Mathematisch-Naturwissenschaftliche Fakultät

Methoden des Maschinellen Lernens

Bachelorarbeit

Evaluating Uncertainty Quantification Techniques in Fourier Neural Operators

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Methoden des Maschinellen Lernens
B.Sc. Informatik
Nicolas Seng, 6050809, nicolas.seng@uni-tuebingen.de, 2025

Bearbeitungszeitraum: 05.01.2025 - 05.05.2025

Gutachter: Prof. Dr. Philipp Hennig, Universität Tübingen
Betreuerin: Emilia Magnani, Universität Tübingen

Abstract

In recent years, deep learning has attracted considerable interest and led to numerous contributions in the scientific community. Deep learning models are typically based on neural networks trained on input-output pairs within finite-dimensional vector spaces, such as $\mathbb{R}^{d_{\text{input}}}$ and $\mathbb{R}^{d_{\text{output}}}$.

In contrast to standard neural networks, a novel framework known as neural operators, introduced by Kovachki et al. in 2020, aims to learn mappings between possibly infinite-dimensional function spaces, such as Banach spaces. This approach is particularly promising for solving or approximating solutions to Partial Differential Equations (PDEs), which are central to modeling physical phenomena.

However, despite their strong performance, predictions made by neural operators may carry significant uncertainty—especially for complex PDEs or extrapolated time domains (e.g., from $t = 0$ to $t = 10$). Therefore, quantifying this uncertainty is crucial for trustworthy deployment in real-world applications.

In this thesis, we explore and compare several methods for uncertainty quantification (UQ) applied to Fourier Neural Operators (FNOs). We demonstrate that the Linearization-based Uncertainty Quantification method for Neural Operators (LUNO) recently proposed by Magnani et al. consistently achieves state-of-the-art performance across various PDE types. Additionally, we show that Monte Carlo Dropout, applied prior to every Fourier layer of the neural operator, serves as a competitive and computationally efficient UQ technique in contrast to deep ensembles.

Acknowledgments

I would like to express my sincere gratitude to Emilia for her continuous support throughout the entire process. I am also grateful to Tobi for his valuable help with the code and implementation aspects of the thesis. Special thanks go to Prof. Philipp Hennig for his structural guidance and insightful feedback. Lastly, I thank Franzi for her assistance with all the organisational matters.

Contents

List of Tables

1. Introduction

In recent years, machine learning has often been used to solve complex problems. Such complex problems can be, for example, solving partial differential equations. Partial differential equations (PDEs) describe complex physical phenomena like the propagation of heat from a heat source or the flow of linear-viscous liquids (Navier-Stokes equations). Given that these equations are intended to model real-world phenomena, they typically involve numerous parameters—such as friction coefficients, flow velocities, fluid viscosities, and material densities—that must be carefully accounted for.

Neural operators or in particular the Fourier Neural Operator (FNO) has proved particularly useful for these problems due to its ability to generalize across different input configurations (Li et al., 2021).
Although Fourier Neural Operators are a preferred choice for learning solution mappings of PDEs, their predictions may still carry uncertainty. Therefore we introduce uncertainty quantification (UQ) to FNOs. The goal of UQ is to capture and model uncertainties in the parameters and the predictions of a given model. There are various methods, such as training several models and then calculating the mean value and the standard deviation of the predictions or perturbing the input with some noise and directly train for multiple predictions via a statistical loss function (Lessig et al., 2023).

The Laplace approximation (LA) is a widely used method to quantify uncertainty in the models parameters. It fits a Gaussian distribution over the point-estimate of a trained model. This Gaussian distribution, specified by a mean and a covariance matrix then allows to draw sample weights and perform inference. Another technique to quantify uncertainty in the models predictions, involves applying probabilistic currying and linearizing a given model to transform a neural operator into a function-valued Gaussian Process (GP), as described by Magnani et al..

Another compelling approach leverages dropout layers to randomly mask weights during inference, effectively reducing model complexity and approximating ensemble predictions—without the overhead of training multiple separate models. This technique forms an implicit ensemble of subnetworks by sampling different dropout masks and averaging their predictions. Since it requires neither training multiple models nor calibrating a parameter like σ, evaluating the performance of Monte Carlo Dropout in the context of FNOs becomes particularly interesting.

The goal of this thesis is to compare different approaches of uncertainty quantification like input perturbations, weight perturbations, the Laplace approximation, the linearized version of the Laplace approximation called LUNO (Magnani et al., 2024), deep ensemble

and Monte Carlo dropout on different kind of 1 and 2-dimensional PDEs. We evaluate different methods using specific metrics and also analyze structures such as the standard deviation. The overarching goal is to identify the best possible method. Furthermore, we seek to assess the effectiveness of Monte Carlo Dropout applied to various components of the network, benchmark it against alternative approaches like deep ensembles, and determine the circumstances under which dropout yields reliable or suboptimal results.

2. Theoretical Background

This chapter deals with the theoretical background needed to understand the fundamental concepts covered in the thesis.

2.1. Partial Differential Equations (PDEs)

Partial differential equations describe many physical phenomena in the real world. A PDE is an equation that depends on the partial derivatives of a function. To get familiar with PDEs, let's start with the heat equation. This equation describes the propagation of heat coming from a source over time, formally it is described as:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x_1^2} + ... + \frac{\partial^2 u}{\partial x_n^2} \tag{2.1.1}$$

where n is the amount of dimensions in the domain, e.g. 2 for a plane. Solving partial differential equations—essentially finding the function u that satisfies the condition stated in (2.1.1)—is a challenging task. That function u is then called the solution of the PDE. The dimension of the PDE is specified by the order of the highest derivative involved. In case of the heat equation, the solution is a function $u : U \times I \to \mathbb{R}$, where U is a open subset of $\mathbb{R}^n$ and I a subinterval of $\mathbb{R}$. Figure 2.1 shows the solution of the 1D-Heat equation as a surface plot. As you can see the temperature is decreasing over time and around the heat source in the spatial domain.

There are various analytical and numerical methods to find or approximate the solution of a given PDE. Popular methods are the finite element method (FEM) and the application of Fourier and Laplace transformations to solve the PDE in the frequency domain (COMSOL, 2016). The choice of method depends on the type of PDE, such as linear, nonlinear, or parabolic, as each requires a specific approach. This complexity is one reason why finding a solution can be challenging, as exemplified by the Navier-Stokes equations.

Numerical methods approximates the solution using a grid, which consists of discrete points that represent the problem in a specific spatial or temporal domain. The finite element method then divides the grid into smaller subdomains called elements, which are typically triangles, squares or higher-dimensional shapes and tries to solve the system for each element with numerical techniques like the Newton-Raphson method [(Dedieu, 2015), COMSOL (2016)]. This process is repeated until the solution converges to a good

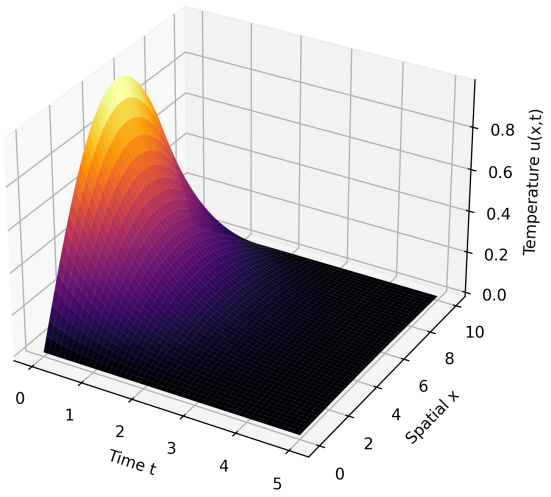

Figure 2.1.: 3D-Surface-Plot of the 1D-Heat Equation (for code see C.1)

approximation. Numerical solvers are dependent on the grid size used to solve the problem, if you want a higher resolution you need to start the solving process again. Since numerical methods require a fine computational grid to guarantee convergence of the iterations, it is unfeasible to run numerical simulations at the scale to replace physical experimentation completely. Especially in the case of climate forecasting, where physical experimentation is not possible, simulations are the only option. Another disadvantage of numerical methods is the requirement of massive computing power to resolve physics at a finer scale as needed in case of climate forecasting (Azizzadenesheli et al., 2024).

In contrast to analytical and numerical methods, neural operators do not directly solve the PDE. Rather, they learn a solution mapping that maps a initial condition or state to another state in t timesteps. This mapping can then be used to make predictions about the appearance of an initial condition in, for example, 10 seconds. This concept will be further elaborated upon in the subsequent chapter. Neural operators enjoy the advantage against numerical solvers as proposed by Kovachki et al. (2021): "Neural operators are the only known class of models that guarantee both discretization-invariance and universal approximation.". This is why it could be beneficial to get familiar with neural operators.

2.2. Neural Operators

A neural operator, like a neural network, processes input data, applies transformations through weighted operations, and produces an output. The output is compared to the

ground truth, and the error is used to adjust the weights iteratively until the optimal parameters are found.

More precisely it is a deep-learning framework, which directly maps between function spaces on bounded domains Kovachki et al. (2021). From a formal point of view, a neural operator is a mapping $\mathbf{F}\colon \mathbb{A} \times \Theta \to \mathbb{U}$, where $\mathbb{A}$ and $\mathbb{U}$ are Banach spaces of functions and Θ is a parameter space. A Banach space is a complete normed linear space, in which every Cauchy sequence converges. In case of learning solutions of partial differential equations (PDE), the neural operator learns a solution mapping $\mathcal{G}$, which maps the input function (e.g, initial condition of a PDE) to the solution of the PDE after t timesteps.

Let $\mathbb{A}$ and $\mathbb{U}$ be Banach spaces. We assume that the input functions $a \in \mathbb{A}$ take values in $\mathbb{R}^{d_a}$ and are defined on the bounded subset $D \subset \mathbb{R}^d$. Similarly, the output functions $u \in \mathbb{U}$ take values in $\mathbb{R}^{d_u}$ and are defined on the bounded domain $D' \subset \mathbb{R}^{d'}$ (Kovachki et al., 2021). Formally, this means that $a : D \to \mathbb{R}^{d_a}$ and $u : D' \to \mathbb{R}^{d_u}$. The structure of a neural operator as introduced by (Kovachki et al., 2021), is the following:

1. **Lifting layer**: The input function gets passed through a lifting layer. This is a pointwise function $\mathbb{R}^{d_a} \to \mathbb{R}^{d_{v_0}}$, which maps the input $\{a : D \to \mathbb{R}^{d_a}\} \mapsto \{v_0 : D \to \mathbb{R}^{d_{v_0}}\}$ to its encoded representation. This is similar to a linear layer in standard neural networks.

2. **Iterative Kernel Integration**: Kovachki et al. proposes an iterative transformation to get from the initial representation $v_0 : D_0 \to \mathbb{R}^{d_{v_0}}$ to the final representation $v_T : D_T \to \mathbb{R}^{d_{v_T}}$. For $t = 0, ..., T-1$ we replace the current representation $v_t : D_t \to \mathbb{R}^{d_{v_t}}$ with the sum of the following components into the next representation $v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}$. In this case, t is the iteration in the model architecture and not the time as stated previously.

   a) **Local linear operator**: The local linear operator $\mathcal{L}$ is a transformation, which only takes information from the neighbourhood of a single point $x \in D_t$. Formally this is a pointwise operation: $\mathcal{L}(v_t)(x) = W * v_t(x)$, where

      i. $v_t(x) \in \mathbb{R}^{d_{v_t}}$ is the current representation of the input function v at the point x,

      ii. $W \in \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}}$ is the weight matrix, which defines the transformation.

   b) **Non-local integral kernel operator**: We define the kernel operator like Li et al. (2021) with:

$$(\mathcal{K}(a;\phi)v_t)(x) := \int_D \kappa(x, y, a(x), a(y); \phi) v_t(y)\, dy, \qquad \forall x \in D, \quad (2.2.1)$$

   where $\kappa_\phi : \mathbb{R}^{2(d+d_a)} \to \mathbb{R}^{d_v \times d_v}$ is a neural network parameterized by $\phi \in \Theta_{\mathcal{K}}$. Unlike the local linear operator, the non-local integral kernel operator captures dependencies over the whole domain, by integrating a kernel function $\kappa(x, y, a(x), a(y); \phi)$, which depends on both spatial locations and features.

c) **Bias function**: The bias function is a simple component, which adds a constant offset to each point x. Formally, $b_t : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}$.

3. **Projection layer**: This layer uses a pointwise function (similar to a linear layer) to map from the last representation v_T to the output function u. Formally, this is a mapping $\mathbb{R}^{d_{v_T}} \to \mathbb{R}^{d_u} : \{v_T : D' \to \mathbb{R}^{d_{v_T}}\} \mapsto \{u : D' \to \mathbb{R}^{d_u}\}$.

As a result we get the solution mapping $\mathcal{G}$, that is learned by the neural operator:

$$\mathcal{G}_\theta := \mathcal{Q} \circ \sigma_T(W_{T-1} + \mathcal{K}_{T-1} + b_{T-1}) \circ ... \circ \sigma_1(W_0 + \mathcal{K}_0 + b_0) \circ \mathcal{P}$$

where $\mathcal{P}$ and $\mathcal{Q}$ are the lifting and projection layers respectively, W_t are the local linear operators, $\mathcal{K}_t$ are the non-local integral kernel operators, b_t are the bias functions and σ_t are the activation functions acting as maps $\mathbb{R}^{v_{t+1}} \to \mathbb{R}^{v_{t+1}}$ in each layer (Kovachki et al., 2021).

In comparison to numerical solvers, neural operators don't depend on the size of the grid of the data, e.g. one can train a neural operator on a grid of size 256×256 and predict on a grid of size 1024×1024 (Kovachki et al., 2021). There are various neural operator architectures, each tailored to address specific kind of problems. Other famous neural operator architectures are the Graph Neural Operator (GNO), the Low-Rank Neural Operator (LNO) and the Fourier Neural Operator (FNO). The main differences between different neural operator architectures are different kinds of integral kernel operators. In this thesis we want to focus on the Fourier neural operator, since it is specialized to generalize across different input configurations due to Fourier transformations and thus is suitable for learning partial differential equations Li et al. (2021).

2.3. Fourier Neural Operators (FNOs)

The Fourier neural operator (FNO) has proved to be very good in learning partial differential equations (PDEs). FNOs, introduced by Li et al. work like the previously introduced neural operators, but instead of a kernel operator they use a convolution operator defined in Fourier space. These kind of neural operators use the Fourier transform to efficiently capture global patterns of the input data, which is similar to how music is broken down into individual frequencies. That allows FNOs to model long-range dependencies in PDE solutions (e.g. temperature distribution over an entire room) directly in frequency space instead of using local approximations such as finite differences. So far, we know convolutions from convolutional neural networks (CNNs), which are neural networks specialized in processing data in a grid-like topology (e.g images) (Marques et al., 2022). When learning the evolution of a PDE over time, which essentially involves an image-to-image mapping between the initial condition and the condition after t timesteps, it makes sense to use a convolution operator as well. This convolution operator, as already mentioned, is defined by the discrete Fourier transform $\mathcal{F} : L^2(D; \mathbb{C}^n) \to l^2(\mathbb{Z}^d; \mathbb{C}^n)$ of a function $v : D \to \mathbb{C}^n$ and its inverse $\mathcal{F}^{-1}$ with $D = \mathbb{T}^d$ is the unit torus and $L^2(D; \mathbb{C}^n)$ is a

Hilbert space with the scalar product $\langle f, g \rangle = \int_{(D;\mathbb{C}^n)} f(x)g(x)dx$. The discrete Fourier transform is used as a foundation for methods like the Fast Fourier Transform (FFT) in order to improve computational performance [Kovachki et al. (2021), Li et al. (2021)]:

$$(\mathcal{F}v)_j(k) = \langle v_j, \psi_k \rangle_{L^2(D;\mathbb{C}^k)}, \qquad j \in \{1, ..., n\}, \quad k \in \mathbb{Z}^d \tag{2.3.1}$$

$$(\mathcal{F}^{-1}w)_j(x) = \sum_{k \in \mathbb{Z}^d} w_j(k)\psi_k(x), \quad j \in \{1, ..., n\}, \quad x \in D \tag{2.3.2}$$

where for each $k \in \mathbb{Z}^d$, we define

$$\psi_k(x) = e^{2\pi i k_1 x_1} \cdots e^{2\pi i k_d x_d} \qquad \forall x \in D \tag{2.3.3}$$

with $i = \sqrt{-1}$ the imaginary unit. As already mentioned this convolution operator replaces the integral kernel operator defined in 2.2.1. This Fourier integral operator $\mathcal{K}$ is then given by:

$$(\mathcal{K}(\phi)_{v_t})(x) = \mathcal{F}^{-1}(R_\phi * (\mathcal{F}v_t))(x) \qquad \forall x \in D \tag{2.3.4}$$

where R_ϕ is the Fourier transform of a periodic function $\kappa : \overline{D} \to \mathcal{R}^{d_v \times d_v}$ parameterized by $\phi \in \Phi_\mathcal{K}$. For every iterative update in each layer, we have

$$v_{t+1} := \sigma(W v_t(x) + \mathcal{F}^{-1}(R_\phi * (\mathcal{F}v_t)(x))) \qquad \forall x \in D$$

Kovachki et al. proposes to truncate the Fourier series at a maximal number of modes

$$k_{\max} = Z_{k_{\max}} = |\{k \in \mathbb{Z}^d : |k_j| \leq k_{\max,j} \quad \text{for} \quad j = 1, ..., d\}|,$$

which improves the empirical performance, as higher-frequency modes typically carry less relevant information. This is achieved by applying the linear transform R in frequency domain. For our experiments, we will truncate the Fourier series at 12 modes. The full architecture then looks like this:

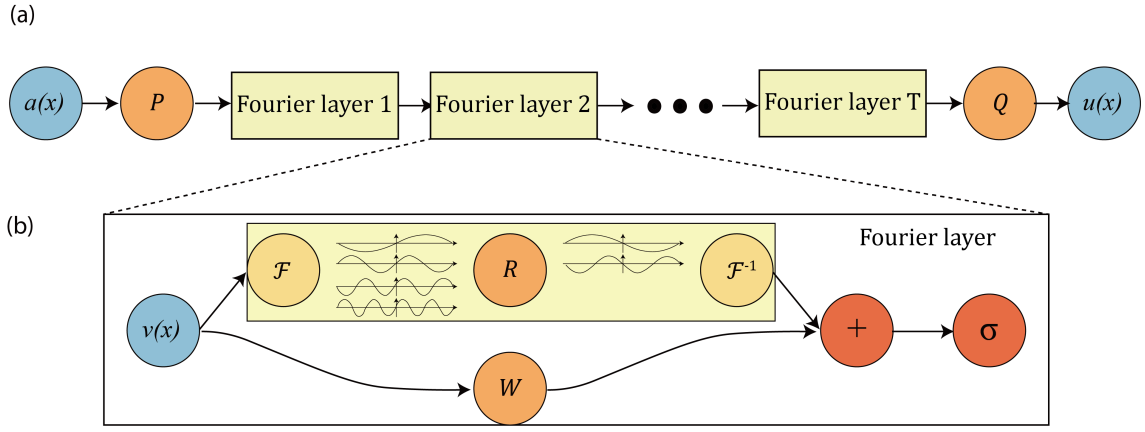

Figure 2.2.: Architecture of the Fourier Neural Operator. Source: Reproduced from Li et al. (2021), Fourier Neural Operator for Parametric Partial Differential Equations, ICLR 2021, licensed under CC BY 4.0.

2.4. Gaussian Processes (GP)

A Gaussian Process (GP) is a family of stochastic processes in which any finite subset of random variables follows a joint multivariate normal distribution [(Rasmussen and Williams, 2006), (García López et al., 2024)]. It is characterized by a mean function $m(x)$ and a covariance function (kernel) $k(x, x')$ that defines the correlation between function values:

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)], \\ k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))] \end{aligned}$$

These functions specify the Gaussian process:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

We will need Gaussian processes later for methods like LUNO, which transform a FNO into a function-valued Gaussian process (Magnani et al., 2024).

3. Bayesian Deep Learning (BDL)

In order to model uncertainty in neural networks, we need to find a probability distribution for the parameters of our model. This can be achieved using Bayesian Deep Learning. This is different to standard deep learning, where we want to compute the optimal parameters in order to get one set of weights. Bayesian Deep Learning is based on two simple ideas. The frequentist paradigm is predicated on the interpretation of probability as the long-run frequency of an event as the number of samples approaches infinity. In contrast, the Bayesian perspective views probability as a measure of belief in an event's occurrence. Additionally, prior beliefs influence the posterior distribution, incorporating prior knowledge into inference (Jospin et al., 2022). This is also what the famous Bayes' theorem states:

$$P(H|D) = \frac{P(D|H) * P(H)}{P(D)} = \frac{P(D|H) * P(H)}{\int_H P(D|H)\, dH}, \tag{3.0.1}$$

where H is a hypothesis with an associated prior belief, and D is the data used to update this belief (Jospin et al., 2022). This belief is about how our parameters should look like to fit the model to our data. $P(H)$ is the prior and $P(D) = \int_H P(D|H)\, dH$ the evidence, while $P(H|D)$ is the posterior. $P(D|H)$, the likelihood, represents the probability of the data given the hypothesis and encodes the statistical uncertainty in the model, i.e., the uncertainty caused by noise in the process (Jospin et al., 2022). Or in words:

$$\text{Posterior} = \frac{\text{Likelihood} * \text{Prior}}{\text{Evidence}}$$

In order to design a Bayesian Neural Network (BNN), we have to choose a statistical model $p(x|\theta)$, like a neural network or in our case a neural operator where $\theta \in \Theta$ are the parameters of the model. Then we have to choose the prior distribution $p(\theta)$ over the model parameters, which is commonly Gaussian (Magnani et al., 2024). The model parametrization is our hypothesis H and the training set is the data D, we denote the parameters as $\theta \in \Theta = \mathbb{R}^n$ and the training set as $D := \{(x_i \in \mathbb{R}^{d_x}, y_i \in \mathbb{R}^{d_y})\}_{i=1}^N$. The goal is then to approximate the posterior $p(\theta|D)$ according to 3.0.1 with

$$p(\theta|D) = \frac{p(D|\theta) * p(\theta)}{\int_\theta p(D|\theta)p(\theta)\, d\theta} \propto p(D|\theta) * p(\theta) \tag{3.0.2}$$

This is also called the unnormalized posterior. Maximizing that product with respect to θ is the maximum a-posteriori (MAP) estimate:

$$\theta_{\text{MAP}} = \arg\max_\theta p(\theta|D) = \arg\max_\theta p(D|\theta) * p(\theta)$$

Maximizing a product is challenging and can result in extremely small or large values. Therefore, we work with the log-likelihood and the log-prior instead:

$$\theta_{\text{MAP}} = \arg\max_{\theta} \log(p(D|\theta)) + \log(p(\theta))$$

The maximum a-posteriori (MAP) estimate can be computed using standard optimization methods like stochastic gradient descent (SGD) (Beznosikov et al., 2023) or the Adam optimizer (Kingma and Ba, 2017). This is similar to training a standard Deep Neural Network (DNN) with an additional regularizer (the log-prior). Common methods for obtaining the approximation of the posterior are Variational Inference (VI) or Markov Chain Monte Carlo (MCMC) (Li et al., 2024). Variational inference (Ganguly and Earp, 2021) tries to find another distribution $q(\theta)$ that is "close" to the posterior and tractable. This can be done by using the KL-Divergence (Kullback-Leibler divergence) as a measure of closeness (Bernstein, 2021):

$$D_{\text{KL}}(p||q) = \sum p(\theta) * \log \frac{p(\theta)}{q(\theta)}$$

In contrast, MCMC is a chain of samples $\theta \rightarrow \theta_{t+1} \rightarrow \theta_{t+2} \rightarrow \theta_{t+3}...$ that converge to the posterior $p(\theta|D)$ (Ulm, Ulm). However, this method is computationally inefficient. Therefore, the Laplace approximation is employed. The Laplace approximation computes the weights in a manner analogous to that of a standard deep neural network, using conventional optimization methods. Subsequent to identifying a point estimate, a Gaussian is fitted over it, and weights can be sampled thereafter. The Laplace approximation is explained in greater detail in Chapter 4.

4. Uncertainty Quantification for Deep Neural Networks (DNN)

Since neural operators only provide an approximation of the solution mapping, the correctness of the prediction cannot always be guaranteed. This is particularly critical when learning the solution mapping of PDEs in real-world applications. In such cases, it is highly beneficial to quantify the uncertainty of the prediction. When discussing uncertainty quantification, several approaches exist to quantify uncertainty in predictive models. One approach involves capturing uncertainty in the model parameters, such as the weights of a neural network, while another focuses on quantifying uncertainty in the predictions themselves. This chapter aims to provide an overview of various techniques employed to capture uncertainty both in model parameters and in the predictions.

4.1. Laplace Approximation

The Laplace approximation is an efficient method to capture uncertainty in models that have already been trained. As already mentioned, the Laplace approximation provides a Gaussian approximation centered around the MAP estimate. This Gaussian approximation is characterized by a Gaussian distribution $\mathcal{N}(\theta; \theta_{\text{MAP}}, \Sigma)$, where
$\Sigma := (\nabla^2_\theta \mathcal{L}(D, \theta)|_{\theta_{\text{MAP}}})^{-1}$ and $\mathcal{L}(D; \theta)$ is the loss-function used in the training process (Daxberger et al., 2022). The Hessian of a neural network has the following structure:

$$\nabla^2_\theta \mathcal{L}(D, \theta) := \begin{bmatrix} \frac{\partial^2 \mathcal{L}}{\partial \theta_1^2} & \frac{\partial^2 \mathcal{L}}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 \mathcal{L}}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \mathcal{L}}{\partial \theta_2^2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial \theta_2 \partial \theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 \mathcal{L}}{\partial \theta_n \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial \theta_n^2} \end{bmatrix}$$

It contains all second-order partial derivatives of the loss function with respect to the weights and provides insights into the curvature of the loss landscape.

Daxberger et al. provided a derivation of the Laplace approximation, which is used as a reference in the following. To obtain the Laplace approximation of the posterior, we can rewrite the evidence as $\int \exp(\log(p(D|\theta) * p(\theta))\, d\theta$. Using the second-order Taylor expansion of the unnormalized posterior around θ_{MAP} gives

$$\log(p(D|\theta) * p(\theta)) \approx p(D|\theta_{\text{MAP}}) * p(\theta_{\text{MAP}}) - \frac{1}{2}(\theta - \theta_{\text{MAP}})^T \Lambda (\theta - \theta_{\text{MAP}}), \qquad (4.1.1)$$

with $\Lambda := -\nabla^2 p(D|\theta) * p(\theta)|_{\theta_{\text{MAP}}}$ the Hessian matrix of the unnormalized posterior evaluated at θ_{MAP}. To approximate the evidence, one can use 4.1.1 to get:

$$\begin{aligned} Z &\approx \exp\Big(\log\big(p(D|\theta_{\text{MAP}}) * p(\theta_{\text{MAP}})\big)\Big) \int \exp\left((-\frac{1}{2}(\theta-\theta_{\text{MAP}})^T \Lambda(\theta-\theta_{\text{MAP}}))\right) d\theta \\ &= p(D|\theta_{\text{MAP}}) * p(\theta_{\text{MAP}}) \frac{(2\pi)^{\frac{d}{2}}}{\det(\Lambda)^{\frac{1}{2}}} \end{aligned}$$

Normalizing $p(D|\theta) * p(\theta)$ with the evidence to obtain the posterior results in:

$$p(\theta|D) = \frac{1}{Z} p(D|\theta) * p(\theta) \approx \frac{\det(\Lambda)^{\frac{1}{2}}}{(2\pi)^{\frac{d}{2}}} \exp\left(-\frac{1}{2}(\theta-\theta_{\text{MAP}})^T \Lambda(\theta-\theta_{\text{MAP}})\right),$$

which can be immediately identified as a Gaussian distribution $\mathcal{N}(\theta; \theta_{\text{MAP}}, \Sigma)$ with mean θ_{MAP} and covariance matrix $\Sigma = \Lambda^{-1}$ (Daxberger et al., 2022). Finally, we can say that

$$p(\theta|D) \approx \mathcal{N}(\theta; \theta_{\text{MAP}}, \Sigma) \qquad \text{with} \qquad \Sigma = (\nabla^2_\theta \mathcal{L}(D, \theta)|_{\theta_{\text{MAP}}})^{-1}.$$

To summarize the findings in the previous lines: the Laplace approximation constructs a Gaussian posterior distribution centered at θ_{MAP}, with a covariance matrix derived from the inverse of the Hessian matrix of the log-posterior. The Hessian, which captures the curvature of the log-posterior, quantifies the uncertainty around θ_{MAP}. However, computing the Hessian for models such as neural operators is computationally intensive as we have to compute all second-order partial derivatives of the loss function. As noted earlier, the Hessian comprises all second-order partial derivatives of the loss function, resulting in a quadratic growth in size with respect to the number of parameters (Daxberger et al., 2022). Additionally the Hessian matrix is not guaranteed to be positive-definite, which complicates its direct use as a covariance matrix. Solutions for that can be the Fisher information matrix (Amari, 1998) defined by

$$F := \sum_{n=1}^{N} \mathbb{E}_{\hat{y} \sim p(y|f_\theta(x_n))}[(\nabla_\theta \log(p(\hat{y}|f_\theta(x_n))|_{\theta_{\text{MAP}}})(\nabla_\theta \log(p(\hat{y}|f_\theta(x_n))|_{\theta_{\text{MAP}}})^T] \quad (4.1.2)$$

or the generalized Gauss-Newton Matrix (GGN) (Schraudolph, 2002) defined by

$$G := \sum_{n=1}^{N} J(x_n)(\nabla^2_f \log(p(y_n|f)|_{f=f_{\theta_{\text{MAP}}}(x_n)}) J(x_n)^T, \quad (4.1.3)$$

where $J(x_n) = \nabla_\theta f_\theta(x_n)|_{\theta_{\text{MAP}}}$ is the NN's Jacobian matrix (Daxberger et al., 2022). The GGN is also utilized in the implementation for the experiments in chapter 6. Now that we can compute the approximations of the Hessian quite fast, we have another problem: storing the approximations, which is despite using the Fishers or GGN still quadratically large (Daxberger et al., 2022). To address that problem one can factorize the matrix according to different techniques. The code utilized for the results employs a technique

known as low-rank approximation. This technique can be understood as an approximation of a matrix by one with a lower rank than the original matrix. Let A be a $m \times n$ Matrix then the approximation is given by

$$A_{m\times n} \approx B_{m\times k}\, C_{k\times n}.$$

That would only need $k * (m + n)$ space, while you had to store $m * n$ entries before (Kumar and Shneider, 2016).

The important thing now is to predict with the Laplace approximation. This can be done with Monte Carlo integration using S samples $(\theta_s)_{s=1}^{S}$ from

$$p(\theta|D) : p(y|f(x_*), D) \approx S^{-1}\sum_{s=1}^{S} p(y|f_{\theta_s}(x_*)), \tag{4.1.4}$$

which is basically drawing S weight samples from the Gaussian distribution and then predicting for each weight sample resulting in S ensemble predictions (Daxberger et al., 2022). We call this approach the sample-based Laplace approximation. Another approach involves linearizing the network.

4.2. Linearized Laplace Approximation

The linearized Laplace approximation is variant of the sample-based version. As the name suggests, the goal is to linearize the network using a first-order Taylor approximation around θ_{MAP}:

$$f(x, \theta) \approx f(x, \theta_{\text{MAP}}) + J_f(\theta_{\text{MAP}}) * (\theta - \theta_{\text{MAP}})$$

This linearization transforms the output distribution into a Gaussian Process (GP) over the function values. In particular, we obtain a posterior distribution over the networks outputs with a Gaussian structure. The networks output distribution is then given by

$$\begin{aligned} p(f(x,\theta)|f(x,\theta_{\text{MAP}}), x, D) &= \int \delta(f(x,\theta) - f(x,\theta_{\text{MAP}}))q(\theta|D)d\theta \\ &\approx \mathcal{N}(f(x,\theta)|f(x,\theta_{\text{MAP}}), J_f^T \Sigma J_f) \end{aligned}$$

with $q(\theta|D) \sim \mathcal{N}(\theta|\theta_{\text{MAP}}, \Sigma)$ (Daxberger et al., 2022). This method is more computationally demanding because it requires computing the Jacobians of the network. Magnani et al. proposes a method called LUNO, which uses the linearized Laplace approximation on neural operators.

4.3. Dropout Monte Carlo

The Dropout layer, initially proposed by Hinton et al., is a method for avoiding overfitting by randomly setting specific network weights to zero. This is achieved by implementing

Bernoulli masks with a specified probability to specific layers of the network. A common dropout rate of 0.5 implies that each weight is independently set to zero with a probability of 50%. In 2016, Gal and Ghahramani demonstrated that implementing dropout prior to each weight layer of a neural network is mathematically equivalent to an variational approximation of a deep Gaussian process. The main idea is to train a model with dropout enabled and then perform evaluation by forwarding the same input multiple times, each time with a different dropout mask but using the same dropout rate. The result is an ensemble of predictions, from which we can compute the mean and standard deviation. This approach is particularly interesting for us, as it allows ensemble-like predictions without the need to train multiple models, leveraging dropout as a form of approximate Bayesian inference. The predictive mean under dropout is given by (Gal and Ghahramani, 2016):

$$\mathbb{E}_{q(y|x)}[y] \approx \frac{1}{T} \sum_{t=1}^{T} \hat{y}(x, \hat{z}_{1,t}, \ldots, \hat{z}_{L,t})$$

with T the number of forward passes. The dropout masks $\hat{z}_{i,t}$ are sampled as

$$\hat{z}_{i,t} \sim \mathrm{Bern}(p_i)$$

with p_i the probability of weights being dropped out.

4.4. Deep Ensemble

Deep Ensemble is the most simple method to obtain uncertainty. The idea is to train multiple models with different seeds and pass an input to all of the models resulting in n predictions, where n is the number of trained models. We want to compare how ensemble predictions behave for different ensemble sizes and also how the methods performs in contrast to dropout monte carlo.

4.5. Input Perturbations

This method represents another approach to incorporating uncertainty into our model. The core idea is to introduce noise to each input value of the function $u_n(x,t)$ before passing it to the network. This noise follows a Gaussian distribution, $\epsilon_{x,t} \sim \mathcal{N}(0, \sigma^2)$, with a calibrated parameter σ (Magnani et al., 2024). Pathak et al. proposes a method to process a batch of perturbed inputs and compute the mean of the corresponding outputs.

4.6. Weight Perturbations

Weight perturbations offer a similar approach to introduce uncertainty, but instead of modifying inputs, noise is directly added to the model weights. The idea is to perturb the parameters of the network, typically by adding Gaussian noise $\epsilon_W \sim \mathcal{N}(0, \sigma^2)$ to the weight matrices before inference is made. Similarly to input perturbations, multiple forward passes with perturbed weights can be averaged to obtain a more robust estimate of the output distribution.

5. Uncertainty Quantification for Neural Operators

As already discussed in the previous chapters, neural operators are very different in terms of architecture compared to standard neural networks. In particular, we are not mapping from $\mathbb{R}^{d_x}$ to $\mathbb{R}^{d_y}$, but from a Banach space to another Banach space. That means, that the input and output of the neural operator are functions.

5.1. The LUNO Method

The so-called LUNO method extends the idea of model linearization from standard neural networks to neural operators and was proposed by Magnani et al.. The difference to neural networks is, by definition, that the output of neural operators are potentially infinite-dimensional Banach spaces of functions instead of $\mathbb{R}^{d_y}$ (Magnani et al., 2024).

To handle this, Magnani et al. apply a technique called uncurrying (reverse currying) to transform the neural operator

$$F\colon \mathbb{A} \times \Theta \to \mathbb{U}$$

into a function with values in $\mathbb{R}^{d_u}$

$$f\colon (\mathbb{A} \times \mathbb{D}_{\mathbb{U}}) \times \Theta \to \mathbb{R}^{d_u}, \quad ((a, x), \theta) \mapsto F(a, \theta)(x).$$

After obtaining a Gaussian posterior over the parameters, $\theta \sim \mathcal{N}(\mu, \Sigma)$—e.g., via Laplace approximation—we linearize the model around the mean μ:

$$f((a, x), \theta) \approx f_{\mu}^{\text{lin}}((a, x), \theta) := f((a, x), \mu) + J_f(\mu) \cdot (\theta - \mu),$$

where $J_f(\mu)$ denotes the Jacobian of f with respect to θ evaluated at μ.

This linearization yields an approximate Gaussian process belief over f:

$$f \sim \mathcal{GP}(m, K),$$

with

$$m(a, x) = f(a, \mu)(x), \quad K((a_1, x_1), (a_2, x_2)) = J_f(\mu)\Sigma J_f(\mu)^{\top}.$$

Using probabilistic currying, one constructs a Gaussian random operator from f. The operator is then given by:

$$F\colon \mathbb{A} \times \Theta \to \mathbb{U}, \quad (a, \theta) \mapsto (x \mapsto f((a, x), \theta)),$$

and defines a $\mathbb{U}$-valued Gaussian process with:

$$\mathbb{E}[F(a)(x)] = F(a, \mu)(x),$$
$$\mathrm{Cov}[F(a_1)(x_1), F(a_2)(x_2)] = J_{F(a_1,\theta)(x_1)}(\mu)\Sigma J_{F(a_2,\theta)(x_2)}(\mu)^\top.$$

In the case of Fourier neural operators, the Gaussian belief is restricted to the parameters of the last Fourier block, denoted by $w_{L-1} := (R^{L-1}, W^{L-1})$.

Applying LUNO to the FNO, the resulting operator takes the form:

$$F(a)(x) = \hat{q}(m_{z^{(L-1)}}(x)) + \left(J_{\hat{q}(m_{z^{(L-1)}}(x))}(x) \cdot (z^{(L-1)} - m_{z^{(L-1)}})\right),$$

with $\hat{q} = q(\cdot, \theta_q) \circ \sigma_{(L-1)}$ the projection layer with its corresponding activation function and

$$F(a) \sim \mathcal{GP}(m_a, K_a),$$

where

$$m_a(x) = F(a, \theta_{\text{MAP}})(x),$$
$$K_a(x_1, x_2) = J_{\hat{q}(m_{z^{(L-1)}})}(x_1) \cdot K_{z^{(L-1)}}(x_1, x_2) \cdot J_{\hat{q}(m_{z^{(L-1)}})}(x_2)^\top,$$

and $z^{(L-1)} \sim \mathcal{GP}(m_{z^{(L-1)}}, K_{z^{(L-1)}})$ is a multi-output parametric Gaussian process (Magnani et al., 2024).

5.2. Monte Carlo Dropout

Applying dropout to an FNO is considerably simpler than implementing the LUNO method. The FNOs used in our experiments consist of four layers, with dropout applied to both the real and imaginary parts of the Fourier weights in each layer. For reference, see fig. 2.2. In the case of Fourier block dropout, the masks are applied to the input before the Fourier transform $\mathcal{F}$, leaving the skip connection W unaffected. For full-network dropout, the masks are additionally applied before the lifting and projection layers $\mathcal{P}$ and $\mathcal{Q}$. For an FNO with a single Fourier layer and dropout applied exclusively to the Fourier layer, the solution operator takes the form

$$\mathcal{G} = \mathcal{Q} \circ \sigma(W + \mathcal{K} \cdot \hat{z} + b) \circ \mathcal{P}$$

In contrast, when dropout is applied throughout the entire network, the solution operator becomes

$$\mathcal{G} = \mathcal{Q} \cdot \hat{z}_3 \circ \sigma(W + \mathcal{K} \cdot \hat{z}_2 + b) \circ \mathcal{P} \cdot \hat{z}_1$$

with $\hat{z}_i \sim \mathrm{Bern}(p_i)$.

6. Experiments

In this chapter, a comparison is finally made between the various methods described in the previous chapter and different types of PDEs.

6.1. Considered PDEs

Burgers Equation The Burgers equation is given by:

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} &= \mu\frac{\partial^2 u}{\partial x^2} \\ \Leftrightarrow \frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial}{\partial x}(u^2) &= \mu\frac{\partial^2 u}{\partial x^2} \end{aligned}$$

where μ is the viscosity parameter. Because of the similarity to the nonlinear part of the Navier-Stokes equation, the Burgers equation can be interpreted as a 1-dimensional flow (Orlandi, 2000).

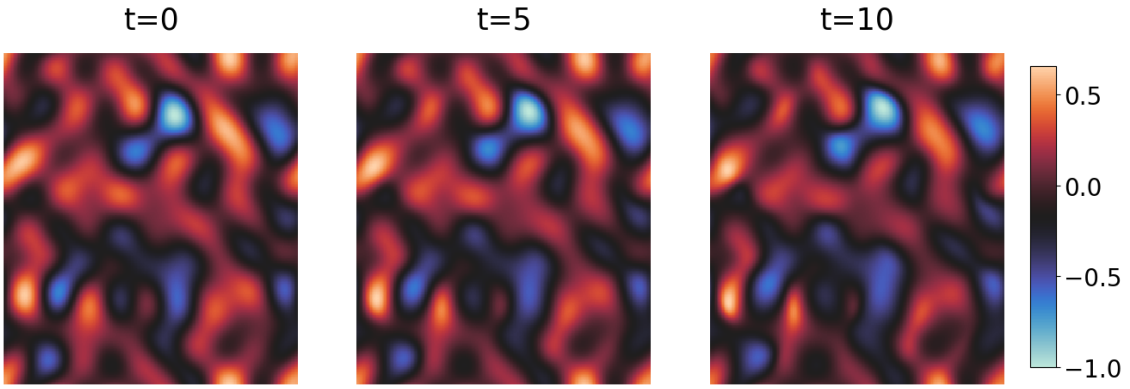

Figure 6.1.: Burgers Equation in 2d

Diffusion Advection The diffusion-advection PDE consists of two parts (Crank, 1975):

1. Diffusion, which describes the transport trough molecular movement and
2. Advection, which describes the transport though a flow.

Putting both terms together, the diffusion-advection equation could describe how temperature is moving through water for example. For one-dimension we have:

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = D\frac{\partial^2 u}{\partial x^2}$$

with v the velocity of the flow and D the diffusion coefficient (Cushman-Roisin, Cushman-Roisin). For more than one dimension the equation is given by:

$$\frac{\partial u}{\partial t} + \mathrm{v} \cdot \nabla u = D\nabla^2 u$$

where v is the velocity vector in n dimensions and $\nabla^2 u$ the Laplace operator.

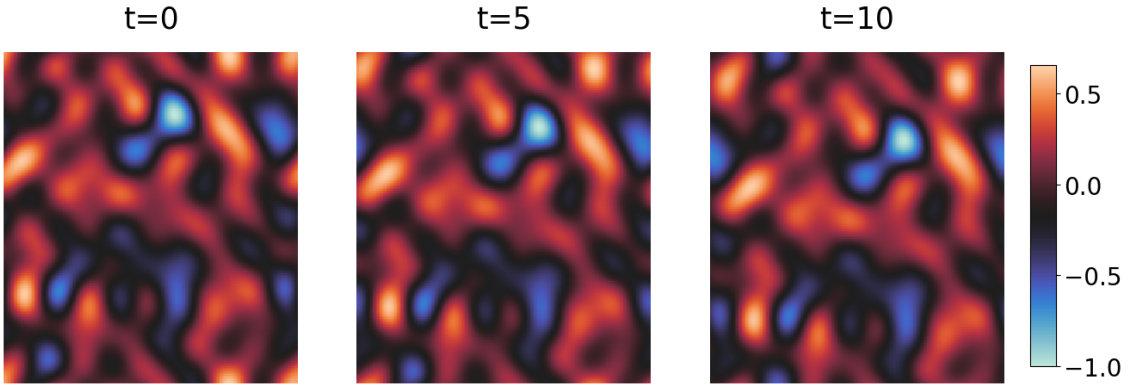

Figure 6.2.: Diffusion Advection Equation in 2d

Kolmogorov Flow In 2D the Kolmogorov flow is a variant of the incrompressible Navier Stokes flow and given by

$$\frac{\partial u}{\partial t} + \nabla \cdot (u \otimes u) = \nu\nabla^2 u - \frac{1}{\rho}\nabla p + \mathrm{f}$$

with ν the kinematic viscosity, ρ the fluid density, p the pressure field and f the external force Lippe et al. (2023). The Kolmogorov flow is known for being chaotic for high Reynolds numbers. The Reynolds number describes the ratio of inertial forces to viscous forces and is defined by

$$Re = \frac{vL}{\nu}$$

with L the length of the object (Lippe et al., 2023). Given the potential complexity of this equation, it is an ideal candidate for benchmarking different uncertainty quantification methods.

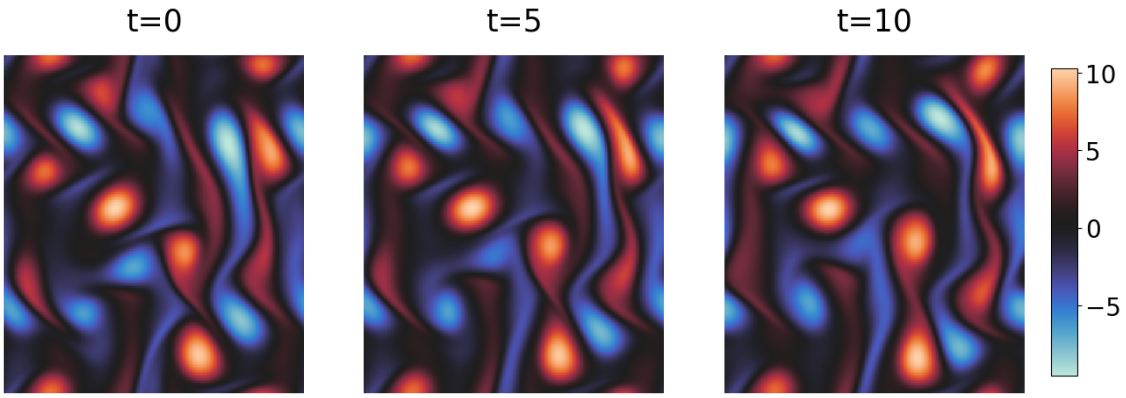

Figure 6.3.: Kolmogorov Flow in 2d

6.2. Data Configurations

The data is generated with ApeBench (Koehler et al., 2024) with the following configuration.

Equation	Dim.	Training Traj.	Valid. Traj.	Test Traj.	Spatial Res.	Temp. Res.
Kolmo. Flow	2D	50	20	20	120	60
Diff.-Adv.	1D	50	20	20	256	80
	2D	50	20	20	120	60
Burgers	1D	50	20	20	256	80
	2D	50	20	20	120	60

Table 6.1.: Data configurations for different PDEs and dimensions.

6.3. Model

The present study utilizes an FNO (see section 2.3) with four layers, each measuring 18 width and possessing 12 modes within the Fourier block, as previously recommended by Koehler et al. (2024). The AdamW optimizer (Loshchilov and Hutter, 2019) is employed in conjunction with a cosine decay learning rate scheduler, inclusive of a warmup phase. The gelu activation function (Hendrycks and Gimpel, 2023) is used and applied to two output projections. The input data comprises 10 timesteps, with the subsequent time step being predicted. The code of Magnani et al. (2024) is utilized and extended by the dropout layer that was implemented by the author. The implementation was executed in

JAX (Bradbury et al., 2018), Flax NNX (Heek et al., 2024), and Optax (DeepMind et al., 2020). The code can be found at [1].

Layer	Shape	#Parameters
Input Projection	Dense ($10 \to 18$)	198
FNO Block 1	Conv + Skip ($2\times12\times12\times18\times18 + 18\times18 + 18$)	93.654
FNO Block 2	Conv + Skip	93.654
FNO Block 3	Conv + Skip	93.654
FNO Block 4	Conv + Skip	93.654
Output Projection 1	Dense ($18 \to 36$)	684
Output Projection 2	Dense ($36 \to 1$)	37
Total	—	**375.535**

Table 6.2.: Number of parameters (including bias) in a 2D Fourier Neural Operator (FNO) with four Fourier blocks, using `modes=12` and `width=18`.

6.4. Training

We train the FNOs for 20 epochs on 25 trajectories and 50 training, 20 test and 20 validation samples with a batch size of 3. We use the AdamW optimizer with a learning rate of 1×10^{-3} and a weight decay aka regularizer rate of 1×10^{-4}. All experiments were executed on an NVIDIA RTX 5090 with CUDA 12.8, except for the evaluation using the LUNO method, which probably failed because JAX (v0.5.3, the full environment can be found in appendix D) is not yet built with CUDA 12.8 support. Therefore the CPU was utilized for LUNO evaluation. A fixed random seed (420) was used in all experiments to ensure reproducibility. The seed was chosen arbitrarily by modifying the commonly used default value 42.

6.5. Evaluation Details

Input Perturbations

As already explained, the noise of the input perturbations follows a Gaussian distribution $\epsilon_{x,t} \sim \mathcal{N}(0, \sigma^2)$, where σ is calibrated. For these experiments, we are using grid search (Liashchynskyi and Liashchynskyi, 2019) to find the optimal σ by minimizing the marginal negative log-likelihood.

[1] `https://github.com/2bys/lugano-experiments/tree/dropout_nicolas`

Ensemble

Since ensemble predictions is the most simple uncertainty quantification method, we analyze how predictions behave for different ensemble sizes. The evaluation is done with ensemble sizes of 5, 10, 20, and 30, where each model is trained with a different seed. Apart from the increased computational cost of training 20 models, the evaluation using ensemble predictions is as fast as dropout, as no additional parameter calibration is required.

Weight Perturbations

This method follows the same logic as input perturbations. We add some Gaussian noise $\epsilon_{x,t} \sim \mathcal{N}(0, \sigma^2)$ to the weights in the Fourier block and average over the predictions. The parameter σ is calibrated by grid search, as for input perturbations.

Laplace Approximation

As mentioned in Chapter (4), we use the GGN (4.1.3) for the covariance matrix and approximate it using a low-rank approximation. This low-rank approximation is given by $\mathrm{H}_{\mathrm{GGN}} = VV^T$, with $\mathrm{rank}(V) = 100$. Magnani et al. used an approach of Dangel et al. to select the largest eigenspace of the GGN and apply an isotropic Gaussian prior to all weights. This ensures that, within regions of high uncertainty, the model reverts to the prior belief, while remaining a Gaussian distribution. This Gaussian distribution is given by

$$\mathcal{N}(\mathcal{F}(x, \theta), J_f \Sigma J_f^T)$$

with $\Sigma = (nVV^T + \sigma I)^{-1}$ and n the number of training samples. J_f is the Jacobian of the model with respect to the weights θ. The isotropic Gaussian prior is given by $\mathcal{N}(\theta_{\mathrm{MAP}}, \sigma^2)$. As with input perturbations, the σ parameter is calibrated by grid search.

Dropout

In contrast to the other methods, the evaluation with dropout does not require a calibration, since there is no σ parameter calibrate. This makes the evaluation highly efficient. We compare FNOs where the dropout masks are only applied to the real and imaginary weights in the Fourier block with FNOs where the masks are also applied to the lifting and projection layers. We use dropout rates of 0.2, 0.5, 0.7, and 0.9. To obtain an ensemble of predictions, we pass a batch of inputs through the network multiple times with different dropout masks at the same dropout rate and average the results.

Rate	Scope	Active Params	Zeroed Params
0.2	Fourier only	300.886	74.649
	All layers	300.702	74.833
0.5	Fourier only	188.911	186.624
	All layers	188.452	187.083
0.7	Fourier only	114.262	261.273
	All layers	113.619	261.916
0.9	Fourier only	39.612	335.923
	All layers	38.785	336.750

Table 6.3.: Estimated number of active and zeroed parameters per training step under different Dropout settings for FNO setup table 6.2. Calculations assume Dropout is applied to layer inputs and affects the same number of parameters each time.

Sample Based Methods

For all sample based methods (e.g. Dropout, input-and weight perturbations, sample-based Laplace approximation) we forward a batch of 100 samples trough the network and average over the predictions.

6.6. Considered Metrics

Root Mean Squared Error (RSME)

The RMSE is given by

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

and computes the average squared error between the ensemble mean and the ground truth. A lower RMSE indicates a better prediction, while a RMSE of 0 signifies the ideal value.

Marginal Negative Log-Likelihood (NLL)

The NLL is given by

$$\text{NLL} = -\sum_{i=1}^{N} \log\left(\frac{1}{2\pi\sigma_i^2}\exp\left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma_i^2}\right)\right)$$

and quantifies how well the model fits the given data. Lower values indicate a better fit, while higher values suggest a poorer fit. Since the NLL considers both the predicted mean

and the associated uncertainty, a model with poorly calibrated uncertainty can have a high NLL despite having a low RMSE.

χ^2-Statistics (Q-Value)

This metric quantifies the average squared error normalized by the variance. The Q-value is always positive and serves as an indicator of calibration quality. A value close to 1 signifies well-calibrated uncertainty, while values greater than 1 suggest overconfidence, and values below 1 indicate underconfidence. In UQ, it is preferable to have an underconfident rather than an overconfident model, as an overconfident model may make definitive predictions without adequately accounting for uncertainty (Magnani et al., 2024). This can be particularly critical in industries such as healthcare. The Q-value is given by

$$\mathrm{Q} = \frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \hat{y}_i)^2}{\sigma_i^2}$$

where σ^2 is the variance.

6.7. Results

Burgers Equation

Method	Avg. σ	NLL (↓)	χ^2	RMSE (↓)
Input pert.	2.68×10^{-2}	-2.243	**0.961**	2.52×10^{-2}
Weight pert.	2.78×10^{-2}	-2.282	0.777	2.48×10^{-2}
Ensemble (N Models)				
5	6.29×10^{-3}	3.782	16.448	2.50×10^{-2}
10	6.51×10^{-3}	1.471	11.624	2.57×10^{-2}
20	6.65×10^{-3}	0.593	9.780	2.57×10^{-2}
30	6.60×10^{-3}	0.558	9.726	2.55×10^{-2}
LUNO	2.64×10^{-2}	-2.287	0.869	$\mathbf{2.47 \times 10^{-2}}$
Laplace	2.71×10^{-2}	$\mathbf{-2.294}$	0.810	2.48×10^{-2}
Dropout (Fourier Block)				
$p = 0.2$	1.04×10^{-1}	-0.786	5.830	2.50×10^{-2}
$p = 0.5$	1.81×10^{-2}	-2.041	2.212	2.72×10^{-2}
$p = 0.7$	2.33×10^{-2}	-2.133	1.511	2.98×10^{-2}
$p = 0.9$	3.55×10^{-2}	-2.127	0.676	3.06×10^{-2}
Dropout (Full Network)				
$p = 0.2$	5.25×10^{-2}	-1.927	0.547	3.62×10^{-2}
$p = 0.5$	8.99×10^{-2}	-1.429	0.445	7.14×10^{-2}
$p = 0.7$	1.31×10^{-1}	-1.020	0.454	1.03×10^{-1}
$p = 0.9$	9.54×10^{-2}	2.883	8.702	3.01×10^{-1}

Table 6.4.: Evaluation of UQ methods for the 1D Burgers equation.

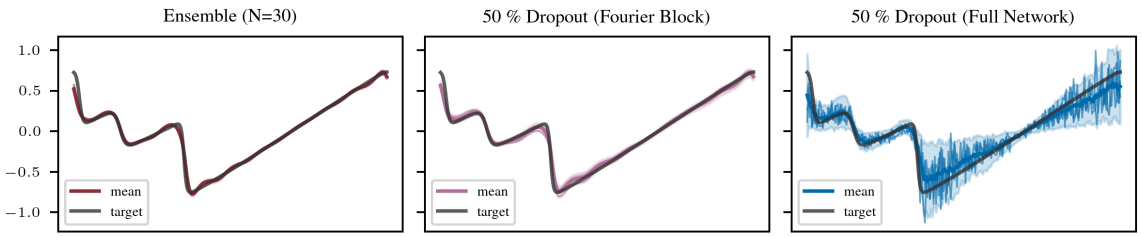

Figure 6.4.: The plot demonstrates a comparison between the ground truth and the prediction of each method, with a mean calculated from the samples of each method. We draw a 1.96 standard deviation around the mean. The input perturbations method manifests the characteristic frizzy look, while we observe a large and constant standard deviation for weight perturbations and a smooth standard deviation for the LUNO method.

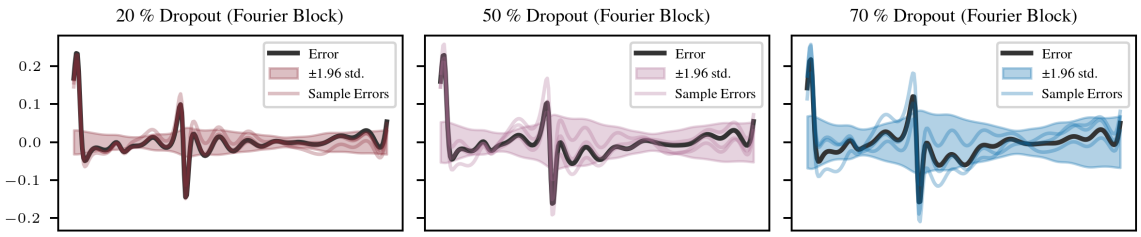

Figure 6.5.: The plot compares errors of different dropout rates applied to the Fourier block. The black line demonstrates the error with 1.96 standard deviations computed from 100 samples during evaluation. We also plot three sample outputs from forward passes with dropout enabled.

Diffusion Advection Equation

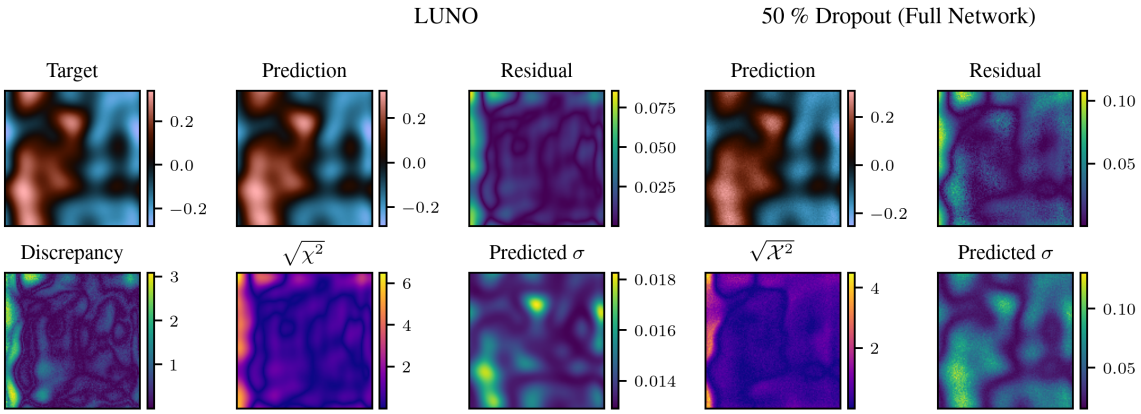

Figure 6.6.: Comparison between the LUNO method and 50% dropout applied to the full network. The top row shows the target, the mean predictions, and the residuals. The bottom row displays the square root of the χ^2-statistic (Q-value), the difference in Q-values between both methods, and the predicted standard deviations.

Method	Avg. σ	NLL (↓)	χ^2	RMSE (↓)
Input pert.	1.55×10^{-2}	-2.681	1.137	1.46×10^{-2}
Weight pert.	1.39×10^{-2}	-2.742	1.225	$\mathbf{1.44 \times 10^{-2}}$
Ensemble (N Models)				
5	3.71×10^{-3}	9.617	29.091	1.61×10^{-2}
10	3.98×10^{-3}	3.342	16.302	1.56×10^{-2}
20	4.19×10^{-3}	1.480	12.346	1.55×10^{-2}
30	4.29×10^{-3}	0.755	10.831	1.52×10^{-2}
LUNO	1.44×10^{-2}	$\mathbf{-2.856}$	$\mathbf{1.000}$	1.45×10^{-2}
Laplace	1.48×10^{-2}	-2.801	1.050	1.49×10^{-2}
Dropout (Fourier block)				
$p = 0.2$	5.00×10^{-3}	0.146	9.166	1.46×10^{-2}
$p = 0.5$	9.25×10^{-3}	-2.505	2.631	1.48×10^{-2}
$p = 0.7$	1.29×10^{-2}	-2.795	1.374	1.52×10^{-2}
$p = 0.9$	2.14×10^{-2}	-2.695	0.577	1.64×10^{-2}
Dropout (Full network)				
$p = 0.2$	2.29×10^{-2}	-2.599	0.841	1.63×10^{-2}
$p = 0.5$	4.18×10^{-2}	-2.214	0.361	2.49×10^{-2}
$p = 0.7$	6.10×10^{-2}	-1.777	0.408	4.55×10^{-2}
$p = 0.9$	6.91×10^{-2}	-0.197	3.155	1.23×10^{-1}

Table 6.5.: Evaluation of UQ methods for the 2D Diffusion Advection equation.

Kolmogorov Flow

Method	Avg. σ	NLL (↓)	χ^2	RMSE (↓)
Input pert.	0.433	0.676	1.219	0.418
Weight pert.	0.440	0.603	**1.074**	0.420
Ensemble (N Models)				
5	0.155	2.872	7.866	0.364
10	0.160	1.469	4.894	0.353
20	0.169	1.003	3.824	0.353
30	0.175	0.741	3.222	**0.338**
LUNO	0.395	**0.567**	1.167	0.419
Laplace	0.391	0.6015	1.260	0.435
Dropout (Fourier block)				
$p = 0.2$	0.113	7.100	16.743	0.441
$p = 0.5$	0.195	2.377	6.207	0.465
$p = 0.7$	0.290	1.280	3.224	0.502
$p = 0.9$	0.465	0.805	1.323	0.513
Dropout (Full network)				
$p = 0.2$	0.572	1.338	2.297	0.635
$p = 0.5$	0.987	1.280	1.083	0.843
$p = 0.7$	1.229	1.538	1.147	1.351
$p = 0.9$	1.402	2.415	2.569	2.501

Table 6.6.: Evaluation of UQ methods for the 2D Kolmogorov Flow.

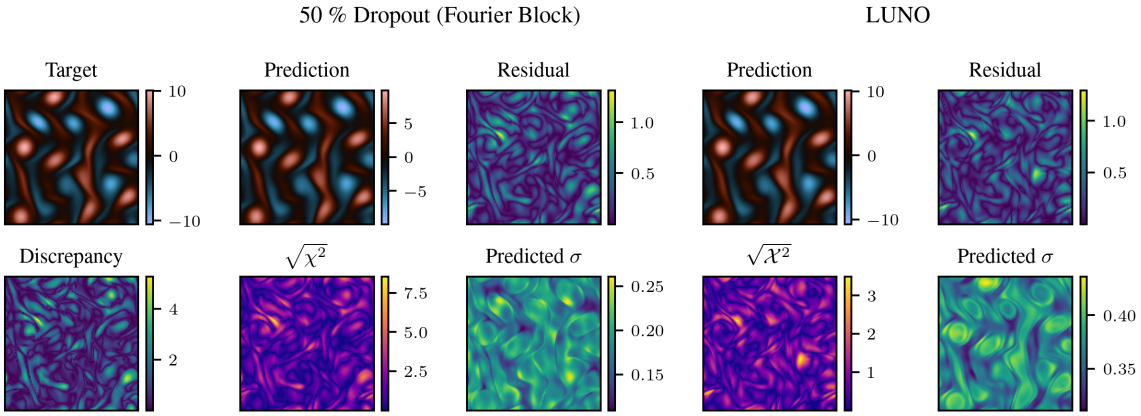

Figure 6.7.: Comparison between dropout applied to 50% in the Fourier block and the LUNO method. The top row shows the target, the mean predictions, and the residuals. The bottom row displays the square root of the χ^2-statistic (Q-value), the difference in Q-values between both methods, and the predicted standard deviations.

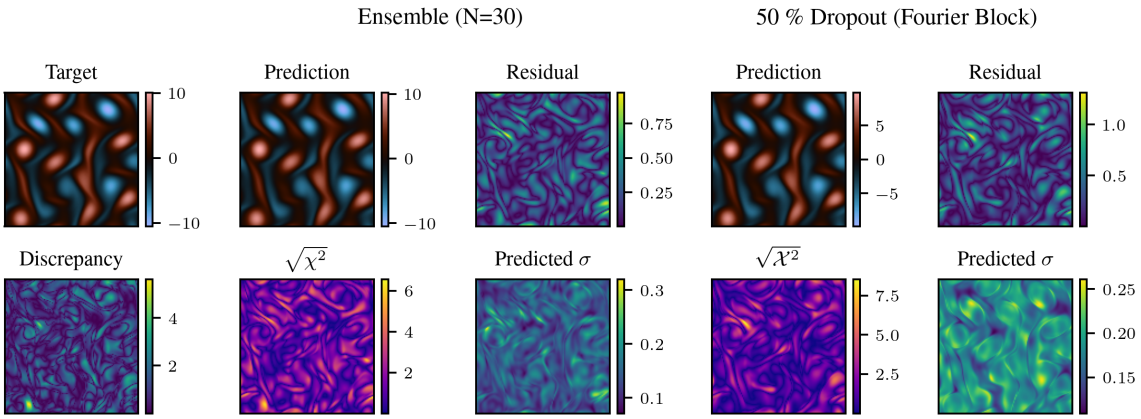

Figure 6.8.: This plot compares ensemble predictions with 30 members to 50% dropout in the Fourier block.

7. Discussion

We trained Fourier Neural Operators (FNOs) using 50 training samples, 20 validation samples, and 20 test samples. Each training sample consisted of 25 trajectories. The models were trained for 20 epochs. Our experiments included the one- and two-dimensional Burgers equation, the one- and two-dimensional advection-diffusion equation, and the two-dimensional Kolmogorov flow.

While the metrics for the advection-diffusion and Burgers equations indicate that the LUNO method and the sample-based Laplace approximation achieve the best overall performance, the results for the 2D Kolmogorov flow show a significant deterioration in key metrics such as the NLL and χ^2 value. This decline can be attributed to the increased complexity of the Kolmogorov flow, which involves strong turbulence and more complex dynamics.

Nevertheless, LUNO and the Laplace approximation still yield χ^2 values close to 1, suggesting good calibration despite the more challenging setting. Interestingly, the weight perturbation method not only achieves the best trade-off between NLL and calibration for the Kolmogorov flow, but also performs really well in other equations.

Dropout-based uncertainty quantification, however, proves to be highly sensitive to the choice of dropout rate and to the considered PDE. Higher dropout rates seem to achieve better NLL and Q-values when dropout is applied solely to the Fourier block. Lower Q-values can be attributed to a higher standard deviation combined with a smaller increase in RMSE. In case of applying dropout to the entire network, we observe better metrics for lower dropout rates such as $0.2 \leq p \leq 0.5$. Applying dropout throughout the whole network results in improved NLL and χ^2 values, indicating better uncertainty calibration, but at the cost of predictive accuracy as reflected by increased RMSE values. Moreover, dropout across the full network leads to a more caotic and overall higher predictive standard deviation. Predictions obtained from dropout applied to the full network appear to be very frizzy.

As shown in table 6.3, the difference in the number of active versus inactive parameters between Fourier block dropout and full network dropout is minimal. Nevertheless, the resulting difference in predictive quality is substantial. This highlights the crucial role of the output projection: zeroing the input with probability p and relying solely on the bias term leads to the observed frizzy appearance.

Since dropout relies on a Bernoulli distribution, the standard deviation of the sampling process is maximized at a dropout rate of 0.5, as the variance $\sigma^2 = p(1-p)$ describes

a downward-opening parabola with its maximum at $p = 0.5$. This effect is not observed when dropout is employed. Reasons for that could be a more noisy network with higher dropout rates, inducing worse predictions and therefore higher deviations from the mean.

The method (applied to the Fourier block) performs particularly poorly on more complex PDEs, where NLL and χ^2 values rise significantly, suggesting unreliable uncertainty estimates compared to LUNO.

Compared to deep ensembles, dropout achieves superior uncertainty calibration, as the predictions from different ensemble members tend to be too similar to each other, limiting the diversity necessary for reliable uncertainty estimation. However, deep ensembles provide better predictive performance than full-network dropout. When dropout is restricted to the Fourier block, the RMSE values are comparable to those of deep ensembles.

8. Conclusion

This thesis aimed to provide a comparative analysis of uncertainty quantification methods in Fourier Neural Operators, with a detailed examination of Monte Carlo dropout under varying configurations. Our experiments demonstrated that both LUNO and the sample-based Laplace approximation achieve state-of-the-art performance across various metrics, including NLL and χ^2-statistics. However, as shown by Magnani et al., these methods also require the longest calibration times. One way to circumvent the need for calibrating a parameter σ is to train multiple models with different random seeds and average their predictions—a technique known as deep ensembles. We showed that Monte Carlo dropout yields ensemble-like predictions by effectively creating an ensemble of different subnetworks within the same model and averaging their outputs. This approach not only performs better across the considered metrics but also eliminates the need for training multiple models. However, applying dropout across the entire network leads to significantly higher RMSE values, indicating poorer predictive performance compared to deep ensembles, as also confirmed by the plots.

When selecting the most suitable method, one must consider available resources such as memory and computational power, as well as the implementation complexity. For example, implementing a linearized Laplace approximation like LUNO can be quite involved. In contrast, Monte Carlo Dropout is easy to implement, offers flexibility regarding the layers where dropout is applied, and enables fast inference. However, it often yields unreliable predictions and only moderately calibrated uncertainty estimates. Weight perturbations, on the other hand, offer a favorable trade-off between short calibration times, reliable predictive performance, and well-calibrated uncertainty estimates. However, in high-stakes applications such as healthcare, the priority must be placed on achieving the most accurate uncertainty estimates—where methods like LUNO demonstrate clear advantages.

Future research could explore longer training durations or larger training datasets when dealing with more complex PDEs, such as the Kolmogorov flow. Investigating the effect of applying Dropout to specific subsets of Fourier layers—such as only the final layer, both the first and last layers, or the second and third layers—may offer valuable insights into its impact on uncertainty quantification in Fourier Neural Operators. Another promising direction would be to vary the architecture of the FNO, for instance by adjusting the number of layers or the layer width, to better match the complexity of the target PDE. Furthermore, extending the prediction horizon to multiple timesteps and evaluating how uncertainty metrics evolve over time could provide deeper insights into the temporal consistency and robustness of different UQ methods.

A. Additional Results

Method	Avg. σ	NLL (↓)	χ^2	RMSE (↓)
Input pert.	2.37×10^{-2}	-2.524	0.613	1.63×10^{-2}
Weight pert.	1.88×10^{-2}	-2.673	$\mathbf{0.771}$	1.60×10^{-2}
Ensemble (N Models)				
5	6.40×10^{-3}	0.638	9.933	1.60×10^{-2}
10	6.59×10^{-3}	-0.981	6.555	$\mathbf{1.52 \times 10^{-2}}$
20	6.50×10^{-3}	-1.381	5.696	1.54×10^{-2}
30	6.56×10^{-3}	-1.528	5.363	$\mathbf{1.52 \times 10^{-2}}$
LUNO	1.88×10^{-2}	$\mathbf{-2.685}$	0.749	1.60×10^{-2}
Laplace	1.92×10^{-2}	-2.671	0.739	1.60×10^{-2}
Dropout (Fourier block)				
$p = 0.2$	1.15×10^{-2}	-2.329	2.553	1.70×10^{-2}
$p = 0.5$	2.21×10^{-2}	-2.513	0.960	1.96×10^{-2}
$p = 0.7$	2.99×10^{-2}	-2.345	0.593	2.25×10^{-2}
$p = 0.9$	4.56×10^{-2}	-1.916	0.603	3.53×10^{-2}
Dropout (Full network)				
$p = 0.2$	5.31×10^{-2}	-2.042	0.228	2.29×10^{-2}
$p = 0.5$	9.01×10^{-2}	-1.482	0.285	5.82×10^{-2}
$p = 0.7$	1.30×10^{-1}	-1.071	0.339	9.59×10^{-2}
$p = 0.9$	9.22×10^{-2}	3.509	10.004	2.95×10^{-1}

Table A.1.: Evaluation of UQ methods for the 1D Diffusion Advection equation.

Method	Avg. σ	NLL (↓)	χ^2	RMSE (↓)
Input pert.	2.80×10^{-2}	-2.055	1.239	2.53×10^{-2}
Weight pert.	2.56×10^{-2}	-2.235	1.054	2.50×10^{-2}
Ensemble (N Models)				
5	7.14×10^{-3}	4.295	17.060	2.46×10^{-2}
10	7.37×10^{-3}	0.508	9.333	2.35×10^{-2}
20	7.59×10^{-3}	-0.125	7.968	2.32×10^{-2}
30	7.67×10^{-3}	-0.206	7.777	2.32×10^{-2}
LUNO	2.41×10^{-2}	$\mathbf{-2.332}$	$\mathbf{1.025}$	2.50×10^{-2}
Laplace	2.37×10^{-2}	-2.311	1.105	2.55×10^{-2}
Dropout (Fourier block)				
$p = 0.2$	8.32×10^{-3}	0.403	8.61	2.47×10^{-2}
$p = 0.5$	1.46×10^{-2}	-1.993	2.693	2.42×10^{-2}
$p = 0.7$	1.93×10^{-2}	-2.310	1.499	2.38×10^{-2}
$p = 0.9$	2.97×10^{-2}	-2.332	9.589	$\mathbf{2.24 \times 10^{-2}}$
Dropout (Full network)				
$p = 0.2$	4.37×10^{-2}	-2.098	0.528	3.16×10^{-2}
$p = 0.5$	6.03×10^{-2}	-1.593	0.510	6.03×10^{-2}
$p = 0.7$	8.13×10^{-2}	-0.933	1.512	1.15×10^{-1}
$p = 0.9$	7.07×10^{-2}	1.528	6.555	1.80×10^{-1}

Table A.2.: Evaluation of UQ methods for the 2D Burgers equation.

B. Additional Plots

1D Burgers Equation

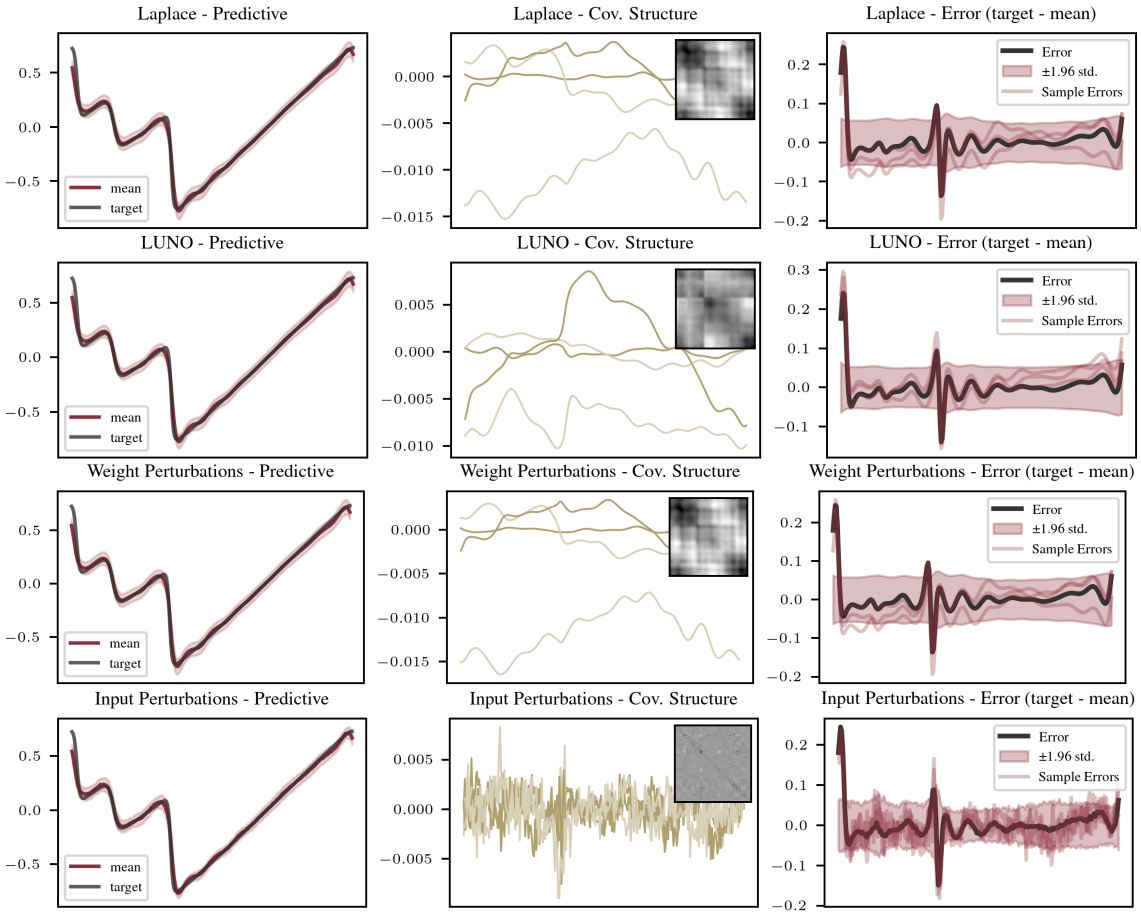

Figure B.1.: This figure displays the sample-based Laplace approximation, the linearized LUNO method, the Weight Perturbations and the Input Perturbations method. From left to right: the first panel shows the mean prediction obtained via multiple forward passes with different weight samples, as defined in eq. (4.1.4), alongside the target. The middle panel presents a heatmap of the covariance matrix and its top four eigenvectors. The rightmost panel displays the prediction error using three samples, along with a 1.96-standard-deviation uncertainty band.

2D Burgers Equation

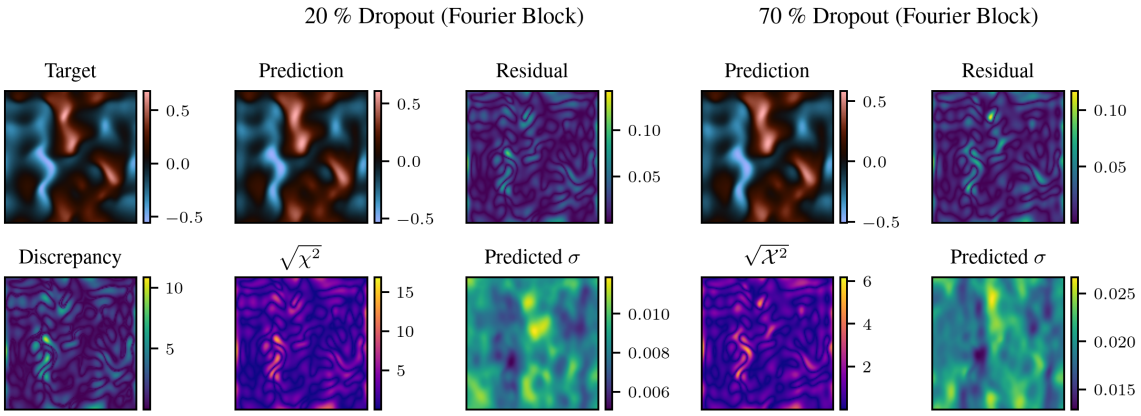

Figure B.2.: Comparison between 50% dropout to 70% dropout applied to the Fourier block. The top row shows the target, the mean predictions, and the residuals. The bottom row displays the square root of the χ^2-statistic (Q-value), the difference in Q-values between both methods, and the predicted standard deviations.

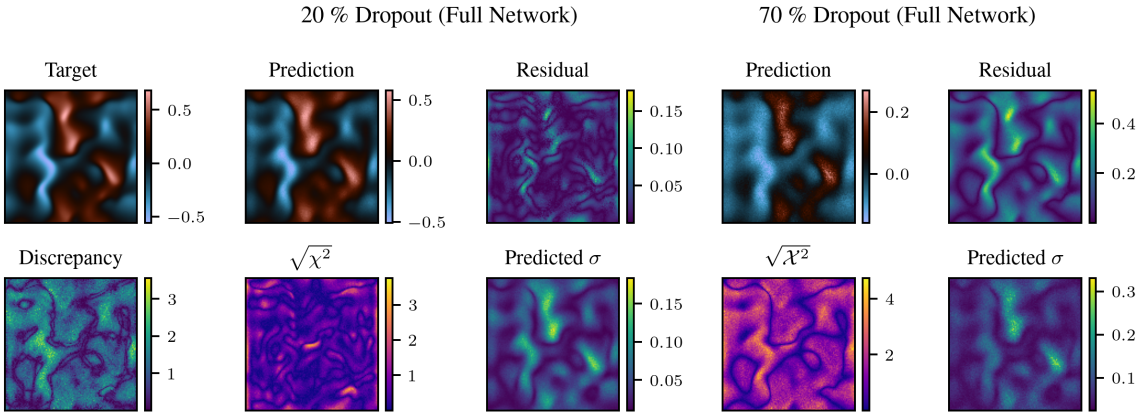

Figure B.3.: Comparison between 50% dropout to 70% dropout applied to the full network.

1D Advection Diffusion Equation

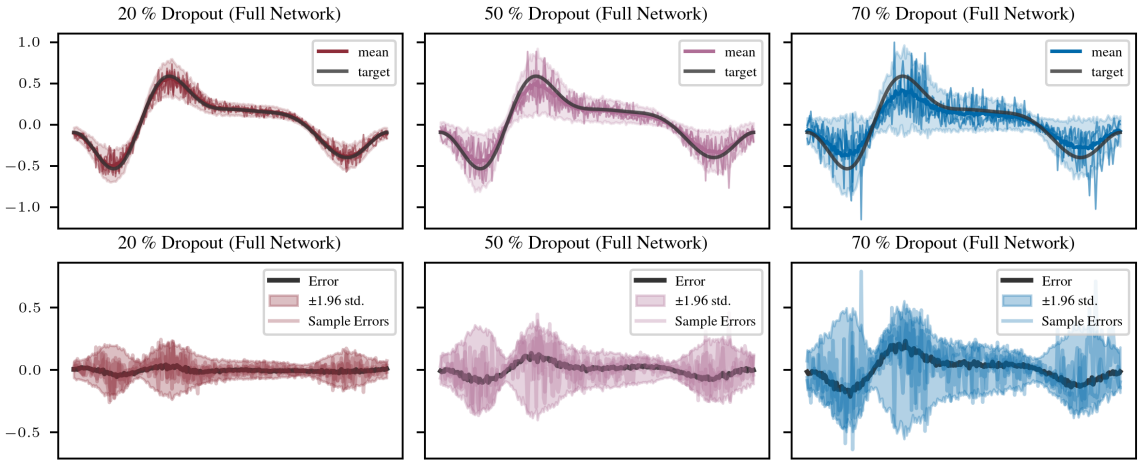

Figure B.4.: This figure shows a comparison of 20%, 50% and 70% dropout applied to the full network. Top row shows the predictions and multiple samples, bottom row displays the corresponding error.

2D Advection Diffusion Equation

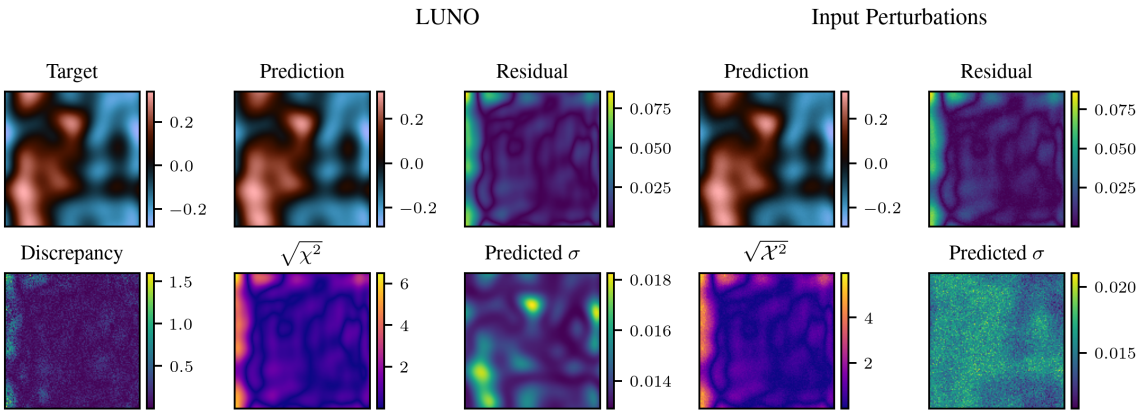

Figure B.5.: Comparison between the LUNO method and Input Perturbations. The top row shows the target, the mean predictions, and the residuals. The bottom row displays the square root of the χ^2-statistic (Q-value), the difference in Q-values between both methods, and the predicted standard deviations.

Kolmogorov Flow

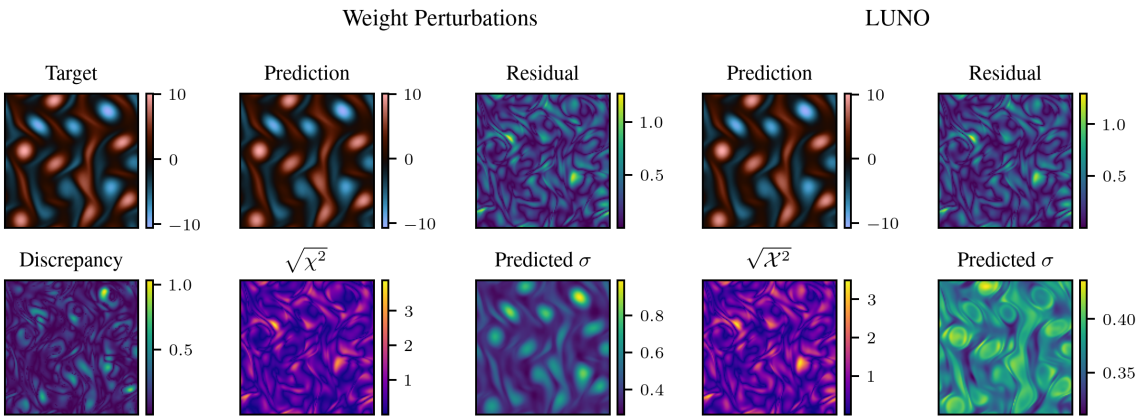

Figure B.6.: Comparison between Weight Perturbations and the LUNO method. The top row shows the target, the mean predictions, and the residuals. The bottom row displays the square root of the χ^2-statistic (Q-value), the difference in Q-values between both methods, and the predicted standard deviations.

C. Heat Equation Code

```python
import matplotlib.pyplot as plt
import numpy as np

# Parameters
nx = 50  # Num spatial points
nt = 100  # Num time points
L = 10  # Height of source
T_max = 5  # Max time

x = np.linspace(0, L, nx)  # Spatial
t = np.linspace(0, T_max, nt)  # Time
X, T = np.meshgrid(x, t)  # 2D-Grid

# Calc solution of Heat equation
U = np.exp(-T) * np.sin(np.pi * X / L)

# 3D-Surface-Plot
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection="3d")

ax.plot_surface(T, X, U, cmap="inferno")

# Axis
ax.set_xlabel("Time t")
ax.set_ylabel("Spatial x")
ax.set_zlabel("Temperature u(x,t)")
plt.savefig("heat_equation_curve.png", dpi=300)

plt.show()
```

Listing C.1: Solution of the 1D-Heat equation in a 3D surface plot

D. Conda Environment

The Conda environment was set up using Python 3.11.11 on a Windows 11 machine running Ubuntu 24.04 via WSL2.

Package	Version
absl-py	2.2.1
aiohappyeyeballs	2.6.1
aiohttp	3.11.16
aiosignal	1.3.2
annotated-types	0.7.0
apebench	0.1.1
array_record	0.7.1
asttokens	3.0.0
attrs	25.3.0
beartype	0.20.2
certifi	2025.1.31
charset-normalizer	3.4.1
chex	0.1.89
click	8.1.8
cloudpickle	3.1.1
comm	0.2.2
contourpy	1.3.1
cycler	0.12.1
debugpy	1.8.11
decorator	5.2.1
dm-tree	0.1.9
docker-pycreds	0.4.0
einops	0.8.1
equinox	0.12.1
etils	1.12.2
executing	2.2.0
exponax	0.1.0
filelock	3.18.0
flax	0.10.5
fonttools	4.56.0
frozenlist	1.5.0
fsspec	2025.3.2
gitdb	4.0.12

Package	Version
GitPython	3.1.44
gpustats	0.0.1
grain	0.2.6
h5py	3.13.0
humanize	4.12.2
idna	3.10
importlib_metadata	8.6.1
importlib_resources	6.5.2
ipykernel	6.29.5
ipython	9.0.2
ipython_pygments_lexers	1.1.1
jax	0.5.3
jax-cuda12-pjrt	0.5.3
jax-cuda12-plugin	0.5.3
jaxlib	0.5.3
jaxtyping	0.3.1
jedi	0.19.2
Jinja2	3.1.6
jupyter_client	8.6.3
jupyter_core	5.7.2
kiwisolver	1.4.8
laplax	0.0.1
lightning-utilities	0.14.2
linox	0.0.1
loguru	0.7.3
lugano	0.0.1
lugano-experiments	0.0.1
markdown-it-py	3.0.0
MarkupSafe	3.0.2
matplotlib	3.10.1
matplotlib-inline	0.1.7
mdurl	0.1.2
ml_collections	1.0.0
ml_dtypes	0.5.1
more-itertools	10.6.0
mpmath	1.3.0
msgpack	1.1.0
multidict	6.2.0
nest-asyncio	1.6.0
networkx	3.4.2
numpy	2.2.4
nvidia-cublas-cu12	12.4.5.8
nvidia-cuda-cupti-cu12	12.4.127
nvidia-cuda-nvcc-cu12	12.8.93
nvidia-cuda-nvrtc-cu12	12.4.127

Package	Version
nvidia-cuda-runtime-cu12	12.4.127
nvidia-cudnn-cu12	9.1.0.70
nvidia-cufft-cu12	11.2.1.3
nvidia-curand-cu12	10.3.5.147
nvidia-cusolver-cu12	11.6.1.9
nvidia-cusparse-cu12	12.3.1.170
nvidia-cusparselt-cu12	0.6.2
nvidia-ml-py	12.570.86
nvidia-nccl-cu12	2.21.5
nvidia-nvjitlink-cu12	12.4.127
nvidia-nvtx-cu12	12.4.127
nvitop	1.4.3.dev8+g2fbf791
opt_einsum	3.4.0
optax	0.2.4
orbax-checkpoint	0.11.10
packaging	24.2
pandas	2.2.3
parso	0.8.4
pdequinox	0.1.2
pexpect	4.9.0
pillow	11.1.0
pip	25.0.1
platformdirs	4.3.7
plum-dispatch	2.5.7
prompt_toolkit	3.0.50
propcache	0.3.1
protobuf	5.29.4
psutil	7.0.0
ptyprocess	0.7.0
pure_eval	0.2.3
pydantic	2.11.1
pydantic_core	2.33.0
Pygments	2.19.1
pyocclient	0.6
pyparsing	3.2.3
python-dateutil	2.9.0.post0
pytorch-lightning	2.5.1
pytz	2025.2
PyYAML	6.0.2
pyzmq	26.2.0
requests	2.32.3
rich	14.0.0
scipy	1.15.2
seaborn	0.13.2
sentry-sdk	2.25.1

Package	Version
setproctitle	1.3.5
setuptools	75.8.0
simplejson	3.20.1
six	1.17.0
skerch	0.10.1
smmap	5.0.2
stack-data	0.6.3
sympy	1.13.1
tabulate	0.9.0
tensorstore	0.1.73
the_well	1.0.1
toolz	1.0.0
torch	2.6.0
torch-dct	0.1.6
torchmetrics	1.7.0
torchvision	0.21.0
tornado	6.4.2
tqdm	4.67.1
trainax	0.0.2
traitlets	5.14.3
treescope	0.1.9
triton	3.2.0
tueplots	0.2.0
typing_extensions	4.13.0
typing-inspection	0.4.0
tzdata	2025.2
urllib3	2.3.0
wadler_lindig	0.1.4
wandb	0.19.9
wcwidth	0.2.13
wheel	0.45.1
wrapt	1.17.2
yarl	1.18.3
zipp	3.21.0

E. Erklärung zum Einsatz von KI

In folgendem Umfang wurde KI eingesetzt:

1. Korrektur Rechtschreibung und Grammatik: Ich habe KI für Korrekturen der Recht- schreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.
2. Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

Nutzen	AI Tool	Version
Softwareentwicklung	ChatGPT	GPT-4o
Softwareentwicklung	DeepSeek	R1
Rechtschreibung und Grammatik	DeepL	25.4.11926442

Table E.1.: Liste der verwendeten AI tools

Bibliography

Amari, S.-i. (1998, 02). Natural gradient works efficiently in learning. *Neural Computation 10*(2), 251–276.

Azizzadenesheli, K., N. Kovachki, Z. Li, M. Liu-Schiaffini, J. Kossaifi, and A. Anandkumar (2024). Neural operators for accelerating scientific simulations and design.

Bernstein, M. N. (2021). Variational inference. Accessed 24. April 2025.

Beznosikov, A., E. Gorbunov, H. Berard, and N. Loizou (2023). Stochastic gradient descent-ascent: Unified theory and new efficient methods.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang (2018). JAX: composable transformations of Python+NumPy programs.

COMSOL (2016). Detailed explanation of the finite element method (fem). Accessed 10. Februar 2025.

Crank, J. (1975). *The Mathematics of Diffusion* (2nd ed.). Oxford University Press.

Cushman-Roisin, B. Chapter 2 – diffusion – part 5: With advection. Accessed 10.03.2025.

Dangel, F., L. Tatzel, and P. Hennig (2022). Vivit: Curvature access through the generalized gauss-newton's low-rank structure.

Daxberger, E., A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig (2022). Laplace redux – effortless bayesian deep learning.

Dedieu, J.-P. (2015). *Newton-Raphson Method*, pp. 1023–1028. Berlin, Heidelberg: Springer Berlin Heidelberg.

DeepMind, I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, A. Dedieu, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, G. Papamakarios, J. Quan, R. Ring, F. Ruiz, A. Sanchez, L. Sartran, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, M. Stanojević, W. Stokowiec, L. Wang, G. Zhou, and F. Viola (2020). The DeepMind JAX Ecosystem.

Gal, Y. and Z. Ghahramani (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning.

Ganguly, A. and S. W. F. Earp (2021). An introduction to variational inference.

García López, S., S. Hauberg, and W. Boomsma (2024). Probabilistic multiple sequence alignment using spatial transformations. *bioRxiv*.

Heek, J., A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee (2024). Flax: A neural network library and ecosystem for JAX.

Hendrycks, D. and K. Gimpel (2023). Gaussian error linear units (gelus).

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov (2012). Improving neural networks by preventing co-adaptation of feature detectors.

Jospin, L. V., H. Laga, F. Boussaid, W. Buntine, and M. Bennamoun (2022, May). Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine 17*(2), 29–48.

Kingma, D. P. and J. Ba (2017). Adam: A method for stochastic optimization.

Koehler, F., S. Niedermayr, R. Westermann, and N. Thuerey (2024). APEBench: A benchmark for autoregressive neural emulators of PDEs. *Advances in Neural Information Processing Systems (NeurIPS) 38*.

Kovachki, N. B., Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. M. Stuart, and A. Anandkumar (2021). Neural operator: Learning maps between function spaces. *CoRR abs/2108.08481*.

Kumar, N. K. and J. Shneider (2016). Literature survey on low rank approximation of matrices.

Lessig, C., I. Luise, B. Gong, M. Langguth, S. Stadtler, and M. Schultz (2023). Atmorep: A stochastic model of atmosphere dynamics using large scale representation learning.

Li, L., J. Chang, A. Vakanski, Y. Wang, T. Yao, and M. Xian (2024). Uncertainty quantification in multivariable regression for material property prediction with bayesian neural networks. *Scientific Reports*.

Li, Z., N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar (2021). Fourier neural operator for parametric partial differential equations.

Liashchynskyi, P. and P. Liashchynskyi (2019). Grid search, random search, genetic algorithm: A big comparison for nas.

Lippe, P., B. S. Veeling, P. Perdikaris, R. E. Turner, and J. Brandstetter (2023). Pde-refiner: Achieving accurate long rollouts with neural pde solvers.

Loshchilov, I. and F. Hutter (2019). Decoupled weight decay regularization.

Magnani, E., M. Pförtner, T. Weber, and P. Hennig (2024). Linearization turns neural operators into function-valued gaussian processes.

Marques, C. R., V. G. dos Santos, R. Lunelli, M. Roisenberg, and B. B. Rodrigues (2022). Analysis of deep learning neural networks for seismic impedance inversion: A benchmark study. *Energies 15*(20).

Orlandi, P. (2000). *The Burgers equation*, pp. 40–50. Dordrecht: Springer Netherlands.

Pathak, J., S. Subramanian, P. Harrington, S. Raja, A. Chattopadhyay, M. Mardani, T. Kurth, D. Hall, Z. Li, K. Azizzadenesheli, P. Hassanzadeh, K. Kashinath, and A. Anandkumar (2022). Fourcastnet: A global data-driven high-resolution weather model using adaptive fourier neural operators.

Rasmussen, C. E. and C. K. I. Williams (2006). *Gaussian Processes for Machine Learning*. The MIT Press.

Schraudolph, N. N. (2002, 07). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation 14*(7), 1723–1738.

Ulm, U. Markov chains, stochastik ii skript. Accessed 23.04.2025.