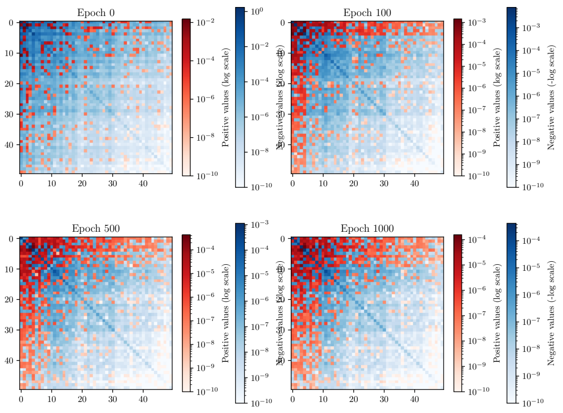


Figure 4.6: ...

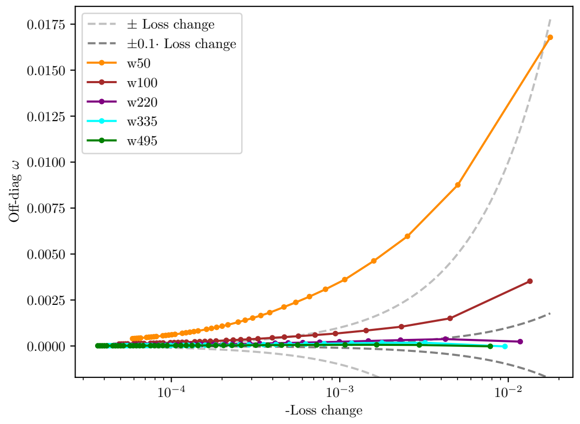


Figure 4.7:

lower test and training losses.

We can now compare the unstacked net to the stacked net. For the stacked net, one finds $\omega = 0$, since we can divide the stacked nets 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 4.2 the performance of stacked and unstacked SVDONets optimized using Adam was investigated. If GD is used, the unstacked SVDONet consistently performs better on both test and training data. Since a detrimental coupling, $\omega > 0$ is observed for many examples, the mode coupling can not be the main cause of performance difference between stacked and unstacked SVDONets.

4.3.2 Adam training

We can also apply this mode coupling analysis to SVDONets trained using Adam. We however continue to look at the entries of S, see Eq. 4.7, whose entries sum up to the loss change according to the 1st order Taylor expansion of the loss with a gradient descent update, not an Adam update. Thus, it should be seen less as an analysis of the optimization procedure, more the state (produced by the optimization procedure).
Firstly, when considering d and ω over the epochs, see Figure 4.8 left most column, we see that there are many epochs in which $\omega \approx 0$ and $d < 0$, but the actual loss change, through Adam, is ≈ 0 too. This shows the difference between Adam and GD updates.
The entries of S_{tr} show an interesting pattern, the modes that are well approximated, i.e. the first few, are beneficially coupled: optimizing one of them (through a gradient descent step) leads to a lower loss for the others. However, the coupling to the worse approximated modes is detrimental: optimizing a good mode leads to higher loss on a bad mode (and vice versa). These bad modes are beneficially coupled to each other again. This effect strengthens over time, even if the 'bad' modes get better. Additionally, since more and more modes become well approximated over time, they move from one set of beneficially coupled modes to the other.
For narrower nets, the similarity between S_{tr} and S_{te} is high, showing a similar coupling structure for train and test, which is inline with the narrower nets overfitting less. Note, that S_{te} is not symmetric, since we consider the effect of the training gradients on the test loss. For wider nets, the similarity decreases showing a less structured coupling.
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.
The inspection of S_{te}'s entries gives insight into another phenomenon, we call mode-internal overfitting (MIO). We find MIO of mode i, 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., $L_{i,te}(\theta + \delta\theta_i) > L_{i,te}(\theta)$. An alternative,

more strict criterion is: mode i is internally overfitting, if $L_{i,tr}(\theta + \delta\theta_i) < L_{i,tr}(\theta)$ and $L_{i,te}(\theta + \delta\theta_i) - L_{i,te}(\theta) > \frac{1}{2}(L_{i,tr}(\theta + \delta\theta_i) - L_{i,tr}(\theta))$, i.e., the test loss reduction is not big enough, compared to the training loss reduction. We consider $\delta\theta_i = -\alpha\nabla_\theta L_{i,tr}(\theta)$ and the 1st order Taylor expansion of L_i. Thus, the question whether MIO takes place reduces to the comparison of the signs of the diagonals of S_{tr} and S_{te}. As seen in Fig. 4.8 we clearly see MIO for some modes, especially for wider nets, which in general overfit more. Out of the 50 modes, 10-20, depending on strict or less strict criterion and epoch, display MIO. This shows that MIO is a widely occuring phenomenon, but we also see that far from all overfit modes display MIO.

Note, that neither MIO or the structured coupling are necessarily unique to Adam. They might be observed in e.g. GD too, but since they strengthen over time / with lower loss, we havent observed them for GD yet.

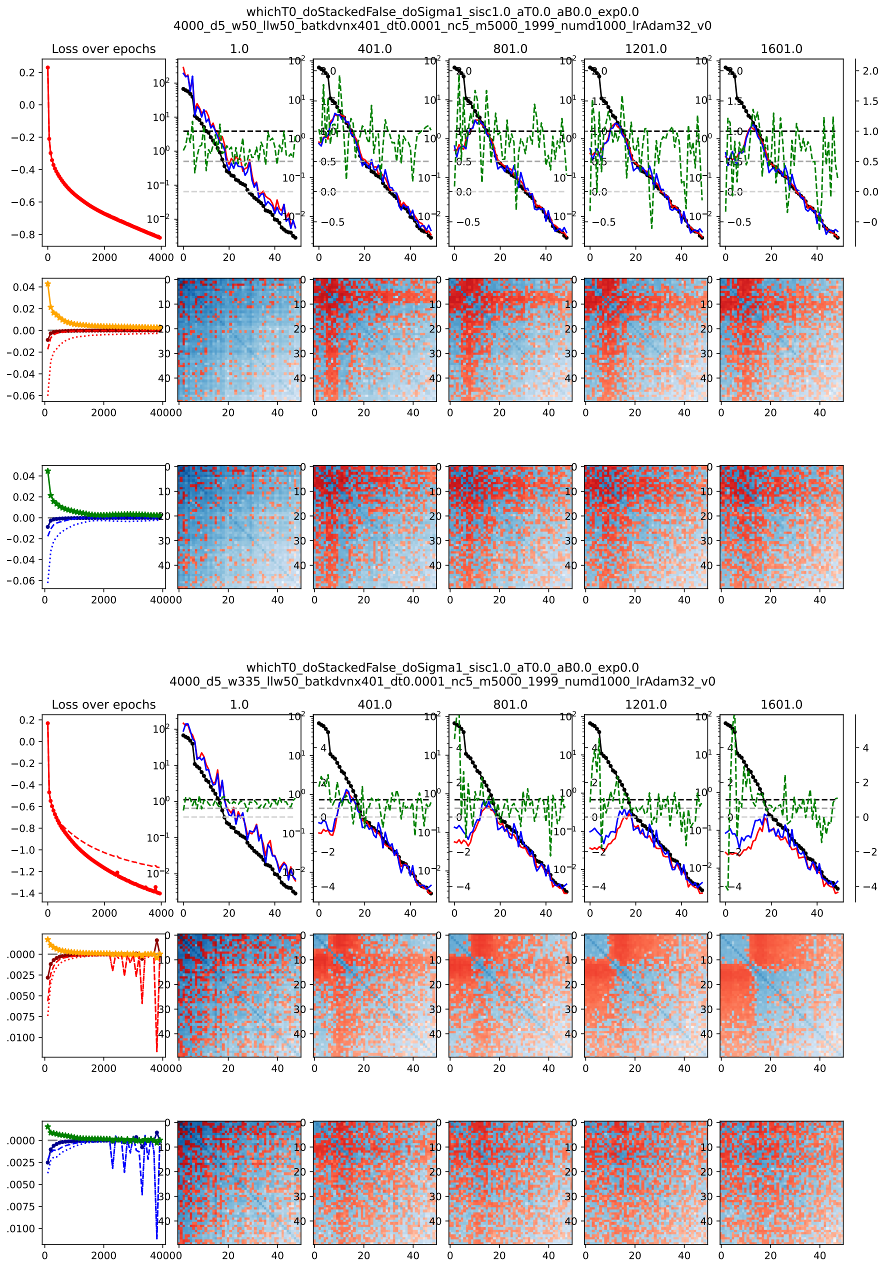


Figure 4.8: top $w = 50$, bottom $w = 335$

Chapter 5

Discussion

5.1 Conclusion

Identified loss in branch
Constructed SVD ONet
looked at mode losses
(de)coupling in GD nets
difference in stacked and unstacked, beyond diags / offdiags
$\rightarrow$ mode framework opens a new point of view on many things from architecture (stacked vs unstacked) to optimizer

Bibliography

[1] Kaushik Bhattacharya, Bamdad Hosseini, Nikola B. Kovachki, and Andrew M. Stuart. Model Reduction And Neural Networks For Parametric PDEs. *The SMAI Journal of computational mathematics*, 7:121–157, 2021.

[2] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, March 2021.

[3] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[4] Sanghyun Lee and Yeonjong Shin. On the training and generalization of deep operator networks, 2023.

[5] Sifan Wang, Hanwen Wang, and Paris Perdikaris. Improved architectures and training algorithms for deep operator networks. *Journal of Scientific Computing*, 2022.

[6] V. S. Fanaskov and I. V. Oseledets. Spectral neural operators. *Doklady Mathematics.*

[7] Lu Lu, Xuhui Meng, Shengze Cai, Zhiping Mao, Somdatta Goswami, Zhongqiang Zhang, and George Em Karniadakis. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. *Computer Methods in Applied Mechanics and Engineering*, 393:114778, April 2022.

[8] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier Neural Operator for Parametric Partial Differential Equations, May 2021. arXiv:2010.08895 [cs, math].

[9] Bogdan Raonić, Roberto Molinaro, Tim De Ryck, Tobias Rohner, Francesca Bartolucci, Rima Alaifari, Siddhartha Mishra, and Emmanuel

de Bézenac. Convolutional Neural Operators for robust and accurate learning of PDEs, May 2023. arXiv:2302.01178 [cs].

[10] Qianying Cao, Somdatta Goswami, and George Em Karniadakis. Laplace neural operator for solving differential equations. *Nature Machine Intelligence*, 6(6):631–640, June 2024. Publisher: Nature Publishing Group.

[11] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[12] Somdatta Goswami, Aniruddha Bora, Yue Yu, and George Em Karniadakis. Physics-informed deep neural operator networks, 2022.

[13] Emily Williams, Amanda Howard, Brek Meuris, and Panos Stinis. What do physics-informed deeponets learn? Understanding and improving training for scientific computing applications, 2024.

[14] Samuel Lanthaler, Siddhartha Mishra, and George E Karniadakis. Error estimates for deeponets: a deep learning framework in infinite dimensions. *Transactions of Mathematics and Its Applications*, 6(1):tnac001, 03 2022.

[15] Francesca Bartolucci, Emmanuel de Bézenac, Bogdan Raonić, Roberto Molinaro, Siddhartha Mishra, and Rima Alaifari. Representation equivalent neural operators: a framework for alias-free operator learning, 2023.

[16] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 1989.

[17] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[19] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. 1986.

[20] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[21] Geoffrey Hinton. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude"adam: A method for stochastic optimization. Accessed 21.07.2025.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[23] Christopher Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[24] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

[25] David Zwicker. py-pde: A python package for solving partial differential equations. *Journal of Open Source Software*, 5(48):2158, 2020.

Appendix A

Simplified DeepONet

Consider a fixed trunk output matrix T and a one-layer linear branch network $B(p) = Wp$.

Appendix B

Example problems

I.e., we consider the operator G_* mapping an initial condition $p(r)$ onto the solution $u^*_p(r)$ of the time-dependent PDE at time τ. We use the same PDEs, as in **??**.

B.1 Burger's Equation

B.2 KdV Equation

B.3 Advection Diffusion Equation

$$\frac{\partial u}{\partial t} + 4\frac{\partial u}{\partial r} - 0.01\frac{\partial^2 u}{\partial r^2} = 0 \tag{B.1}$$

with $r \in [0,1]$ and $u(r,t=0) = p(r)$ and $p(r) = \sum_{i=1}^{2} 0a_i \sin(2\pi i r)$
...