AISE Project 2: Neural Operators in practice

Patrick Dowd

June 26, 2024

1 Task 1

1.1 Solving the coupled ODE system

To solve the coupled ODE system, I implemented a simple RK4 solver. I modelled the system with a four-dimensional state y representing $[x, \dot{x}, \theta, \dot{\theta}]$. The dynamics of the system are therefore:

$$\begin{aligned}
\dot{y}[0] &= y[1] \\
\dot{y}[1] &= \frac{\mathrm{F}(t) - mg\cos(y[2])\sin(y[2]) + (mly[3]^2)\sin(y[2])}{M + m(1 - \cos(y[2])^2)} \\
\dot{y}[2] &= y[3] \\
\dot{y}[3] &= \frac{g\sin(y[2]) - \cos(y[2])\dot{y}[1]}{l}
\end{aligned}$$

I implemented the solver using `jax`, so it could be used in the loss function of the model in section 1.2.

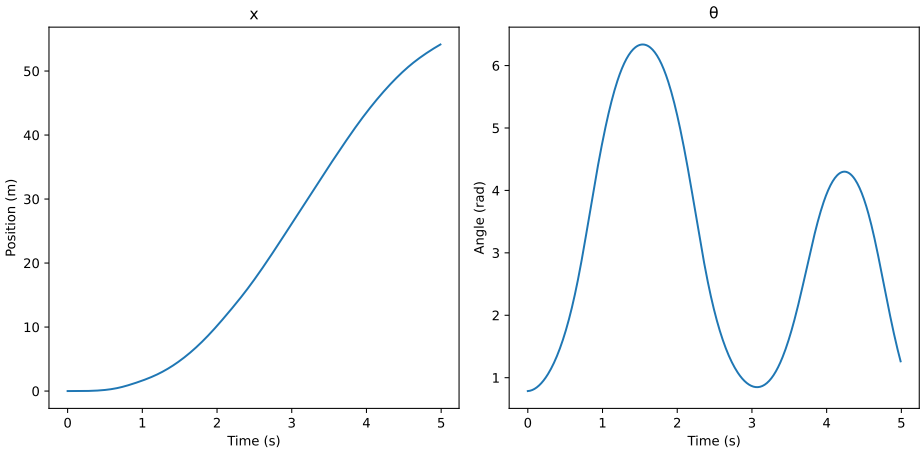


Figure 1: Trajectory of the system for the given initial conditions and parameters, as calculated by my RK4 solver. θ is the angular position of the pendulum and x is the linear position of the cart.

Figure 1 shows the output of the solver for the given initial conditions and parameters ($M = 1.0, m = 0.1, l = 1.0, g = 9.81, x(t_0) = 0, \dot{x}(t_0) = 0, \theta(0) = \pi/4, \dot{\theta}(0) = 0, F(t) = 10\sin(t)$). The plotted motion is reasonable, implying the correctness of the solver.

1.2 Learning to balance the pendulum

The given task was to learn a forcing function $F(t)$ that causes the pendulum to remain in the inverted position (unstable equlibrium) for the final quarter of the trajectory. Since the quantity we wish to control is the output of the ODE solver, the solver must be incorporated into the training loop of the model. As mentioned above, the solver was written in the `jax` autodifferentiation framework. This

allows losses to be defined on the solver output and propagated back to the model parameters, enabling them to be updated in the direction of the loss gradient.

Some relevant lines from my implementation are shown below:

```
def __init__(self, input_dim, hidden_dim, output_dim, key, y0, t0, tf, h):
    ...
    self.ode_solver = partial(ODESolver, y0, t0, tf, h)

def _loss(self, forcing_model):
    _, y = self.ode_solver(forcing_model)
    ...
```

To control the trajectory in the desired way, I defined the loss below:

$$L(y) = \frac{1}{n-k}\sum_{i=k}^{n}(y[2]_i^2 + y[3]_i^2)$$

where k is the index of the start of the penalisation and n is the length of the discretized trajectory. Initially I took $k = \lfloor 0.75n \rfloor$. However, I noticed that the model had difficulty learning to stabilise the pendulum prior to the onset of the penalisation. To address this, I took $k = \lfloor \frac{11}{16}n \rfloor$. The loss also includes the angular velocity of the pendulum in addition to the angular position. Including this extra term seemed to improve the convergence of the model.

The architecture that I chose for F(t) was a fully connected network with two hidden layers and a width of 64. I trained the model for 500000 steps and used an Adam optimiser with a learning rate of 0.001.

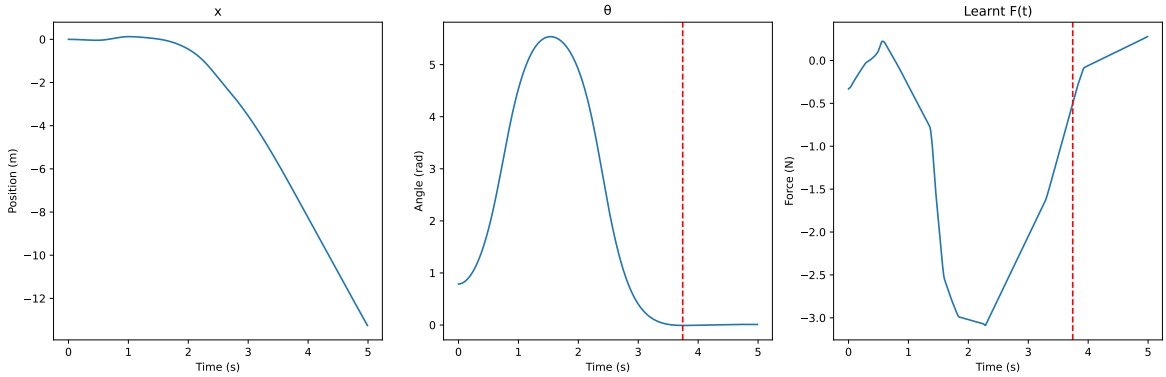


Figure 2: Trajectory of the system for the learnt forcing function, given the initial conditions and system constants. The red dashed lines indicate the start of the period in which the pendulum should be inverted according to the task description.

Figure 2 shows the learnt forcing function and the resulting system trajectory. As the figure shows, the pendulum remains almost perfectly in the inverted position over the desired period.

2 Task 2 - Recreating PDE-FIND

NOTE: Correct predictions are highlighted in yellow.

The algorithm that I used for this task follows the algorithm given in PDE-FIND closely [1]. Specifically, I first generate features from the data consisting of either numerical derivatives, polynomials, or a mixture of the two. The specific library I use for each dataset varied and will be explained further below. I then calculate the numerical time derivative of the observed data. Finally, I use the sparse solver provided in the PDE-FIND codebase to solve the linear system:

$$\mathbf{u}_t = \Theta(\mathbf{u})\xi$$

where $\Theta(\mathbf{u})$ is a matrix whose columns are the features described above.

When deciding which method I should use to calculate the numerical derivatives, I first visualised the datasets. All three of the datasets showed very smooth solution surfaces, indicating that there was no noise. Datasets 1 and 2 can be seen in Figures 5a and 5b, respectively. Additionally, I implemented a simple function to test if the grid was perfectly regular in all dimensions for each dataset. The spatial grids for all datasets were perfectly regular. The temporal grids for datasets two and three were not perfectly regular, but the deviation was extremely small and clearly arose from some truncation or rounding error. An example of the function output is provided below, for dataset three:

```
Grid is uniform along x dimension
Grid is uniform along y dimension
Grid is not uniform along t dimension
Max delta: 0.05000019
Min delta: 0.049999237
Mean delta: 0.05
Std dev of delta: 2.3049132e-07
```

Since the data appeared noise-free and the grids were effectively regular, I opted to calculate numerical derivatives using a simple centred finite-difference approach.

For dataset 1, I used the following library:

$$[\mathbf{1}, u, u^2, u^3, u_x, uu_x, u^2u_x, u^3u_x, u_{xx}, uu_{xx}, u^2u_{xx}, u^3u_{xx}, u_{xxx}, uu_{xxx}, u^2u_{xxx}, u^3u_{xxx}]$$

The predicted equation was:

$$u_t = (-1.000830)uu_x + (0.100173)u_{xx}$$

This is Burger's equation, which fits the data well based on visual inspection. The algorithm worked out of the box for this dataset, requiring no tweaking.

For dataset 2, I initially used the same library as for dataset 1 above. However, the sparse solver output the following prediction:

$$u_t = (-7.245200)u^2u_x$$

While this equation may be reasonable, the visual similarity between the provided data and that in the Korteweg-De Vries example in the supplementary material of PDE-FIND [1] prompted me to tweak the algorithm to find a better solution. I noticed that in their KdV example, they only use up to second order polynomials (not third order as above). I therefore reduced the library to the following set:

$$[\mathbf{1}, u, u^2, u_x, uu_x, u^2u_x, u_{xx}, uu_{xx}, u^2u_{xx}, u_{xxx}, uu_{xxx}, u^2u_{xxx}]$$

With this reduced library of functions, the algorithm produced the KdV equation:

$$u_t = (-5.955390)uu_x + (-0.987654)u_{xxx}$$

It is not reassuring that the algorithm performs differently in these two cases. In both cases, numerical derivatives were calculated using the finite difference approach and the correct features were present in the matrix. This experiment reveals that the algorithm as proposed in PDE-FIND is not robust.

The final dataset involves a set of coupled equations. The linear system to be solved is therefore:

$$[\mathbf{u}_t\mathbf{v}_t] = \Theta(\mathbf{u}, \mathbf{v})[\xi_u\xi_v]$$

It can be solved in the same way as the uncoupled systems by separating the first and last matrices into their columns and solving separately:

$$\mathbf{u}_t = \Theta(\mathbf{u}, \mathbf{v})\xi_u$$
$$\mathbf{v}_t = \Theta(\mathbf{u}, \mathbf{v})\xi_v$$

The library used for this dataset is shown in Equation 2 in Appendix A, and includes both pure features in each solution and mixed features involving information from both solutions. It includes polynomials up to the third degree and derivatives up to the third order.

Another new aspect of dataset 3 compared to the two previous datasets is that its spatial grid is two-dimensional. This introduces the issue of how to calculate mixed derivatives. It also introduces the need to downsample the linear system since the system resulting from flattening across three dimensions is too large to be solved efficiently on an ordinary computer.

My first approach was to use finite differences and calculate the pure derivatives for all points in the spatio-temporal grid. To calculate mixed derivatives I took the relevant pure partial derivative in the x-direction (which is defined on the whole grid) and used finite differences in the y-direction. To subsample the system, I took random rows from both the time derivatives and feature matrix.

The resulting equations from this approach were as follows:

$$\begin{aligned} u_t &= (0.252823)v + (0.705980)v^3 + (0.930026)u + (-0.930882)uv^2 + (0.706592)u^2v \\ &\quad + (-0.927188)u^3 + (0.094984)u_{xx} + (0.093430)u_{yy} \\ v_t &= (0.893074)v + (-0.888904)v^3 + (-0.274821)u + (-0.682719)uv^2 + (-0.890469)u^2v \\ &\quad + (-0.680951)u^3 + (0.089600)v_{xx} + (0.093970)v_{yy} \end{aligned}$$

This predicted set of coupled equations does not correspond with a well known physical law. Specifically, it does not correspond with the reaction-diffusion equation that the dataset appears to correspond to. The most notable error is that the time derivative of each solution is predicted to rely linearly on the other solution.

To address this discrepancy, I reviewed the code that the authors of PDE-FIND [1] used in their reaction-diffusion example. Even for the dataset without additional noise, they fit a polynomial function around each sampled datapoint and used this approximation when calculating derivatives. Aiming to replicate their results, I implemented a very similar method.

To begin, I repeatedly sample (x_i, y_i, t_i) from the dataset. To calculate, for example, the first pure derivative in the x-direction, I take 9 points centred around (x_i, y_i, t_i) along the x-axis. I fit a 4th order Chebyshev polynomial using these points and calculated pure derivatives at (x_i, y_i, t_i) using the function `numpy.polynomial.chebyshev.Chebyshev.deriv`.

To calculate mixed derivatives, I fit polynomials in the x-direction at the grid points with y-values one above and one below the chosen point. That is, I applied the method above at the points (x_i, y_{i+1}, t_i) and (x_i, y_{i-1}, t_i). I then use a centred finite difference in the y-direction to calculate the mixed derivative at (x_i, y_i, t_i).

With this approach, downsampling is built into the algorithm and the resulting system can be used directly. The output of the solver is given below:

$$\begin{aligned} u_t &= (0.999979)v^3 + (0.999913)u + (-0.999882)uv^2 + (0.999977)u^2v + (-0.999900)u^3 \\ &\quad + (0.099972)u_{xx} + (0.100004)u_{yy} \\ v_t &= (1.000278)v + (-1.000293)v^3 + (-0.999971)uv^2 + (-1.000268)u^2v + (-0.999978)u^3 \\ &\quad + (0.100032)v_{xx} + (0.099968)v_{yy} \end{aligned}$$

This output corresponds to the standard reaction-diffusion equation in 2D and seems correct.

Again, it is not reassuring that using the centred difference approach on seemingly clean data on a regular grid produced the wrong results. In fact, across the three datasets I had to follow the implementation in PDE-FIND very closely in order to reproduce their results. This fact seriously undermines the robustness and generalisability of the PDE-FIND approach and limits its effectiveness on data from a truly unknown equation.

To improve the generality and convergence of the algorithm, it is necessary to have some sort of validation layer. For example, since the algorithm is susceptible to a changing library one could iterate over a number of libraries. For each library, the least squares error of the output solution could be recorded and displayed alongside the predicted solutions. This would give the user some metric upon which to make their choice, as well as a measure of the model's uncertainty. Further, the validation layer could be more advanced and dynamically reduce or expand the library based on the algorithm's performance on certain subsets of the features. This would help the convergence towards the right solution and would also improve the generality of the method, as the user does not need to closely guess the equation's terms when designing the feature matrix.

Another idea is to train a PINN that incorporates the predicted equation as a residual loss term. The boundary conditions could be taken from the provided data. The mean squared error between the

PINN output at the domain points and the provided dataset could be used to quantify the accuracy of the predicted equation if the noise level in the dataset is approximately known. More importantly, it would allow the user to identify particular areas in the domain where the model/equation breaks down (can also be done by finding the rows in the linear system with the largest errors) and demonstrating what the solution should be in that area according to the predicted equation. Standard numerical methods could also be used to calculate the solution according to the predicted equation.

3 Bonus Task

3.1 Dataset Creation

To generate the datasets, I implemented random samplers for the three function classes as well as a simple solver for the Poisson equation. The full implementation of my random samplers can be found in the provided code, so I will just give an overview here.

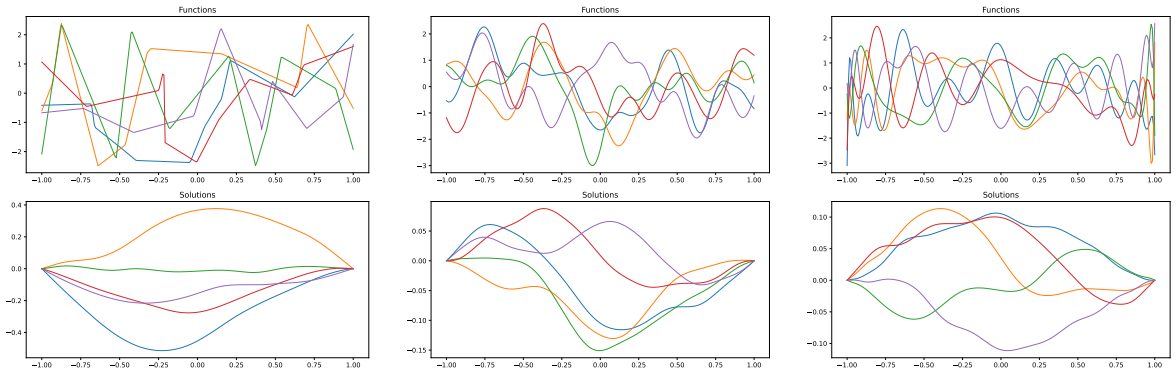
The Gaussian Process sampler used an RBF kernel (Equation 1) with $\sigma = 0.1$ to define the covariance matrix, so that there were approximately seven local maxima and minima in $[-1, 1]$. The mean for the Gaussian Processes was 0.

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) \tag{1}$$

The Chebyshev sampler used the first 20 Chebyshev functions and drew coefficients from $\mathcal{N}(0, 1)$

The Piecewise Linear sampler divided the domain $[-1, 1]$ into 14 segments based on random breakpoints. The value of the function at the breakpoints was randomly drawn from a uniform distribution centred on 0. This set of information fully defines a continuous piecewise-linear function, so the slopes were calculated from it.

For all datasets, I chose parameters so that each function should have around 7 local maxima and minima over the domain $[-1, 1]$. Additionally, I used `sklearn.preprocessing.StandardScaler` on each GP and Chebyshev function individually. I did this because it was difficult to control the range of the functions (particularly for the Chebyshev polynomials). By normalising the range, I ensure that each model sees data in the same range and the generalisability of the operator mapping can be properly assessed. Each function was discretized on a regular 256 dimensional grid on the domain $[-1, 1]$. Some sample functions and their corresponding solutions can be seen in Figure 3.



(a) Piecewise Linear (b) Gaussian Process (c) Chebyshev Polynomial

Figure 3: Samples of functions and the corresponding solutions of the Poisson equation for each of the three function classes.

To find the corresponding solution to the Poisson equation, I used `np.linalg.solve` to solve the

below linear system.

$$f = \frac{1}{\Delta x^2} \begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & \dots & -1 & 2 \end{bmatrix} u$$

In the system above, Δx is the grid spacing, f is the sampled forcing function and u is the discretized solution. Note that the system is defined in the interior of the grid. On the boundaries, the homogeneous boundary conditions are used.

3.2 Training and fine-tuning

I experimented with both FNOs and CNOs to learn the operator that maps the forcing function to the PDE solution. FNOs tended to produce extremely oscillatory solutions, even after experimenting with limiting the number of fourier modes. I therefore focussed on CNOs for the remainder of the analysis.

I trained the model for each class on 100 function/solution pairs drawn from the random sampler, up to 500 epochs. I validated the models on another 50 function/solution pairs drawn separately from the random sampler. The retained model for each class was the one with the lowest validation score during training. I then drew another 100 function/solution pairs from the random sampler to serve as the testing dataset. This ensures that the same testing dataset is used for both of the testing function classes. Sample model outputs after training are shown in Figure 4.

During the testing phase, I took the first 20 pairs from the testing dataset to use for train the model during finetuning. I took the last 12 pairs from the testing set to use for validation during finetuning. To evaluate the model's performance I took the mean squared error of the model's output on the 88 pairs not used for validation during finetuning. For the zero-shot error, this occurred before finetuning and for the finetuning-error it occurred after finetuning. During finetuning, the model could train for up to 1000 epochs.

3.3 Analysis

Training Class	**Loss on Class**	**Testing Class**	**Testing Losses**
Piecewise Linear	1.81e-05	Chebyshev	Zero-shot: 9.39e-06 Finetuned: 1.36e-04
		Gaussian Process	Zero-shot: 1.36e-05 Finetuned: 1.13e-04
Gaussian Process	1.75e-06	Piecewise Linear	Zero-shot: 2.42e-04 Finetuned: 3.48e-04
		Chebyshev	Zero-shot: 8.91e-06 Finetuned: 1.29e-04
Chebyshev	1.26e-06	Piecewise Linear	Zero-shot: 1.42e-03 Finetuned: 3.15e-04
		Gaussian Process	Zero-shot: 5.63e-05 Finetuned: 1.06e-04

Table 1: Intra and inter-class losses for Poisson solution operators on forcing functions drawn from different function classes.

Table 1 shows the results of the experiments. One notable feature of the results is that finetuning harmed the model's performance in most cases. The only case in which finetuning resulted in a performance gain was for the model trained on Chebyshev polynomials and tested on piecewise linear functions. The model trained on gaussian processes and tested on piecewise linear function showed only a modest decline in performance. The other models saw their losses increase by one or even two orders of magnitude on the test set. I interpret this as an indication that the zero-shot performance of

most models was already quite good, particularly if the function classes were similar (that is, there was little new information to learn from the finetuning set). In these cases, updating the weights knocked the model out of a local minima in the loss surface and worsened performance. This hypothesis is supported by the fact that the loss curves during training are very noisy (indicating that the loss surface is very non-convex).

Overall, the performance of the GP and Chebyshev models on the piecewise linear dataset was poor. This is likely because the GP and Chebyshev functions have zero mean by construction, whereas each piecewise linear function may not. This has a large effect on the magnitude of the solution - solutions in Figure 4a range from -0.35 to 0.35 where as solutions in Figure 4b and 4c range from -0.15 to 0.15. Also, the GP and Chebyshev functions are in $C^\infty([-1,1])$ while the piecewise linear functions are in $C^0([-1,1])$. Since $C^\infty([-1,1]) \subset C^0([-1,1])$, it makes sense that the model trained on piecewise linear functions should generalise better than the models trained on the classes of smooth functions. It also makes sense that the models trained on functions in $C^\infty([-1,1])$ should generalise well to other functions in $C^\infty([-1,1])$ but not those in $C^0([-1,1])$. This analysis is supported by the relatively good zero-shot losses for the model trained on piecewise linear functions - the zero-shot losses come within one order of magnitude of the intra-class validation loss. Also, the zero-shot losses for the piecewise-linear model are as low as or lower than the zero-shot losses of the GP/Chebyshev models tested on the other smooth class. Therefore, training on piecewise linear functions seems to offer superior zero-shot performance. However, the minimum intra-class validation loss for the piecewise linear model was an order of magnitude worse than for the smooth functions. In fact, the piecewise-linear model's zero-shot performance on the Chebyshev and GP datasets was better than its performance on its own class. This indicates that the piecewise linear class is more difficult to learn than the smooth classes and reinforces the analysis above.

PINNs are not a suitable choice for solving this task. PINNs are well suited to learning the solution of PDEs without input data. They are also useful for learning the solution to PDEs given noisy observations of the PDEs, and for inferring some unobserved term in the PDE such as a coefficient function. However, all of these use cases are limited to mappings $\Omega \mapsto \mathbb{R}^n$ (that is, for learning a function that maps the domain to the solution output). In this task, we aim to learn a mapping $L^2([-1,1]) \mapsto L^2([-1,1])$ (the discretized form is $\mathbb{R}^{256} \mapsto \mathbb{R}^{256}$). Another limitation of PINNs for this task is that they learn just one PDE (unless they are conditioned). Given the large variation in forcing functions within, and especially between, each function class, it would be very difficult to train a conditioned PINN to accurately learn the whole class of solutions to the Poisson equation.

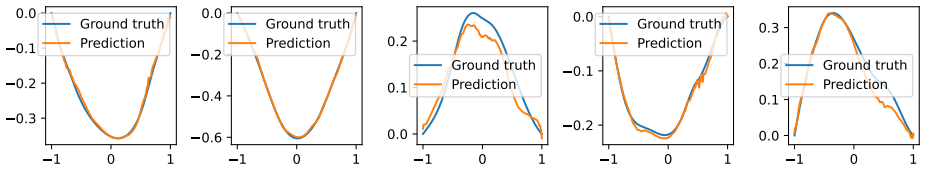
References

[1] Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(4):e1602614, 2017.

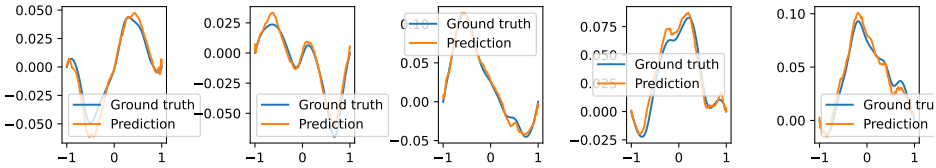
A Appendix A: Additional Information

$$
\begin{aligned}
&\mathbf{1}, v, v^2, v^3, u, uv, uv^2, u^2, u^2v, u^3, u_x, vu_x, v^2u_x, v^3u_x, uu_x, uvu_x, uv^2u_x, u^2u_x, u^2vu_x, u^3u_x,\\
&u_y, vu_y, v^2u_y, v^3u_y, uu_y, uvu_y, uv^2u_y, u^2u_y, u^2vu_y, u^3u_y, u_{xx}, vu_{xx}, v^2u_{xx}, v^3u_{xx},\\
&uu_{xx}, uvu_{xx}, uv^2u_{xx}, u^2u_{xx}, u^2vu_{xx}, u^3u_{xx}, u_{yy}, vu_{yy}, v^2u_{yy}, v^3u_{yy}, uu_{yy},\\
&uvu_{yy}, uv^2u_{yy}, u^2u_{yy}, u^2vu_{yy}, u^3u_{yy}, v_x, vv_x, v^2v_x, v^3v_x, uv_x, uvv_x, uv^2v_x, u^2v_x,\\
&u^2vv_x, u^3v_x, v_y, vv_y, v^2v_y, v^3v_y, uv_y, uvv_y, uv^2v_y, u^2v_y, u^2vv_y, u^3v_y, v_{xx}, vv_{xx},\\
&v^2v_{xx}, v^3v_{xx}, uv_{xx}, uvv_{xx}, uv^2v_{xx}, u^2v_{xx}, u^2vv_{xx}, u^3v_{xx}, v_{yy}, vv_{yy}, v^2v_{yy},\\
&v^3v_{yy}, uv_{yy}, uvv_{yy}, uv^2v_{yy}, u^2v_{yy}, u^2vv_{yy}, u^3v_{yy}, u_{xy}, vu_{xy}, v^2u_{xy}, v^3u_{xy},\\
&uu_{xy}, uvu_{xy}, uv^2u_{xy}, u^2u_{xy}, u^2vu_{xy}, u^3u_{xy}, v_{xy}, vv_{xy}, v^2v_{xy}, v^3v_{xy}, uv_{xy},\\
&uvv_{xy}, uv^2v_{xy}, u^2v_{xy}, u^2vv_{xy}, u^3v_{xy}
\end{aligned}
\tag{2}
$$

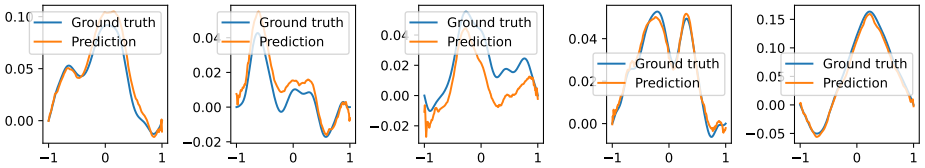
Library of features used for Dataset 3 in Task 2



(a) Piecewise Linear

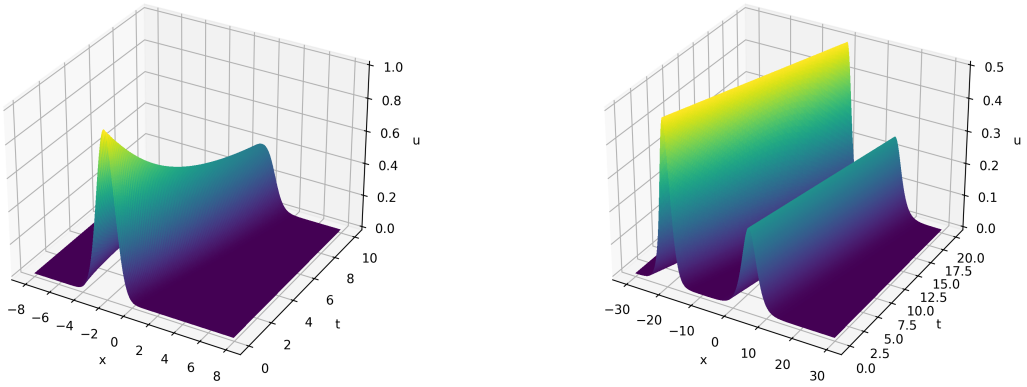


(b) Gaussian Process



(c) Chebyshev Polynomial

Figure 4: Sample model outputs and ground truth solutions for each of the three function classes.



(a) Visualisation of dataset 1

(b) Visualisation of dataset 2

Figure 5: Visualisation of datasets in Task 2, demonstrating the absence of noise.