Technical University Munich

School of Engineering and Design

Report for Deep Learning for PDEs in Engineering Physics

Eduardo Silva

03805057

Supervisors: Dr. Yaohua Zang & Vincent Scholz

July 20, 2025

Contents

Chapter 1

Problem A

1.1 Task 1

1.1.1 Problem Description

One of the most common cases when working with structural dynamics is the analysis of the 1D beam deformation. This is the foundation of many approaches on how to solve static loading problems, as also a way to compare different materials behavior.

Assuming linear elasticity, and that the acceleration of the load is close enough to zero, it is possible to write the governing equation for the static response as:

$$-\frac{d}{dx}\left(E(x)\frac{du}{dx}\right) = f \tag{1.1}$$

, where $u = u(x)$ represents the vertical displacement of the beam as a function of the horizontal position. Additionally, the variable f represents the magnitude of the force applied per unit length.

This way, the coefficient $E(x)$ is responsible by characterizing the different behavior of the system for different materials. As the values for $E(x)$ are usually known from testing, it is necessary to implement a model that can quickly and accurately predict the displacement $u(x)$ of the beam.

Lastly, considering that the rod is fixed at both ends, by theory we can implement that:

$$u(x = 0) = u(x = 1) = 0 \tag{1.2}$$

, with $x \in [0, 1]$.

1.1.2 Methodology

For this case, we want to evaluate the response of the system for the following young modulus:

$$E(x) = \begin{cases} 5, & \text{if } 0.15 < |x - 0.5| < 0.35 \\ 2, & \text{otherwise} \end{cases} \tag{1.3}$$

, such that it is represented as:

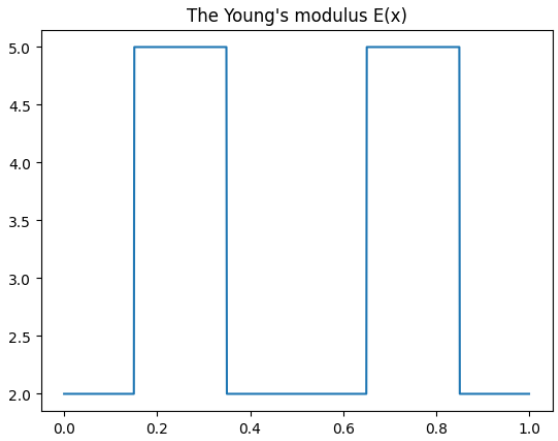

Figure 1.1: Theoretical Plot of $E(x)$

The method selection

When looking into the data that is available for the training of the model, it is possible to check that there are values only for the $E(x)$ discretized distribution. Apart from this, it is possible to check the accuracy of the model by using the values of x_{test} and u_{test}. However, these can not be used for training and only for validation and testing.

As the values of $E(x)$ can be know in any point between zero and one, this problem is not characterized as unsupervised, meaning that our data is labeled. However, the labels for the data are created within the training of the model, meaning that it is self-supervised.

Contrary to regular MLPs used for many applications in machine learning, both PINNs and DeepRitz models [1] use the data known for the inputs and the labels come from the physics involved in the system.To be more precise, in this case, the models will create a loss function that is constructed from the PDE in equation 1.1. The target behaviour is implemented therefore as a good satisfication of the PDE for the model's predictions.

Another important thing to realize, it that this first task is a forward problem. This means that the values of the parameter space $E(x)$ are known, and that the model should only correctly predict the solution space $u(x)$.

For this case, I decided to use the DeepRitz model due to a collection of reasons. As presented in the lectures, the PINN model struggles with sharp gradients in the parameter space, and as the function for $E(x)$ is only piece-wise linear, this can lead to a greater difficulty on learning the behavior close to the discontinuity points. On the other hand, when the variational form of the PDE is know (explained in next section) the DeepRitz model handles better non-smooth solutions and singularity points.

Lastly, DeepRitz works extremely well with elliptic PDEs (the case of this first task), since the solutions are based on the minimization of the system's energy. Contrary to this, when solving hyperbolic PDEs, the solution is not always the energy minimum case.

The loss function

To be more precise, the PDE equation 1.1 is described as the Poisson's equation. This is a well know equation in physics used to model many different scenarios of elliptic PDEs. As the

study of this specific formula has evolved, it is possible through the usage of test functions to arrive to the variational form of the PDE, described as:

$$\min_{u \in \mathcal{U}} I(u(x)) = \int_0^1 \frac{1}{2} E(x) \left(\frac{du(x)}{dx} \right)^2 - f(x)u(x) \; dx \tag{1.4}$$

, where the boundaries of the integral represent the space of x and $\mathcal{U}$ represents the solution space of $u(x)$.

As seen above, this is the variational form of the PDE for this problem. The trick used for the DeepRitz is that it is possible to use this variational form as the loss function. This means that the model will minimize the energy of the system, while respecting the physical properties of such. Therefore, the main goal of the model is to find $u(x)$ such that $I(u(x))$ is the lowest value.

With this mind, the complex PDE problem becomes a simple gradient descent problem in machine learning. And the loss function for the training process will be the variational form (equation 1.4).

Now another important part of solving the equation is to make sure that the boundary conditions are met. For this purpose, I decided to use a mollifier class that imposes this conditions directly. Since we want to have the following:

$$u(0) = u(1) = 0 \tag{1.5}$$

, it is possible to write a polynomial solution that forces the solution of the model to be zero only in the boundary conditions. I used the following second order polynomial:

$$Mollifier : u(x)_{bc} = u(x) \cdot (1 - x) \cdot x \tag{1.6}$$

For this reason, it is not needed to create weights for the loss of the PDE (w_{PDE}) and for the boundary conditions (w_{BC}) on the forward problem. This proves to be very helpful as the model will automatically find the best configuration that leads to the minimization of the loss function.

To further improve the model, the integration points were initialized using the Gaussian quadrature. This is much more accurate as the variational form will weight the different integrations points differently, and therefore not have a homogeneous grid of weights for the points. This can be seen in the following figure:

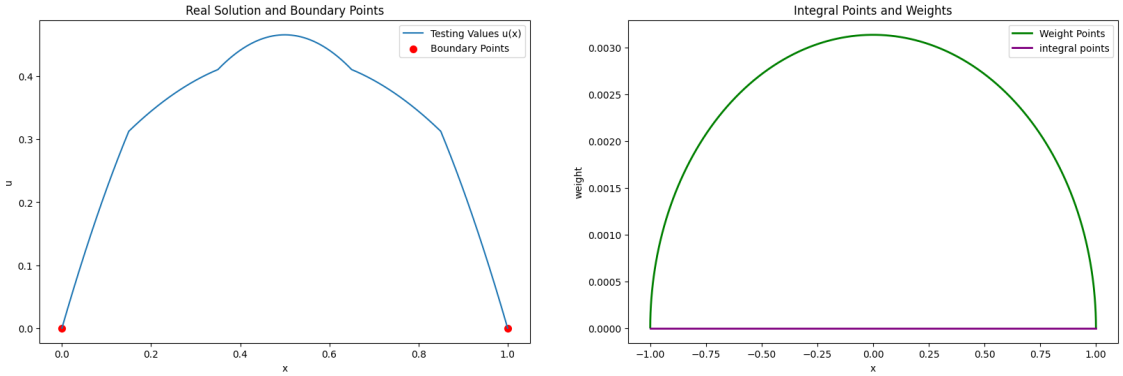

Figure 1.2: Real Solution and Integration Points

It is important to see here that the domain of x in the real space is different from the one used to parametrize the integration points using Gaussian quadrature. For this reason, after initializing the weights of the points, we need to perform the following operation:

$$x_{real} = x_{integration} \cdot \frac{1}{2} + \frac{1}{2} \tag{1.7}$$

After this, the model can correctly train based on the expression given for $E(x)$. As a last point it is important to know that when using weighted integral points, the calculation of the loss function should change a little. Defining equation 1.4 as $\mathcal{L}_{Ritz}$ we have that the loss to minimize is:

$$\mathcal{L}_{total} = \mathcal{L}_{Ritz} \cdot weights \cdot \frac{1}{2} \tag{1.8}$$

, where the $\frac{1}{2}$ term comes from the fact that it is the central point of the x space.

The network structure

For this exercise, many different models were tested, by varying the depth and width of the hidden layers, as also as activation functions. At the end the following model was chosen:

Table 1.1: Architecture of Neural Network Model

Layer	Size / Neurons	Activation Function	Notes
Input Layer	1 (value of x)	–	Automates Batch Size
Hidden Layer 1	40	Sin(Tanh)	Fully connected
Hidden Layer 2	40	Sin(Tanh)	Fully connected
Hidden Layer 3	40	Sin(Tanh)	Fully connected
Hidden Layer 4	40	Sin(Tanh)	Fully connected
Output Layer	1	–	Prediction of u(x)

Some important things to notice is the input size and the activation functions.

Firstly, the model should only receive one value for input, as the main goal and application is to determine what the vertical displacement $u(x)$ is at every point separately. Secondly, after testing many different activation functions, the one that created the better results was the hyperbolic tangent together with a sine function.

These activation functions were shown in class, and showed the best results for this problem. However, since the learning of the model works inside a black box, more options can and should be tested in order to find any other better combination. When unsure on what activation function to use, literature recently points to the usage of ReLU as a good starting point, as it does suffer from the gradient vanishing problem.

Lastly, at the output layer no activation function is used. This is because we want to train the model without knowing initially if the displacements will be negative or positive. Therefore using a ReLu or SiLU function(or any other function with a positive values bias) at the end could disturb the results. Usually it is good practice to not use activation at the last neuron.

Additionally, the following information shows the size of the model used:

- Total Parameters = 5041
- Trainable Parameters = 5041
- Total Size = 0.02 MB

The code link

https://github.com/MiRoSi-52wab/DeepLearningPDEs/blob/main/TaskA.ipynb

1.1.3 Numerical Experiments

Experiment Setups

For the training process, the following setup was used for the best result of the $u(x)$ prediction:

Parameter	Value
Optimizer	Adam
Weight Decay	1×10^{-4}
Learning Rate (initial)	1×10^{-3}
Batch Size	1001 (total size of x)
Number of Epochs	5000
Learning Rate Scheduler	StepLR
Decay Rate (α/N epochs)	0.5/833
Hardware	AMD Ryzen 7 5700U CPU
Precision	Float32
Early Stopping	No
Weight Clipping	No
Xavier Initialization	No

Table 1.2: Numerical Setup for Training the DeepRitz Model in PyTorch

For this exercise, the CPU was used as it did not take a long enough time to justify the usage of GPU. Another tricks were implemented but they proved not upgrade the quality of the result; however, it is good practice to have them implemented as for larger and more complex problems it influences the quality of the results.

The tricks implemented are: weight clipping, early stopping and Xavier initialization of the model's parameters. These are techniques used to avoid exploding gradients, overfitting and bad initial values for weights, respectively.

Additionally, after testing different batch sizes, it was possible to check that giving the entire values of x led to the better result. This can be the case since the dataset is rather small, and for that reason the model can have a bigger picture of the shape of the function at every point simultaneously.

It is important to notice that changing the initial learning rate is crucial. Initially, the value had been set to 5×10^{-2}, leading to wiggles in the descent of the L^2 error. This means that the value was too high and that the parameters were changing to much in order to converge to a good result. Together with changing this value to 1×10^{-3}, the patience of the learning rate scheduler was changed to a better training step.

Numerical Results

After using the setup explained in the table above, the following results were obtained:

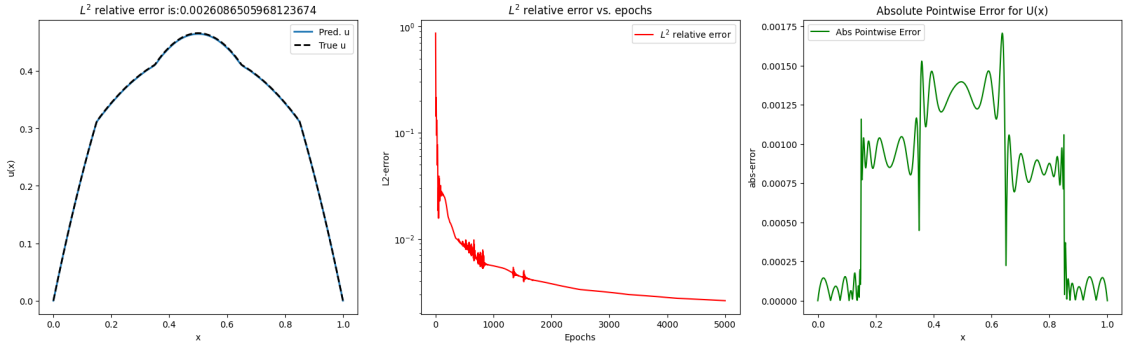

Figure 1.3: Experiment Results of Task 1

As seen above, the prediction of the displacement $u(x)$ was very precise. The model was able to correctly take care of the discontinuity of the Young's modulus and create a quicker and efficient way to retrieve values.

It is also important to notice that for this case there are no wiggles in the descent of the L^2 error, since the learning rate was adjusted to better converge to the correct solution. The L^2 error is calculated for every task using the following expression:

$$L^2 = \sqrt{\frac{\sum_i^n (X_{pred} - X_{true})^2}{\sum_i^n X_{true}^2}} \tag{1.9}$$

, for any quantity X.

Lastly, it is interesting noticing that even with the highest point-wise error being around 0.18%, the discontinuity points on $E(x)$ match exactly with the x points where the point-wise error suffers a bigger change of value. This can indicates that these are the values of x where the model has a harder time understanding the relation between $E(x)$ and $u(x)$. Nevertheless, for most applications this level of error is negligible.

For this case, the following was obtained for time consumption on the training process:

- Total Time $\simeq$ 2.5 min
- Iterations per second $\simeq$ 55

1.2 Task 2

1.2.1 Problem Description

For this scenario, now the challenge is to solve the inverse problem. This means that given sparse and noisy values of $u_{obs}(x)$, the model should be able to correctly predict both parameter function $E(x)$ and solution function $u(x)$. However, it is important to know that given the values of $u(x)$ contain noise and for that reason, the predictions of $E(x)$ and $u(x)$ should be distributed over two different models.

Additionally, even without knowing the values of $E(x)$, it is possible to know that the boundaries have the following criteria:

$$E(0) = E(1) = 1 \tag{1.10}$$

1.2.2 Methodology

The method selection

For the inverse problem, since it is still a self-supervised model usage case, either PINNs or DeepRitz can be used. However, the second one proves to be unfit for inverse problems.

As presented in the section above, the DeepRitz model key concept is to train the model based on the variational form of the PDE. This form tries to minimize the energy of the system, while respecting the physics of such. The issue arises for inverse problems due to the non-specification of the parameter space $E(x)$. While $E(x)$ is known for the forward problem, the unknown of both $u(x)$ and $E(x)$ leads to an aggregated optimization of both instead of separately.

In another words, for DeepRitz the models will try to minimize the energy by varying both $u(x)$ and $E(x)$. An example would be that if $E(x) = 0$, then the variational form of the PDE would have a lower value, however it would not respect the physics of the system. For this reason, PINNs show to be better solvers for inverse problems [2], since both $E(x)$ and $u(x)$ are trained with diverse loss criteria and weights.

The loss function

For the PINN model, the same general concept is used, with the difference that instead of using the variational form of the PDE, the strong form is used. This means that the model will try to minimize the error between the terms of equation 1.1. Using this equation it is possible to write:

$$-\frac{d}{dx}\left(E(x)\frac{du}{dx}\right) - f = \epsilon \tag{1.11}$$

And for that reason, calculating the value of the PDE for each point of x, we can create a loss function for the PDE as[1]:

$$\mathcal{L}_{PDE} = MSE(\epsilon, 0) \tag{1.12}$$

, meaning that we want to make epsilon be as close to zero as possible. For that scenario, both $E(x)$ and $u(x)$ are predicted in a way to correctly satisfy the PDE equation.

However, this is not enough for the model, as the values of $u_{obs}(x)$ are noisy and not continuous over all the x space. For this reason, using another model to correctly predict and interpolate $u(x)$, another additional loss function is used:

$$\mathcal{L}_{data} = MSE(u_{predicted}, u_{observed}) \tag{1.13}$$

, meaning that the predicted $u_{pred}(x)$ should be the one that minimizes the mean-squared error given the points of the observed field $u_{obs}(x)$.

Lastly, I decided to implement another loss functions for the boundaries conditions of $u(x)$ (known form task 1) and the recent introduced knowledge for $E(x)$:

$$\mathcal{L}_E^{BC} = MSE(E_{prediced}^{BC}, 1) \tag{1.14}$$

, and:

$$\mathcal{L}_u^{BC} = MSE(u_{predicted}, 0) \tag{1.15}$$

[1] All the mean squared errors are calculated as $MSE(X) = \frac{1}{N}\sum_{i=1}^{N}(X_{true} - X_{predicted})^2$ for any quantity X

Looking above it is possible to check that the loss functions will be minimized in order for the boundary points to respect their conditions, namely $E(x)$ being equal to one and $u(x)$ equal to zero at boundaries.

With all of these losses, I compute a total loss that is a weighted sum. This allows for more flexibility in the training process and allows to find the best combination of weights that creates lower errors:

$$\mathcal{L}_{total} = w_{PDE} \cdot \mathcal{L}_{PDE} + w_{data} \cdot \mathcal{L}_{data} + w_E^{BC} \cdot \mathcal{L}_E^{BC} + w_u^{BC} \cdot \mathcal{L}_u^{BC} \tag{1.16}$$

Initially, the mollifier was used for the boundary conditions of $E(x)$ and $u(x)$; however, the results proved to be worse and for that reason it was not used for the inverse problem. The mollifier can over-constrain the solution space with hard boundaries, and for inverse problem it is important that the model learns these conditions flexibly.

Just like in task 1, the weights were initialized using the Gaussian quadrature, and this can be seen in the following figure:

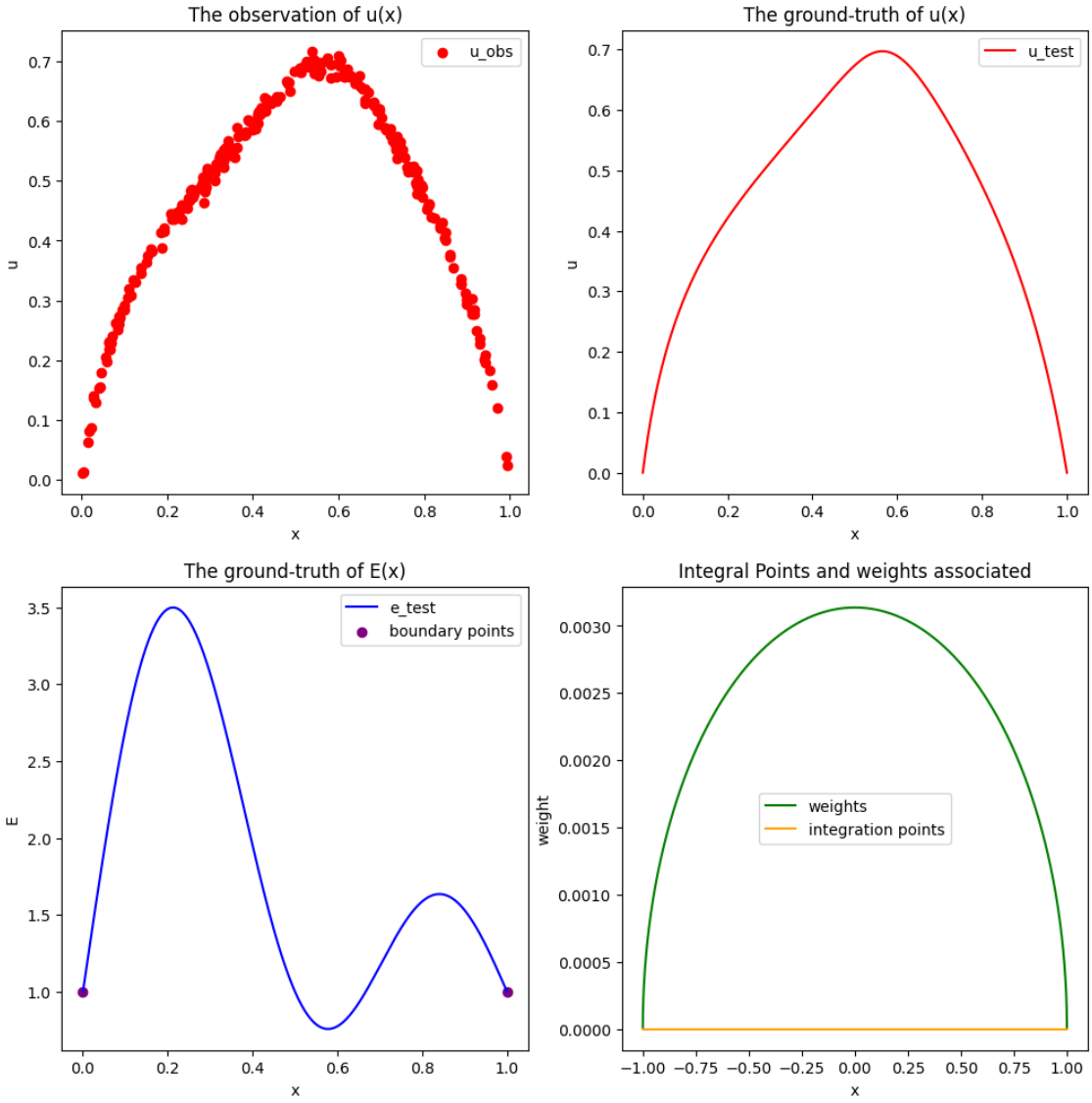

Figure 1.4: Data observed, true data and initialization of integral points.

Once again the values of the x were transformed using equation 1.7, such that they match with the coordinate space for $E(x)$ and $u(x)$.

The network structure

As explained before, for this exercise two different models were used to correctly predict both $u(x)$ and $E(x)$ spaces. The following network structures were used:

Layer	Size / Neurons	Activation Function	Notes
Input Layer	1 (value of x)	–	Flattened input
Hidden Layer 1	128	Sin(Tanh)	Fully connected
Hidden Layer 2	128	Sin(Tanh)	Fully connected
Hidden Layer 3	128	Sin(Tanh)	Fully connected
Hidden Layer 4	128	Sin(Tanh)	Fully connected
Output Layer	1	–	For $u(x)$ prediction

Table 1.3: Architecture of Neural Network Model for $u(x)$

And doing the same for the model predicting the values of $E(x)$:

Layer	Size / Neurons	Activation Function	Notes
Input Layer	1 (value of x)	–	Flattened input
Hidden Layer 1	128	Sin(Tanh)	Fully connected
Hidden Layer 2	128	Sin(Tanh)	Fully connected
Hidden Layer 3	128	Sin(Tanh)	Fully connected
Hidden Layer 4	128	Sin(Tanh)	Fully connected
Output Layer	1	–	For $E(x)$ prediction

Table 1.4: Architecture of Neural Network Model for $E(x)$

For this case, the same activation functions were used (since it was the best configuration for task 1) and the models for $u(x)$ and $E(x)$ are identical. This is a good approach to the problem, since it is important for the optimizer to correctly approximate both functions and both of them input the same number of parameters into the optimizer. One of the differences compared to task 1, is that the models show wider and deeper structures, needed for solving inverse problems. This is due to the fact that there is not ground-truth know for either of the functions in the PDE, leading to the need of more complex models.

Lastly, the models have the following information regarding parameters and sizes (for each)[2]:

- Total Parameters = 49 921
- Total Size $\simeq$ 0.19 MB

The code link

https://github.com/MiRoSi-52wab/DeepLearningPDEs/blob/main/TaskA.ipynb

[2]This is calculated by changing the models initialization inside the "MLP Class" Jupyter notebook block, using Pytorch summary library.

1.2.3 Numerical Experiments

Experiment Setups

For this task, the following setup was used:

Parameter	Value
Optimizer	Adam (using both model's parameters)
Weight Decay	1×10^{-4}
Learning Rate (initial)	1×10^{-4}
Batch Size	1001 (total size of x)
Number of Epochs	10000
Learning Rate Scheduler	StepLR
Decay Rate (α/N epochs)	0.5/1000
Hardware	Google Collab T4 GPU
Precision	Float32
w_E^{BC}	1
w_{PDE}	500
w_{data}	10
w_u^{BC}	1
Early Stopping	No
Weight Clipping	Yes
Xavier Initialization	No

Table 1.5: Numerical Setup for Training the PINN Model in PyTorch

For this scenario, the optimizer receives the parameters from both models, so that it can manipulate the weights of the neural networks to achieve the lowest value of $\mathcal{L}_{total}$. Contrary to what was expected, the results were better for higher values of w_{PDE}, such that this is explained in the next section.

In this case, since it is a inverse problem, the learning rate was set to a lower value such that the oscillations on the model's parameter values were not as abrupt. This is because the optimizer will be handling both models parameters, leading to the necessity of carefully managing both model's hyperparameters. So it is better to have a lower learning rate with more epochs and lower patience for the schedule, so that the gradient decent is smoother. Since the epochs were increased, for this case GPU was used instead to run faster the training process, leading to:

- Total Time $\simeq$ 4.5 min
- Iteration per second $\simeq$ 40

The batch size used was again the full sample of x, since it showed better results. Once again, this is due to the fact that (since the data isn't very large) the model can better understand the physical behavior of the functions when it has access to all spatial points simultaneously. For this reason, lower batching sample sizes are not needed, since efficiency is not crucial for this problem.

Lastly, after trial and error, activating weight clipping showed slight better results. And for this reason, it was used for the final training process.

Numerical Results

When using the setup mentioned above, it was possible to obtain the following figure showing the real and predicted value, as also the evolution of the different loss functions:

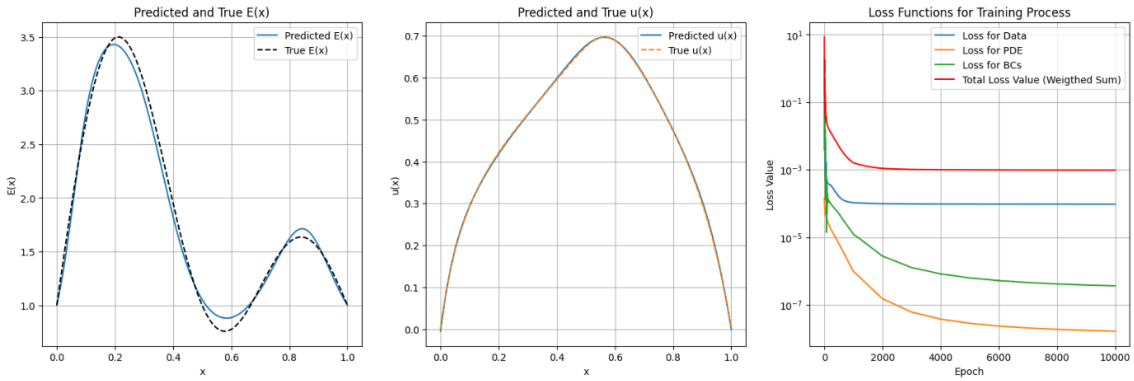

Figure 1.5: Prediction Values and Loss Functions

Additionally, the plots for the point-wise errors were achieved, as also the L^2 errors evolution with the validation process:

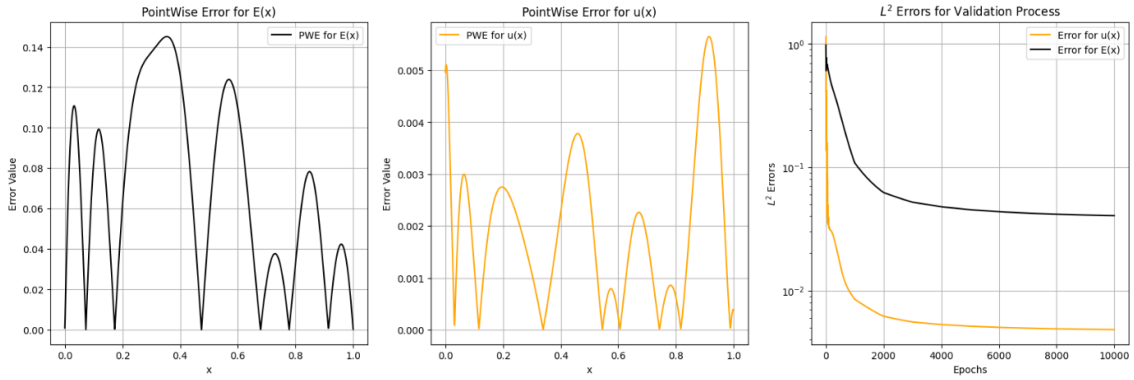

Figure 1.6: Point-wise Errors and L^2 Errors

Looking to both figures 1.5 and 1.6 it is possible to take some important conclusions. Firstly looking into the $E(x)$, the maximum error achieved 14%, showing a worse performance compared to task 1. However, the boundary conditions are correctly understood by the model as the point-wise errors for $E(0)$ and $E(1)$ are very small.

When looking into the $u(x)$ plot, it is possible to notice that this shows a much lower error (a maximum around 0.5%), as the predicted and real values align much better. Nevertheless, the nature of the observation data of $u(x)$ is what causes the prediction of $E(x)$ to be worse. As explained before, the optimizer will be trained with both models parameters, and will find the prediction that leads to the lowest weighted loss of both cases. Since the data from $u(x)$ has some noise, the model is not capable to predict perfectly the distribution of $u(x)$ and this leads to a propagation of error when trying to satisfy the PDE equation.

With this in mind, it was tried to train both models separately. However, this led to worse results since it is important for the models to understand the PDE equation.

Lastly, looking at the plots for the loss functions and L^2 errors, it is possible to conclude that the choice taken for epochs, learning rate and scheduler patience, led to a smooth descent of both curves. This indicates that the rate of learning of the model is well aligned with the

setup choice for these processes. Once again, if there were wiggles present, it could mean that the learning rate was be too high, leading to a higher change of parameters when performing backpropagation, and therefore making it harder for the model to find thee optimal parameter choice.

For this reason, it was needed that $w_{PDE} > w_{data}$ for the result to be better. For the opposite scenario, the prediction of $E(x)$ got much worse, since the prediction of $u(x)$ does not get much better due to the randomness and noise of the observed displacement. This is not common, as usually it is necessary to have an extremely good learning off the observed data for the model to correctly satisfy the PDE. However, the point-wise error never changed too much for $u(x)$ even after setting w_{data} to extremely high values.

If more time and computational cost is accessible, it is possible to iterate over a combinations of values of $(w_{PDE}, w_{data}, w_E^{BC}, w_u^{BC})$ to find the best configuration for a lower aggregated error.

Chapter 2

Problem B

2.1 Problem Description

Now similarly to the first two tasks, the goal is to correctly predict and estimate the solution of a schema of Poisson's equation. In this case, the goal is to solve the 2D steady-state heat equation that is defined as follows:

$$-\nabla \cdot (k(x,y)\nabla T(x,y)) = 0 \tag{2.1}$$

, where $k(x,y)$ represents the 2D bulk thermal conductivity of the material and $T(x,y)$ represents the temperature field of such. Additionally, the following information is know for the boundary conditions of both fields:

- $T(x=0,y)=0 \quad \wedge \quad T(x=1,y)=1$
- $\frac{\partial T}{\partial \vec{n}}(x,y=0) = \frac{\partial T}{\partial \vec{n}}(x,y=1) = 0$

, and for the thermal conductivity:

- $k_1 = 2, \quad (x,y) \in \Omega_1$
- $k_2 = 10, \quad (x,y) \in \Omega_2$

, where Ω_1 and Ω_2 represent the first and second phase regions.

For this problem, the dataset includes known distributions of $k(x,y)$ for many different materials and their correspondent solution $T(x,y)$. For this reason, the data is still labeled and therefore it is not an unsupervised learning mechanism. However, differently to the two first tasks, instead of using the variational form of the PDE or using the PDE as the loss function, for this case the model can learn how to correctly predict the values based on explicit pairs of (k, T). Below is an example of what these pairs look like:

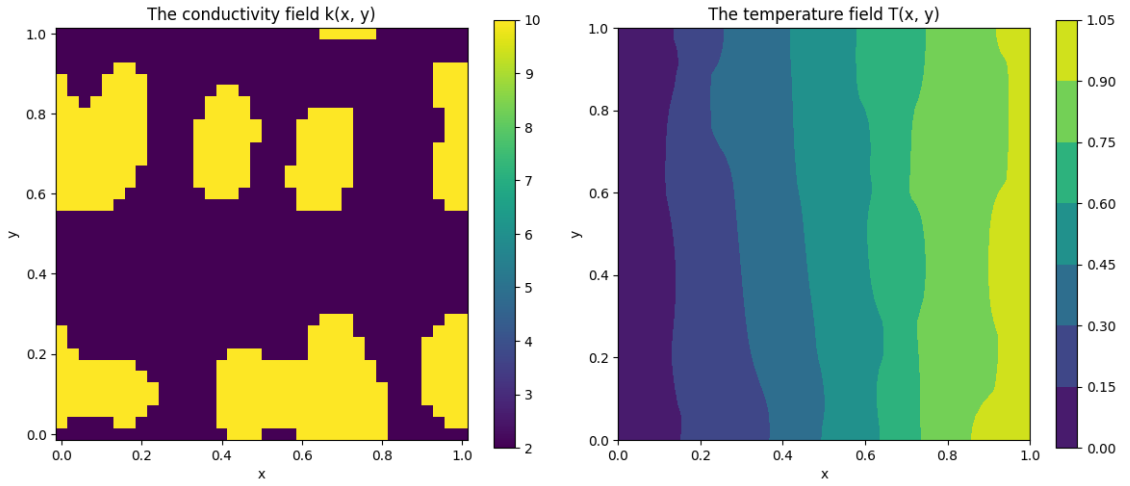

Figure 2.1: Pair of Fields (k, T)

For this reason, it makes sense to use a supervised learning model for this problem. The two options are DeepONet and Fourier Neural Operator (in the scope of this report).

As a last point, it is important to know that the dataset includes 1000 pairs of (k, T) and that both fields are of sizes $(x = 36, y = 36)$.

2.2 Methodology

The method selection

Since both models of DeepONet and FNO are deep neural operators, it means that these will be trained to map input function into output function (instead of value to value). When learning this mapping, the models are a good way to generalize solutions for unseen input functions.

For this first task, both models were tested, in order to better understand which one gave the better results. Nevertheless, it is important to understand the main differences between these two. As the last task is also related to function mapping, **both models are explained in depth in this section**, and the **best one is used** therefore **for both cases**.

DeepONet Architeture

DeepONet was firstly introduced as a neural operator designed to learn nonlinear mappings between function spaces. This means that, for a function $k(x, y)$ and a solution space of $T(x, y) = \mathcal{G}(k(x, y))$ the DeepONet tries to learn the operator:

$$\mathcal{G} : k(x, y) \in \mathcal{K} \rightarrow T(x, y) \in \mathcal{T} \tag{2.2}$$

For the DeepONet scenario, the key innovation idea is to separate the handling of inputs functions $k(x, y)$ and the spatial coordinates (x, y). Using this separation, the whole model is composed of two subset MLPs or CNNs (depending on dimensionality), namely the branch network and the trunk network. The first one is responsible for encoding the information acquired using the input functions [1], while the second encodes the information regarding the

[1] As a side note, usually input functions are discretized in order to match the input size of the branch network.

spatial location information. This decoupling leads to a easier model's architecture and more lightweight when inferring for new values.

At last, both of these information are combined using a simple inner product and bias operation such that:

$$\mathcal{G}_\theta(k(\Xi))(x,y) = \sum(Branch(k(\Xi)) \quad \cdot \quad Trunk(x,y)) \quad + b_0 \tag{2.3}$$

The DeepONet is restrained to the fact that $k(x,y)$ needs to be represented at fixed input sensors, meaning that for all the fields of conductivity measured, the sensors must be at the same position of (x,y). Additionally, as the DeepONet uses a flatten trunk networks for its architecture, some important information can be lost due to the lack of spatial awareness. This means that when the space parameters (x,y) the model has no direct learning of how neighborhood spatial points influence the satisfaction of the PDE.

The same happens for the branch network, as the conductivity is often flatten to a one dimensional vector.

Nevertheless, the DeepONet is a great structure for a general operator learning framework at it can be applied to any continuous operator. For some cases even, the decoupling of the spatial space with the input functions can lead to a more free learning process and arbitrary resolution of the predictions made by the model. Lastly, one of the main advantages of DeepONet is the mesh-free characteristic, meaning that the spatial does not need to be regular, but only consistent through different measures.

Fourier Neural Operator

Now the focus is to understand the Fourier Neural Operator, its components, structure and usage. Just like the DeepONet, this model is considered supervised, since it is trained on measured pairs of data (k, T). Therefore, one can predict that the loss function needs to be similar.

Usually CNN models are used for computer vision problems, since usually images have local patterns that can lead to the learning of different features. However, parametric PDEs exhibit global correlations, meaning that the solution at one point is not only affected by its neighbors. FNO models replace the local convolution operations with global convolutions, while using the Fourier transform to compact the values in the frequency domain.

Regarding the FNO's architecture there are three main components: lifting, global transformations and projection.

The first step (lifting) is applied to the input function (in this case $k(x,y)$) to expand the number of dimensions, leading to a more suitable set for learning complex function mappings. This is represented as:

$$v_0 = P_\theta(k(x,y),(x,y)) \tag{2.4}$$

, where P_θ represents a point-wise MLP usually. This allows the model to learn more patterns only noticeable for higher dimensions.

Secondly, the core of the neural operator model is characterized by the global transformations that happen in each layer. To predict the values of the neurons of thee FNO at layer $l+1$, the following expression is used:

$$v_{l+1} = \sigma(\mathcal{F}^{-1}(R \cdot \mathcal{F}(v_l)) + W v_l) \tag{2.5}$$

, where $\mathcal{F}$ represents the Fourier transform, R the complex spectral weights applied to a number of the lower modes, and $W v_l$ a 1×1 convolution operation connected to the spatial

domain. It is important to notice that while the first two operations are used for capturing the global interactions of the input, the latter is used to enhance the local correlation between values of the field $k(x, y)$.

Lastly, after the input is passed through a number of layers using equation 2.5, the solution needs to be brought back to a lower dimension space, matching the output space of $T(x, y)$. For this, similar to the lifting case, a MLP is used such that:

$$T(x, y)_{predicted} = Q_\theta(v_{last}) \tag{2.6}$$

This leads to the following pipeline inside the FNO model:

$$Input\, k(x, y) \rightarrow Lifting\, P_\theta \rightarrow Operations\, (\mathcal{F}, R, W) \rightarrow Projection\, Q_\theta \rightarrow T(x, y) \tag{2.7}$$

During this process, it is important to know that the modes with higher frequency are usually discarded by the R operator, as information is gathered in lower frequencies for PDEs. Additionally, the activation function σ is only applied after the core operation is not anymore in the Fourier space, leading to the recover of some of the higher-frequency details as also non-periodic boundary behaviors.

One of the disadvantages of the FNO compared to the DeepONet model, is the fact that due to the application of Fourier transform, it is needed that the data for the functions is measured with a uniform grid. This comes from the fact that the eigen-states in the frequency space can only be calculated if there is a consistent space step between points.

Nevertheless, FNO models usually show higher accuracy than DeepONet models, higher computational speed and a faster inference. As it will be shown in the next section, FNO proves to be a better choice when accuracy and precision are crucial for solving a determined problem.

The loss function

An interesting topic mentioned before, is that since both models are supervised, the loss function for both is the same. This is because both models will be trying to predict the values of $T(x, y)$. Therefore, as many of the simpler case of training machine learning models, the loss functions is defined by the mean squared error:

$$\mathcal{L}(\theta) = \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} (\mathcal{G}_\theta(k^{(i)}(x, y), (x, y)) - T^{(i)}(x, y))^2 = \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} (T^{(i)}_{pred} - T^{(i)}_{true})^2 \tag{2.8}$$

This way, the optimizer will try to find the best configuration of the model parameters θ, such that the value of $\mathcal{L}(\theta)$ is minimized. For lower values of loss, the model on average predicted the values of $T(x, y)$ closer to the real values.

The network structure

Initially, for the **DeepONet** model, the following networks structure was used:

Layer	Size / Neurons	Activation	Notes
Input Layer	2 (values of (x, y))	–	Flattened input
Hidden Layer 1	256	ReLU	Fully connected
Hidden Layer 2	256	ReLU	Fully connected
Hidden Layer 3	256	ReLU	Fully connected
Hidden Layer 4	256	ReLU	Fully connected
Output Layer	256	–	Features of Spatial Coordinates

Table 2.1: Architecture of Trunk Network - DeepONet

And for the branch network:

Layer	Size / Neurons	Activation	Notes
Input Layer	1296 (values of $k(x, y)$)	–	Flattened input of size 36×36
Hidden Layer 1	256	ReLU	Fully connected
Hidden Layer 2	256	ReLU	Fully connected
Hidden Layer 3	256	ReLU	Fully connected
Hidden Layer 4	256	ReLU	Fully connected
Output Layer	256	–	Features of Conductivity Field

Table 2.2: Architecture of Branch Network - DeepONet

At the end, both networks are combined to approximate the operator $\mathcal{G}$, using the equation 2.3. The final output will be of size $[B, 1296]$, where B represents the batch size and 1296 represents the value of $T(x, y)$ for each point of the 2D grid of size 36×36. So for this case, the model has the following information:

- Total Parameters = 859 137
- Forward/Backward Size = 13.28 MB
- Parameters Size = 3.44 MB
- Total Size $\simeq$ 17 MB

Secondly, for the **Fourier Neural Operator** model, the following structure was used:

Layer	Neurons	Activation	Notes
Lifting Input Layer	3	–	Values of (k, x, y)
Lifting Layer 1	40	ReLU	Fully Connected
SpectralConv2D	40	ReLU	Conv1D performed after Conv2D
Conv1D Layer 1	40	ReLU	Used as piecewise convolution
SpectralConv2D	40	ReLU	Conv1D performed after Conv2D
Conv1D Layer 1	40	ReLU	Used as piecewise convolution
SpectralConv2D	40	ReLU	Conv1D performed after Conv2D
Conv1D Layer 1	40	ReLU	Used as piecewise convolution
Projection Input Layer	40	ReLU	From $(\mathcal{F}, R, W)$ gets values
Projection Hidden Layer	128	ReLU	Fully Connected
Output Layer	1	–	Predicts value of $T(x, y)$

Table 2.3: Architecture of Fourier Neural Operator Model

And for the FNO is it important to define what are the number of modes to keep in each spatial direction. This means, that the SpectralConv2D layer will be responsible for analyzing the information for the lowest frequency modes, since these contain the most useful information (as explained before). For this case, it was set that:

- Number of modes in x-direction $= 6$
- Number of modes in y-direction $= 6$

Since the grid is 36×36, the number of modes is the same for each direction. However, for problems where it is known that one direction heavily influences the solution, these values can be changed accordingly.

For the FNO, the following information is obtained:

- Total Parameters $= 356\ 057$

The code link

https://github.com/MiRoSi-52wab/DeepLearningPDEs/blob/main/TaskB.ipynb

2.3 Numerical Results

Experiment Setups

For the DeepONet training process, the following setup was used:

Parameter	Value
Optimizer	Adam
Weight Decay	1×10^{-5}
Learning Rate (initial)	1×10^{-4}
Batch Size	50
Number of Epochs	300
Learning Rate Scheduler	StepLR
Decay Rate (α/N epochs)	0.5/75
Hardware	Google Collab T4 GPU
Precision	Float32 / Complex64
Early Stopping	No
Weight Clipping	No
Xavier Initialization	No

Table 2.4: Numerical Setup for Training DeepONet and FNO in Pytorch

It is important to notice that for this problem, since the data is much larger than compared to the first two tasks (instead of 1 value for each sample, we have 36×36 values), it is necessary to use batching. This will make training easier and more efficient as the model is trained on random shuffles of 50 samples for each epoch. The number of epochs is smaller to keep the training process reasonable in time consumption.

For the FNO structure, in order to compare it with the DeepONet structure, the same setup was used. The exception is that for the FNO scenario, since the Fourier transformation handles complex numbers, the precision needs to be increased to Complex64, while for the DeepONet it is set at Float32.

Numerical Results

For the DeepONet case, only the predicted and true results are shown. This is used only to compare with the FNO model, which proves to be the better one to choose. Below, it is possible to see the input used for the first instance of the testing data set for $k(x, y)$:

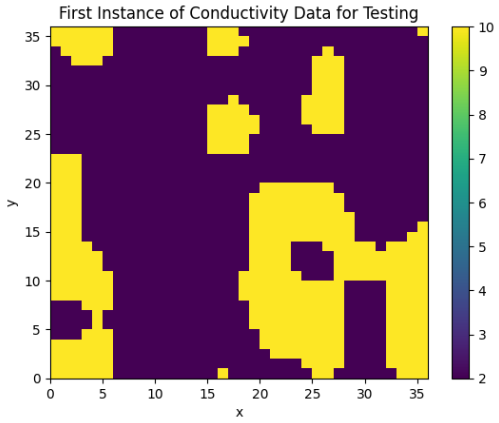

Figure 2.2: 1st Conductivity Field of Testing Dataset

This input was used for DeepONet and FNO, in order to compare the results. The following image shows the results, firstly for the DeepONet structure:

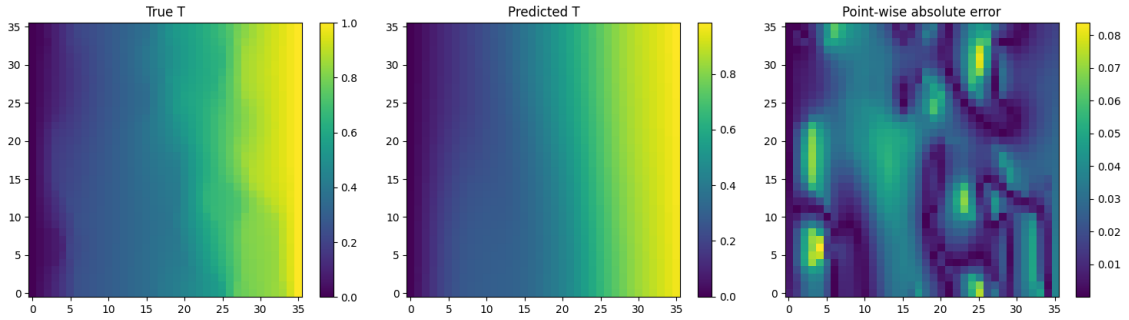

Figure 2.3: Results for DeepONet Model

As seen above, the DeepONet model reaches a maximum absolute error of 8%. This can be accepted depending on the application, but the main concern is that the model seems to approximate the temperature field linearly. This is a good indication that the model is not correctly understanding the local and global relations inside the conductivity field.

When using the FNO architeture, the following results were obtained:

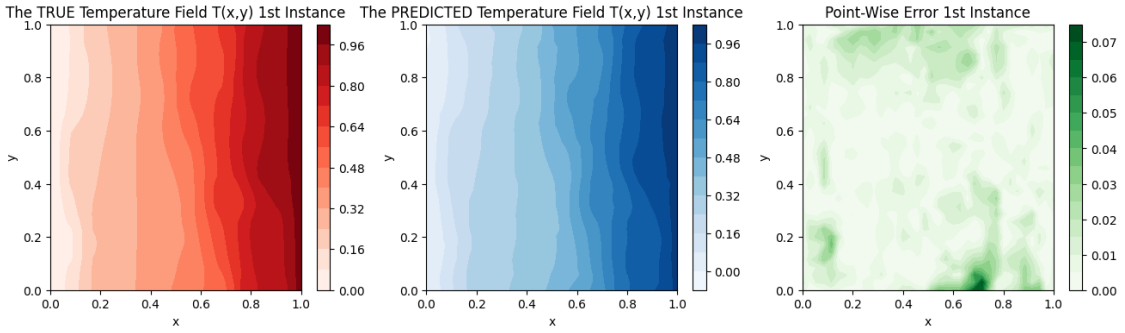

Figure 2.4: Results for FNO Model

As it is possible to see, the FNO model only reaches a maximum of around 7% absolute error, and additionally it is able to understand much better the shape of the temperature field. This is due to the fact that the FNO looks into the eigen-states of the PDE in the Fourier space, and correctly predicts the local and global relations inside the parameter space $k(x, y)$. To add, it is possible to see that the decision of the setup for the training process matched the models capability of learning. This is seen in the figure below:

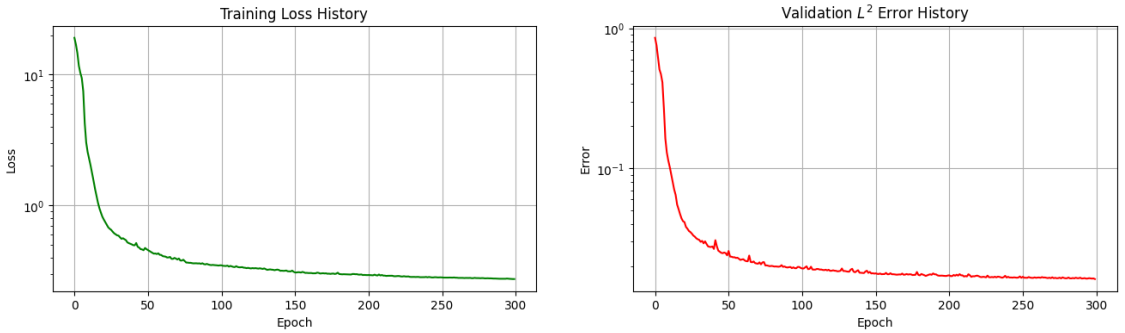

Figure 2.5: Loss and L^2 Values for Training and Validation (resp.)

It is possible to observe that there are no big "wiggles" in any of the plots above, showing that the gradient descent was smooth and converged to the better choice of model parameters θ, minimizing the value of the loss function in equation 2.8.

So after looking into both, results and noticing that the DeepONet used around double the amount of parameters of the FNO model, it becomes clear why the FNO are usually used for problems that have a uniform grid. The underlying understanding of the different components of the parameter space is highly enhanced when transformed into the Fourier space, and for that reason the FNO should be used.

As a last point, both models took the following amout of time to run the training process:

- DeepONet Training Time $\simeq$ FNO Training Time $\simeq$ 2 min

With this mind, since the values are very close to each other, the FNO proves to better understand the nature of the PDE, while still being able to run the training process with a reasonable time and a much lower number of parameters. Additionally, in general the FNO structure shows errors lower than 7%, meaning that this first instance of testing is actually one of the cases where the models fail more. To further analyze the results of the FNO structure refer to appendix A for confirmation

As a result, it is clear that for problems with a uniform grid of the parameter and solution fields it is better to recur to the usage of the Fourier Neural Operator architecture. The discussion and analysis between a regular CNN and FNO is left out of this report, but it is another interesting topic to discuss in further projects.

Chapter 3

Problem C

3.1 Problem Description

For the last task, the main goal is to correctly predict traffic flow by using Burguer's equation model. This equation is used to analyze the behavior of non-linear system with both convective and diffusion effects. An example is the usage of this equation to estimate the flow of a fluid inside a channel with both advective and dispersive coefficients.

Such modelling is defined using the following equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{3.1}$$

, such that $x \in (-1, 1)$ and $t \in (0, 1]$. For this exercise, the variable $u(x, t)$ represents the velocity of the car andν represents how cars speed react when close to others (set to $\nu = 0.1$). Additionally, by imagining that car starts and stops and positions $x = 0$ and $x = 1$, the following conditions are imposed:

$$u(x = -1, t) = u(x = 1, t) = 0 \quad , for \quad any \quad t \in (0, 1] \tag{3.2}$$

This way, when knowing the initial velocity field $u(x, t = 0) \equiv a(x)$, it is possible to predict the car's behavior by solving the Burguer's equation. As an illustration, in the image below it is possible to understand what the initial velocity and velocity fields can look like:

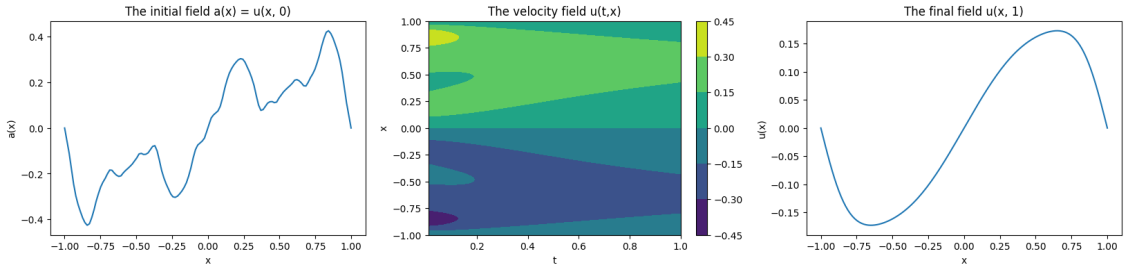

Figure 3.1: Example of Initial Conditions and Velocity Field

As seen above, the goal of our model is to receive as an input the initial condition of the velocity field $a(x)$ and correctly predict the velocity field for any time stamp $u(x, t)$.

3.2 Methodology

The method selection

By realizing that this problem is once again related to supervised learning models (DeepONet and FNO), the decision made was to use FNO once again since it showed better results for task B. With this mind, the trick used to solve this problem comes from a physical understanding of the Burguer's PDE and how to manipulate the data to be used by the FNO model.

When dealing with physical problems that encompass a temporal evolution, our brain is biased to thinking that there is only direction for the "arrow" of time. Meaning that it makes sense that time only moves forward. However, as presented by the great minds studying relativity and quantum mechanics in the physics world, the equations need to be respected in any direction of such. For this reason, it is possible to use the space-temporal grid of points (x, t) as a analogy to the spatial grid (x, y) used in task B. For this reason, the implementation of the FNO model should not change, and only the data processing needs to be done accordingly.

The main idea here is to use both x-axis and t-axis as any other direction in out solution space. This way, the FNO model will behave in a similar manner, looking for the eigen-states of the solution in both spatial and dimensional physical spaces. This is a good way of using the architecture of the FNO, as it damps the necessity of having two models: one for prediction of $u(x)$ and one to evaluate the temporal evolution of this field.

Summarizing, due to the efficiency and accuracy of the FNO model, this one is also used for this last task by creating the equivalency of spatial and temporal dimensions, as being intrinsic quantities of the PDE equation:

$$(x, y) \equiv (x, t) \tag{3.3}$$

Now, the model will receive the initial velocity $a(x)$, evaluate the lower modes in the directions of (x, y), and correctly predict the solution of $u(x, t)$. However, there is a conflict of dimensions between the inputs.

Since $a(x)$ is only measured for $t = 0$, how can the input have the knowledge of the entire temporal grid? The solution is to replicate manually the values of $a(x)$ for every time stamp of t. This way, the input will be a grid (x, t) with values of $a(x)$ replicated in the t-axis, leading to a similar problem statement as in task B.

The loss function

Once again, as the problem stated is similar to task B, the loss function used is:

$$\mathcal{L}(\theta) = \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} (\mathcal{G}_\theta(a^{(i)}(x, t), (x, t)) - u^{(i)}(x, t))^2 = \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} (u^{(i)}_{pred} - u^{(i)}_{true})^2 \tag{3.4}$$

This is simply a mean squared error used to analyze the differences between the predicted and real values of the velocity field $u(x, t)$. Since this is a gradient descent setup, the optimizer will adjust the model parameters θ such that the values of the loss function $\mathcal{L}$ is minimized. For lower values of loss, the predicted value from the model is closer to the real velocity field.

The network structure

The skeleton used for the FNO architecture is similar to the one used before. Once again, one layer of lifting is used to enhance the properties of the input, together with three global

transformation operations to analyze the eigen-states and modes of the solution in the Fourier space. Lastly, a projection block is used to bring down the dimensionality of the solution back to the (x, t) grid.

Layer	Neurons	Activation	Notes
Lifting Input Layer	3	–	Values of (a, x, t)
Lifting Layer 1	20	ReLU	Fully Connected
SpectralConv2D	20	ReLU	Conv1D performed after Conv2D
Conv1D Layer 1	20	ReLU	Used as piecewise convolution
SpectralConv2D	20	ReLU	Conv1D performed after Conv2D
Conv1D Layer 1	20	ReLU	Used as piecewise convolution
SpectralConv2D	20	ReLU	Conv1D performed after Conv2D
Conv1D Layer 1	20	ReLU	Used as piecewise convolution
Projection Input Layer	20	ReLU	From $(\mathcal{F}, R, W)$ gets values
Projection Hidden Layer	128	ReLU	Fully Connected
Output Layer	1	–	Predicts value of $u(x, t)$

Table 3.1: Architecture of Fourier Neural Operator Model

Additionally, the FNO model will consider for this scenario the following number of lower modes when evaluating the parameters in the Fourier space:

- Number of modes in x-direction $= 8$
- Number of modes in t-direction $= 8$

With this setup, the FNO model is created, after which the following information is obtained:

- Total Parameters $= 157\ 757$

This model is three times smaller than the FNO used for task B; however, it has a higher number for the modes considered in the Fourier space. After some research [3], it is noticeable that for temporal PDEs oscillations can happen in higher frequencies. For this reason, it is important that more modes are considered as these can contain very useful information for the learning process.

The code link

https://github.com/MiRoSi-52wab/DeepLearningPDEs/blob/main/TaskC.ipynb

3.3 Numerical Results

Experiment Setups

For this FNO model, the folliwing training setup was used:

Parameter	Value
Optimizer	Adam
Weight Decay	1×10^{-3}
Learning Rate (initial)	1×10^{-3}
Batch Size	50
Number of Epochs	100
Learning Rate Scheduler	StepLR
Decay Rate (α/N epochs)	0.5/25
Hardware	Google Collab T4 GPU
Precision	Float32 / Complex64
Early Stopping	No
Weight Clipping	No
Xavier Initialization	No

Table 3.2: Numerical Setup for Training FNO in Pytorch

In this case, the learning rate was increased and therefore the number of epochs was reduced. This allows the optimizer to change the parameters more quickly, leading to a faster training process (if the loss function shows a stable decay). Once again, the precision changes to Complex64 in order for the FNO to be able to identify the mods in the complex Fourier space.

Numerical Results

Lastly, the FNO is trained and tested for the first instance of the testing dataset. Below it is possible to see the initial velocity for the input $a(x)$:

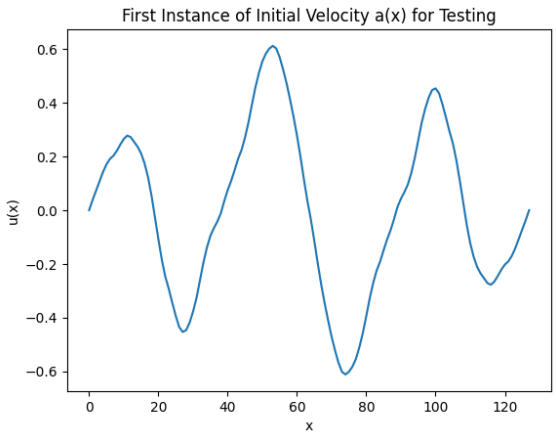

Figure 3.2: First instance of $a(x)$ for Testing

When this input was parsed into the trained FNO, the following results were obtained:

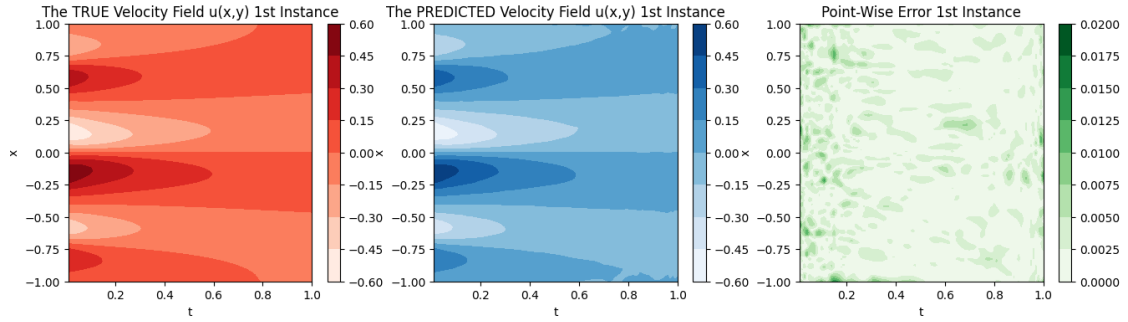

Figure 3.3: Prediction, Truth and Error for $u(x, t)$

As it is possible to see above, the differences between the real and predicted values of $u(x, t)$ are almost inexistent. This is further confirmed by the fact that the point-wise error only achieves a maximum value of 2%. For this reason, it is possible to confirm that this model is able to correctly predict the total velocity field given just the initial condition. For further plot regarding the accuracy of the FNO model, please refer to appendix B.

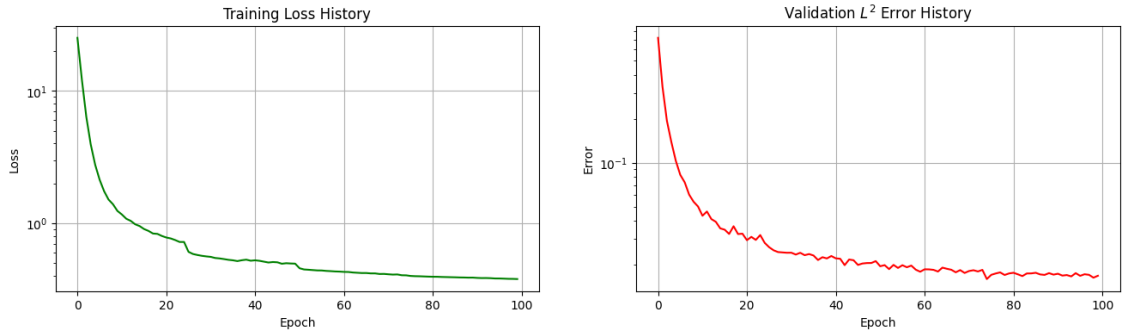

Figure 3.4: Loss and L^2 Error Values for Training and Validation (resp.)

The two plots above show that the training process parameters were well fitted for the learning of the FNO model. As explained before, this is due to the fact that there are no "wiggles" in both plots, showing that the learning rate and scheduler patience are well matched with the model's need to change the parameters θ.

Lastly, for this training process, the model structure and experiment setup led to the following time consumption:

- Training Time $\simeq$ 3.5 min

This further confirms that the FNO is well suited for problems with uniform grids in any direction (either spatial and/or temporal), leading to quick training process, great learning abilities and therefore optimal predictions of the desired quantities.

Chapter 4

Conclusions

After analyzing all the different models and their applications, it is possible to build a bigger idea of the advantages and disadvantages of each of them.

Initially, looking into problems where data was not labeled in pairs, the usage of PINNs and DeepRitz self-supervised showed to be crucial. For the forward problem, the DeepRitz method was selected due to the presence of the variational form of the PDE, together with an implementation of a mollifier to deal with hard-imposed boundary conditions. This combination, together with the fact that the PDE is of elliptic nature, showed a great result for the DeepRitz model, since this one handles non-smooth coefficients and sharp gradients better than PINNs.

However, for the inverse problem, it was possible to see that the DeepRitz approximates the solutions of $E(x)$ poorly. While for the forward problem the models only have to predict one field, for the inverse problem it is necessary to approximate both the parameter and solution space. For this reason, using the variational form of the PDE can lead to undesired results, for example setting $E(x) = 0$ creates a low value of the variational form, without actually respecting the physical properties of the system.

Moving forward to problems where data as labeled in pairs, the DeepONet and FNO models were introduced. For this purpose, it was possible to verify that as long as the grids are uniform, the FNO models shows a better performance with fewer number of parameters. This can be crucial when developing models that need to be trained and inferred quickly.

This is due to the fact that the FNO structure analyzes the global and local interactions between parameters and solution space in the Fourier space. For this reason, it is ideal for uniform grids, but not applicable to cases where the grid is not uniform. In those cases, the DeepONet is a better approach as it only requires consistent grids (meaning that they don't change between samples).

As a last point, it was interesting to investigate and understand that temporal problems can also be solved with the FNO model, just by correctly manipulating the input data. For such scenarios, using more Fourier modes is crucial to capture sharper temporal features.

Summarizing these comments:

- DeepRitz is excellent for forward problems with variational form and fixed boundary conditions.
- PINNs are more versatile for inverse problems but sensitive to noise and complex to understand the best configuration of losses weights.
- DeepONet is mesh-free and flexible, but less able to capture global and local relations in the solution space.

- FNOs excel on structured and uniform grids due to the analysis on the Fourier space, but are restricted to this scenario.

As future steps for investigation, it can be interesting to understand what models work better for non-uniform grids. As all the tasks were defined in structured grids, these models are only studied in this domain. However, in many applications it is not possible to have uniform grids, for example a meshing mechanism close to right-angle corners.

Another interesting topic, that requires more computational power, is to evaluate what the best selection of loss weights is. This study can lead into insights of what the models prioritize for certain applications.

Lastly, it would be great to test models that can predict temporal values out of the training scope. This means, for example, incorporating the FNO model with a LSTM (Long Short-Term Memory) model [4]. With this structure, the model is able to create sequences of events and predict new values based on the solutions context.

References

[1] E. Weinan and Yu. Bing. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *arXiv preprint arXiv:1710.00211*, 2017.

[2] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 2019.

[3] S. Qin, F. Lyu, W. Peng, D. Geng, J. Wang, X. Tang, S. Leroyer, N. Gao, X. Liu, and L. L. Wang. Toward a better understanding of fourier neural operators from a spectral perspective. *arXiv preprint arXiv:2404.07200*, 2024.

[4] K. Michalowska, S. Goswami, G. Karniadakis, and S. Riemer. Neural operator learning for long-time integration in dynamical systems with recurrent neural networks. *https://ar5iv.labs.arxiv.org/html/2303.02243*.

Appendix A

FNO Accuracy for Task B

In this section, some additional plots are added to show the accuracy of the FNO model for the prediction of $T(x, y)$. As explained before, the first instance of the testing data is actually one of the cases where both models perform worse. So selecting three random instances from the testing data, leads to the following:

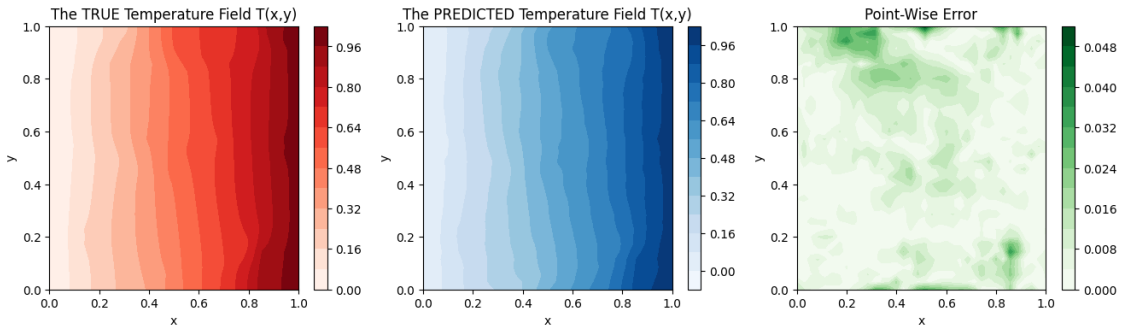

Figure A.1: FNO Results for a 2nd Instance of Testing - $T(x, y)$

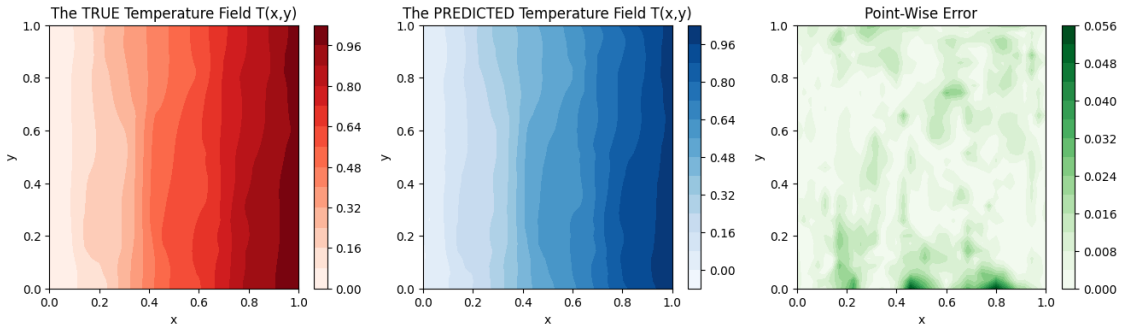

Figure A.2: FNO Results for a 3rd Instance of Testing - $T(x, y)$

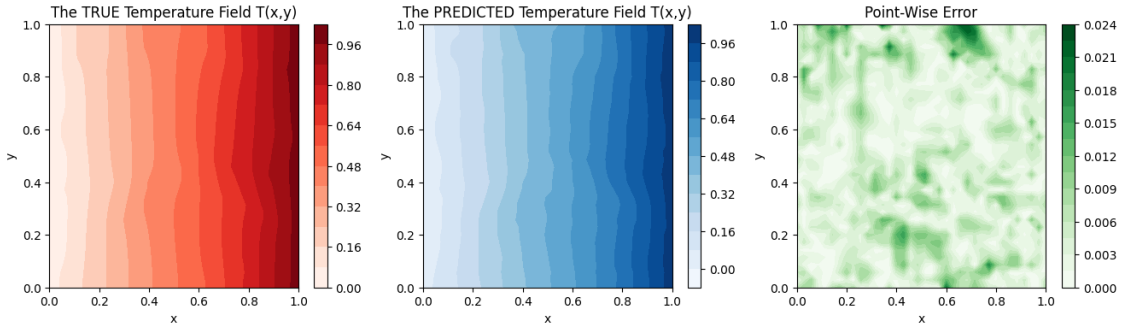

Figure A.3: FNO Results for a 4th Instance of Testing - $T(x, y)$

As seen above, the point-wise error is always lower than 6%, and given the reasons mentioned already previously, it proves that the FNO is the better suited model for uniform grids.

Appendix B

FNO Acuracy for Task C

In this section, some additional plots are added to show the accuracy of the FNO model for the prediction of $u(x, t)$. As explained before, the first instance of the testing data is actually one of the cases where both models perform worse. So selecting three random instances from the testing data, leads to the following:

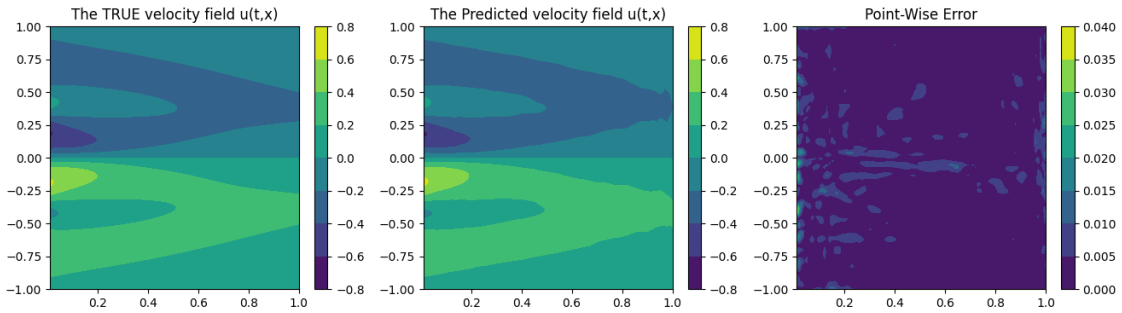

Figure B.1: FNO Results for a 2nd Instance of Testing - $u(x, t)$

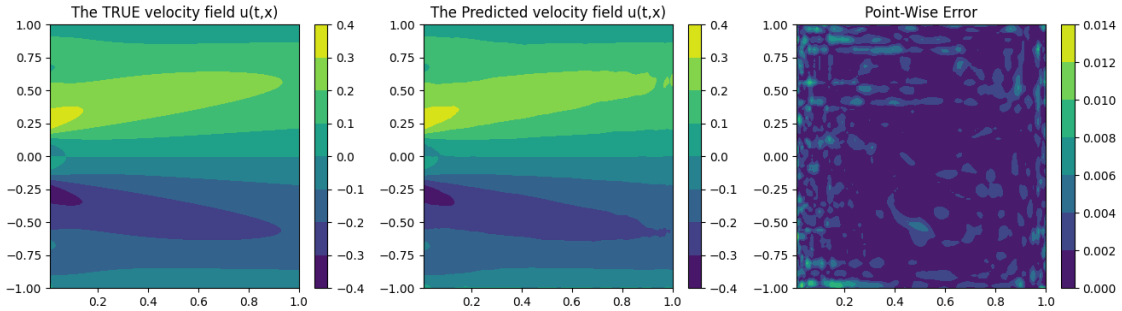

Figure B.2: FNO Results for a 3rd Instance of Testing - $u(x, t)$

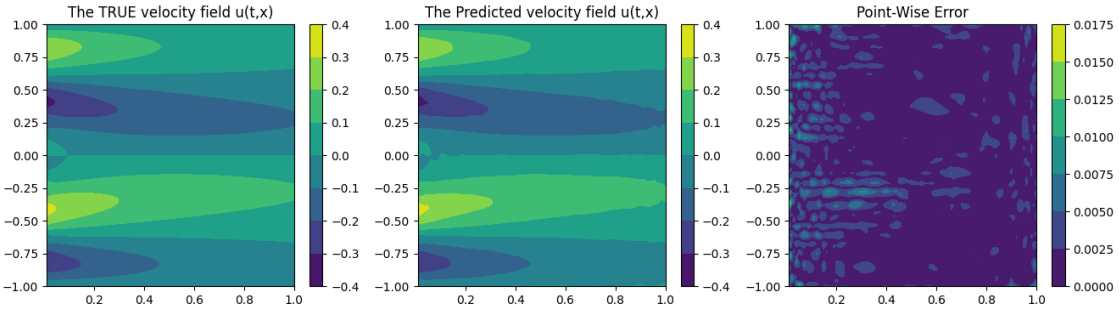

Figure B.3: FNO Results for a 4th Instance of Testing - $u(x, t)$

As seen above, the point-wise error is always lower than 4%, and given the reasons mentioned already previously, it proves that the FNO is the better suited model for uniform grids.