Fourier Neural Operators: Hyperparameter Optimization and Autoregressive Time Step Prediction

Shawn J. Koohy
Department of Mechanical Engineering and Applied Mechanics
University of Pennsylvania
Philadelphia, PA 19014, USA
skoohy@seas.upenn.edu

Abstract

Fourier Neural Operators (FNOs) are a powerful tool for learning mappings between infinite-dimensional function spaces, offering efficiency and resolution invariance for solving partial differential equations (PDEs). This paper explores the application of FNOs on two benchmark problems: the Burgers' equation and the Korteweg–De Vries (KdV) equation. For the Burgers' equation, we conduct an extensive hyperparameter optimization study to minimize the L^2 error and loss allowing the model to achieve high accuracy across all examples. For the KdV equation, we extend the FNO framework to an autoregressive setting, predicting multiple future time steps based on previous predictions. Our results demonstrate the effectiveness and expressivity of FNOs, achieving an average L^2 error of 0.0016 for Burgers' and 0.08 for KdV. This showcases the potential of FNOs to solve complex PDE systems in a wide range of applications.

1 Introduction

Neural operators are a class of machine learning techniques used to learn mappings between infinite-dimensional function spaces. In many applications of machine learning, data exists and is assumed to be within finite-dimensional vector spaces, examples include images, videos, words/text, time series, and graphs. However, in the setting of science and engineering problems, many of the systems we wish to model reside in infinite-dimensional vector spaces. A common technique within deep learning is finding ways to modify a neural network's inductive bias to better exploit the structure of the data we are working with. This type of thinking has been very successful in the past and several alternatives to the vanilla feed-forward neural network (FNN) or multi-layer perceptron (MLP) have been proposed. In the case where we are working with images the convolutional neural network (CNN) [7] was popularized, for sequences or time-series data the recurrent neural network (RNN) [5, 12], for graph data the graph neural network (GNN) [13], and many others [4, 14, 15]. We are interested in working with infinite-dimensional or functional data to predict a wide range of continuous physical quantities such as velocity, pressure, temperature, and magnetic, or electric fields. This is what the neural operator [9] was introduced for. An MLP performs a series of alternating compositions between affine and non-linear transformations given by

$$W^{(L)} \circ \sigma \circ W^{(L-1)} \circ \cdots \circ \sigma \circ W^{(1)}, \tag{1}$$

where W is the affine transformation and σ is the activation function. The affine transformation used above, call it f, defines a mapping on some input vector $\mathbf{x}$ such that $f(\mathbf{x}) = A\mathbf{x} + b$, where A would be the weight matrix and b the bias vector in the context of MLPs. Using this form of an affine transformation assumes that we are in a finite-dimensional space, therefore we seek to replace this with one that exists in an infinite-dimensional space. To move into the infinite-dimensional case we replace A with a linear operator $\mathcal{K}$ and b with some fixed function c, our new affine transformation

.

is then $\mathcal{G}(f)(x) = \mathcal{K}(f)(x) + c(x)$, where f is now a function and $\mathcal{G}$ the operator we wish to learn (as opposed to learning the transformation W in the MLP sense). To maintain the form of an affine transformation we choose $\mathcal{K}$ to be a (bounded) linear operator, a common choice is the kernel integral operator defined as

$$\mathcal{K}(v)(x) := \int \kappa(x, y)v(y)\,\mathrm{d}y, \tag{2}$$

where κ is a kernel function. We consider $\mathcal{K}$ to represent a global operator, to also retain local behaviors of our continuous field we let $c(x) = Wx$, where W is now a local transformation, typically a convolution. Therefore the $(l+1)$-th layer of a neural operator $v^{(l+1)}$ is defined to be

$$v^{(l+1)}(x) = \sigma\left(Wv^{(l)}(x) + \int \kappa(x, y)v^{(l)}(y)\,\mathrm{d}y\right). \tag{3}$$

Note that it's possible that our kernel function $\kappa(x, y)$ may depend on the input features of our data $(a(x), a(y))$. It is also possible that the dimension of the input function space does not equal the dimension of the output function space, because of this we apply a lifting layer on the inputs to lift them into higher dimensional space to extract abstract features and then forward pass the lifted inputs through the neural operator layers and finally apply a projection layer to project the outputs into the desired dimension of the output space. We parametrize the unknowns W and κ by a set parameters ϕ and the full neural operator is then written as

$$\mathcal{G}_\phi := \mathcal{P} \circ v^{(L)} \circ v^{(L-1)} \circ \cdots \circ v^{(1)} \circ \mathcal{Q}, \tag{4}$$

where $\mathcal{P}$, $\mathcal{Q}$ are the projection and lifting layers, v the neural operator layer, and $\mathcal{G}_\phi$ the operator we wish to learn parameterized by ϕ. In the first introduction of neural operators, it was proposed to approximate the kernel integral operator by a "Monte Carlo sum via a message passing graph network with edge weights" [9]. Without going into too much detail, since it's not the aim of this project, the computation essentially came down to letting

$$v^{(l+1)}(x) = \sigma\left(Wv^{(l)}(x) + \frac{1}{|N(x)|}\sum_{y\in N(x)} \kappa_\phi(x, y, a(x), a(y))v^{(l)}(y)\right), \tag{5}$$

where $N(x)$ is the neighborhood of x. Monte Carlo generally has a slow convergence rate and new methods have been developed to improve this [8].

In section 2 we discuss the background of the main focus of this paper, the Fourier neural operator (FNO), in section 3 we present two numerical experiments performed using the FNO including one on the Burgers' equation (see section 3.1) and another on the Korteweg–De Vries equation (see section 3.2). Concluding remarks are given in section 4 and additional results of the trained FNOs can be seen in appendix A.

2 Fourier Neural Operators

The Fourier neural operator aims to provide a more efficient and expressive way of computing the kernel integral operator $\mathcal{K}_\phi$ by applying the convolution theorem. The convolution theorem states that the Fourier transform (denoted $\mathcal{F}$) of the convolution (denoted $\star$) between two functions f and g is the product of the Fourier transform of the f and the Fourier transform of g. This means that

$$\mathcal{F}\{f \star g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}. \tag{6}$$

The first thing to note is that from the convolution theorem, we can directly write the convolution between f and g by simply taking the inverse Fourier transform and obtaining

$$f \star g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}. \tag{7}$$

In the FNO framework, we assume that the kernel $\kappa_\phi(x, y)$ is stationary meaning that the output of the kernel depends only on the difference of the inputs, not the individual inputs and find that $\kappa_\phi(x, y) = \kappa_\phi(x - y)$ .Our operator $\mathcal{K}(v)(x)$ then becomes

$$\mathcal{K}(v)(x) = \int \kappa_\phi(x, y)v(y)\,\mathrm{d}y = \int \kappa_\phi(x - y)v(y)\,\mathrm{d}y = (\kappa_\phi \star v)(x), \tag{8}$$

we can see this as nothing but a convolution. It then follows that

$$(\kappa_\phi \star v)(x) = \mathcal{F}^{-1}\{\mathcal{F}\{\kappa_\phi\} \cdot \mathcal{F}\{v\}\}(x), \quad R_\phi = \mathcal{F}\{\kappa_\phi\}. \tag{9}$$

Our neural operator layer, now referred to as a Fourier layer can be written as

$$v^{(l+1)}(x) = \sigma\left(Wv^{(l)}(x) + \mathcal{F}^{-1}\{R_\phi \cdot \mathcal{F}\{v\}\}(x)\right), \tag{10}$$

this eliminates any need for approximating integrals and allows the model to learn the solution space within the frequency domain. Note that we call this new form of the kernel integral operator the spectral convolution. The Fourier transform admits an infinite series which we cannot do in practice, due to this we truncate the Fourier series at a user-specified number of modes labeled k_{max}. The fewer modes we use the more the FNO will focus on learning purely global features. Including the later modes will allow the network to learn more local behaviors (in addition to the global ones). The choice of determining the number of modes to use depends on the problem being worked on, for highly localized functions more modes may be beneficial to capture those features, for very smooth functions fewer modes may be needed, in this context domain knowledge can be useful. Typical values for k_{max} appear to range roughly between 10 to 30. We do not know the actual form of R_ϕ which is why we parametrized it and we do so by a tensor of size $(2, k_{\text{max}}, w, w)$ where the two elements of the first channel represent the real and complex part of the Fourier transform, k_{max} the number modes and the last two channels the width or size of the tensor (similar to the height and width of an image). All computations in the FNO are well-established and commonly used operations except the spectral convolution. In JAX [2] an example implementation of the spectral convolution may look like

```
class SpectralConv1d(nn.Module):
    width: int
    modes: int

    def setup(self):
        scale = 1/(self.width*self.width)
        self.weights = self.param("global_kernel", lambda rng, shape:
            random.uniform(rng, shape, minval=-scale, maxval=scale),
                                (2, self.modes, self.width, self.width))

    @nn.compact
    def __call__(self, x: jnp.ndarray) -> jnp.ndarray:
        spatial_resolution = x.shape[0]

        x_ft = jnp.fft.rfft(x, axis=0)
        x_ft_trunc = x_ft[:self.modes,:]

        R = jax.lax.complex(self.weights[0,...], self.weights[1,...])

        R_x_ft = jnp.einsum("Mio,Mi->Mo", R, x_ft_trunc)

        result = jnp.zeros_like(x_ft, dtype=x_ft.dtype)
        result = result.at[:self.modes,:].set(R_x_ft)

        inv_ft_R_x_ft = jnp.fft.irfft(result, n=spatial_resolution, axis
            =0)
        return inv_ft_R_x_ft
```

The class `SpectralConv1d` takes in the number of modes k_{max} and the width used for the parametrization of R_ϕ, the scale is used for initialization purposes. The call of `SpectralConv1d` takes an input of size (spatial grid, channels). During the call the Fourier transform, $\mathcal{F}\{v\}$, is computed, truncated and from here we compute $R_\phi \cdot \mathcal{F}\{v\}$ by `jnp.einsum`. A zero matrix is created such that the first k_{max} rows will be the results of the $R_\phi \cdot \mathcal{F}\{v\}$, and the rest of the elements are kept zero. This ensures that only the first k_{max} modes contribute to the convolution operation while the remaining elements are kept as zeros to preserve the original shape and dimensionality of the Fourier-transformed input. We are informing the inverse Fourier transform, $\mathcal{F}^{-1}\{R_\phi \cdot \mathcal{F}\{v\}\}$, of the dimensionality of the original Fourier transformed input but only allow the truncated modes to influence the output. The axis names in `jnp.einsum` refers to the modes (M), in channels (i), and

out channels (o). There may be scenarios where the in channels and out channels are different but in this case,e we consider `i=o=width`. Let d_g be the number of spatial points (spatial resolution) and d_v be the width. The axis names are setup as `"Mio, Mi->Mo"` because $v \in \mathbb{R}^{d_g \times d_v}$, $\mathcal{F}\{v\} \in \mathbb{C}^{Z \times d_v}$ which is truncated to be $\mathbb{C}^{k_{\max} \times d_v}$ and $R_\phi \in \mathbb{C}^{k_{\max} \times d_v \times d_v}$, where Z is the total number of available modes. Therefore to find $R_\phi \cdot \mathcal{F}\{v\}$ we multiply the corresponding modes together and end up with an output of $\mathcal{F}^{-1}\{R_\phi \cdot \mathcal{F}\{v_t\}\} \in \mathbb{R}^{k_{\max} \times d_v}$. Note that we use `rfft` because we know the inputs will be real-valued and `irfft` because we know the outputs should be real-valued. Some benefits of the FNO, beyond eliminating the need for integral approximations, include its use of the fast Fourier transform (FFT) for efficient computation of Fourier transforms and its resolution invariance. Resolution invariance means that we can train the FNO on low-resolution data and evaluate it on higher-resolution data while maintaining accuracy. This property is intrinsic to the design of neural operators like the FNO. Unlike traditional numerical methods that depend heavily on fine meshes and effective discretization strategies, the FNO learns mappings directly between function spaces. This allows it to generalize across various resolutions, rather than simply optimizing for a solution on a fixed grid.

3 Numerical Experiments

In this section, we present numerical results of the FNO on two separate problems. The first is a 1-dimensional Burgers' equation and the second is a 1-dimensional Korteweg–De Vries (KdV) equation. Simulations were run on Google Colab using the T4 GPU.

3.1 Burgers' Equation

In the first example, we test the FNO on the 1-dimensional Burgers' equation given by

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} &= \nu \frac{\partial^2 u}{\partial x^2}, \quad (x,t) \in (0, 2\pi) \times (0, 1], \\ u(0,t) &= u(2\pi, t), \quad t \in (0, 1], \\ u(x,0) &= u_0(x), \quad x \in (0, 2\pi), \end{aligned} \tag{11}$$

where the viscosity is fixed to $\nu = 0.1$. We train the FNO to predict solutions at $t = 1$, $u(x, t = 1)$, given an initial condition $u_0(x)$. The FNO learns an operator that maps the initial condition space to the solution space at $t = 1$. The dataset was created by sampling initial conditions from a Gaussian random field, $u_0 \sim \mathcal{N}(0, 625(\Delta + 25I)^{-2})$. As this is a supervised learning task we also require the solutions to our training dataset which were solved using a fine forward Euler method. The resolution of the dataset is a uniform grid of $N_R = 2^{13} = 8192$ points however, using this many points can be computationally expensive so we uniformly subsample $N_R = 2^{11} = 2048$ points and use those for both training and testing instead. We train by optimizing the mean-squared error (MSE) between the FNO's output $\hat{u}$ and the desired function solution u over all elements of the batch. The loss function is given by

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{N_b N_R} \sum_{i=1}^{N_b} \sum_{j=1}^{N_R} (\hat{u}_{i,j} - u_{i,j})^2, \tag{12}$$

where θ are the network's parameters, N_R is the resolution size, and N_b is the batch size. In scenarios where input and output functions may differ largely in scale, a relative or normalized MSE (NMSE) may be more beneficial. This is typically computed as

$$\mathcal{L}_{\text{NMSE}}(\theta) = \frac{1}{N_b N_R} \sum_{i=1}^{N_b} \sum_{j=1}^{N_R} \frac{(\hat{u}_{i,j} - u_{i,j})^2}{(u_{i,j})^2}. \tag{13}$$

An extensive hyperparameter sweep was conducted to see how accurate the FNO can become on this dataset. The dataset included 1200 examples, 75% were used for training and the rest for validation/testing. In all examples, the validation and testing datasets are the same but are not necessarily used to infer any potential updates to the model. We begin with an initial FNO model of $k_{\max} = 16$, $w = 64$, $n_L = 4$ and a batch size $N_b = 128$. In all examples the Adam optimizer [6] was used with an initial learning rate of 10^{-3}, and an exponential decaying learning rate scheduler with a decay rate of 0.9 applied every 250 epochs over a total of 8000 epochs. An average L^2 error was

computed over all test examples and reported alongside the terminal training and validation loss at the final epoch. The average L^2 error is defined as

$$L^2_{\text{avg}} = \frac{1}{N_{\text{test}}} \sum_{k=1}^{N_{\text{test}}} \frac{\|\hat{u}_k - u_k\|_2}{\|u_k\|_2}, \tag{14}$$

where N_{test} is the number of test examples, $\hat{u}_k$ is the predicted output for the k-th test example, and u_k is the corresponding ground truth. In table 1 the default FNO ($k_{\text{max}} = 16$, $w = 64$, $n_L = 4$, $N_b = 128$) was used and a sweep of activation functions was performed. In an ideal situation, a full sweep should be done over all possible combinations of hyperparameters but due to time constraints, we start with a sweep over activations functions, choose the best-performing model, and use those hyperparameters as the starting point of the next sweep and repeat. It's also best to average these statistics over 5-10 independent runs but this also was not done due to time constraints, all statistics are from single independent runs of the model.

Table 1: Activation function sweep using $k_{\text{max}} = 16$, $w = 64$, $n_L = 4$, $N_b = 128$.

Activation Function	L^2_{avg}	Training Loss	Validation Loss
Leaky ReLU	0.0105	1.9281×10^{-5}	1.3491×10^{-4}
ReLU	0.0105	1.8738×10^{-5}	1.3327×10^{-4}
ELU	0.0074	3.0228×10^{-5}	8.0688×10^{-5}
SiLU	0.0047	1.0384×10^{-5}	3.0236×10^{-5}
GELU	0.0043	7.7389×10^{-6}	2.6171×10^{-5}
Sigmoid	**0.0036**	1.5383×10^{-5}	2.7204×10^{-5}
Tanh	**0.0036**	$\mathbf{6.7951 \times 10^{-6}}$	$\mathbf{1.7603 \times 10^{-5}}$

In table 1 we can see that L^2_{avg} ranges between 0.0036-0.0105 which is already impressive. The training and validation loss also generally stay in the range of 10^{-5} whereas in the case of using $\tanh$ the training loss achieves an order of 10^{-6} and the validation is not far off either. Sigmoid and the $\tanh$ activation functions had a comparable L^2_{avg} however both the training and validation losses were smaller in the case of $\tanh$ (although not by a significant amount), so we take $\tanh$ to be used in all other experiments.

Table 2: Modes sweep using $\tanh$ activation, $w = 64$, $n_L = 4$, $N_b = 128$.

Modes (k_{max})	L^2_{avg}	Training Loss	Validation Loss	Parameters
4	0.0130	7.7474×10^{-5}	9.9261×10^{-5}	147,969
8	0.0031	6.7074×10^{-6}	1.4943×10^{-5}	279,041
12	0.0031	6.0287×10^{-6}	$\mathbf{1.4919 \times 10^{-5}}$	410,113
16	0.0036	6.7949×10^{-6}	1.7601×10^{-5}	541,185
20	0.0030	4.9436×10^{-6}	1.7395×10^{-5}	672,257
24	**0.0027**	$\mathbf{3.9959 \times 10^{-6}}$	1.6424×10^{-5}	803,329

When it comes to the number of modes (table 2), there is not a significant difference between different choices of k_{max}, though this example does not generalize to all other use cases of the FNO. Setting $k_{\text{max}} = 4$ may be considered to be too few modes and cannot capture enough relevant features of the function to obtain substantial results, though the L^2_{avg} is still within a reasonable margin of error. We believe that using $k_{\text{max}} = 4$ would allow the FNO to learn the overall global features of the solution space but local features may be approximated poorly. As we increase the number of modes (also increasing the number of parameters or model complexity) the L^2_{avg} drops by almost an order of magnitude however, further increasing the number of modes past $k_{\text{max}} = 8$ does not seem to make a significant impact as it did when going from $k_{\text{max}} = 4$ to $k_{\text{max}} = 8$.

Table 3: Fourier layer sweep using $\tanh$ activation, $k_{\max} = 24$, $w = 64$, $N_b = 128$.

Layers (n_L)	L^2_{avg}	Training Loss	Validation Loss	Parameters
2	0.0048	2.0240×10^{-5}	4.8067×10^{-5}	401,793
4	0.0027	3.9957×10^{-6}	1.6424×10^{-5}	803,329
6	0.0025	2.7788×10^{-6}	1.3840×10^{-5}	1,204,865
8	0.0018	1.9763×10^{-6}	8.2767×10^{-6}	1,606,401
10	0.0018	1.7106×10^{-6}	6.6709×10^{-6}	2,007,937
12	**0.0016**	$\mathbf{1.5638 \times 10^{-6}}$	$\mathbf{6.6348 \times 10^{-6}}$	2,409,473

The number of layers used plays a major role in the model complexity, as seen in table 3 the number of parameters our model has enters the region of 1-2 million, with increasing model complexity we hope to see an improvement in model performance. In both tables 2 and 3 there is a relationship between the L^2_{avg} and the training/validation loss (with one exception in the case of $k_{\max} = 12$, table 2), that as model complexity increases, these measures decrease.

Table 4: Width sweep using $\tanh$ activation, $k_{\max} = 24$, $n_L = 12$, $N_b = 128$.

Width (w)	L^2_{avg}	Training Loss	Validation Loss	Parameters
16	0.0032	7.0570×10^{-6}	1.5389×10^{-5}	150,785
32	0.0024	2.7606×10^{-6}	8.6501×10^{-6}	602,625
48	0.0020	1.7035×10^{-6}	8.5413×10^{-6}	1,355,521
64	**0.0016**	$\mathbf{1.6060 \times 10^{-6}}$	$\mathbf{6.5671 \times 10^{-6}}$	2,409,473
80	0.0019	1.9991×10^{-6}	8.6621×10^{-6}	3,764,481
96	0.0019	3.7933×10^{-6}	6.9058×10^{-6}	5,420,545

The relationship between model complexity and performance (being inversely proportional) doesn't always hold as recorded in table 4. As we go from 2.4 M parameters up to 5.4 M performance drops by a small amount. The width of the convolutions does not seem to play a significant role in this example. Although we are increasing the number of parameters in the model, it's entirely possible these are not the "correct" parameters to be increasing. It may be more beneficial to add more Fourier layers, making the model deeper rather than increasing the width of these convolutions. In terms of the batch size table 5, it was seen that it doesn't play much of an important role in the scenario where the model is already of quite high complexity. It would be interesting to explore further if fine-tuning the batch size for a smaller model would allow it to reach a similar performance as these larger models.

Table 5: Batch size sweep using $\tanh$ activation, $k_{\max} = 24$, $w = 64$, $n_L = 12$.

Batch Size (N_b)	L^2_{avg}	Training Loss	Validation Loss
32	0.0021	2.2595×10^{-6}	7.7501×10^{-6}
64	0.0020	2.4817×10^{-6}	7.6333×10^{-6}
96	0.0018	2.0149×10^{-6}	6.7963×10^{-6}
128	**0.0017**	$\mathbf{1.6578 \times 10^{-6}}$	6.7415×10^{-6}
160	0.0020	2.1975×10^{-6}	8.4056×10^{-6}
192	**0.0017**	1.6779×10^{-6}	$\mathbf{6.6647 \times 10^{-6}}$

In fig. 1 the loss convergence and distribution of errors across all test cases are plotted for the best performing model which corresponds the following combination of hyperparameters, $\tanh$ activation, $k_{\max} = 24$, $n_L = 12$, $w = 64$, $N_b = 128$. About 98.2% of the L^2 errors were below 0.005. At the point of 8000 epochs, the training loss has not necessarily fully converged yet however the validation loss overall has and training for more epochs will likely lead to a minimal increase in performance. In fig. 2 we randomly select three test cases and in the first row plot the initial condition along with the FNO prediction on top of the ground truth solution. In the second row of fig. 2 are the absolute

point-wise errors between the FNO prediction and ground truth above it along with the L^2 error for that specific test case. The FNO captures the ground truth quantitatively and qualitatively to high accuracy.

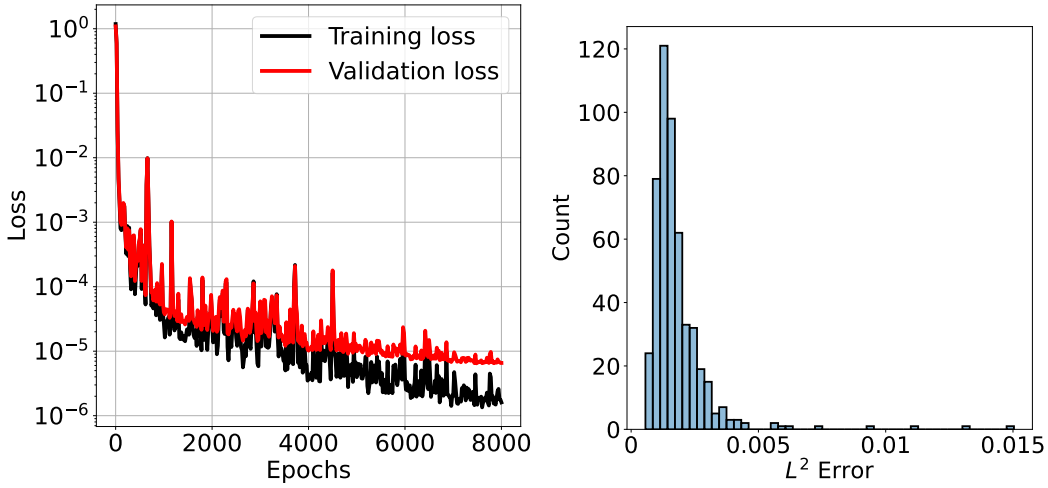

Figure 1: Left: Training and validation loss history for the best performing model. Right: Distribution of L^2 errors over the entire test dataset using the best-performing model.

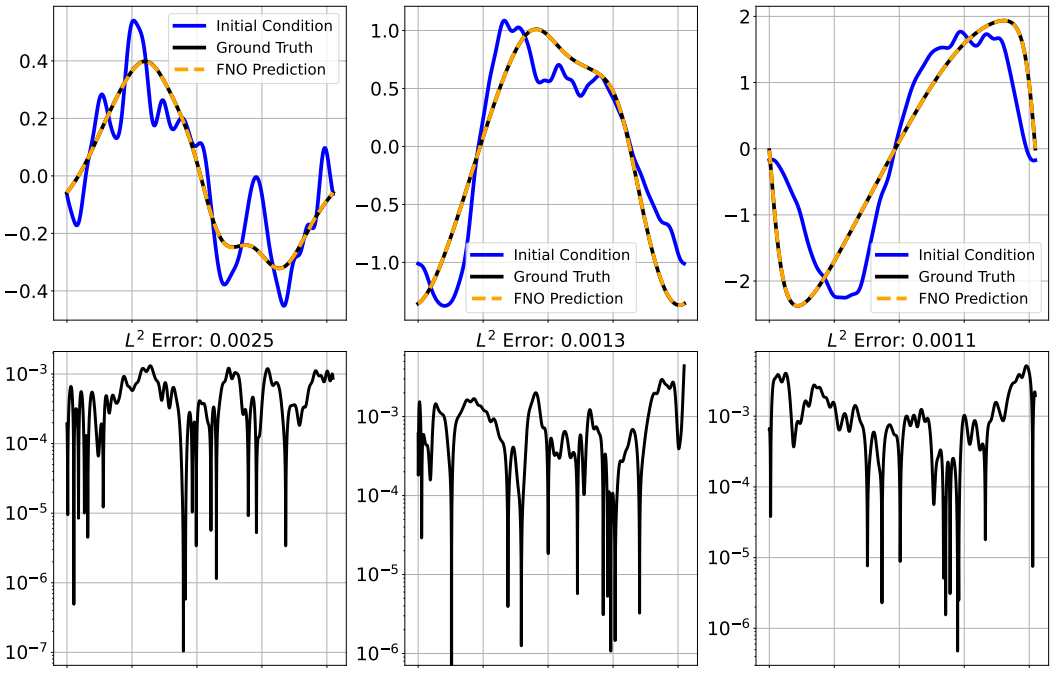

Figure 2: Randomly selected test examples. Top: The given initial condition and FNO prediction alongside the ground truth solution. Bottom: Point-wise absolute errors and L^2 between the FNO prediction and ground truth corresponding to the plot above it.

3.2 Korteweg–De Vries Equation

In the second and final example, we test the FNO on a 1-dimensional KdV equation given by

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} &= 0, \quad (x,t) \in (0,L) \times (0,T], \\ \frac{\partial u(0,t)}{\partial x} = \frac{\partial u(L,t)}{\partial x} &= 0, \quad t \in (0,T], \\ u(x,0) &= u_0(x), \quad x \in (0,L), \end{aligned} \tag{15}$$

where $L = 128$ and $T = 140$. The initial conditions are generated by a truncated Fourier series with random coefficients [1, 3] given as

$$u_0(x) = \sum_{k=1}^{N} A_k \sin(2\pi \ell_k x/L + \phi_k), \tag{16}$$

where A_k controls the amplitude, ℓ_k the frequency, and ϕ_k the phase change. In the case of the Burgers' equation, we trained the FNO to take in an initial condition and output the solution at the next time step $t = 1$. We now go beyond this and train an FNO to take in a series of inputs corresponding to t_H (time history) time steps of the solution and predict the next t_F (time future) time steps of the solution. This can then be extended to predict long-time solutions to the PDE in an autoregressive manner. In all simulations, $t_H = t_F = 20$, the FNO takes in the 20 time steps of the PDE solution and will be trained to predict the next 20 in an autoregressive fashion. The solutions to this KdV equation were produced using a pseudo-spectral method. The dataset used [10] contains three splits, training, validation, and testing, only the training and testing will be used. Each dataset contains 512 trajectories, and each trajectory has a spatial domain of $x \in [0, 128]$ with a spatial resolution of $N_x = 256$ and a temporal domain of $t \in [0, 140]$ with a temporal resolution of $N_t = 140$. The training was done by optimizing the sum of squared errors (SSE) between the FNO's output $\hat{u}$ and the desired solution u over all elements of the batch. The loss function is given by

$$\mathcal{L}_{\text{SSE}}(\theta) = \sum_{i=1}^{N_b} \sum_{j=1}^{N_x} \sum_{k=1}^{t_F} (\hat{u}_{i,j,k} - u_{i,j,k})^2, \tag{17}$$

where θ are the network's parameters and N_b is the batch size. The FNO has a lifting layer followed by 4 Fourier layers, a dense layer, and a final projection layer. The lifting and projection layers, unlike in the case of the Burgers' equation are not convolutions but dense layers. The lifting layer takes in an input of size (N_x, t_H) which is fed through a dense layer of width 64 with no activation function. Following the lifting layer 4 Fourier layers are used with each using a GeLU activation function. Once the 4 Fourier layers are applied, a dense layer of width 128 is used with a GeLU activation function. Since our goal is to take in t_H time steps and predict the next t_F time steps projection layer must have an output size of (N_x, t_F). To achieve this a dense layer of width t_F (20) with no activation function is used. A batch size of 16 was used, along with 8400 epochs using the AdamW optimizer [11] with a weight decay of 0.01, a learning rate scheduler was also used. For the first 10 epochs, a learning rate of 0.001 was used. Starting at epoch 1200, the learning rate was scaled by 0.4 every 1200 epochs, including the first 10 epochs in the count, until epoch 3600, after which it was kept constant.

The process of training this autoregressive model is slightly different than the simple process for the Burgers' FNO. We begin by defining a maximum start time which is simply $t_{\text{start}}^{\text{max}} = (T - t_H) - t_F$ $(100$ in our case). A range of all possible start times between t_H and $t_{\text{start}}^{\text{max}}$ with a step-size of t_H is then created. We do not include $T = 0$ as a possible starting time during our training process because during testing our FNO will take in time steps between $[0, t_H]$ as inputs and are therefore already known and do not need to be learned. From here we create a batch of N_b trajectories each with random start times, and call them t_R. Note here that these batches are not random but the start times are, in our example, we set $N_b = 16$ and since we have 512 trajectories the data loader will be looped through 32 times, meaning each trajectory will be trained over but at different start times. Each of these trajectories are split into an input trajectory (from t_R to $t_R + t_H$) and a label/expected trajectory (from $t_R + t_H$ to $(t_R + t_H) + t_F$). This is then repeated until all trajectories in the data loader have been trained over (which we define as one epoch). After each batch, the loss is recorded, and divided by the batch size, and after each epoch, this quantity is then averaged over the total number of batches.

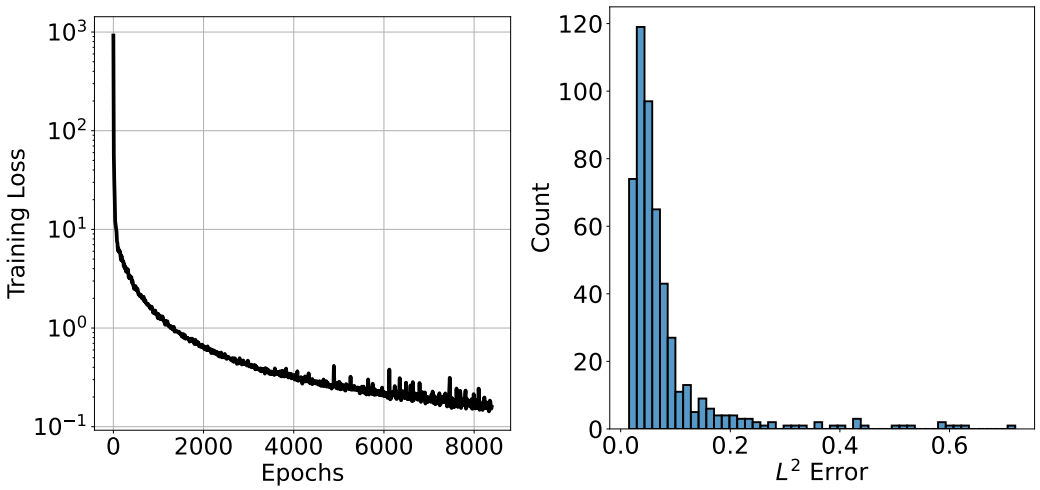

Figure 3: Left: Training loss history. Right: Distribution of L^2 errors over the entire test dataset.

In fig. 3 we can see on the left that there is a steady convergence of the loss over the total number of epochs. The loss within the first few epochs decreases quickly due to the larger learning rate being used. Our experiments showed that using a smaller learning rate during the initial training steps led to a convergence of a local minimum causing the training loss to oscillate between 45 and 50, significantly worse than what our model currently achieves. It may be beneficial to keep the larger initial learning rate for longer than 10 epochs before applying any scaling to speed up convergence even more, though we did not test this. The right plot of fig. 3 shows the distribution of the relative L^2 errors. The average L^2 error was 0.08, with the minimum being 0.0153 and the maximum 0.72. The majority of the resulting L^2 errors are generally acceptable (83% were below 0.1) however, there do exist several outlier cases which have considerably larger errors. In fig. 4 and fig. 5 are the results from the FNO on the worst and best case example from the test dataset, respectively. In the worst case scenario fig. 4, we can see that qualitatively the true and predicted trajectories have similar characteristics but quantitatively there are large deviations. This could be due to the autoregressive nature of the prediction. Since we use our previous/current predictions to predict the next set of time steps, if our current prediction is slightly off the error will propagate through the solution. As time increases we expect the error to increase due to this error propagation which can be seen in the plots of figs. 4 to 6. It should also be noted here that the size of the training dataset was the same as the test dataset

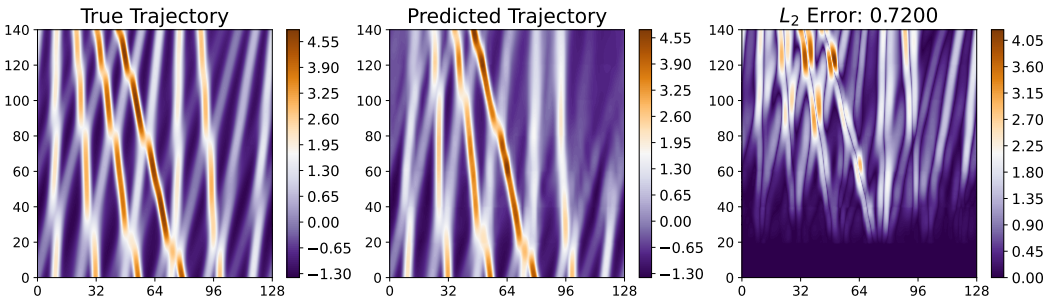

Figure 4: Largest L^2 error test case for the KdV FNO.

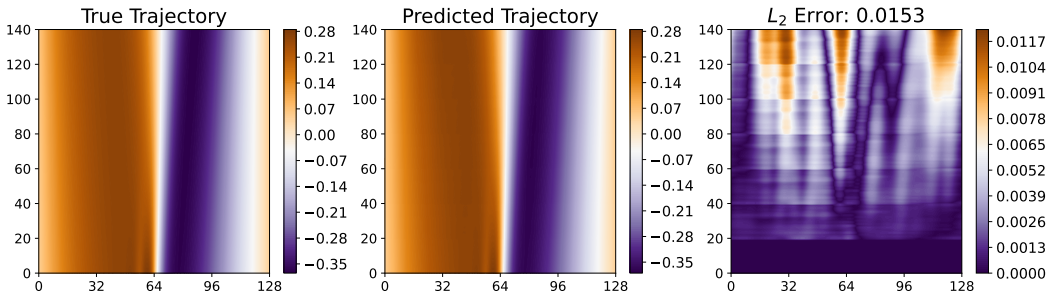

Figure 5: Smallest L^2 error test case for the KdV FNO.

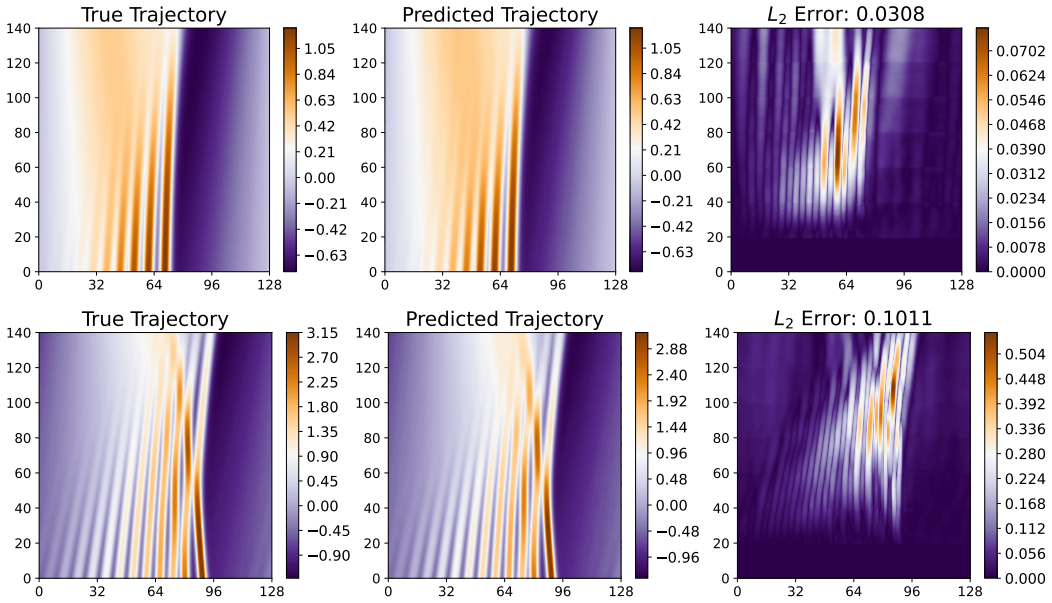

Figure 6: Randomly selected test cases for the KdV FNO.

4 Conclusion

In this paper, we explored the capabilities of the proposed Fourier neural operator in the context of a single time step prediction involving the Burgers' equation and an autoregressive multi-time step prediction task for the Korteweg–De Vries equation. Through an extensive hyperparameter sweep for the Burgers' equation, we were able to drive the average L^2 error across the entire dataset to about 0.0016 with a training loss of 1.6×10^{-6} and a validation loss of 6.6×10^{-6} using a small model of only 2.4 million parameters. For the Korteweg–De Vries equation, we've shown the architecture and training process used for the Burgers' equation can be extended to allow for the prediction of multiple time steps through autoregression. Our model showed smooth convergence of the loss and an average L^2 error of 0.08.

References

[1] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

[3] Johannes Brandstetter, Max Welling, and Daniel E Worrall. Lie point symmetry data augmentation for neural pde solvers. In *International Conference on Machine Learning*, pages 2241–2256. PMLR, 2022.

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[5] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[6] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[8] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[9] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*, 2020.

[10] Phillip Lippe. UvA Deep Learning Tutorials. `https://uvadlc-notebooks.readthedocs.io/en/latest/`, 2024.

[11] I Loshchilov. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[12] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[13] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[14] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in neural information processing systems*, 33:7537–7547, 2020.

[15] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

A Additional Results

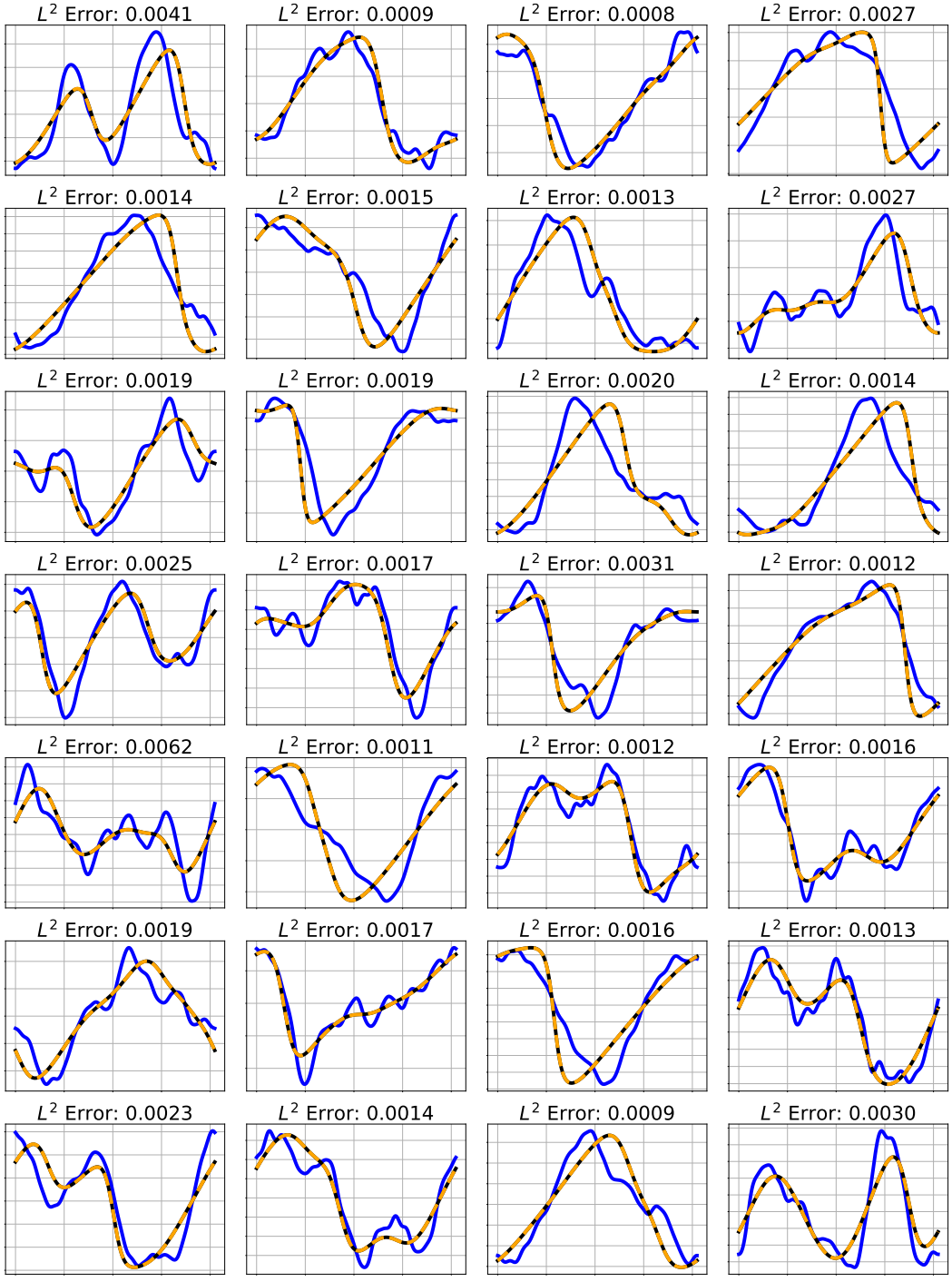

Figure 7: Additional results of randomly sampled test cases from the Burgers' FNO.

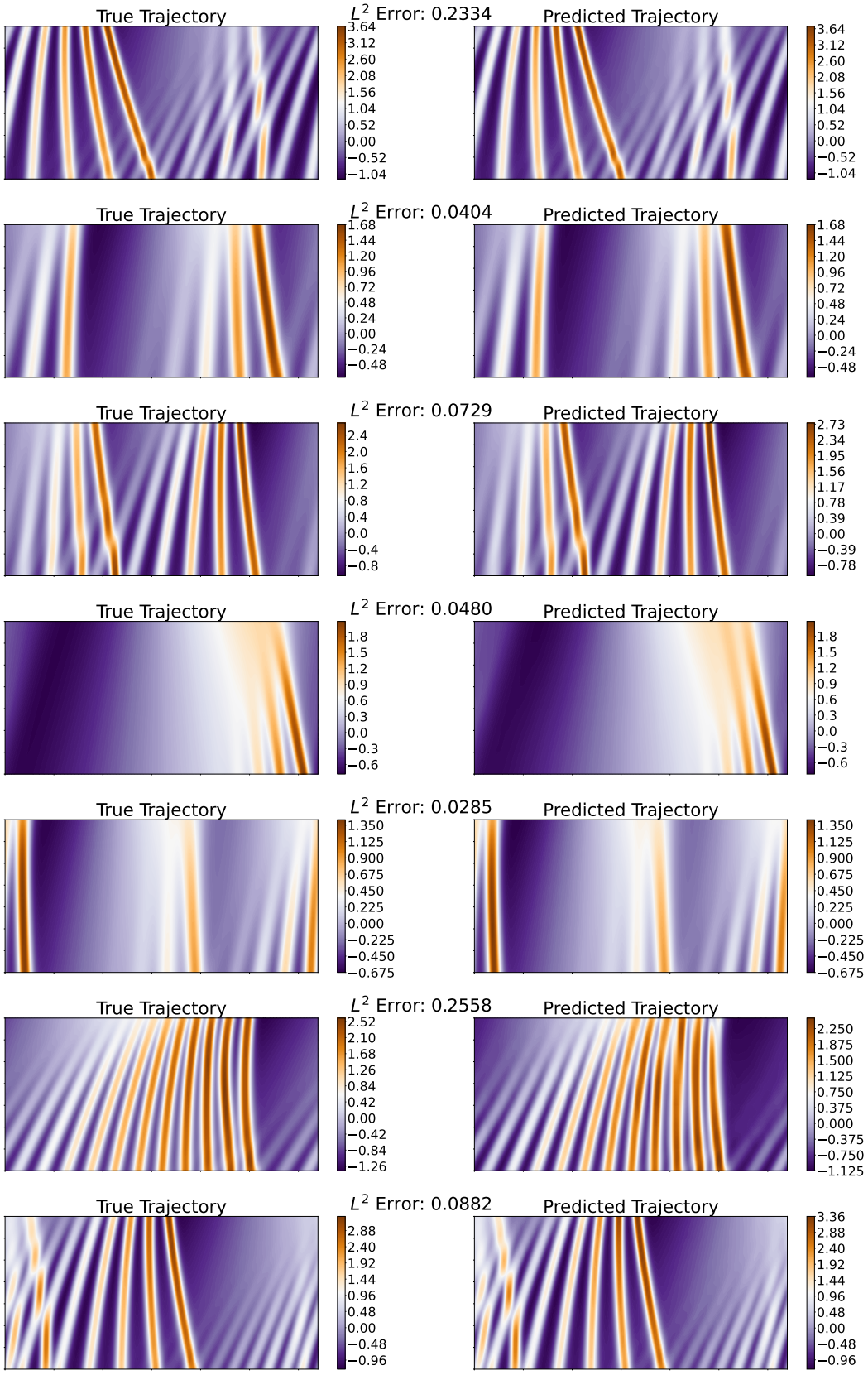

Figure 8: Additional results of randomly sampled test cases from the KdV FNO.