Student Eirik Fagerbakke
Supervisor Brynjulf Owren
Co-supervisor Sølve Eidnes
Co-supervisor Benjamin Kwanen Tapley

Energy-preservation in Operator Learning

Specialization Project

Trondheim, 2024

NTNU
Norwegian University of Science and Technology
Faculty of Faculty of Information Technology and Electrical Engineering
Department of Mathematics

NTNU
Norwegian University of
Science and Technology

Contents

List of Figures

List of Tables

Introduction 1

1.1 Motivation

The use of neural networks to learn functions has in general seen a lot of success, where we learn a mapping between finite-dimensional Euclidean spaces. A different and more recent approach is to learn operators, and has garnered a lot of attention [1]. The first architecture with the goal of learning *operators*, rather than functions, was introduced in 2019 [2, 3]. This was named the *DeepONet*, and allowed for mapping from a discretized function to an infinite-dimensional function space. Since then, a plethora of other architectures and extensions have been proposed [4–9]. Perhaps most notably, a family of methods dubbed *Neural Operators* were introduced in [10]. Of these, the method known as the *Fourier Neural Operator* (FNO) introduced in [10, 11] has seen a lot of success.

The use of operator learning allows us to learn the solution operator of a partial differential equation (PDE) directly from data. It is previously known that methods informed about the underlying physics of the problem, so-called Physics-Informed Neural Networks (PINNs), can outperform purely data-driven methods [12, 13]. This has also been explored for Neural Operators in [14] and for the DeepONet in [15]. These approaches do however require that we know the underlying PDE, or at least its general form, which is not always the case.

The *Hamiltonian Neural Networks* (HNN) [16] and *Lagrangian Neural Networks* (LNN) [17] instead work by strictly imposing the Hamiltonian or Lagrangian structure of the problem, respectively, to learn an ordinary differential equation (ODE). These approaches have been further explored for Pseudo-Hamiltonian systems in [18], and were then extended to partial differential equations (PDEs) in [19]. A similar approach to the HNN, but in the context of Hamiltonian PDEs and operator learning, were also proposed in [20]. This paper introduces the *Energy-consistent Neural Operator* (ENO), where they apply an energy-penalty to the loss function, similar to a PINN. The paper does however use the more traditional multilayer perceptron (MLP) architecture, instead of the operator learning methods previously mentioned.

This project will focus on comparing different DeepONet and FNO architectures for solving Hamiltonian PDEs, and will in particular build on the work of [20] by introducing a *Hamiltonian Operator Network* (HON). In this architecture, we will investigate the implementation and use of the different operator learning frameworks previously mentioned. We will also investigate imposing the Hamiltonian structure through a penalty term as in the ENO approach, or directly through the architecture, akin to the HNN approach.

Our proposed HON architecture allows for using the existing operator learning architectures, while also imposing a Hamiltonian structure on the learned

[1]: Kovachki et al. (2024), *Operator Learning: Algorithms and Analysis*

[2]: Lu et al. (2021), *DeepONet: Learning Nonlinear Operators for Identifying Differential Equations Based on the Universal Approximation Theorem of Operators*

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

[11]: Li et al. (2021), *Fourier Neural Operator for Parametric Partial Differential Equations*

[12]: Raissi et al. (2019), *Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations*

[13]: Karniadakis et al. (2021), *Physics-Informed Machine Learning*

[14]: Li et al. (2023), *Physics-Informed Neural Operator for Learning Partial Differential Equations*

[15]: Wang et al. (2021), *Learning the Solution Operator of Parametric Partial Differential Equations with Physics-Informed DeepOnets*

[16]: Greydanus et al. (2019), *Hamiltonian Neural Networks*

[17]: Cranmer et al. (2020), *Lagrangian Neural Networks*

[18]: Eidnes et al. (2023), *Pseudo-Hamiltonian Neural Networks with State-Dependent External Forces*

[19]: Eidnes et al. (2024), *Pseudo-Hamiltonian Neural Networks for Learning Partial Differential Equations*

[20]: Tanaka et al. (2024), *Neural Operators Meet Energy-based Theory: Operator Learning for Hamiltonian and Dissipative PDEs*

operator, with no previous knowledge of the underlying PDE. The learned operator will then enjoy the benefits of both the operator learning architecture and the learned Hamiltonian structure. Specifically, we will then achieve networks that are universal approximators of operators. The use of such an architecture also poses the problem of how one should compute the derivatives of the learned operator, which is not trivial in general, and is something we have had to address for the networks discussed in this project.

1.2 Problem setting

Following similar notation as [20] and [10], we want to learn an approximation to an operator $\mathcal{S}$, which maps from the input function space $\mathcal{A}$ to the output function space $\mathcal{U}$. $\mathcal{A}$ and $\mathcal{U}$ are assumed to be Banach spaces. $\mathcal{A}$ consists of functions defined on the bounded spatial domain $\mathcal{X} \subset \mathbb{R}^d$, and $\mathcal{U}$ consists of functions defined on the spatial-temporal domain $\mathcal{Y} = \mathcal{T} \times \mathcal{X} \subset \mathbb{R}^{d+1}$.

[20]: Tanaka et al. (2024), *Neural Operators Meet Energy-based Theory: Operator Learning for Hamiltonian and Dissipative PDEs*

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

The input function is denoted by $a : \mathcal{X} \to \mathbb{R}^{d_a}$, with $a \in \mathcal{A}$. It may in general correspond to the initial condition, a forcing term or a coefficient in the PDE. In this project, a shall refer to the initial condition of the PDE. The output function is denoted by $u : \mathcal{Y} \to \mathbb{R}^{d_u}$, with $u = \mathcal{S}[a] \in \mathcal{U}$. $\mathcal{S}$ is then the solution operator for the PDE.

We assume that we have some observations $\{a^{(i)}, u^{(i)}\}_{i=1}^N$ of the input-output pairs, where $a^{(i)} \sim \mu$ are assumed to be i.i.d. samples drawn from some probability measure μ supported on $\mathcal{A}$, and $u^{(i)} = \mathcal{S}[a^{(i)}]$ are the corresponding outputs.

Our goal is then to learn a neural network that approximates this operator, i.e. $\mathcal{S}_\theta[a] \approx \mathcal{S}[a]$, where $\theta \in \mathbb{R}^q$ are the parameters of the neural network.

We are further going to consider PDEs which have an energy functional, and we want the predictions of the neural network to preserve an approximation to the energy (or more precisely the *Hamiltonian*) of the governing PDE, which we discuss in Chapter 3.

A summary of the notation introduced here, and in other sections of the project, can be found in Table 3.

Background on DeepONets and Neural Operators

2

DeepONet

The Vanilla DeepONet

In [2], the authors introduce the DeepONet. The vanilla DeepONet is a network that takes a discretized function, a, as input (evaluated at p fixed "sensors"). It also takes a query point $y \in \mathcal{Y}$, which is where we want the output to be evaluated in the output domain. We then want the network to learn an operator $\mathcal{S}$, such that $\mathcal{S}[a](y) = u(y)$. It is important to note that the query points y are independent of the placements of the fixed sensors. This then leads to a network architecture that maps from a finite-dimensional space to an infinite-dimensional one.

[2]: Lu et al. (2021), *DeepONet: Learning Nonlinear Operators for Identifying Differential Equations Based on the Universal Approximation Theorem of Operators*

The motivation behind the network architecture stems from the Universal approximation theorem by Chen & Chen [2, 21], where they originally consider a shallow neural network. The authors of [2] then build a deep structure based on this. The theorem is reiterated below, with notation that matches ours:

[2]: Lu et al. (2021), *DeepONet: Learning Nonlinear Operators for Identifying Differential Equations Based on the Universal Approximation Theorem of Operators*

[21]: Tianping Chen et al. (1995), *Universal Approximation to Nonlinear Operators by Neural Networks with Arbitrary Activation Functions and Its Application to Dynamical Systems*

Theorem 2.0.1 (Universal approximation theorem for operators)
Suppose that σ is a continuous nonpolynomial function, X is a Banach Space, $\mathcal{X} \subset X, \mathcal{Y} \subset \mathbb{R}^{d+1}$ are two compact sets, $\mathcal{A}$ is a compact set in $C(\mathcal{X})$, $\mathcal{S}$ is a nonlinear continuous operator, which maps $\mathcal{A}$ into $C(\mathcal{Y})$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}, w_k \in \mathbb{R}^{d+1}, x_j \in K_1, i = 1, \dots, n, k = 1, \dots, p, j = 1, \dots, m$, such that

$$\left| \mathcal{S}[a](y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k a(x_j) + \theta_i^k \right)}_{branch} \underbrace{\sigma\left(w_k \cdot y + \zeta_k\right)}_{trunk} \right| < \epsilon \tag{2.1}$$

holds for all $a \in \mathcal{A}$ and $y \in \mathcal{Y}$.

[2]: Lu et al. (2021), *DeepONet: Learning Nonlinear Operators for Identifying Differential Equations Based on the Universal Approximation Theorem of Operators*

Based on Theorem 2.0.1, the DeepONet is split into two parts: the *trunk* and the *branch* nets. The trunk net takes the query point y as input, and the branch net takes the discretized function a as input. The outputs of the branch and trunk nets result in two p-dimensional vectors, which are then combined in a linear fashion via a dot product. It should be noted that the DeepONet is origianlly defined for scalar-valued functions, but can be generalized to vector-valued functions as well, see for instance [22].

[22]: Wang et al. (2021), *Long-Time Integration of Parametric Evolution Equations with Physics-Informed DeepONets*

The authors discuss some slightly varying architectures. One option they suggest uses a "stacked DeepONet", meaning that we have p different branch networks, each one producing one element in the branch net vector. Another way is to use the "unstacked DeepONet", where we use only a single network to produce all the p branch elements directly. Although Theorem 2.0.1 does not include a final bias term, the authors also

discuss the possibility of adding a bias to the final output of the DeepONet, to allow for a more expressive model. In this project, we have chosen to use the unstacked DeepONet with a bias, and we will be referring to this as the "Vanilla DeepONet". The predictions can then be written as

$$\mathcal{S}_\theta[a](y) = \sum_{i=1}^{p} b_k(a)t_k(y) + b_0, \tag{2.2}$$

where $b_k(a)$ are the branch net outputs, $t_k(y)$ are the trunk net outputs, and b_0 is the bias term. The branch and trunk nets can in general be parametrized by any neural network architecture, but we have chosen to use MLPs for both in this project.

The general DeepONet architecture is illustrated in Figure 2.1.

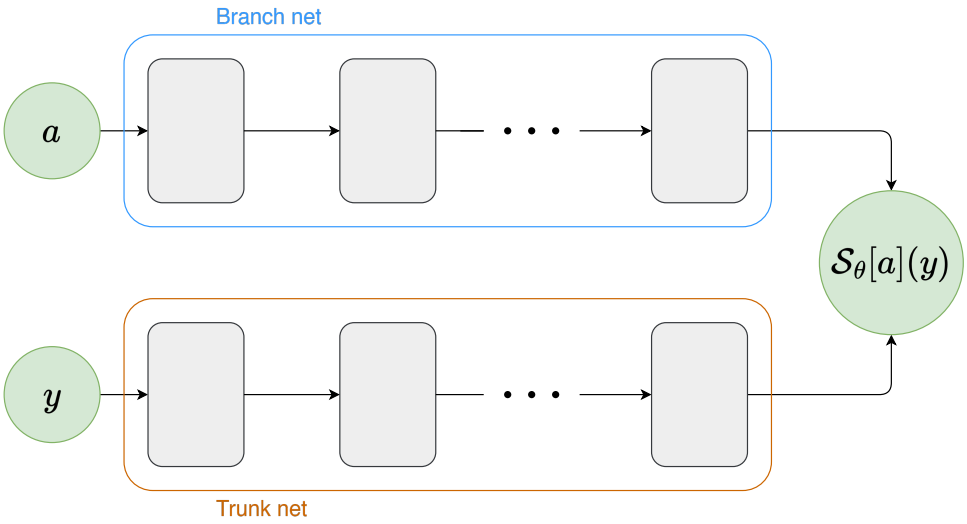

Figure 2.1: DeepONet architecture.

Evaluation of the DeepONet

The DeepONets are evaluated on single coordinates, so in order to get a prediction for the whole domain, we have to evaluate the model multiple times. This can be done by for instance utilizing `vmap` in `JAX`, which vectorizes a function automatically. However, as discussed in [3], we can make the computation even more efficient by re-using predictions from the branch and trunk nets. If we have a single input function which we want to evaluate at multiple points, we may evaluate the branch net once only, and combine it with the predictions of the trunk net by a matrix multiplication. The same holds for when we have multiple input functions and a single query point, or when we have multiple input functions and multiple query points.

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*

DeepONet extensions

There have been made several extensions to the vanilla DeepONet architecture. One noteable example includes the POD-DeepONet, introduced in [3], where some of the authors of the original DeepONet paper discuss an enhancement of their original

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*

architecture. The POD-DeepONet uses a pre-computed POD-basis, denoted by $\{\phi_k\}$, generated from the training data, in place of the trunk net. Instead of learning the basis for the query points, these are now pre-computed, and the network only learns the branch net parameters. The output can then be written as

$$\mathcal{S}_\theta[a](y) = \sum_{i=1}^{p} b_k(a)\phi_k(y) + \phi_0(y). \tag{2.3}$$

With the improved results seen in [3], this could be an interesting extension to explore. However, we have chosen to focus on the original DeepONet architecture as it is more general and expressive.

One criticism of the DeepONet, is that it is not able to take input functions at any point; the number of "sensors" and their placement remains fixed. A possible extension which allows for variable input in the branch-net has been discussed in [9, 23], where they develop an enhanced architecture which allows for a varying number and placement of sensors. In this project, we have however chosen to generate the data such that the input functions are evaluated at a fixed set of sensors, as originally proposed in [2].

[9]: Prasthofer et al. (2022), *Variable-Input Deep Operator Networks*
[23]: Hoop et al. (2022), *The Cost-Accuracy Trade-Off In Operator Learning With Neural Networks*

Lastly, we note the "Modified DeepONet", introduced in [7]. The authors of [7] argue that their proposed architecture should help the inputs propagate throughout the network, and their modification empirically led to more accurate predictions in all examples discussed in [7]. In addition, another criticism of the DeepONet is that the input function and the query point is combined in a linear fashion, which leads to a linear approximation of the *operator* that we want to learn [10]. The Modified DeepONet circumvents this by creating encoders for both the branch and trunk nets, which are combined throughout the networks. The predictions are now written as

[7]: Wang et al. (2022), *Improved Architectures and Training Algorithms for Deep Operator Networks*

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

$$\begin{aligned}
B_E &= \sigma(W_a a + b_a), & T_E &= \sigma(W_y y + b_y),\\
B^{(1)} &= \sigma(W_a^{(1)} a + b_a^{(1)}), & T^{(1)} &= \sigma(W_y^{(1)} T + b_y^{(1)}),\\
B^{(l+1)} &= (1 - \sigma(W_a^{(l)} B^{(l)} + b_a^{(l)})) \odot B_E + \sigma(W_y^{(l)} T^{(l)} + b_y^{(l)}) \odot T_E\\
T^{(l+1)} &= (1 - \sigma(W_y^{(l)} T^{(l)} + b_y^{(l)})) \odot B_E + \sigma(W_a^{(l)} T^{(l)} + b_a^{(l)}) \odot T_E\\
B^{(L)} &= \sigma(W_a^{(L)} B^{(L-1)} + b_u^{(L)}), & T^{(L)} &= \sigma(W_y^{(L)} T^{(L-1)} + b_y^{(L-1)}),\\
\mathcal{S}_\theta[a](y) &= \langle B^{(L)}, T^{(L)} \rangle + b_0,
\end{aligned}$$

where B_E and T_E are the encoders for the branch and trunk nets, respectively, and σ is the activation function. The W and b are the weights and biases of the networks, and the $\odot$ denotes element-wise multiplication. $B^{(l)}$ and $T^{(l)}$ are the outputs of the branch and trunk nets at layer l, respectively. Finally, we also allow for a final bias term, b_0, to be added to the output.

This network architecture is illustrated in Figure 2.2.

Motivated by the results seen in [7], we have chosen to also implement and compare this modified architecture agains the vanilla DeepONet.

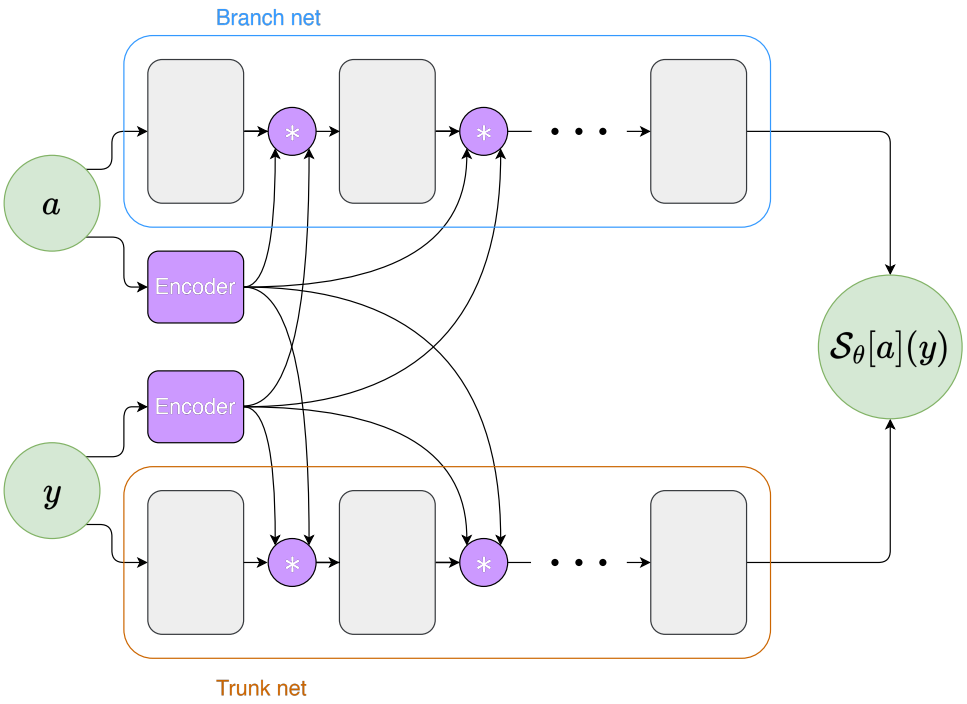

Figure 2.2: Modified DeepONet architecture, based on diagram in [7].

Neural Operators

A different and more recent approach to operator learning is discussed in [10]. While the (vanilla) DeepONets take discretized functions as inputs, and therefore technically map from a finite dimensional space to a function space, the Neural Operators are defined as mappings between function spaces. In this section, we will start by quickly reiterating the building blocks of a general Neural Operator, as they are defined in [10].

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

The network architechture can be seen as an extension of MLPs to function spaces. Instead of the traditional fully-connected layers, the networks employ "integral kernel operators", which similarly are meant to capture the global features through an integral over the domain:

Definition 2.0.1 (Integral Kernel Operators)

$$(\mathcal{K}v_l)(z) = \int_D \kappa^{(l)}(z, \bar{z}) v_l(\bar{z})\, dv_l(\bar{z}). \tag{2.4}$$

Here, v_l is the input function, $\kappa^{(l)}$ is the kernel function, and dv_l is the measure over the domain D. We write z to denote the inputs to the functions, as the input to the model can vary. Later, we will investigate how to handle the time dimension, and z may represent either only a spatial coordinate, $z = x$, or it may refer to points in the spatial-temporal domain $z = y = (x, t)$.

The paper also defines two other alternative kernel integral operator definitions, but we will use the basic form defined in Definition 2.4 with dv_l being the Lebesgue measure. In addition to the integral kernel operators, the article also defines three additional pointwise operators.

1. A lifting layer, $\mathcal{P}$, which lifts the input function's codimension: $\{a : D \to \mathbb{R}^{d_a}\} \mapsto \{v_0 : D \to \mathbb{R}^{d_{v_0}}\}$.
2. Linear layers, $\mathcal{W}_l$, which are the local transformations of each Neural Operator layer.
3. A projection layer, $\mathcal{Q}$, which maps back to the desired codimension: $\{v_L : D' \to \mathbb{R}^{d_{v_L}}\} \mapsto \{u : D' \to \mathbb{R}^{d_u}\}$.

It is also usual to incorporate bias terms in neural networks. In the context of neural operators, these are included as bias *functions*, $b_l(z)$.

This all comes together as

Definition 2.0.2 (Neural Operator $\mathcal{S}_\theta$)

$$\mathcal{S}_\theta = \mathcal{Q} \circ \sigma_L(\mathcal{W}_L + \mathcal{K}_L + b_L) \circ \cdots \circ \sigma_1(\mathcal{W}_1 + \mathcal{K}_1 + b_1) \circ \mathcal{P} \tag{2.5}$$

where σ_l denotes non-linear activation functions. A visualization of the Neural Operator architecture can be seen in Figure 2.3.

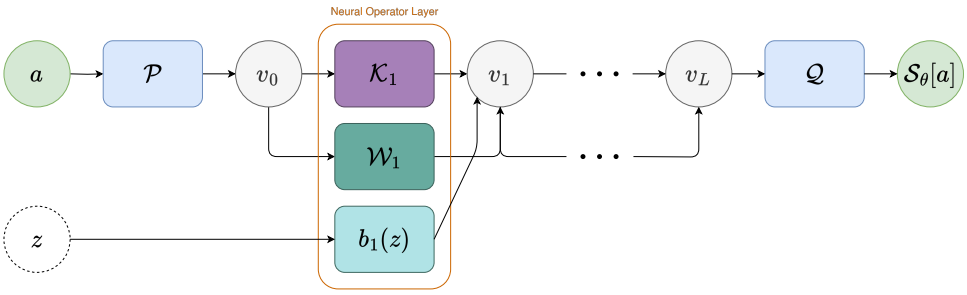

Figure 2.3: Neural Operator architecture.

The methods are then implemented by using discretized versions of the operators and functions. The local linear operators, $\mathcal{W}_l$, are usually implemented as either 1×1 convolutions or by the multiplication with a matrix $W_l \in \mathbb{R}^{d_{v_l} \times d_{v_{l-1}}}$ [3, 24]. In this project, we have chosen to do the latter.

The bias functions are usually implemented simply as a learnable vector, $b_l \in \mathbb{R}^{d_{v_l}}$ [24]. This means that the we have a constant function which does not depend on z. Other alternatives could be to parametrize it by a shallow neural network or with coefficients multiplied with a fixed basis.

Lastly, [10] define three different suggestions for discretized integral kernel operators: the "Graph Neural Operator" (GNO), the "Low-rank Neural Operator" (LNO) and the "Fourier Neural Operator" (FNO). Especially the FNO has garnered a lot of attention for its good results on PDEs and its efficiency by utilizing the fast Fourier transform (FFT). This is done by assuming a kernel of the form $\kappa(z, \bar{z}) = \kappa(z - \bar{z})$ and using the convolution theorem of the Fourier Transform [10]:

$$\mathcal{K}(v)(z) = \int_D \kappa(z - \bar{z}) v(\bar{z})\, d\bar{z} = \mathcal{F}^{-1}(\mathcal{F}(\kappa)\mathcal{F}(v))(z) = \mathcal{F}^{-1}(R\mathcal{F}(v))(z), \tag{2.6}$$

where $\mathcal{F}(\kappa)$ is parametrized directly by a matrix R of learnable weights. The Fourier transform and its inverse are then replaced by the FFT and the inverse fast Fourier

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*
[24]: Kossaifi et al. (2024), *A Library for Learning Neural Operators*
[24]: Kossaifi et al. (2024), *A Library for Learning Neural Operators*

transform (IFFT), leading to an efficient approximation of the integral kernel operator [10].

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

In this project, we have decided to limit our exploration of the Neural Operators to the FNO.

Appending the grid to the input

From the definition of the Neural Operators, we see that there is no need to give the grid that the input function is evaluated on as input to the network. This is however often done in practice by appending the grid to the input function, and usually gives better results [3, 10]. This essentially means that we now view our input function as a vector-valued function, which also outputs the grid points. If we for instance add the spatial coordinates to our input function $a : x \mapsto a(x)$, we can instead evaluate $v : x \mapsto (x, a(x))$. We do this for all of the FNO-based models considered.

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*
[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

Incorporating the time dimension in the FNO

As was also discussed in [3], the FNO requires D and D' to be from the same domain. Since we are trying to learn the mapping from an initial condition, at time $t = 0$, to the solution for $t \in [0, T]$, this has to be accounted for. In our problem setting, we have $D = \mathcal{X}$ and $D' = \mathcal{Y} = \mathcal{X} \times \mathcal{T}$. We present three possible solutions:

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*

Method 1: Timestepping. We learn an operator $\tilde{\mathcal{S}}_\theta : u(x, t) \mapsto u(x, t + \Delta t)$, where Δt is a fixed step size. We can then iteratively apply this operator to get predictions for discrete time points. In order to make this method continuous in time, we will have to interpolate between the discrete time points. We do this using the `Akima` splines, implemented in `interpax`. This method performs the FFT in space only, and will be referred to as "`FNOTimestepping`". This method was also mentioned in [3, 10].

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*
[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

Method 2: Extending the domain. We extend the domain of our initial condition to $\mathcal{Y}$, by viewing it as a function that is constant in time, $a(x, t) = a(x)$. The integral kernel operator will then act on the whole spatial-temporal domain, meaning that we need to perform the 2D FFT. This method will therefore be referred to as "`FNO2d`". The method was also discussed in [3].

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*

Method 3: Query points in time. We introduce here a third method that, as far as we know, has not been discussed in the literature before. In this method, we also view a as a function on $\mathcal{Y}$ that is constant in time, but we only learn the spatial mapping. The time coordinate then essentially acts as a "query point", telling us where in the temporal output domain we want the prediction. With this method, it is necessary to append the time coordinate to the input function, as discussed in Chapter 2, to allow the network to differentiate between different time points. The input function is then $v : x \mapsto (a(x), t)$, where t is the time point we want to evaluate the solution at. We can also add the spatial grid points to the input as before. This method will be referred to as "`FNO1d`".

A brief comparison between the different approaches

Both the DeepONet and the Neural Operators are shown to be universal approximators of operators. Specifically, the Neural Operators are shown to uniformly approximate any continuous operator defined on a compact set of Banach spaces in [10], and from before we have the universal approximation theorem for the DeepONet seen in Theorem 2.0.1. This property does not apply to the traditional neural networks [10], and is a big motivation for using DeepONets or Neural Operators for operator learning. Even though the DeepONet is a universal approximatior, it combines the branch and trunk nets in a linear fashion, leading to a linear approximation of the operator . The Neural Operators, on the other hand, are non-linear approximations to the operator, which in general should allow for a more expressive model [10].

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

The Neural Operators do however encompass a property that sets them apart from the DeepONets. They are *discretization-invariant*, a property proposed in [10]. According to [10], they are also the only known class of methods to encompass this property. A discretization-invariant model is defined by the following properties [10]:

Definition 2.0.3 (Discretization-invariant operator)
A discretization-invariant model satisfies the following properties:

1. *acts on any discretization of the input function, i.e. accepts any set of points in the input domain,*
2. *can be evaluated at any point of the output domain,*
3. *converges to a continuum operator as the discretization is refined.*

In comparison, the vanilla DeepONet fulfills the second property, but it does not fulfill the first or third. The Vanilla DeepONet cannot take the input function at any point, as the branch net limits the input to the fixed "sensors". As discussed previously, there have however been made some generalizations which circumvent this [9, 10, 23].

[9]: Prasthofer et al. (2022), *Variable-Input Deep Operator Networks*
[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*
[23]: Hoop et al. (2022), *The Cost-Accuracy Trade-Off In Operator Learning With Neural Networks*

The DeepONet does not fulfill the third property, as its predictions does not converge with a finer grid size. Since the network takes query points, rather than a grid, the model's prediction at the same query point will not change with the mesh size. This contrasts with the Neural Operators, which take values on a grid, and the predictions converge as the step size decreases [10].

Another big difference between the DeepONet and the Neural Operators, is in the inputs and outputs of the methods. As stated previously, the DeepONet takes a query point y from the output domain and predicts $\mathcal{S}_\theta[a](y) \approx u(y)\big|_a$. The inputs and outputs of the DeepONet (and the "Modified" variant) can then be written as

$$\text{DeepONet}: \underbrace{\mathbb{R}^N}_{\substack{\text{branch input,}\\ a(x) \text{ at } N \text{ sensors.}}} \times \underbrace{\mathbb{R}^{d+1}}_{\substack{\text{trunk input,}\\ \text{query point } y.}} \to \underbrace{\mathbb{R}}_{\substack{\text{prediction,}\\ u \text{ at } y}} . \tag{2.7}$$

The Neural Operators do not take query points, but rather directly compute the prediction at all given input points, $\mathcal{S}_\theta[a] \approx u$. The number of "input points" depends

on the method we use. Below we list the mappings for the different methods we have considered:

$$\text{FNO2d}: \underbrace{\mathbb{R}^{N_x \times N_t}}_{\substack{\text{initial condition,} \\ a(x,t) = a(x)}} \to \underbrace{\mathbb{R}^{N_x \times N_t}}_{\substack{\text{prediction,} \\ u(x,t)}} \tag{2.8}$$

$$\text{FNO1d}: \underbrace{\mathbb{R}^{N_x}}_{\substack{\text{initial condition,} \\ a(x)}} \times \underbrace{\mathbb{R}}_{\text{time point } t^*} \to \underbrace{\mathbb{R}^{N_x}}_{\substack{\text{prediction,} \\ u(x,t^*)}} \tag{2.9}$$

$$\text{FNOTimeStepping}: \underbrace{\mathbb{R}^{N_x}}_{\substack{\text{input function,} \\ u(x,t^*)}} \to \underbrace{\mathbb{R}^{N_x}}_{\substack{\text{prediction,} \\ u(x,t^*+\Delta t)}} \tag{2.10}$$

This difference in input will also affect both the loss function and how we can compute derivatives. For the DeepONets we are able to sample the query points randomly, so we can then compute the loss function by evaluating the network at these points only. This also applies to the FNO1d approach, where we can sample the time points randomly. The same holds for the derivatives, where we for the DeepONets can compute them directly at the query points only. The query points of the DeepONet therefore makes it more flexible in general. To evaluate on a grid, we simply vectorize over the grid points. We will discuss both the loss function used, and the computation of the derivatives, in more detail in the following sections.

Operator Learning for Hamiltonian PDEs

3

Hamiltonian PDEs

A Hamiltonian PDE can be written on the form

$$u_t = \mathcal{G}\frac{\delta \mathcal{H}}{\delta u} \tag{3.1}$$

where $\mathcal{G}$ is a Hamiltonian operator, $\mathcal{H} : \mathcal{U} \to \mathbb{R}$ is the Hamiltonian functional of the system, and $\frac{\delta \mathcal{H}}{\delta u}$ is the variational derivative [25, 26]. In order for $\mathcal{G}$ to be a Hamiltonian operator, it needs to be a constant linear operator, and it also has to satisfy two conditions on its Poisson bracket. These are the conditions of skew-symmetry and the Jacobi identity. We skip the details here, but refer the reader to [26]. We will in our project assume that $\mathcal{G} = -\partial_x$, which does indeed satisfy the conditions of a Hamiltonian operator. The Hamiltonian functional can furthermore be written as

$$\mathcal{H} = \int_{\mathcal{X}} F(x; u, u_x, u_{xx}, \ldots)dx,$$

where $F : \mathcal{U} \to \mathbb{R}$ is the *energy density function* [20, 26, 27].

We can then follow the formula in [20, 26, 27], which states that

$$\frac{\delta \mathcal{H}[\boldsymbol{u}]}{\delta u_m} = \frac{\partial F}{\partial u_m} - \sum_{d=1}^{D}\left[\frac{\partial}{\partial x_d}\frac{\partial F}{\partial u_{m,d}}\right] + \ldots \tag{3.2}$$

where u_m is the m-th component of $\boldsymbol{u}$, x_m is the m-th component of the spatial coordinate, and F is the energy density function of the Hamiltonian. $u_{m,d}$ refers to the derivative of u_m with respect to x_d.

[20]: Tanaka et al. (2024), *Neural Operators Meet Energy-based Theory: Operator Learning for Hamiltonian and Dissipative PDEs*
[26]: Olver (1986), *Applications of Lie Groups to Differential Equations*
[27]: Celledoni et al. (2012), *Preserving Energy Resp. Dissipation in Numerical PDEs Using the "Average Vector Field" Method*

The Energy-consistent Neural Operator

There are now two main approaches to satisfying the constraint of Equation 3.1 in our network: hard or soft constraints. The easiest of the two to employ, is usually the soft constraints, where we add a term to the loss function that we wish to minimize. This term then pushes the network towards satisfying the constraints, but does not necessarily guarantee that they are satisfied. Hard constraints, on the other hand, enforce the constraints directly, thereby ensuring that they are exactly satisfied [13].

In [20], the authors introduced the "Energy-consistent Neural Operator" (ENO). This network consists first of an MLP, which takes a stacked vector of a discretized initial condition $a(x)$ and a query point (x, t) as input. It then outputs an approximation to $\mathcal{S}[a](x, t) = u(x, t)$. They then compute derivatives u_t and $u_x, u_{xx}, \ldots$ by automatic differentiation. These values are then the inputs to another neural network, F_ϕ, which parametrizes the energy density function F. They then proceed by computing the

[13]: Karniadakis et al. (2021), *Physics-Informed Machine Learning*
[20]: Tanaka et al. (2024), *Neural Operators Meet Energy-based Theory: Operator Learning for Hamiltonian and Dissipative PDEs*

right-hand side of Equation 3.1, again through automatic differentiation, by using the variational derivative of the Hamiltonian, as seen in Equation 3.2.

The ENO is "informed" about the Hamiltonian structure of the PDE, in the sense that $\dot{u}_m^\theta$ and $\mathcal{G}\frac{\delta \mathcal{H}_\phi[\boldsymbol{u}^\theta]}{\delta u_m^\theta}$ are pushed to equality by adding a penalty term to the loss function. This is a "soft constraint", as the network is not guaranteed to satisfy Equation 3.1 exactly. This follows the same general strategy as we see in the well-known physics-informed neural networks (PINNs) [12].

[12]: Raissi et al. (2019), *Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations*

It should also be noted that the ENO, as it was introduced in [20], does not use the DeepONet or a Neural Operator in its approach, but rather an MLP. They do mention the DeepONet and FNO, but as we shall see, the inclusion of the FNO in the architectur is in particular not straightforward. The MLP is not a universal approximator of operators, and is not discretization-inveriant. We therefore wish to investigate how we can implement the architecture with the use of the DeepONets and the FNOs, and compare how well these perform, in addition to investigating the use of a hard constraint.

The Hamiltonian Operator Network

We now propose an extension to the ENO, which uses the DeepONet or the FNO as the operator network, in addition to a hard-constrained approach. Rather than using the predictions from the operator network, we could instead impose $\dot{u}_m^\theta = \mathcal{G}\frac{\delta \mathcal{H}_\phi[\boldsymbol{u}^\theta]}{\delta u_m^\theta}$ directly. We then have a network which predicts temporal derivatives that are guaranteed to preserve the approximated Hamiltonian functional, $\mathcal{H}_\phi = \int_{\mathcal{X}} F_\phi(u^\theta, u_x^\theta, \ldots)dx$. To obtain the final predictions for u, we can then integrate $\dot{u}_m^\theta$ in time:

$$u_m^\theta(x,t) = a(x) + \int_0^t \mathcal{G}\frac{\delta \mathcal{H}_\phi[\boldsymbol{u}^\theta]}{\delta u_m^\theta} d\bar{t}. \tag{3.3}$$

This approach follows a similar structure as the one proposed in [16] for ODEs, but in this case for PDEs used with the Operator Learning framework.

As we cannot find the exact antiderivative to $\mathcal{G}\frac{\delta \mathcal{H}_\phi[\boldsymbol{u}^\theta]}{\delta u_m^\theta}$, we will have to approximate it by a numerical method. To this end, we here list three possible methods to approximate the antiderivative:

1. **Cumulative integration:** This is the simplest method where we first predict values for the integrand on grid points. We then cumulatively integrate, using for instance Simpson's rule or the trapezoid rule. This can easily be done with `cumulative_trapezoid` or `cumulative_simpson` functions, found in libraries such as `scipy` or `quadax`. This approach does however introduce discretization errors which accumulate over time, and is therefore not the most suitable approach.

2. **Exact antiderivative of interpolated function:** With this method, we first predict the integrand on grid points, and then interpolate the function. We can then find the exact antiderivative of the interpolated function, and evaluate it at the desired time points. This is easily done by leveraging functions such as `interp1d` in `scipy` or the `interpax` library. If we use sufficiently many points, and use a suitable interpolation method, this should give better results than the cumulative integration as we avoid errors that accumulate over time.
3. **ODE integration:** A final approach is to numerically integrate by solving the initial value problem

$$f(u_m^\theta, t) = \mathcal{G}\frac{\delta \mathcal{H}_\phi[\boldsymbol{u}^{\boldsymbol{\theta}}]}{\delta u_m^\theta}, \qquad u_m^\theta(x, 0) = a(x).$$

   We should also solve this using a symplectic integrator, which will preserve a discrete version of the *learned* Hamiltonian. Note that we have a directly integrable ODE, as we know the right-hand sides direct dependence on t, and we therefore do not need to solve any implicit equations. This method should be the preferred one.

3.1 Computing the functional derivative

In order to compute our model's prediction for the right-hand side of Equation 3.1, we need to compute the functional derivative of the Hamiltonian.

In this project, we will only consider scalar-valued functions in one spatial dimension. In addition, we assume that our energy density function F only depends on u and u_x. The formula then simplifies to

$$\frac{\delta \mathcal{H}[u]}{\delta u} = \frac{\partial F}{\partial u} - \frac{\partial}{\partial x}\frac{\partial F}{\partial u_x}. \tag{3.4}$$

We will further assume that the operator $\mathcal{G}$ is $-\partial_x$. The right-hand side of Equation 3.1 can then be written as

$$\mathcal{G}\frac{\delta \mathcal{H}[u]}{\delta u} = -\frac{\partial}{\partial x}\frac{\partial F}{\partial u} + \frac{\partial^2}{\partial x^2}\frac{\partial F}{\partial u_x}. \tag{3.5}$$

As F depends on x through u and u_x, we can either consider total derivatives and compute them directly, or we can use the chain rule.

For the approach using the chain rule, we note that the chain rule for multivariable functions gives

$$\begin{aligned}\frac{\partial}{\partial x}\frac{\partial F}{\partial u} &= \frac{\partial^2 F}{\partial u^2}u_x + \frac{\partial^2 F}{\partial u \partial u_x}u_{xx} \\ \frac{\partial}{\partial x}\frac{\partial F}{\partial u_x} &= \frac{\partial^2 F}{\partial u_x \partial u}u_x + \frac{\partial^2 F}{\partial u_x^2}u_{xx}.\end{aligned} \tag{3.6}$$

We now differentiate Equation 3.6 with respect to x once again, which after a thourough calculation gives

$$\frac{\partial^2}{\partial x^2}\frac{\partial F}{\partial u_x} = \frac{\partial^2 F}{\partial u \partial u_x}u_{xx} + \frac{\partial^2 F}{\partial u_x^2}u_{xxx} + \frac{\partial^3 F}{\partial u_x u^2}u_x^2 + 2\frac{\partial^3 F}{\partial u_x^2 \partial u}u_x u_{xx} + \frac{\partial^3 F}{\partial u_x^3}u_{xx}^2. \tag{3.7}$$

The full expression for the right-hand side of Equation 3.1 using the multivariable chain rule is then

$$\begin{aligned}\mathcal{G}\frac{\delta \mathcal{H}[u]}{\delta u} = &-\left(\frac{\partial^2 F}{\partial u^2}u_x + \frac{\partial^2 F}{\partial u \partial u_x}u_{xx}\right) \\ &+ \frac{\partial^2 F}{\partial u \partial u_x}u_{xx} + \frac{\partial^2 F}{\partial u_x^2}u_{xxx} + \frac{\partial^3 F}{\partial u_x u^2}u_x^2 + 2\frac{\partial^3 F}{\partial u_x^2 \partial u}u_x u_{xx} + \frac{\partial^3 F}{\partial u_x^3}u_{xx}^2.\end{aligned} \tag{3.8}$$

3.2 Computing spatial and temporal derivatives

How should one go about computing spatial and temporal derivatives in the context of DeepONets and Neural Operators? As we want the outputs of our network to satisfy Equation 3.1, we will need to compute derivatives of the output $u^\theta = \mathcal{S}_\theta[a]$ with respect to x and t.

One strategy is to simply use traditional numerical methods, such as finite differences or FFT-based differentiation. However, this introduces discretization errors, as well as requiring sufficiently small step sizes and equidistant grids. A more elegant and direct approach might be to utilize automatic differentiation, which can compute derivatives exactly down to floating point precision.

For the DeepONets, which take query points as input, this is possible, and we simply compute the derivatives as e.g.

$$u_t^\theta = \frac{\partial \mathcal{S}_\theta}{\partial t}[a](x,t). \tag{3.9}$$

However, the situation for the Neural Operators is quite different. As discussed previously, these networks do not take query points, but are defined to map between function spaces directly. Luckily, two possible solutions have been discussed in [14], in addition to the use of traditional numerical differentiation methods.

[14]: Li et al. (2023), *Physics-Informed Neural Operator for Learning Partial Differential Equations*

The solution proposed in [14] is to use the chain rule, but only for the very last layers in the neural operator. This is not only efficient, but if we were to backpropagate through the whole network, we would in the end reach the input layer, and we do not in general know the derivative of the initial condition.

We will now illustarte how this works for the spatial derivatives by ignoring the model's dependence on t. We start by writing our resulting function $u(x)$ as it depends on the output of the previous layer, $v_L(x)$:

$$\begin{aligned} u(x) &= Q(\sigma(v_L(x))), \\ v_L(x) &= \mathcal{K}_L(v_{L-1}(x)) + \mathcal{W}_L(v_{L-1}(x)) + b_L(x). \end{aligned} \tag{3.10}$$

We now use the chain rule *once*, and observe the following:

$$\frac{du}{dx} = \frac{dQ}{dv_L}\left(\frac{d\mathcal{K}_L(v_{L-1})}{dx} + \frac{d\mathcal{W}_L(v_{L-1})}{dx} + \frac{db_L}{dx}\right). \tag{3.11}$$

From the definition of the integral kernel operator, Definition 2.4, we see that we can compute the exact derivative of $\mathcal{K}_L(v_{L-1}(x))$ with respect to x directly, without iterating further through the network. Following the definition of the kernel operator, we have in general that

$$\frac{d\mathcal{K}(v)}{dx} = \int_D \frac{\partial \kappa(x,y)}{\partial x} v(y)\, dv(y). \tag{3.12}$$

Since we use the Fourier Neural Operator, the derivatives are given by

$$\frac{d\mathcal{K}(v)}{dx} = \frac{1}{k_{max}} \sum_{k=0}^{k_{max}} (R_k(\mathcal{F}v_{L-1})_k) \frac{d}{dx} \exp \frac{i2\pi k}{D}(x) \tag{3.13}$$

and from $\frac{d}{dx} \exp \frac{i2\pi k}{D}(x) = \frac{i2\pi}{D} \exp \frac{i2\pi k}{D}(x)$, we can simply compute the derivatives as

$$\frac{d\mathcal{K}(v)}{dx} = \mathcal{F}^{-1}\left(\frac{i2\pi}{D} R\mathcal{F}(v)\right), \tag{3.14}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the FFT and IFFT, respectively [14].

[14]: Li et al. (2023), *Physics-Informed Neural Operator for Learning Partial Differential Equations*

As we do not in general know directly how the bypass-layer, $\mathcal{W}_L(v_L(x))$, depends on x, we simply remove this term, as suggested in [14].

Lastly, the bias function is easily differentiated, as it only depends on x. Lastly, $\frac{dQ}{dv_L}$ can be computed by automatic differentiation. Furthermore, since $\mathcal{Q}$ is a pointwise operator, we can do this efficiently by computing the gradient of $\sum_j Q(v_L)_j$ with respect to v_L, rather than computing the full Jacobian. This is a simple "trick" that we have utilized in multple parts of the implementation.

We can now either compute the derivatives at query points, which is the first method suggested in [14], or we can use the second method, which is to simply alter the forward

pass of the network slighly, to account for the $\frac{i2\pi}{D}$-term that arises in the differentiation. In this project, we have chosen to do the latter, as it is more efficient.

Note that for the spatial derivatives of the `FNOTimeStepping` method, we will again have to interpolate in time to get derivatives that are continuous in time. We have again chosen to do this using `Akima` splines.

Computing the Hamiltonian PDE right-hand side

We now wish to compute Equation 3.4 for the networks. For the DeepONets, we can use automatic differentiation directly to compute the total derivatives of the terms in the energy density function. The implementation in `JAX` then follows the formula very closely:

```
def __call__(self, a, x, t):
    """
    Input:
        x: scalar
        t: scalar
    Output:
        GdH(x,t) (= u_t(x,t)): scalar, at x=x and t=t
    """

    u = lambda x : self.u.decode_u(self.u(a, x, t))
    u_x = lambda x : grad(self.u, 1)(a, x, t)*self.u.u_std/self.u.x_std

    dFdu = grad(self.F)
    dFdu_x = lambda x : grad(self.F, 1)(u(x), u_x(x))

    dH = lambda x : dFdu(u(x), u_x(x)) - grad(dFdu_x)(x)/self.u.x_std
    GdH =  -grad(dH)(x)/self.u.x_std

    return GdH
```

Listing 3.1: Computing the Hamiltonian PDE right-hand side for the DeepONets

Note that we have included the scalings of the equations, as we have discussed in Section 4.1.

For the Neural Operators, we cannot compute the derivatives through automatic differentiation directly, as shown by the discussion in the previous section. Instead we can compute the formula according to Equation 3.8, and use the method suggested earlier to compute the spatial derivatives of u^θ. We have here chosen to highlight the implementation in `JAX` for the `FNO1d` model. The implementation for the other FNO-based models are however very similar, but vary slightly in their inputs and outputs. For details, refer to the code in the repository linked in the appendix.

```
def __call__(self, a, x, t):
    """
    Input:
        x: (Nx,) array
```

Listing 3.2: Computing the Hamiltonian PDE right-hand side for the FNO1d model

```
        t: scalar
    Output:
        GdH(x,t) (= u_t(x,t)): (Nx,) array at t=t
    """

    u, u_x, u_xx, u_xxx = ... # Compute as discussed previously

    # Notation: write u=y and u_x=z.
    # dF/du is then F_y, dF/du_x is F_z, etc.

    F_y = grad(F)
    F_yz_val = vmap(grad(F_y, 1))(u, u_x)
    F_yy_val, F_yyz_val = vmap(value_and_grad(grad(F_y), 1))(u, u_x)
    F_zz_val, (F_yzz_val, F_zzz_val) = vmap(value_and_grad(grad(grad(F, 1), 1)
    , (0,1)))(u, u_x)

    F_yx = F_yy_val * u_x  + F_yz_val * u_xx
    F_zxx = F_yz_val*u_xx + F_zz_val*u_xxx + F_yyz_val*u_x**2 + 2*F_yzz_val*
    u_x*u_xx+F_zzz_val * u_xx**2

    GdH = - F_yx + F_zxx
    return GdH.reshape((len(x)))
```

Method 4

4.1 Data generation

The data used to train the different networks has been generated, as we are trying to compare the models in a systematic way on familiar problems. The data structure is consistent across all different problems and models, and consists of input-output function pairs $\{a_i, u_i\}_{i=1}^N$.

We generated $N = 1000$ samples, divided into:

- **Training set:** 800 samples
- **Validation set:** 100 samples
- **Test set:** 100 samples

The data was generated on equispaced grids:

$$\begin{aligned} x_m &= m\Delta x \in [0, 20), & m &= 0, \dots, 256 - 1 \\ t_n &= n\Delta t \in [0, 3), & n &= 0, \dots, 384 - 1, \end{aligned}$$

i.e. with step sizes $\Delta x = \frac{20}{256} \approx 0.078$ and $\Delta t = \frac{3}{384} \approx 0.0078$. For the advection problem, we may use an analytical solution, but for the Korteweg–De Vries equation, we need to solve the equation numerically. The step sizes are therefore chosen to be small enough to avoid numerical instability as well, to make sure that our traning data follows the governing PDE.

The number of grid points was meticulously chosen such that we can downsample to a 64×64 grid, which we will be used as training and validation data:

$$\begin{aligned} x_{\hat{m}} &= \hat{m} \cdot 4\Delta x \in [0, 20), & \hat{m} &= 0, \dots, 64 - 1 \\ t_{\hat{n}} &= \hat{n} \cdot 4\Delta t \in [0, 2), & \hat{n} &= 0, \dots, 64 - 1, \end{aligned}$$

In this way, we can explore how the different models are able to interpolate when encountering data on a finer grid, and extrapolate to time points further into the future. The 64×64-grid was chosen specifically, as 64 is a power of 2, which better utilizes the Fast Fourier Transform seen in the Fourier Neural Operators.

To test for extrapolation, we will use the prediction at $t \approx 2$ as the initial condition for the next time steps. This is in particular crucial for the `FNO2d`, which assumes a periodic domain, and simply giving a larger time domain will then lead to wrong predictions for all time steps. We also tried simply using larger values of t to extrapolate for the other methods, but this gave very poor results for all architectures.

Preprocessing and weight initialization

Normalizing the input and output data has been extensively used in deep learning, as it may aid in faster convergence [28]. However, as [7] points out, it is somewhat unclear how to best preprocess the data when working with operator networks, and if this should differ from the regular normalization techniques, as it has not been extensively studied yet.

In [29], the authors show how preprocessing, specifically Z-score normalization, can improve the performance of PINNs. With that in mind, we also choose to normalize the data to have zero mean and unit variance. We then have the new inputs and outputs given by

$$\hat{u} = \frac{u - \mu_u}{\sigma_u}, \quad \hat{x} = \frac{x - \mu_x}{\sigma_x}, \quad \hat{t} = \frac{t - \mu_t}{\sigma_t}, \tag{4.1}$$

and the normalized initial condition is given by $\hat{a} = \hat{u}|_{t=0}$.

As noted in [29], it is now important to also scale the equations we want to impose accordingly. For instance, we will have to use $u_t = \hat{u}_{\hat{t}} \frac{\sigma_u}{\sigma_t}, u_x = \hat{u}_{\hat{x}} \frac{\sigma_u}{\sigma_x}$, which follows from the chain rule. For the energy density net, we will however use the input in its original scale. The reason for this is that the spatial derivatives in particular become very large, due to the small scale of the spatial domain. This leads to large predictions, far away from the actual energy density, which is not ideal.

When we compute metrics on the test data and show predictions, we will however use the actual grids and decode the predictions back to their original scale. A further exploration of different ways of preprocessing the data could give new insights and uncover improved techniques for training neural networks on function spaces, but has not been explored further in this project.

Weight initialization has also not been extensively studied in the context of operator learning. For traditional neural networks, it is common to initialize the weights with either "He" or "Xavier" initializations [30, 31]. In this project, we have used the gelu activation function, and have therefore chosen to use the "He" initialization, as it is specifically designed for ReLU-like activation functions.

[28]: Huang et al. (2020), *Normalization Techniques in Training DNNs: Methodology, Analysis and Application*

[7]: Wang et al. (2022), *Improved Architectures and Training Algorithms for Deep Operator Networks*

[29]: Xu et al. (2024), *On the Preprocessing of Physics-informed Neural Networks: How to Better Utilize Data in Fluid Mechanics*

[30]: Glorot et al. (n.d.), *Understanding the Difficulty of Training Deep Feedforward Neural Networks*

[31]: He et al. (2015), *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*

4.2 Test problems

We have chosen to work with two different test problems: the advection equation and the Korteweg–De Vries equation. The prior is a simple linear PDE, and acts as a good starting point for testing the models, while the latter is a nonlinear PDE, and is therefore a more challenging and more realistic problem.

Advection equation

The governing equation for the advection problem is given by

$$u_t + cu_x = 0. \tag{4.2}$$

That is, a translation of the initial condition at speed c. Here, we have chosen to use a one-soliton as the initial condition and periodic boundary, with analytical solution given by

$$u(x,t) = 2\,c\,\operatorname{sech}^2\left(\left(x - ct + \frac{P}{2} + d\,P\right) \mod P - \frac{P}{2}\right) \tag{4.3}$$

where $c \sim \mathcal{U}(0.5, 1.5)$ and $d \sim \mathcal{U}(0,1)$ are random variables, and $P = 20$ is the periodicity of the domain.

To relate this to the form of Equation 3.1, we note that the energy is given by the Hamiltonian

$$\mathcal{H}[u] = \int_{\mathbb{R}} \frac{c}{2} u^2 \, dx. \tag{4.4}$$

with variational derivative

$$\frac{\delta \mathcal{H}[u]}{\delta u} = cu, \tag{4.5}$$

and we have $\mathcal{G} = -\partial_x$.

Korteweg–De Vries equation

The Korteweg–De Vries equation is given by

$$u_t + \eta u u_x + \gamma^2 u_{xxx} = 0 \qquad x \in [0, 20], \quad t > 0 \tag{4.6}$$

where we have set $\eta = 6$ and $\gamma = 1$. We have also truncated the domain to $[0, 20]$, and use periodic boundary conditions.

Again, we reformulate Equation 4.6 to the form of Equation 3.1. Here, we have $\mathcal{G} = -\partial_x$, and the energy is given by

$$\mathcal{H}[u] = \int_{\mathbb{R}} \left(\frac{\eta}{6} u^3 - \frac{\gamma^2}{2} u_x^2 \right) dx \tag{4.7}$$

with varitional derivative

$$\frac{\delta \mathcal{H}[u]}{\delta u} = \frac{\eta}{2} u^2 - \gamma^2 u_{xx}. \tag{4.8}$$

This gives us the Hamiltonian formulation

$$u_t = -\left(\frac{\eta}{2}u^2 + \gamma^2 u_{xx}\right)_x. \tag{4.9}$$

For the initial condition, we have used a two-soliton, given by

$$a(x) = 2\sum_{l=1}^{2} c_l^2 \operatorname{sech}^2\left(c_l\left(\left(x + \frac{P}{2} - d_l P\right) \mod P - \frac{P}{2}\right)\right), \tag{4.10}$$

where $c_1, c_2 \sim \mathcal{U}(0.5, 1.5)$ and $d_1, d_2 \sim \mathcal{U}(0, 1)$ are random variables, and $P = 20$ is the period.

With a two-soliton as the initial condition, it is however no longer possible to find an analytic solution to the KdV equation, and we therefore need to solve it numerically. As stated in [25], we need to ensure that we use a skew-symmetric approximation of the $\mathcal{G}$-operator in Equation 3.1. Since we have $\mathcal{G} = -\partial_x$, we can approximate it by central differences. In time, we should use a symplectic integrator, so that we can conserve an approximation of the energy of the system. This is important, as we will later be training models which will attempt to conserve the energy of the system.

[25]: Leimkuhler et al. (2005), *Simulating Hamiltonian Dynamics*

To avoid inaccurate solutions, we use a fine grid and methods of high order. We use a sixth-order Gauss-Legendre method for time integration and sixth-order central differences for spatial discretization, to ensure that the data is accurate to the system we want to learn. During each implicit time step, the Newton method is employed with settings `atol` $= 1 \times 10^{-12}$, `rtol` $= 1 \times 10^{-12}$, and `maxiter` $= 10$.

4.3 Loss function

The choice of loss function plays an important role in training any neural network. Since we are working with mappings between function spaces, we should define a loss function that is suitable for this task. Inspired by [10], we want to minimize the $L^2_\mu(\mathcal{A}, \mathcal{U})$ Bochner norm of the difference between the true operator $\mathcal{S}$ and the learned operator $\mathcal{S}_\theta$, that is

[10]: Kovachki et al. (2024), *Neural Operator: Learning Maps Between Function Spaces*

$$\|\mathcal{S} - \mathcal{S}_\theta\|^2_{L^2_\mu(\mathcal{A},\mathcal{U})} = \mathbb{E}_{a\sim\mu}\|\mathcal{S}(a) - \mathcal{S}_\theta(a)\|^2_{L^2(\mathcal{Y})}. \tag{4.11}$$

It has been observed in [7, 32] that DeepONets might be biased towards learning functions with larger magnitudes, and they therefore argue that a relative loss function might be better suited. In [10], they also use a relative loss function due to its "good normalization and regularization effect". We therefore investigate the relative $L^2_\mu(\mathcal{A}, \mathcal{U})$ error, as stated in [10]:

[7]: Wang et al. (2022), *Improved Architectures and Training Algorithms for Deep Operator Networks*
[32]: Leoni et al. (2021), *DeepONet Prediction of Linear Instability Waves in High-Speed Boundary Layers*

$$\mathbb{E}_{a\sim\mu}\frac{\|\mathcal{S}[a] - \mathcal{S}_\theta[a]\|_{L^2(\mathcal{Y})}}{\|\mathcal{S}[a]\|_{L^2(\mathcal{Y})}}. \tag{4.12}$$

Proposition 4.3.1

The expected relative L_2 error can be approximated by the relative ℓ_2 error of a mini-batch:

$$\mathbb{E}\frac{1}{N}\sum_{i=1}^{N}\frac{\|\mathcal{S}[\boldsymbol{a_i}]-\mathcal{S}_{\boldsymbol{\theta}}[\boldsymbol{a_i}]\|_2}{\|\mathcal{S}[\boldsymbol{a_i}]\|_2}=\mathbb{E}_{a\sim\mu}\frac{\|\mathcal{S}[a]-\mathcal{S}_\theta[a]\|_{L^2(\mathcal{Y})}}{\|\mathcal{S}[a]\|_{L^2(\mathcal{Y})}}. \tag{4.13}$$

where N is the batch size. $\mathcal{S}[\boldsymbol{a_i}]=[\mathcal{S}[a_i](y_1),\dots,\mathcal{S}[a_i](y_P)]^\top$ is a vector of the true values at P points; either over the whole grid, or P uniformly sampled query points in the case of the DeepONet. The same also holds for $\mathcal{S}_{\boldsymbol{\theta}}[\boldsymbol{a_i}]=[\mathcal{S}_\theta[a_i](y_1),\dots,\mathcal{S}_\theta[a_i](y_P)]^\top$, but for the predicted values.

Proof. To approximate the expectation value, we use a standard Monte Carlo estimator, i.e. the sample mean of a mini-batch:

$$\mathbb{E}_{MC}\frac{\|\mathcal{S}[a]-\mathcal{S}_\theta[a]\|_{L^2(\mathcal{Y})}}{\|\mathcal{S}[a]\|_{L^2(\mathcal{Y})}}=\frac{1}{N}\sum_{i=1}^{N}\frac{\|\mathcal{S}[a_i]-\mathcal{S}_\theta[a_i]\|_{L^2(\mathcal{Y})}}{\|\mathcal{S}[a_i]\|_{L^2(\mathcal{Y})}},\qquad a_i\overset{\text{i.i.d.}}{\sim}\mu. \tag{4.14}$$

To approximate the L_2 norms, we may use a standard Monte Carlo integral:

$$\|f(y)\|^2_{L^2(\mathcal{Y})}=\int_{\mathcal{Y}}f(y)^2dy\approx\mathrm{Vol}(\mathcal{Y})\frac{1}{P}\sum_{j=1}^{P}f(y_j)^2 \tag{4.15}$$

Equation 4.12 can then be approximated by

$$\frac{1}{N}\sum_{i=1}^{N}\frac{\sqrt{\frac{1}{P}\sum_{j=1}^{P}(\mathcal{S}[a_i](y_j)-\mathcal{S}_\theta[a_i](y_j))^2}}{\sqrt{\frac{1}{P}\sum_{j=1}^{P}(\mathcal{S}[a_i](y_j))^2}}=\frac{1}{N}\sum_{i=1}^{N}\frac{\|\mathcal{S}[\boldsymbol{a_i}]-\mathcal{S}_{\boldsymbol{\theta}}[\boldsymbol{a_i}]\|_2}{\|\mathcal{S}[\boldsymbol{a_i}]\|_2} \tag{4.16}$$

which is a consistent estimator by the law of large numbers. □

For our operator loss, we will therefore use

$$\mathcal{L}_{\text{operator}}(\theta)=\frac{1}{N}\sum_{i=1}^{N}\frac{\|\mathcal{S}[\boldsymbol{a_i}]-\mathcal{S}_{\boldsymbol{\theta}}[\boldsymbol{a_i}]\|_2}{\|\mathcal{S}[\boldsymbol{a_i}]\|_2}. \tag{4.17}$$

To evaluate the validation/test error, we will use the same approximation to 4.12 as above. In the case of the DeepONets, where we randomly sample query points during the training, we will however use all of the grid points during inference. This is to ensure that we get a fair comparison between the different models, and an accurate estimate of the validation/test error.

We also found that some of the models did not predict good results for $t=0$, which should be the identity operator. Therefore, we added a penalty term to penalize violations of the initial condition, given by

$$\mathcal{L}_{\text{init}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\frac{\|\boldsymbol{a_i} - \mathcal{S}_{\boldsymbol{\theta}}[\boldsymbol{a_i}]|_{t=0}\|_2}{\|\boldsymbol{a_i}\|_2}. \tag{4.18}$$

For the vanilla operator networks, we have then used the following loss function:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{operator}}(\theta) + \lambda_{\text{init}}\mathcal{L}_{\text{init}}(\theta) \tag{4.19}$$

where we have set the penalty weight $\lambda_{\text{init}} = 0.5$.

When it comes to the Hamiltonian Operator Networks, we have added an additional energy penalty term to the loss function, as suggested in [20]. The energy penalty term is given by

[20]: Tanaka et al. (2024), *Neural Operators Meet Energy-based Theory: Operator Learning for Hamiltonian and Dissipative PDEs*

$$\mathcal{L}_{\text{energy}}(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left\|\dot{\boldsymbol{u}}_i^{\theta} - \mathcal{G}\frac{\delta\mathcal{H}_{\boldsymbol{\phi}}[\boldsymbol{u}_i^{\theta}]}{\delta\boldsymbol{u}_i^{\theta}}\right\|_2. \tag{4.20}$$

Since we do not have access to the true temporal derivative, we have not used a relative error for this term. We add this to the operator loss function, to get the final loss function for the Hamiltonian Operator Networks:

$$\mathcal{L}(\theta,\phi) = \mathcal{L}_{\text{operator}}(\theta) + \lambda_{\text{init}}\mathcal{L}_{\text{init}}(\theta) + \lambda_{\text{energy}}\mathcal{L}_{\text{energy}}(\phi) \tag{4.21}$$

where we have multiplied with the penalty weight $\lambda_{\text{energy}} = 1e-3$.

4.4 Self-adaptive weights

In addition to the weights of the neural network, which we denote by θ, we also supply our operator networks with the possibility for learning self-adaptive weights, denoted by λ. The self-adaptive weights were first introduced in [33], as a way to improve the learning of PINNs by adaptively weighting the different terms and points in the loss function. Instead of directly minimizing the loss function, we first maximize it with respect to the self-adaptive weigths. The idea behind this, is that it should force the network to focus on the most challenging terms and points in the domain to learn [33].

[33]: McClenny et al. (2023), *Self-Adaptive Physics-Informed Neural Networks Using a Soft Attention Mechanism*

For our problem setting, it makes sense to adaptively weight the different time points, as it is likely easier to learn the solution operator to earlier time points. Since we for all methods are working with training sets with 64 time points, we can define the self-adaptive weights as $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_{64}]$, where λ_i is the weight for the i-th time point. Our relative loss function is then modified to the following:

$$\mathcal{L}_{\text{operator}}(\theta,\lambda) = \frac{1}{N}\sum_{i=1}^{N}\frac{\sqrt{\sum_{j=1}^{P} g(\lambda_{j*})(\mathcal{S}[a_i](y_j) - \mathcal{S}_\theta[a_i](y_j))^2}}{\|\mathcal{S}[\boldsymbol{a_i}]\|_2}, \tag{4.22}$$

and in each update we compute the gradients according to

$$\min_{\theta} \max_{\lambda} \mathcal{L}(\theta, \lambda). \tag{4.23}$$

$g(\lambda)$ is a "masking function", and λ_{j*} corresponds to the temporal coordinate of the point y_j. It is then multiplied with the squared difference between the true and predicted function values. We note that when $g(\lambda) = 1$, the loss function is equivalent to the un-weighted loss presented before, Equation 4.17. These weights are furthermore only used during training, and discarded during inference.

As explained in [33], we want the masking function defined on $[0, \infty)$ to be non-negative, differentiable on $(0, \infty)$, and strictly increasing. We can compute the gradient of the loss function with respect to the self-adaptive weights as follows:

[33]: McClenny et al. (2023), *Self-Adaptive Physics-Informed Neural Networks Using a Soft Attention Mechanism*

$$(\nabla_\lambda \mathcal{L}_{\text{operator}})_{j*} = \frac{1}{2N} \sum_{i=1}^{N} \frac{1}{\|\mathcal{S}[\boldsymbol{a_i}]\|_2} \frac{\sum_{j=1}^{P} g'(\lambda_{j*}) \left(\mathcal{S}[a_i](y_j) - \mathcal{S}_\theta[a_i](y_j)\right)^2}{\sqrt{\sum_{j=1}^{P} g(\lambda_{j*}) \left(\mathcal{S}[a_i](y_j) - \mathcal{S}_\theta[a_i](y_j)\right)^2}}. \tag{4.24}$$

We then know that $(\nabla_\lambda \mathcal{L})_{j*} \geq 0$, with equality if and only if the (original) un-weighted loss is zero, as we have $g'(\lambda) > 0$. The updates should then also be larger where the un-weighted loss is larger, which is the desired behavior.

As candidates for the masking function, we have considered the three following functions:

Exponential Mask

$$g(\lambda) = e^{a(\lambda - 1)}$$

Polynomial Mask

$$g(\lambda) = \begin{cases} \lambda^a, & \lambda \geq 1, \\ e^{a(\lambda-1)}, & \lambda < 1 \end{cases}$$

Logistic Mask

$$g(\lambda) = \frac{2}{1 + e^{a(1-\lambda)}}.$$

Here, the parameter $a \geq 0$ decides the "sharpness" of the function. This is either chosen to be a hyperparameter that we choose before training, or we can let it be a learnable parameter. For the fixed masking functions, we have chosen $a \in \{1, 3\}$ for the exponential and polynomial masks, and $a \in \{5, 50\}$ for the logistic mask. The fixed masks are illustrated in Figure 4.1.

Of these, versions similar to our logistic and polynomial masks were mentioned in [33], but we also allow for learnable masks. We will then let `optuna` decide between these six fixed masks or the learnable versions in the hyperparameter optimization step.

[33]: McClenny et al. (2023), *Self-Adaptive Physics-Informed Neural Networks Using a Soft Attention Mechanism*

In [33] it is also mentioned that we should bound the weights. Otherwise, we run the risk of increasing the loss drastically for masks unbounded from above, or we may encounter vanishing gradients in the case of logistic masks. We therefore propose centering the self-adaptive weights around 1, by updating the weights according to

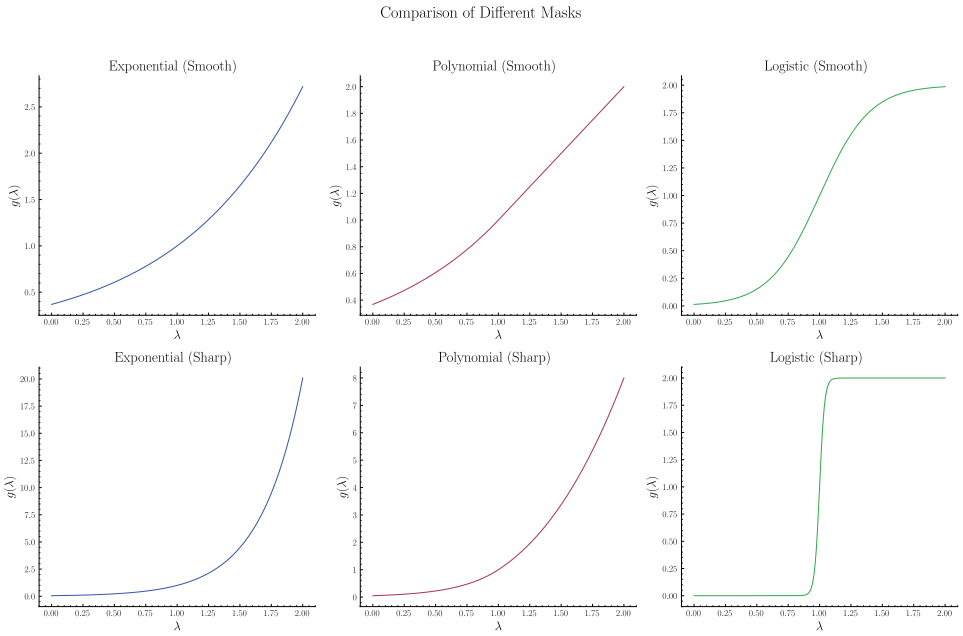

Figure 4.1: Comparison of the different available masking functions.

$$\lambda \leftarrow g^{-1}\left(\frac{\lambda}{\frac{1}{64}\sum_{j=1}^{64}\lambda_j}\right) \tag{4.25}$$

after each step. We then have

$$\frac{1}{64}\sum_{j=1}^{64} g\left(\lambda_j^{(\text{new})}\right) = \frac{1}{64}\sum_{j=1}^{64} g\left(g^{-1}\left(\frac{\lambda_j^{(\text{old})}}{\frac{1}{64}\sum_{j=1}^{64}\lambda_j^{(\text{old})}}\right)\right) = \frac{\frac{1}{64}\sum_{j=1}^{64}\lambda_j^{(\text{old})}}{\frac{1}{64}\sum_{j=1}^{64}\lambda_j^{(\text{old})}} = 1.$$

We also clip the input to the inverse masking function, to avoid logarithms of negative numbers.

4.5 Periodic boundary conditions

We have chosen to work with test problems that have periodic boundary conditions, which we want the solutions of our models to respect. To ensure this, we can choose between hard and soft constraints [13]. Soft constraints are easier and more general to implement, as this involves adding a penalty term to the loss function. We have chosen to work with hard constraints, as this will lead to the boundary conditions being satisfied exactly.

[13]: Karniadakis et al. (2021), *Physics-Informed Machine Learning*

For the DeepONet, we follow the suggestion of [3]. Their solution is to encode the spatial part of our query point in a Fourier basis, i.e. $x \to \left[\sin(\frac{2\pi x}{P}), \cos(\frac{2\pi x}{P})\right]$, where P is the periodicity. The periodicity in x of the model then clearly follows from the periodicity in sin and cos.

[3]: Lu et al. (2022), *A Comprehensive and Fair Comparison of Two Neural Operators (with Practical Extensions) Based on FAIR Data*

For the FNOs, we note that both the discrete Fourier and the inverse discrete Fourier transform are N-periodic (with N being the number of elements in the sequence to transform). The output from the kernel integral operator, $\mathcal{K}$, will therefore strictly obey

the periodic boundary conditions, no matter if the input function is or not. On the other hand, the local transform operator, $\mathcal{W}$, allow for outputs that are not periodic. In fact, since the action is pointwise, the output will observe the same periodicity as the input. Since we append the grid to the initial input function (the initial condition a), we can also perform a Fourier feature expansion of the spatial grid, in order to preserve the spatial periodicity. The bias functions, b, are not periodic in general, but we can enforce this directly. Here are three different approaches:

- **Single vector:** We can parametrize the bias function as a single vector, i.e. a constant function. The bias term is then clearly periodic in both x and t.

- **Fourier series:** We write our bias function as a truncated Fourier series with learnable coefficients:

$$\begin{aligned} b(x,t) = &\sum_{n=0}^{N_{max}} \sum_{m=0}^{M_{max}} \alpha_{m,n} \cos\frac{2\pi m x}{P_x} \cos\frac{2\pi n t}{P_t} \\ &+ \sum_{n=0}^{N_{max}} \sum_{m=0}^{M_{max}} \beta_{m,n} \cos\frac{2\pi m x}{P_x} \sin\frac{2\pi n t}{P_t} \\ &+ \sum_{n=0}^{N_{max}} \sum_{m=0}^{M_{max}} \gamma_{m,n} \sin\frac{2\pi m x}{P_x} \cos\frac{2\pi n t}{P_t} \\ &+ \sum_{n=0}^{N_{max}} \sum_{m=0}^{M_{max}} \delta_{m,n} \sin\frac{2\pi m x}{P_x} \sin\frac{2\pi n t}{P_t} \end{aligned}$$

  To avoid periodicity in t, we can make sure that the "period" is larger than the length of the domain, i.e. $P_t > T$.

- **Fourier basis:** Similarly to the trunk net in the DeepONet, we can encode the spatial part of the query point in a Fourier basis, and take $\left[\sin(\frac{2\pi x}{P}), \cos(\frac{2\pi x}{P}), t\right]$ as input to b. The bias function can then be parametrized as a (shallow) neural network, or we can for instance use a polynomial basis.

In the section on Hamiltonian (informed) neural operators, we discussed the need to remove the local linear transform $\mathcal{W}_L$ of the last layer, in order to be able to compute exact derivatives. To avoid outputs that are periodic in time, we therefore need to add a bias function that is periodic in space, but not in time. In this project, we have decided to use the Fourier series approach explained above.

4.6 Hyperparameter optimization

A challenging task in training neural networks, is to pick the best possible hyperparameters. The hyperparameters are the parameters that are not learned during the training process, but rather have to be set in advance by the user. Choosing the wrong parameters could potentially lead to a model that performs drastically worse than it would do with a better set of hyperparameters. To fairly compare different methods, it is therefore essential to try to find the most optimal hyperparameters for each model and problem.

In this project, we will utilize the package `optuna` to optimize the hyperparameters, which is an automatic hyperparameter optimization framework, in combination with some manual tuning. It should however also be noted that finding good hyperparameters is a computationally expensive task, and we have therefore not been able to explore the full hyperparameter space for each model. In addition, the best hyperparameters will in general also depend on the specific problem, and we have therefore chosen to optimize the hyperparameters for each problem separately. The search spaces for the different models are given in the following tables, with the hyperparameters we ended up using.

DeepONet Hyperparameters

Hyperparameter	Range	Value for advection	Value for KdV
trunk width	[50, 150]	85	66
branch width	[50, 150]	126	57
trunk depth	[3, 10]	6	8
branch depth	[3, 10]	7	9
interact size	[5, 25]	25	24
learning rate	[1e-4, 1e-1]	1.6e-4	2.9e-3
number of sensors	{16,32,64}	64	64
number of query points	[50, 2000]	1568	722
self-adaptive learning rate	[1e-4, 1e-1]	6.7e-2	3.0e-2
mask	-	logistic, sharp	polynomial, sharp

Table 4.1: Hyperparameters for the DeepONet model

Modified DeepONet Hyperparameters

Hyperparameter	Range	Value for advection	Value for KdV
width	[50, 150]	150	125
depth	[3, 10]	3	5
interact size	[5, 25]	6	10
learning rate	[1e-4, 1e-1]	2.4e-3	4.6e-3
number of sensors	{16,32,64}	32	32
number of query points	[50, 2000]	1673	1896
self-adaptive learning rate	[1e-4, 1e-1]	6.8e-3	4.1e-2
mask	-	exponential, smooth	polynomial, learnable

Table 4.2: Hyperparameters for the Modified DeepONet model

We did not perform a hyperparameter search for the energy net using `optuna`, but explored this manually.

Furthermore, we have for all problems used the Adam optimizer with weight decay, a batch size of 16 and `gelu` as the activation function. We also used the learning rate scheduler `optax.contrib.reduce_on_plateau`, which halves the step size when the training loss plateaus. We then train the models for a minimum of 300 epochs, and stop the training if the validation loss does not improve for 25 epochs.

For the energy nets in the Hamiltonian Operator Networks, we have also used the Adam optimizer with weight decay, but we here use `optax.schedules.warmup_cosine_decay_schedule` as the learning rate scheduler. We set the initial learning rate to zero and give the models a warm-up period of 50 epochs, before the learning rate starts to decay. This is done to give the models a head-start, to avoid derivative

Table 4.3: Hyperparameters for the FNOTimeStepping model

FNOTimeStepping Hyperparameters

Hyperparameter	Range	Value for advection	Value for KdV
fourier layers	[3, 6]	4	5
hidden dimension	[50, 150]	77	77
max number of modes	[8, 32]	14	26
Δt	-	$\frac{1}{8}$	$\frac{1}{8}$
learning rate	[1e-4, 1e-1]	8.6e-4	3.7e-3
self-adaptive learning rate	[1e-4, 1e-1]	-	8.2e-2
mask	-	-	exponential, smooth

Table 4.4: Hyperparameters for the FNO1d model

FNO1d Hyperparameters

Hyperparameter	Range	Value for advection	Value for KdV
fourier layers	[3, 6]	6	6
hidden dimension	[50, 150]	110	110
max number of modes	[8, 32]	27	27
learning rate	[1e-4, 1e-1]	6.8e-4	6.8e-4
self-adaptive learning rate	[1e-4, 1e-1]	6.8e-3	3.8e-2
mask	-	exponential, smooth	exponential, smooth

predictions that are too far off from the true values. The peak value is set to 10^{-3} and decays to 10^{-5} over 500 epochs.

We finetuned the energy penalty weights manually. For the KdV equation, we set $\lambda_{\text{energy}} = 10^{-8}$ for `FNO2d` and $\lambda_{\text{energy}} = 10^{-10}$ for `FNOTimeStepping`. For the advection equation, we set $\lambda_{\text{energy}} = 10^{-8}$ for `FNO2d` and $\lambda_{\text{energy}} = 10^{-6}$ for `ModifiedDeepONet`. All other methods had $\lambda_{\text{energy}} = 10^{-4}$. The penalty weights were set to be smaller for the methods where we experienced energy losses that dominated the loss function.

FNO2d Hyperparameters

Hyperparameter	Range	Value for advection	Value for KdV
fourier layers	[3, 6]	4	4
hidden dimension	[50, 150]	90	90
max number of modes	[8, 32]	8	8
learning rate	[1e-4, 1e-1]	1.2e-2	1.2e-2
self-adaptive learning rate	[1e-4, 1e-1]	-	-
mask	-	-	-

Table 4.5: Hyperparameters for the FNO2d model

Energy net Hyperparameters

Hyperparameter	Value
width	75
depth	3

Table 4.6: Hyperparameters for the energy net

4.7 Implementation

A very large portion of this project has been spent on the efficient implementation of the different models. The code for this project is written in `Python`, and utilizes the `JAX` library. There are many deep learning libraries one can use in conjunction with `JAX`, and in this project we have opted for the `Equinox` package [34].

[34]: Kidger et al. (2021), *Equinox: Neural Networks in JAX via Callable PyTrees and Filtered Transformations*

Results 5

5.1 Verifying the implementation

In this section, we wish to empirically verify our implementation of the derivatives for the different models. This includes the computation of u_x, u_t and $\mathcal{G}\frac{\delta \mathcal{H}}{\delta u}$, discussed in Section 3.2. We do this using the models trained on the Korteweg–De Vries equation.

Verification of spatial and temporal model derivatives

In both Figure 5.1 and Figure 5.2, the first row shows the exact derivatives of the model predictions, using automatic differentiation, as discussed in Section 3.2. The second row shows the derivatives of the predictions, using sixth order finite differences. The last row shows the derivatives of the ground truth label, again using sixth order finite differences.

For the temporal derivatives, we have set the values for the finite difference comparisons at the start and end times to zero, as u is not periodic in time. Note that we do not need to make any considerations regarding this for the automatic differentiations, as these do not depend on neighboring points.

We are here interested in the observation that the first and second rows should be seemingly identical, to verify that the model derivatives are correctly implemented. Ideally, these should also be close to the third row, but this depends on the accuracy of the model which will be investigated in the next section.

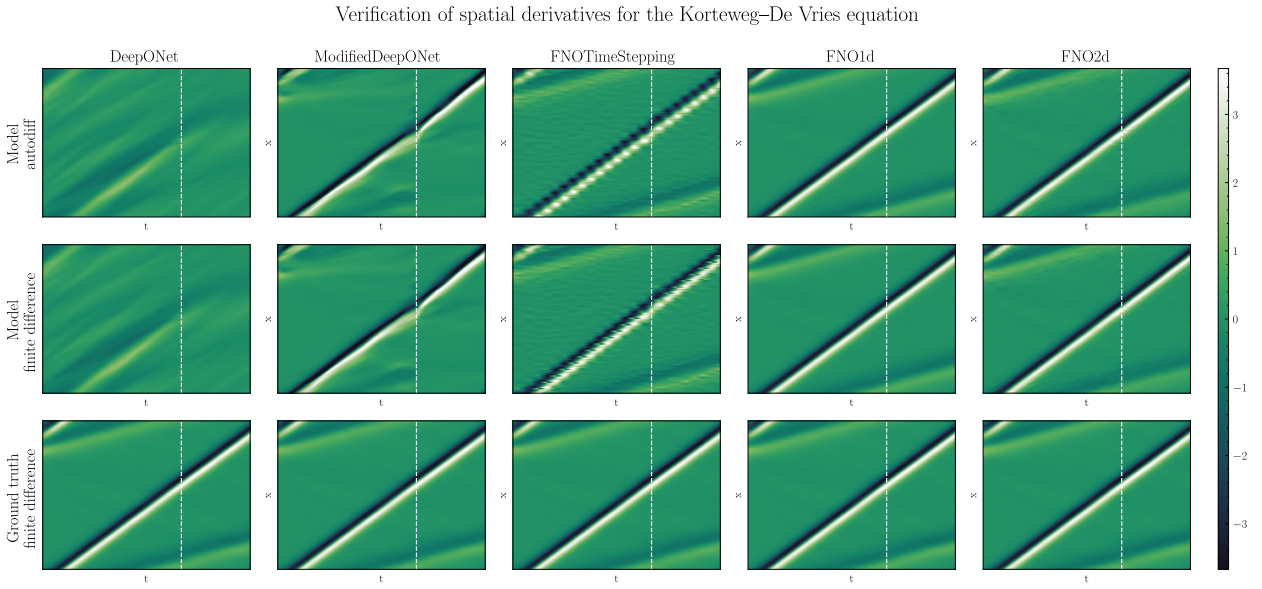

Figure 5.1: Verification of spatial derivatives.

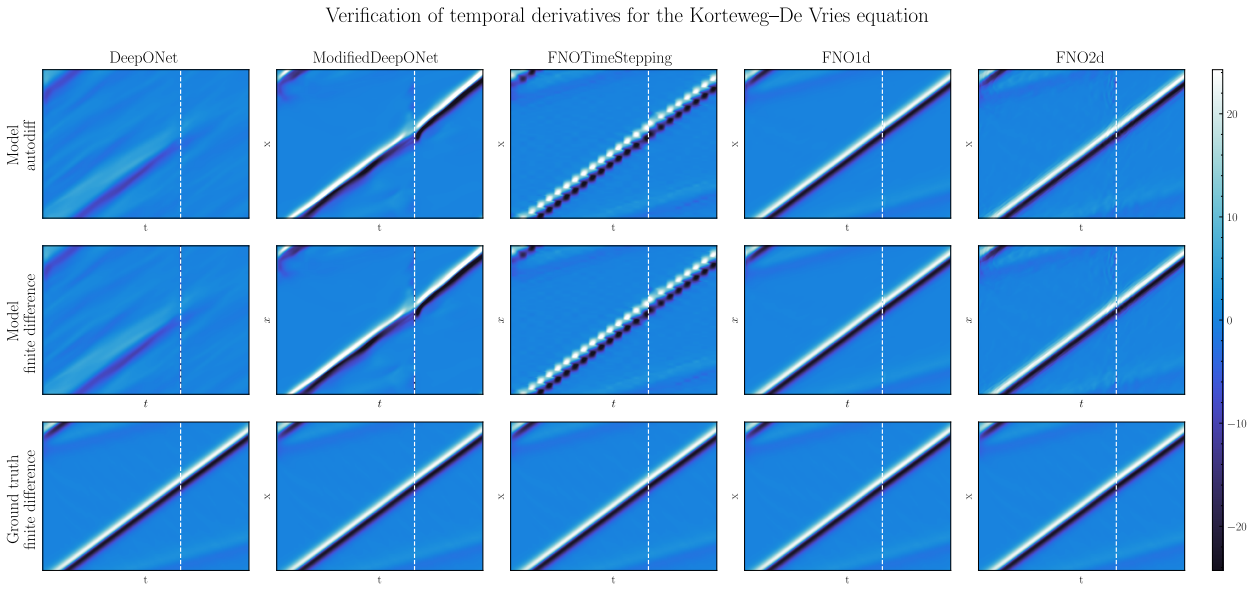

Figure 5.2: Verification of temporal derivatives.

We see that the first and second rows match in both figures, with the exception for the spatial derivatives for `FNOTimeStepping`. As this method is discrete in time, we interpolate the prediction and the spatial derivative using `Akima` splines, and the derivative is therefore only exact at the grid points used in the method.

Verification of Hamiltonian PDE right-hand side implementation

To verify the implementation discussed in Section 3.2, we have used the analytical energy density function of the Korteweg–De Vries equation. u is here a test sample, and $u_t, u_x, u_{xx}, u_{xxx}$ are all computed using sixth order central differences.

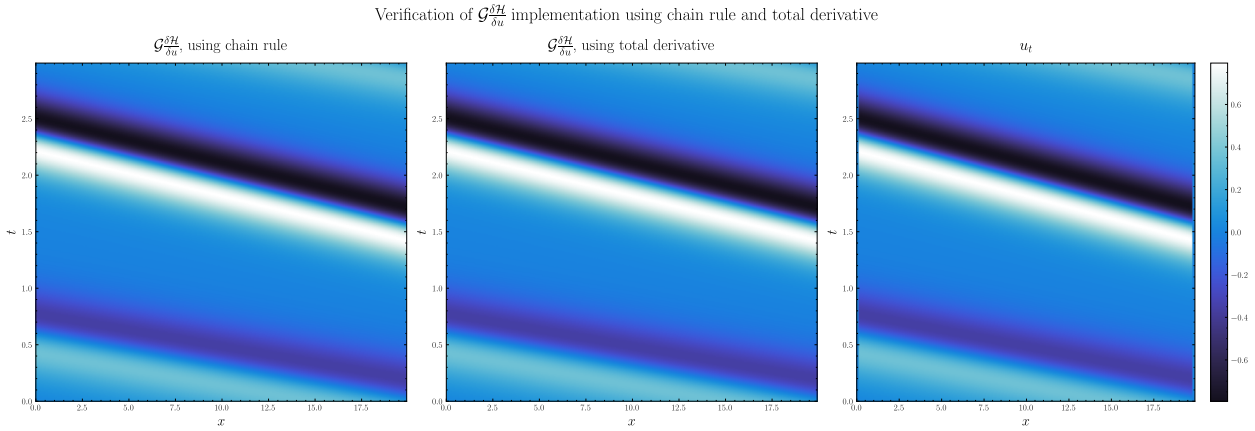

Figure 5.3: Verification of Hamiltonian right-hand side implementation.

We see that both implementations, as shown in the first two plots of Figure 5.3, match that of the last plot which shows u_t. Note that we for u_t again do not plot the first and last time plots, as u is not periodic in time.

We also want to verify the use of ODE integration to retrieve the final result, as discussed in Chapter 3. We do this by first computing $\mathcal{G}\frac{\delta\mathcal{H}}{\delta u}$ according to either the total derivative or chain rule approach, and then use a Gauss-Legendre method of order six. As we see, the integrated value to the left, matches that of the original data seen to the right in Figure 5.4.

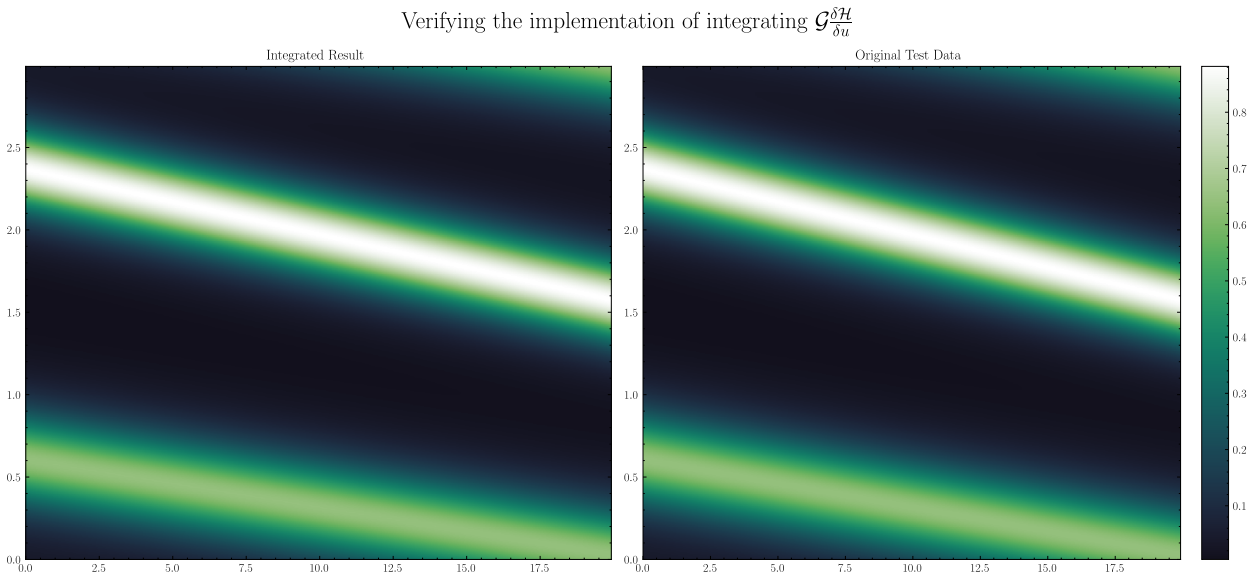

Figure 5.4: Verification of Gauss integration.

Self-adaptive weights

We also show the self-adaptive weights throughout the training on the KdV equation, in Figure 5.5. We here see some slight indication for smaller weights for earlier timesteps, especially for the `FNOTimeStepping` model which enforces the initial condition exactly. However, we feel that more experimentation is required to see the impacts of self-adaptive weights.

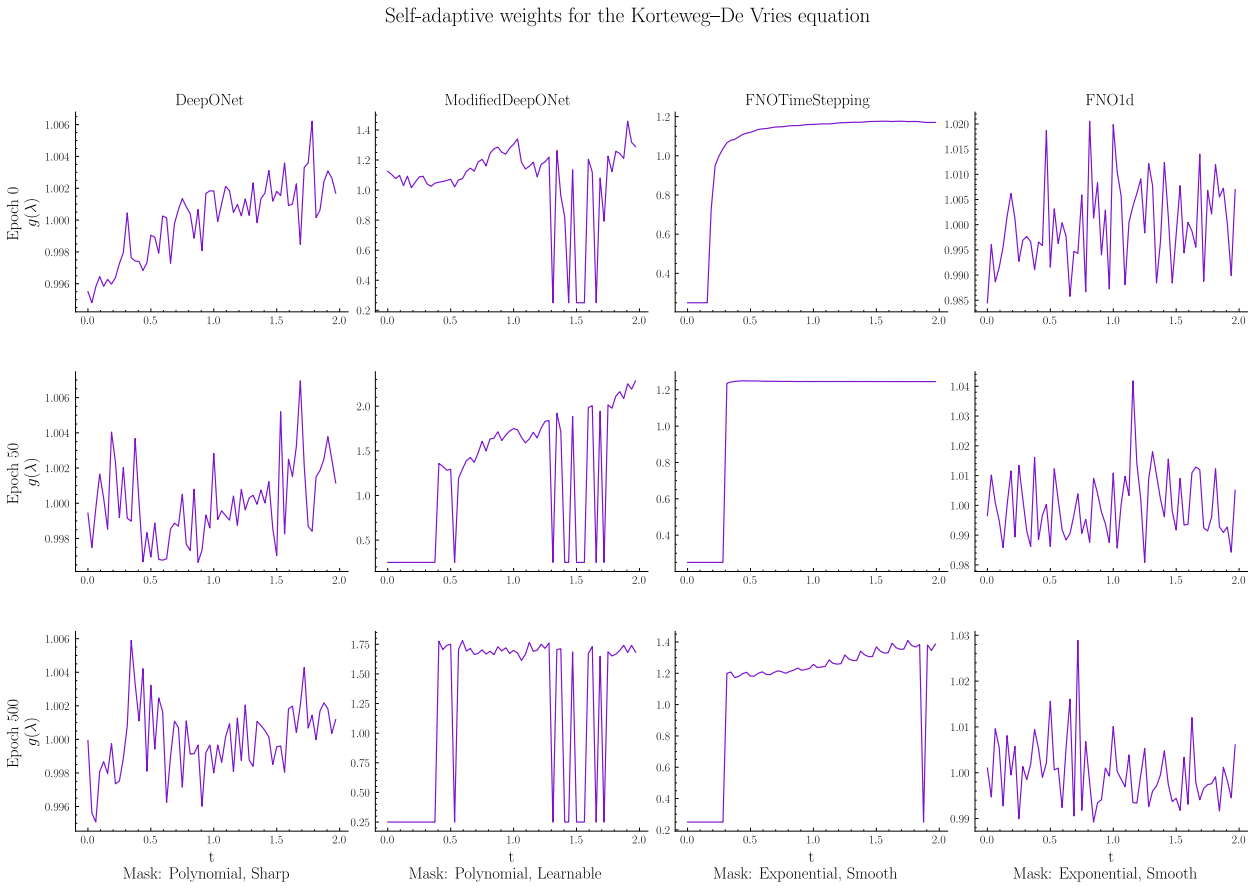

Figure 5.5: The self-adaptive weights throughout the training on the KdV equation.

5.2 Comparison of methods

Vanilla methods

Advection equation

We have chosen to start with the advection equation, as it is the simplest of the two PDEs we are considering. Since this is a relatively easy problem, we expect all methods to be able to perform well, and it acts as a good starting point for comparison.

From the loss plot in Figure 5.6, we see that all models are able to learn quite well, apart from the vanilla `DeepONet` model, which converges very slowly. There is no indication of overfitting, as both the training and validation loss are decreasing. More

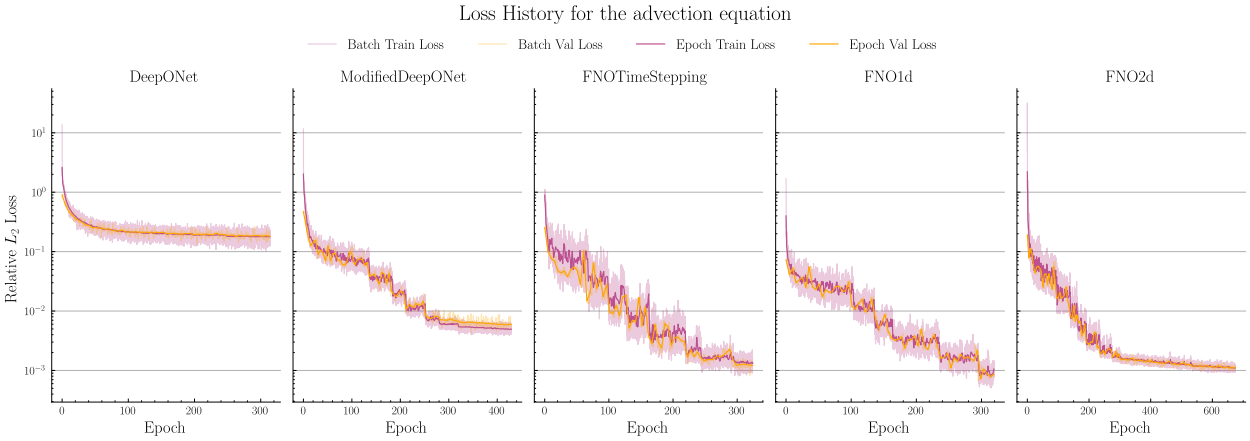

Figure 5.6: Loss plot, using vanilla methods on the advection equation.

experimentation regarding architecture or learning rates might therefore be needed for the `DeepONet` model. All the FNO-based models seem to perform particularly well, reaching a validation loss of around 10^{-3}. It should also be mentioned that the `FNOTimeStepping` model has a particularly high variance in the loss, as indicated by the shaded batch loss. This model might therefore have benefitted more from a lower learning rate or a larger batch size.

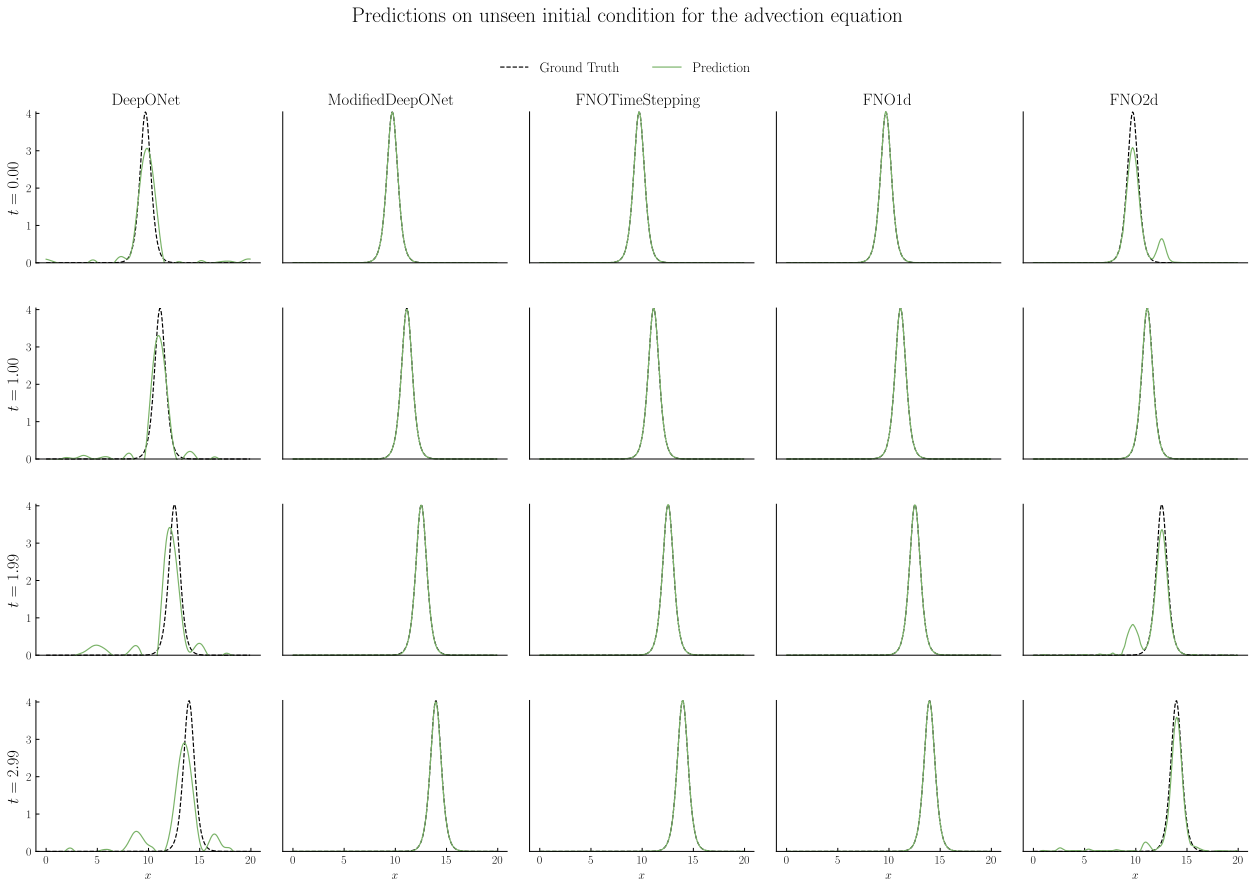

Figure 5.7: Predictions on advection with vanilla methods.

In Figure 5.7, we investigate the predictions of the models at different time steps. The first row marks the start of the time domain, and should ideally produce predictions identical to the initial condition. The second row shows the predictions in the middle of the time domain seen during training, the third row shows the predictions at the end of the time domain, and the last row shows how the predictions extrapolate beyond the training domain.

We clearly see that the `FNO2d` and `DeepONet` models violate the initial condition, and the `FNOTimeStepping` method is in fact the only method that strictly imposes this. Apart from the `FNO2d` and `DeepONet` models, all methods seem to perform well.

We also show the results as heatmaps in Figure 5.11, with the predictions in the first row and the errors in the second row.

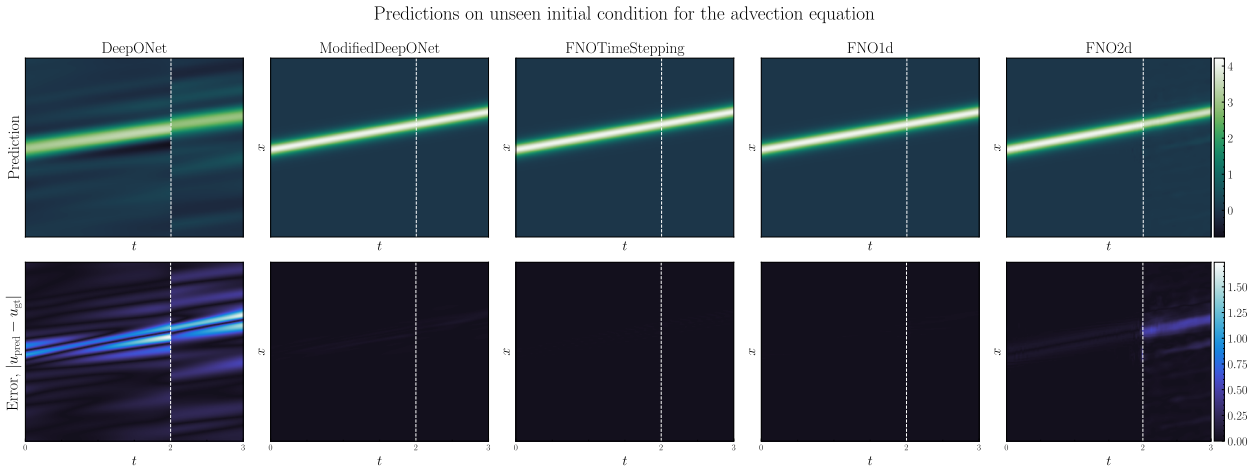

Figure 5.8: Heatmap predictions on advection with vanilla methods.

Korteweg–De Vries equation

We now move on to the more complex KdV equation.

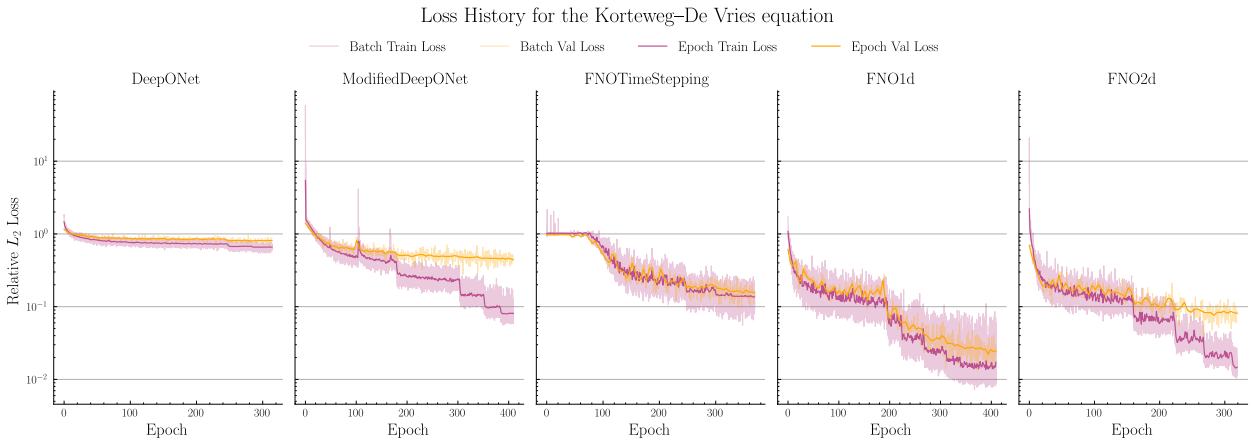

Figure 5.9: Loss plot, using vanilla methods on Korteweg–De Vries.

From the loss plot in Figure 5.9, we see that the `DeepONet` model again converges very slowly, and would likely benefit from more experimentation regarding architecture and hyperparameters. We now also see that the `ModifiedDeepONet` and `DeepONet` models are overfitting, and would likely benefit from more regularization.

From Figure 5.10 and Figure 5.11, we qualitatively see that the FNO-based methods perform quite well, even in the extrapolation region. The DeepONet-based models do not perform as well, with the vanilla `DeepONet` model in particular giving quite poor predictions.

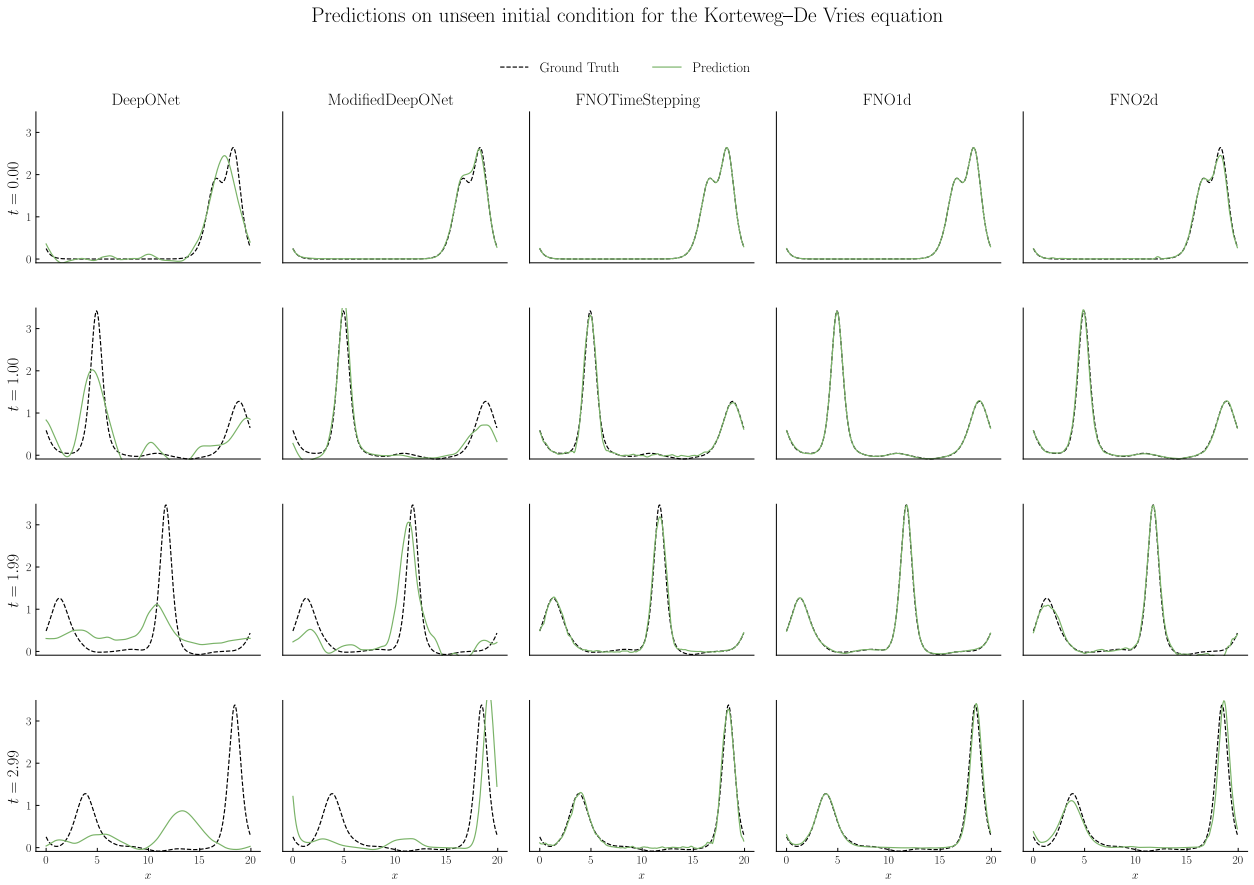

Figure 5.10: Predictions on Korteweg–De Vries with vanilla methods

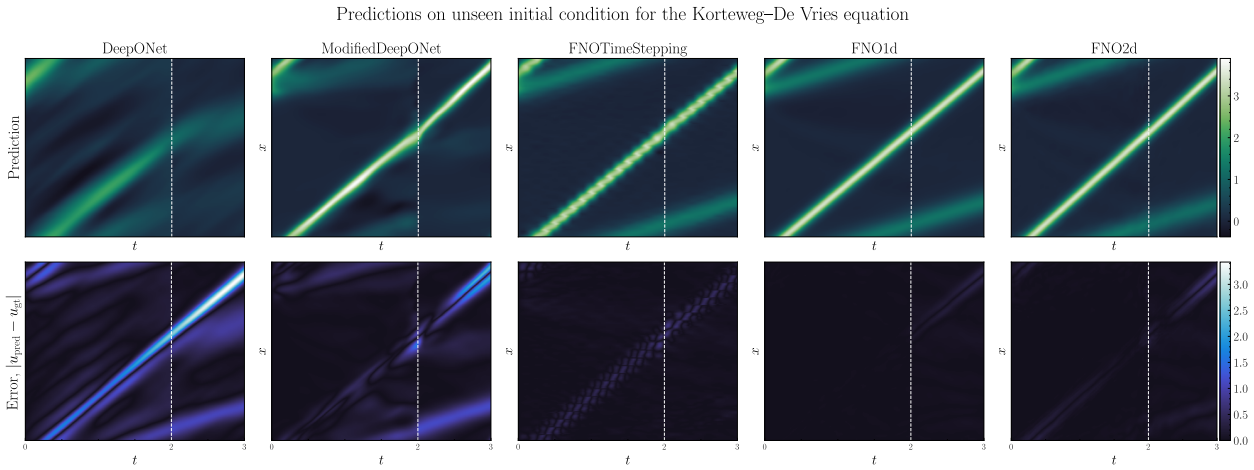

Figure 5.11: Heatmap predictions on Korteweg–De Vries with vanilla methods

Hamiltonian Operator Network

We finally include preliminary results for the Hamiltonian Operator Network, showcasing both the soft and hard-constrained approaches.

Advection equation

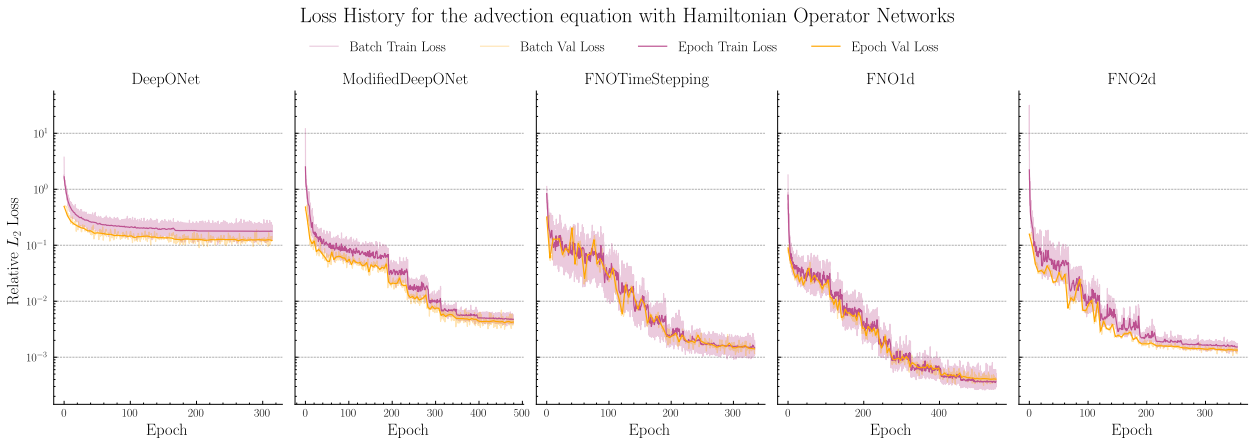

Figure 5.12: Loss history for Hamiltonian Operator Networks on the advection equation.

All models show good loss functions in Figure 5.12, apart from the DeepONet model, which quickly stabilizes at a high loss. This is further confirmed in Figure 5.14, where we see that all models have learned the solution operator. However, the DeepONet model and some of the hard-constrained models, in particular FNOTimeStepping and FNO2d, display higher errors.

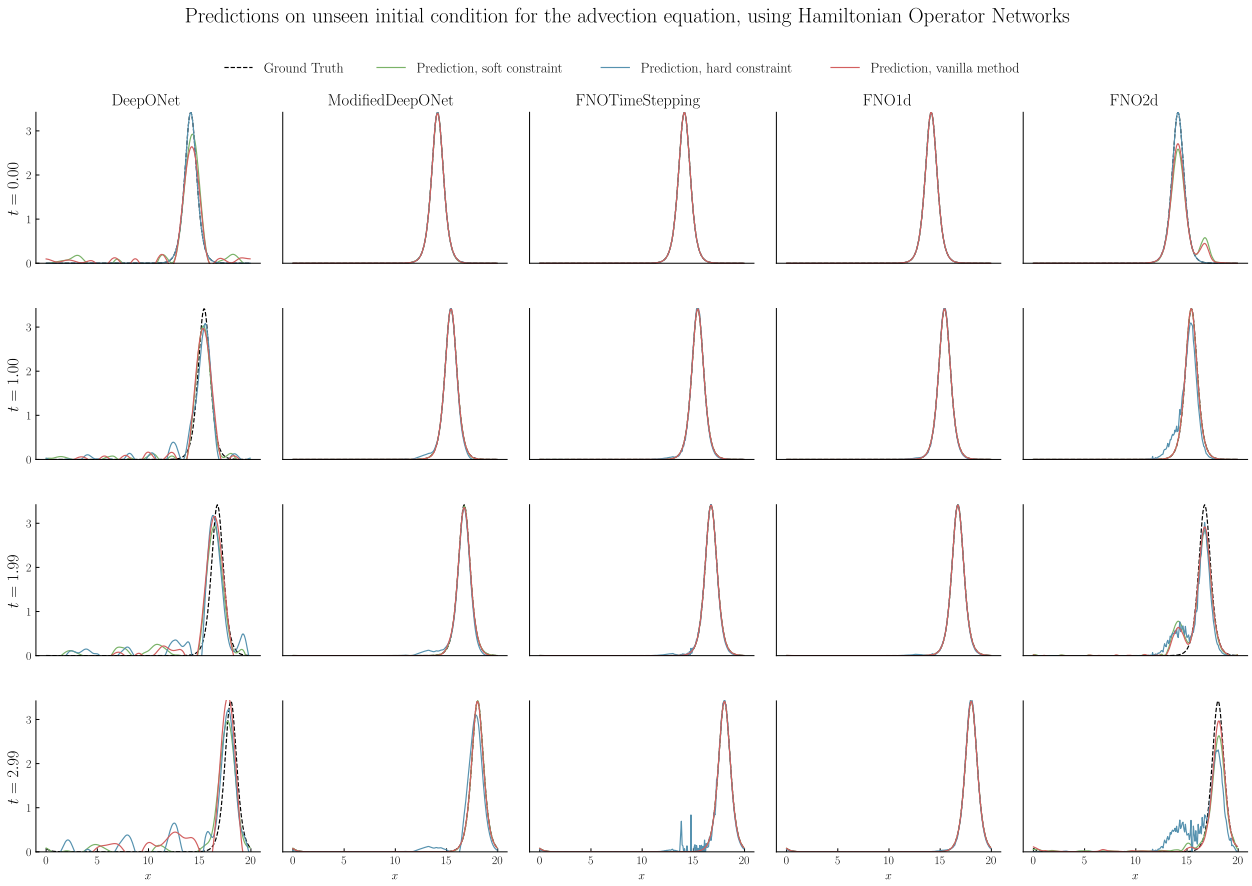

Figure 5.13: Predictions for Hamiltonian Operator Networks on the advection equation.

We now use the analytical Hamiltonian function, as given in Equation 4.4, to compute

the Hamiltonian based on the predictions from one sample. Across all models, it seems that the soft-constrained models are able to better conserve the Hamiltonian, as seen in Figure 5.14.

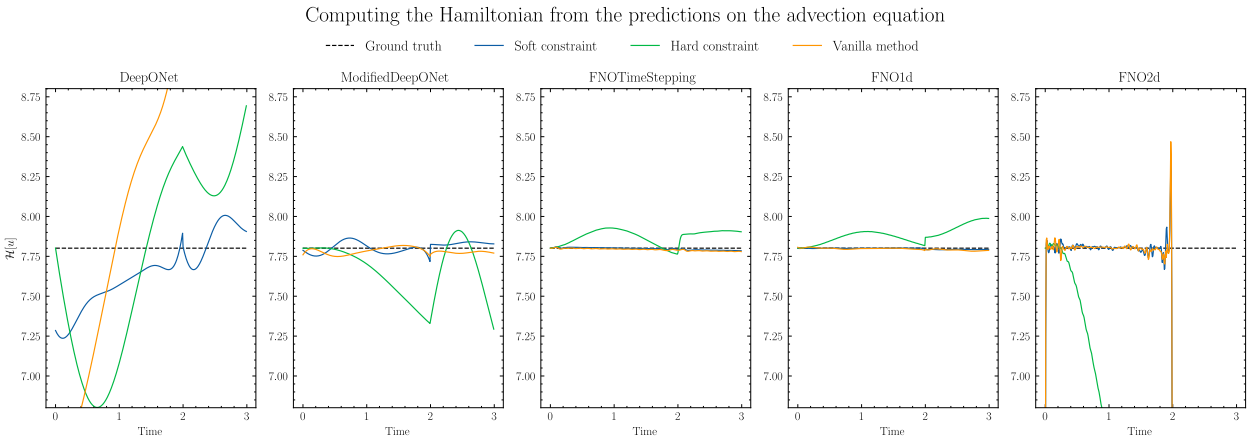

Figure 5.14: Computing the Hamiltonian based on the predictions, for the advection equation.

Korteweg–De Vries equation

We lastly move on to using the Hamiltonian Operator Networks for the KdV equation. For this section, we were not able to retrieve results from the Hamiltonian Operator Network with the Modified DeepONet as the operator network, so these have been left out.

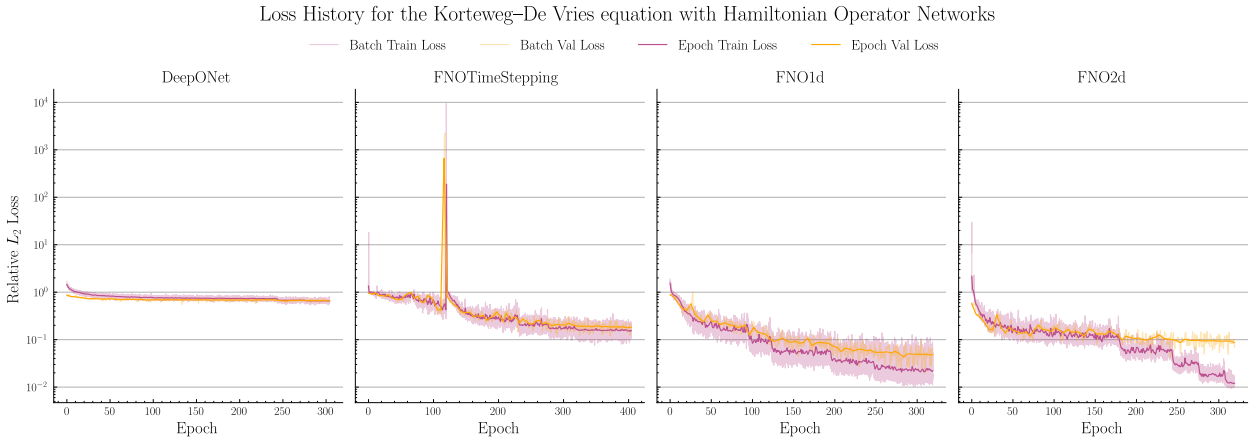

Figure 5.15: Loss history for Hamiltonian Operator Networks on the Korteweg–De Vries equation.

From the loss plot in Figure 5.15, we in particular observe that the loss function appears less stable for the FNOTimeStepping method. This was also observed while choosing hyperparameters, where a higher energy penalty weight quickly led to unstable training.

In Figure 5.16, we see that the hard-constrained models clearly are not able to learn the solution operator, but the other methods seem to perform well. This is further confirmed in Figure 5.17, where the vanilla and soft-constrained methods seem to be the closest to conserving the Hamiltonian.

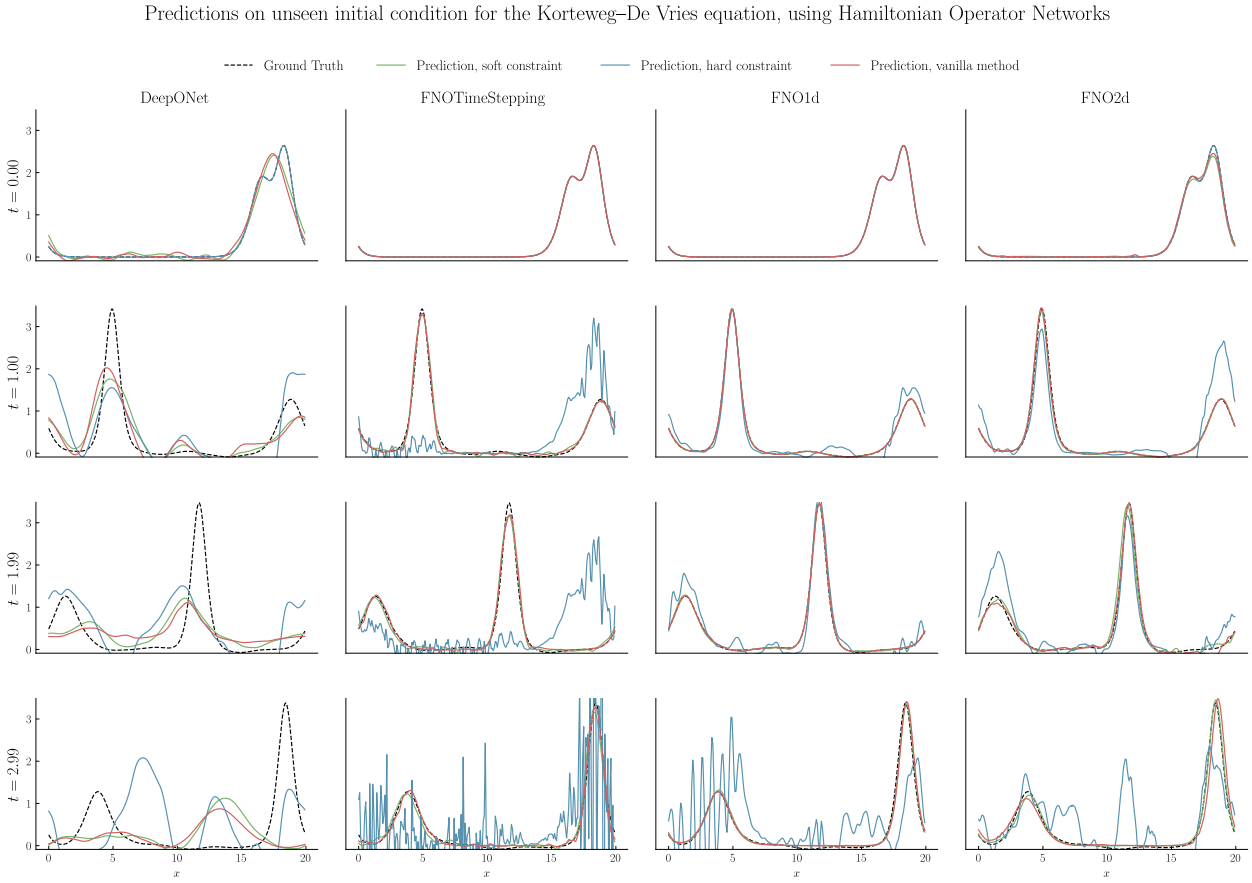

Figure 5.16: Predictions for Hamiltonian Operator Networks on the Korteweg–De Vries equation.

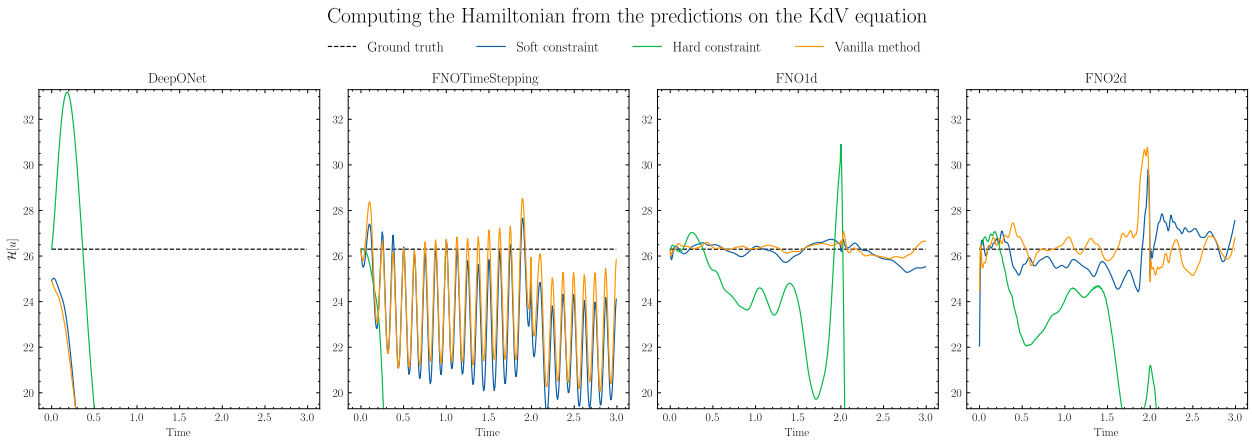

Figure 5.17: Computing the Hamiltonian based on the predictions, for the Korteweg–De Vries equation.

Performance metrics

We finally compute the performance metrics, including relative L_1 and L_2 errors on the test set, as well as the time taken by the methods to predict the full-resolution output on all samples in the test set. In this section, we do not consider extrapolation region $t \in [2, 3)$. The lowest scores are marked in bold.

We see that the FNO1d method gives the lowest relative errors for both test problems, and that the DeepONet method is the fastest. For the KdV-equation, we have in fact

Table 5.1: Performance on the advection equation.

Performance Metrics on the advection equation

Method	# Parameters	Epochs Trained	Relative L_2 Error	Relative L_1 Error	Time test set (s)
FNO1d (Vanilla)	2022085	320	0.0008	0.0012	2.2384
DeepONet (Vanilla)	146482	315	0.1227	0.2142	**0.6403**
FNO2d (Vanilla)	4172567	675	0.0157	0.0152	1.8950
FNOTimeStepping (Vanilla)	350505	325	0.0012	0.0019	1.9211
ModifiedDeepONet (Vanilla)	58277	430	0.0040	0.0054	1.3940
FNO1d (HON, soft)	2033786	550	**0.0003**	**0.0006**	1.3917
FNO1d (HON, hard)	2033786	550	0.0150	0.0201	143.0301
DeepONet (HON, soft)	158183	315	0.1193	0.2058	**0.5669**
DeepONet (HON, hard)	158183	315	0.1469	0.2487	285.0311
FNO2d (HON, soft)	4184268	355	0.0171	0.0150	1.2277
FNO2d (HON, hard)	4184268	355	0.1187	0.1473	10.0974
FNOTimeStepping (HON, soft)	362206	335	0.0011	0.0019	1.2859
FNOTimeStepping (HON, hard)	362206	335	0.0210	0.0275	11.1497
ModifiedDeepONet (HON, soft)	69978	480	0.0041	0.0054	0.6937
ModifiedDeepONet (HON, hard)	69978	480	0.0376	0.0481	458.4253

Table 5.2: Performance metrics on the Korteweg–De Vries equation.

Performance Metrics on the Korteweg–De Vries equation

Method	# Parameters	Epochs Trained	Relative L_2 Error	Relative L_1 Error	Time test set (s)
FNO1d (Vanilla)	2022085	410	**0.0203**	**0.0220**	2.2756
DeepONet (Vanilla)	64436	315	0.5763	0.7719	**0.7680**
FNO2d (Vanilla)	4172567	320	0.0812	0.0803	2.5478
FNOTimeStepping (Vanilla)	635161	370	0.1337	0.1488	2.2338
ModifiedDeepONet (Vanilla)	106336	410	0.3416	0.3585	1.4650
FNO1d (HON, soft)	2033786	320	0.0305	0.0320	1.4009
FNO1d (HON, hard)	2033786	320	0.2704	0.3560	144.5205
DeepONet (HON, soft)	76137	305	0.5815	0.7775	**0.6730**
DeepONet (HON, hard)	76137	305	0.8357	1.1975	342.4109
FNO2d (HON, soft)	4184268	320	0.0946	0.0909	1.2469
FNO2d (HON, hard)	4184268	320	0.6021	0.7350	10.2327
FNOTimeStepping (HON, soft)	646862	405	0.1623	0.1929	1.7002
FNOTimeStepping (HON, hard)	646862	405	1.0213	1.2234	10.7051

that the soft-constrained `FON1d_HON` model gives the lowest relative errors, but this might also be due to longer training times. The hard-constrained models are in general much slower, and give higher errors.

5.3 Conclusion and future work

In this project, we have investigated the use of two popular operator learning architectures, namely the Fourier Neural Operators (FNO) and DeepONets [2, 10], for solving PDEs. We further discussed how to enforce energy conservation in these models, based on the approach of the Energy-consistent Neural Operators (ENO) [20]. We introduced the Hamiltonian Operator Network (HON), which uses either a DeepONet or FNO-based operator network as a part of the architecture. In addition, we investigated the use of a hard-constrained approach, where the discretized learned Hamiltonian is preserved.

From the previous section, we clearly see that the vanilla methods actually perform better than our HON approach. This might be due to the hyperparameters chosen for the energy net, and more experimentation would likely be needed to assess the full potential of the HON approach. Moreover, the ability of the hard-constrained methods to conserve the *true* Hamiltonian depends on both the operator network and the energy network, and if the energy network is unable to accurately predict the true energy density, we will naturally get poor predictions. Another reason why the HON methods

give poor results, might also be due to the required regularity of the predictions. In particular, we see that the chain-rule approach for computing the Hamiltonian PDE right-hand side, as discussed in Section 3.2, require computing third order spatial derivatives. As we do not enforce smoothness in the predictions, we might therefore get high derivative values that are far off from the ground truth.

It would also be interesting to investigate other methods for enforcing energy conservation. For instance, one could also investigate the use of an operator network in the energy net, or other network architectures, rather than an MLP as we have used. One could also investigate modeling the Hamiltonian directly, rather than the energy density, and perhaps incorporate the semi-discrete Average Vector Field (AVF) method [27]. Recently, a library for automatic functional differentiation in `JAX` was developed in [35], which could be highly relevant in the context of this project. Lastly, we have in this project only considered global energy conservation, but it could also be interesting to investigate local conservation laws, in the context of multisymplectic PDEs.

[27]: Celledoni et al. (2012), *Preserving Energy Resp. Dissipation in Numerical PDEs Using the "Average Vector Field" Method*

[35]: Lin (2024), *Automatic Functional Differentiation in JAX*

Appendix

Github repository

All code files used in this document are included in the Github repository linked below.

- https://github.com/eirikfagerbakke/specialization_project

Notation summary

Symbol	Description
$\mathcal{S}$	Operator mapping from input to output function space
$\mathcal{A}$	Input function space (Banach space)
$\mathcal{U}$	Output function space (Banach space)
$\mathcal{X}$	Bounded spatial domain $\subset \mathbb{R}^d$
$\mathcal{Y}$	Spatial-temporal domain $\mathcal{T} \times \mathcal{X} \subset \mathbb{R}^{d+1}$
a	Input function (initial condition of the PDE)
u	Output function, $u = \mathcal{S}[a]$
y	Query point, $y = (x, t) \in \mathbb{R}^{d+1}$
$\{a^{(i)}, u^{(i)}\}_{i=1}^N$	Observations of input-output pairs
μ	Probability measure supported on $\mathcal{A}$
$\mathcal{S}_\theta$	Neural network approximation of the operator $\mathcal{S}$
θ	Parameters of the neural network $\in \mathbb{R}^p$
λ	Self-adaptive weights

Table 3: Summary of notation used in the project.

Bibliography

References given in citation order.

[1] Nikola B. Kovachki, Samuel Lanthaler, and Andrew M. Stuart. *Operator Learning: Algorithms and Analysis*. Feb. 23, 2024. URL: http://arxiv.org/abs/2402.15715 (visited on 10/07/2024). Pre-published (cited on page 1).

[2] Lu Lu, Pengzhan Jin, and George Em Karniadakis. 'DeepONet: Learning Nonlinear Operators for Identifying Differential Equations Based on the Universal Approximation Theorem of Operators'. In: *Nature Machine Intelligence* 3.3 (Mar. 18, 2021), pp. 218–229. DOI: 10.1038/s42256-021-00302-5. (Visited on 10/07/2024) (cited on pages 1, 3, 5, 42).

[3] Lu Lu et al. '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. DOI: 10.1016/j.cma.2022.114778. (Visited on 10/28/2024) (cited on pages 1, 4, 5, 7, 8, 25).

[4] Qianying Cao, Somdatta Goswami, and George Em Karniadakis. *LNO: Laplace Neural Operator for Solving Differential Equations*. May 30, 2023. DOI: 10.48550/arXiv.2303.10528. URL: http://arxiv.org/abs/2303.10528 (visited on 01/03/2025). Pre-published (cited on page 1).

[5] Jacob H. Seidman et al. *NOMAD: Nonlinear Manifold Decoders for Operator Learning*. June 7, 2022. DOI: 10.48550/arXiv.2206.03551. URL: http://arxiv.org/abs/2206.03551 (visited on 12/18/2024). Pre-published (cited on page 1).

[6] Wei Xiong et al. 'Koopman Neural Operator as a Mesh-Free Solver of Non-Linear Partial Differential Equations'. In: *Journal of Computational Physics* 513 (Sept. 2024), p. 113194. DOI: 10.1016/j.jcp.2024.113194. (Visited on 01/03/2025) (cited on page 1).

[7] Sifan Wang, Hanwen Wang, and Paris Perdikaris. 'Improved Architectures and Training Algorithms for Deep Operator Networks'. In: *Journal of Scientific Computing* 92.2 (Aug. 2022), p. 35. DOI: 10.1007/s10915-022-01881-0. (Visited on 10/27/2024) (cited on pages 1, 5, 6, 19, 21).

[8] Samuel Lanthaler. *Operator Learning with PCA-Net: Upper and Lower Complexity Bounds*. Oct. 13, 2023. DOI: 10.48550/arXiv.2303.16317. URL: http://arxiv.org/abs/2303.16317 (visited on 01/03/2025). Pre-published (cited on page 1).

[9] Michael Prasthofer, Tim De Ryck, and Siddhartha Mishra. *Variable-Input Deep Operator Networks*. May 23, 2022. DOI: 10.48550/arXiv.2205.11404. URL: http://arxiv.org/abs/2205.11404 (visited on 12/18/2024). Pre-published (cited on pages 1, 5, 9).

[10] Nikola Kovachki et al. *Neural Operator: Learning Maps Between Function Spaces*. May 2, 2024. DOI: 10.5555/3648699.3648788. URL: http://arxiv.org/abs/2108.08481 (visited on 10/21/2024). Pre-published (cited on pages 1, 2, 5–9, 21, 42).

[11] Zongyi Li et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. May 16, 2021. URL: http://arxiv.org/abs/2010.08895 (visited on 10/07/2024). Pre-published (cited on page 1).

[12] M. Raissi, P. Perdikaris, and G.E. Karniadakis. 'Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations'. In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045. (Visited on 10/07/2024) (cited on pages 1, 12).

[13] George Em Karniadakis et al. 'Physics-Informed Machine Learning'. In: *Nature Reviews Physics* 3.6 (May 24, 2021), pp. 422–440. DOI: 10.1038/s42254-021-00314-5. (Visited on 10/07/2024) (cited on pages 1, 11, 25).

[14] Zongyi Li et al. *Physics-Informed Neural Operator for Learning Partial Differential Equations*. July 29, 2023. DOI: 10.48550/arXiv.2111.03794. URL: http://arxiv.org/abs/2111.03794 (visited on 12/10/2024). Pre-published (cited on pages 1, 14, 15).

[15] Sifan Wang, Hanwen Wang, and Paris Perdikaris. *Learning the Solution Operator of Parametric Partial Differential Equations with Physics-Informed DeepOnets*. Mar. 19, 2021. DOI: 10.48550/arXiv.2103.10974. URL: http://arxiv.org/abs/2103.10974 (visited on 01/03/2025). Pre-published (cited on page 1).

[16] Sam Greydanus, Misko Dzamba, and Jason Yosinski. *Hamiltonian Neural Networks*. Sept. 5, 2019. URL: http://arxiv.org/abs/1906.01563 (visited on 10/07/2024). Pre-published (cited on pages 1, 12).

[17] Miles Cranmer et al. *Lagrangian Neural Networks*. July 30, 2020. DOI: 10.48550/arXiv.2003.04630. URL: http://arxiv.org/abs/2003.04630 (visited on 01/03/2025). Pre-published (cited on page 1).

[18] Sølve Eidnes et al. 'Pseudo-Hamiltonian Neural Networks with State-Dependent External Forces'. In: *Physica D: Nonlinear Phenomena* 446 (Apr. 2023), p. 133673. DOI: 10.1016/j.physd.2023.133673. (Visited on 10/07/2024) (cited on page 1).

[19] Sølve Eidnes and Kjetil Olsen Lye. 'Pseudo-Hamiltonian Neural Networks for Learning Partial Differential Equations'. In: *Journal of Computational Physics* 500 (Mar. 2024), p. 112738. DOI: 10.1016/j.jcp.2023.112738. (Visited on 10/07/2024) (cited on page 1).

[20] Yusuke Tanaka et al. *Neural Operators Meet Energy-based Theory: Operator Learning for Hamiltonian and Dissipative PDEs*. Feb. 14, 2024. URL: http://arxiv.org/abs/2402.09018 (visited on 10/07/2024). Pre-published (cited on pages 1, 2, 11, 12, 23, 42).

[21] Tianping Chen and Hong 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 (July 1995), pp. 911–917. DOI: 10.1109/72.392253. (Visited on 12/18/2024) (cited on page 3).

[22] Sifan Wang and Paris Perdikaris. *Long-Time Integration of Parametric Evolution Equations with Physics-Informed DeepONets*. June 9, 2021. DOI: 10.48550/arXiv.2106.05384. URL: http://arxiv.org/abs/2106.05384 (visited on 01/13/2025). Pre-published (cited on page 3).

[23] Maarten V. de Hoop et al. *The Cost-Accuracy Trade-Off In Operator Learning With Neural Networks*. Aug. 11, 2022. URL: http://arxiv.org/abs/2203.13181 (visited on 10/21/2024). Pre-published (cited on pages 5, 9).

[24] Jean Kossaifi et al. *A Library for Learning Neural Operators*. Dec. 17, 2024. DOI: 10.48550/arXiv.2412.10354. URL: http://arxiv.org/abs/2412.10354 (visited on 12/29/2024). Pre-published (cited on page 7).

[25] Benedict Leimkuhler and Sebastian Reich. *Simulating Hamiltonian Dynamics*. 1st ed. Cambridge University Press, Feb. 14, 2005. (Visited on 10/07/2024) (cited on pages 11, 21).

[26] Peter J. Olver. *Applications of Lie Groups to Differential Equations*. Vol. 107. Graduate Texts in Mathematics. New York, NY: Springer New York, 1986. (Visited on 01/13/2025) (cited on page 11).

[27] E. Celledoni et al. 'Preserving Energy Resp. Dissipation in Numerical PDEs Using the "Average Vector Field" Method'. In: *Journal of Computational Physics* 231.20 (Aug. 2012), pp. 6770–6789. DOI: 10.1016/j.jcp.2012.06.022. (Visited on 01/13/2025) (cited on pages 11, 43).

[28] Lei Huang et al. *Normalization Techniques in Training DNNs: Methodology, Analysis and Application*. Sept. 27, 2020. DOI: 10.48550/arXiv.2009.12836. URL: http://arxiv.org/abs/2009.12836 (visited on 01/13/2025). Pre-published (cited on page 19).

[29] Shengfeng Xu et al. *On the Preprocessing of Physics-informed Neural Networks: How to Better Utilize Data in Fluid Mechanics*. July 11, 2024. DOI: 10.48550/arXiv.2403.19923. URL: http://arxiv.org/abs/2403.19923 (visited on 01/07/2025). Pre-published (cited on page 19).

[30] Xavier Glorot and Yoshua Bengio. 'Understanding the Difficulty of Training Deep Feedforward Neural Networks'. In: () (cited on page 19).

[31] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. Feb. 6, 2015. DOI: 10.48550/arXiv.1502.01852. URL: http://arxiv.org/abs/1502.01852 (visited on 01/13/2025). Pre-published (cited on page 19).

[32] P. Clark Di Leoni et al. *DeepONet Prediction of Linear Instability Waves in High-Speed Boundary Layers*. May 18, 2021. URL: http://arxiv.org/abs/2105.08697 (visited on 10/28/2024). Pre-published (cited on page 21).

[33] Levi McClenny and Ulisses Braga-Neto. 'Self-Adaptive Physics-Informed Neural Networks Using a Soft Attention Mechanism'. In: *Journal of Computational Physics* 474 (Feb. 2023), p. 111722. DOI: 10.1016/j.jcp.2022.111722. (Visited on 12/27/2024) (cited on pages 23, 24).

[34] Patrick Kidger and Cristian Garcia. *Equinox: Neural Networks in JAX via Callable PyTrees and Filtered Transformations*. Oct. 30, 2021. URL: http://arxiv.org/abs/2111.00254 (visited on 10/28/2024). Pre-published (cited on page 29).

[35] Min Lin. *Automatic Functional Differentiation in JAX*. Jan. 28, 2024. DOI: 10.48550/arXiv.2311.18727. URL: http://arxiv.org/abs/2311.18727 (visited on 01/21/2025). Pre-published (cited on page 43).