Mathematics

Thesis - Spring 2023

Analysis for Fourier Neural Operator and its Applications

Jialin Wang

supervised by

Prof. Mathieu Laurière

Analysis for Fourier Neural Operator and its Applications

Jialin Wang wj716@nyu.edu

May 2023

Abstract

In complex PDE settings, the neural operator, an efficient data-driven approach that aims to find the solution operator of PDEs, might be used. In contrast with traditional neural networks, which learn function mapping between finite-dimensional spaces, neural operators expand this learning to include operators between infinite-dimensional spaces. This allows for zero-shot generalization to higher-resolution evaluations and frees the neural operator from the grid's resolution and size for training data. Furthermore, if we rely on Fourier spaces for our training procedures, our solution training process will be more efficient. In comparison to conventional numerical approaches, the Fourier neural operator has quasi-linear time complexity, allowing it to solve PDEs much more quickly. Numerical experiments are performed on Darcy Flow as well as Fokker-Planck equations to prove such properties of the Fourier neural operator.

Contents

1 Introduction

Solving complicated partial differential equation systems repeatedly for various values of parameters is a common task in the field of science and engineering. Micro-mechanics, turbulent fluxes, and molecular dynamics are a few examples of them. Conventional solvers such as purely physics-informed optimization learning, Finite Element Methods (FEM), and Finite Difference Methods (FDM) suffer many disadvantages compared to data-driven methods. The traditional numerical solvers are often inefficient because such complex systems often need fine discretization to capture the phenomenon being modeled and have a trade-off on resolution, while many data-driven methods hold the property of mesh independence and resolution invariance. For instance, when dealing with inverse problems, thousands of evaluations of the forward model will be required, which complicates the calculation, and the output resolution would be affected by the input resolution size. Thus, a fast and efficient method is needed to solve the problem.

Hence, the neural operator, a data-driven approach that tries to find the PDEs' solution operator quickly, can be employed in complex PDE settings. Contrary to traditional neural networks, which learn function mapping between spaces of finite dimensions, neural operators extend this learning to include operators between areas of infinite dimensions. This enables zero-shot generalization to higher-resolution evaluations, and makes the neural operator free of the resolution and size of the grid for training data. Moreover, if we apply our training procedures in the Fourier spaces, our solution training process will be more efficient. In comparison to conventional numerical approaches, we observe that the Fourier neural operator has quasi-linear time complexity, making it much faster in solving the PDEs.

Table 1: Comparison between Conventional Solvers and Data-driven Methods

Type	Conventional Solvers	Data-driven Methods
Advantages and Disadvantages	• only solve one instance, inefficient when changing parameters, boundary or initial conditions • require an explicit form to train • trade-off on the resolution • slow on fine grids • fast on coarse grids • theory is well understood	• require training data • data can be slow to generate • model-free, learn a family of PDE • black-box, data-driven • some are resolution-invariant, mesh-independent • slow to train, fast to evaluate • theory not well understood

1.1 Literature Review and Context

Data-driven methods directly learn the trajectory of a family of PDE through the data provided, by means of some machine learning algorithms. Prominent examples include Graph Neural Operator [15], Fourier Neural Operator [14], Physics-informed Neural Operator [16], and Adaptive Fourier Neural Operator [10]. From a broad literature review and paper reading, we summarize the advantages and disadvantages of conventional PDE solvers and data-driven methods, which are shown in Table 1. However, it is worth mentioning that not all data-driven methods are resolution-invariant and mesh-independent. Only under moderate conditions do these two properties hold. Additionally, though data-driven, some classical neural networks map between finite-dimensional spaces and can only train solutions within a specific discretization. Such classical neural networks include Finite-dimensional Operators and Neural-FEM [14].

1.1.1 Finite-dimensional Operator and Neural-FEM

The finite-dimensional operator learns the mapping between two finite-dimensional Euclidean spaces through convolutional neural networks. Since it is a finite-dimensional operator, it needs modification of resolution and discretization according to different PDE cases to minimize the error. Thus, it is a mesh-dependent operator [11] [26] [1] [3] [13]. As for the Neural-FEM, it directly parameter-

izes an instance of PDE using neural networks. In other words, it is not an operator, rather, it is similar to the traditional solvers like the finite difference method, but trained by neural networks. Although Neural-FEM is mesh-independent, Neural-FEM is actually time-consuming since it needs to be trained for each new instance of parameters, leading to a new neural network each time. Moreover, the approach is restricted to the settings where the underlying PDE is known [5] [18] [2] [19] [17].

1.1.2 Fourier Transform

The Fourier transform (FT) in mathematics is a transformation that changes a function into a form that exhibits the frequencies found in the initial function [24]. According to Li et al. [14], since differentiation is equivalent to multiplication in the Fourier domain, the Fourier transform is widely employed in spectral methods for solving differential equations. Fourier transforms also play a significant role in the development of deep learning.

1.2 Contributions

In the thesis, we first conducted detailed research and analysis of the Fourier neural operator. Generally speaking, the theoretical part of the thesis is a new and ordered illustration of the Fourier neural operator starting from scratch. According to Li et al. [14], we started with the graph construction, then set up the message-passing graph network, and finally built up the graph neural operator by transforming the discrete problem into a continuous format. We then introduced the iterative algorithm for solving the neural operator, and put such algorithm calculation in the Fourier spaces to boost the solution speed and efficiency. Moreover, we summarized the advantages of the Fourier neural operator such as resolution invariance, mesh independence, and quasi-linearity,

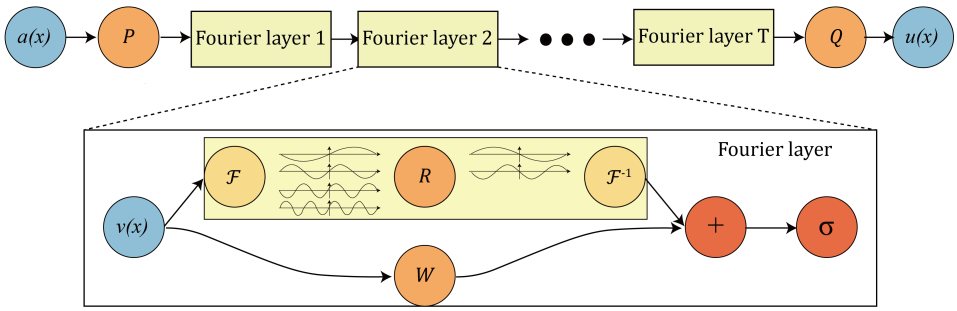

Figure 1: The architecture of the Fourier neural operator by Li et al. [14]

and proved such properties through numerical experiments. Apart from the Darcy Flow experiments containing three different resolutions, we also performed experiments on a brand-new instance: the Fokker-Planck equation, by means of the Fourier neural operator. This is also an innovative part of this thesis.

2 Theory

2.1 Problem Setting and Operators

According to Li et al. [14], the PDE operator learns a mapping between two infinite-dimensional spaces through a finite collection of observations of input-output pairs. Let $D \subset \mathbb{R}^d$ be a bounded and open set, and $\mathcal{A}$ and $\mathcal{U}$ be separable Banach spaces of functions which take values in $\mathbb{R}^d$, where d is the size of the input and output space. Then $G^\dagger \colon \mathcal{A} \to \mathcal{U}$ is a typically non-linear map we would like to investigate. Suppose we have N observations $\{a_i, u_i\}_{i=1}^N$ where $a_i \sim \mu$ is an independently and identically distributed sequence from the probability measure μ supported on $\mathcal{A}$. Furthermore, it is highly likely that $u_i = G^\dagger(a_i)$ is corrupted with noise. Then our goal is to approximate $G^\dagger$ by

building a parametric map:

$$G_\theta : \mathcal{A} \to \mathcal{U}, \quad \theta \in \Theta \tag{1}$$

for some finite-dimensional parameter space Θ. We choose $\theta^\dagger \in \Theta$, the optimal parameter, then we have $G(\cdot, \theta^\dagger) = G_{\theta^\dagger} \approx G^\dagger$, the optimal solution. For such learning in infinite dimensions, we define a cost-function C: $\mathcal{U} \times \mathcal{U} \to \mathbb{R}$ and try to find a minimizer of the problem:

$$\min_{\theta \in \Theta} \ \mathbb{E}_{a \sim \mu}[C(G(a, \theta), G^\dagger(a))] \tag{2}$$

which directly parallels the traditional finite-dimensional setting in Vapnik's work [20]. Though we do not prove the existence of the minimizer here, we will address this problem in test-train settings in the numerical application section where empirical approximations to the cost are calculated. Since the methodology is proposed in the infinite-dimensional approximation scenario, the common set of network parameters that are defined in the infinite-dimensional setting can be implemented into all finite-dimensional approximations as well. To summarize, we consider mappings $G^\dagger$ which take coefficient functions of a PDE as input and then map them to solutions of the PDE, and both input and solutions are real-valued functions on $\mathbb{R}^d$ [15]. Traditionally, we calculate the solution $u \in \mathcal{U}$ of a PDE for a single case of the parameter $a \in \mathcal{A}$ using solvers such as physics-informed neural networks and Neural-FEM. However, these approaches aim at solving one instance of PDE and are thus computationally expensive and not applicable to solving operators $G^\dagger$ for a family of equations from the data. In contrast, the proposed method in our paper designed for solving the operator can directly approximate the operator and is thus much more efficient and faster [14].

Since our data a_i and u_i are generally functions, we consider them as point-wise values to deal with them numerically. To illustrate, let $P_K = \{x_1, ..., x_K\} \subset D$ be a K-point discretization (in applied math, discretization is the conversion of continuous equations, models, variables, and functions into discrete counterparts) of the domain D. Suppose we have a finite collection of input-output value pairs denoted by $a_i|_{P_K}, u_i|_{P_K} \in \mathbb{R}^K$. In the following subsection, a kernel-inspired graph neural network architecture trained on the discretized data pairs is brought up to generate the solution $u(x)$ for any $x \in D$ given a new input parameter $a \sim \mu$. This shows that the method is independent of the discretization P_K. This is a good property of operators since it allows a transfer of solutions between different discretization sizes and grid geometries [15]. Apart from that, answers produced by operators are mesh-independent and resolution-invariant, and the error is independent of the input resolution. We will discuss such properties in later sections.

2.2 Graph Kernel Network

Partial differential equations govern the law of a broad range of important engineering problems and physical phenomena. Recent decades witness significant developments in formulating and solving PDEs in many scientific disciplines. However, according to Li et al. [15], two significant challenges remain. Firstly, formulating the underlying partial differential equations for the specific scientific phenomenon usually demands intensive prior knowledge in the corresponding field; secondly, solving complex non-linear PDE systems is computationally difficult. Luckily, the emergence of neural networks contributes to leveraging the increasing volume of available data in both of these difficulties, and they should be further studied to adapt to mappings between function spaces.

We first propose a graph kernel neural network to help to figure out the operator formulated in

section 2.1. We consider PDEs of the form:

$$(\mathcal{L}_a u)(x) = f(x), \quad x \in D \tag{3}$$

$$u(x) = 0, \quad x \in \partial D \tag{4}$$

with solution u: $D \to \mathbb{R}$ and parameter a: $D \to \mathbb{R}$. $\mathcal{L}_a$ is a differential operator depending on the parameter $a \in \mathcal{A}$, and f is some fixed function living within a proper function space following the structure of $\mathcal{A}$. For instance, $\mathcal{L}_a \cdot = -div(a\nabla \cdot)$ is the elliptic operator [8]. Under general conditions on $\mathcal{L}_a$ [7], we define the Green's function $G : D \times D \to \mathbb{R}$ as the unique solution to the problem:

$$\mathcal{L}_a G(x, \cdot) = \delta_x \tag{5}$$

where δ_x is the delta measure on $\mathbb{R}^d$ centered at x. Since G is dependent on parameter a, we denote it as G_a. The solution to equations (3) and (4) can be expressed as [15]:

$$u(x) = \int_D G_a(x, y) f(y) dy \tag{6}$$

Proof. $\mathcal{L}_a u(x) = \int_D (\mathcal{L}_a G(x, \cdot))(y) f(y) dy = \int_D \delta_x(y) f(y) dy = f(x)$ □

Since Green's function is continuous at points $x \neq y$, the equation could be solved through the neural network. However, we will not discuss the case when $x = y$. Indeed, our goal is to approximate (6) using some proper methods, which is the core of our research topic. Thus, to approximate the solution, we construct an operator using the well-known graph construction, and such an operator is called the graph neural operator.

2.2.1 Graph Construction

According to Li et al. [15], a graph connecting the physical domain D of the PDE is designed to foster the realization of the neural operator. Here, the K discretized spatial locations are chosen to be the graph nodes. For simplicity, we assume working on a standard uniform mesh, but there are also cases like random mesh points according to the provided data. The edge connectivity is chosen with respect to the integration measure, which is the Lebesgue measure restricted to a ball $B(x, r)$. We define the neighborhood set as $N(x)$, and each node $x \in \mathbb{R}^d$ is connected to nodes lying within $B(x, r)$ (a vertex S is a neighbor of a vertex Q in a graph G if there is an edge between them). For each neighbor $y \in N(x)$, the edge feature is assigned as $e(x, y) = (x, y, a(x), a(y))$.

Based on the graph, we further construct a message-passing graph network, which comprises the edge features [9]. Assuming we construct the graph on the spatial domain D of the PDE, the aggregation of messages can be used to represent the kernel integration, which is the key part of our proposed neural operator. To be specific, given the node features $v_t(x) \in \mathbb{R}^n$, where n is the dimension of one layer, edge features $e(x, y) \in \mathbb{R}^{n_e}$, and a graph H, the message passing neural network with averaging aggregation is expressed as:

$$v_{t+1}(x) = \sigma(W v_t(x) + \frac{1}{|N(x)|} \sum_{y \in N(x)} \kappa_\phi(e(x, y)) v_t(y)) \tag{7}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear activation function applied element-wise, $W : \mathbb{R}^n \to \mathbb{R}^n$ is a linear transformation (to be learned from data), $N(x)$ is the neighborhood of x according to the graph, and the kernel $\kappa_\phi : \mathbb{R}^{2(d+1)} \to \mathbb{R}^{n \times n}$ with the parameter ϕ is going to be modeled using neural network and trained from the data. $\kappa_\phi(e(x, y))$ is a neural network taking edge features as input

and a matrix in $\mathbb{R}^{n\times n}$ as output. Moreover, $e(x,y) = (x, y, a(x), a(y)) \in \mathbb{R}^{2(d+1)}$ [15], where x and y are of dimension d, $a(x)$ and $a(y)$ are of dimension 1.

2.2.2 Graph Neural Operator

To construct an algorithmic framework of the neural operator, we aim to transform the discrete summation in equation (7) into a continuous one without the loss of generality. Thus, following equation (6) and the graph construction in equation (7), we introduce the following iterative architecture for $t = 0, ..., T-1$:

$$v_{t+1}(x) := \sigma(W v_t(x) + \int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y) \nu_x(dy)) \tag{8}$$

where the notation and constraints are the same as those in equation (7), and ν_x is a fixed Borel measure for each x in D [15]. The iteration follows $v_0 \to v_1 \to \cdots \to v_T$, where v_j for $j = 0, 1, \ldots, T-1$ is a sequence of functions, each of them taking values in $\mathbb{R}^n$. This neural operator is called a "Graph Neural Operator".

According to Li et al. [14], the initialization $v_0(x)$ to equation (6) can be viewed as the initial input we make for the solution $u(x)$. We first begin with the coefficient $a(x)$ itself as well as the position in physical space x. This forms a vector field of dimension $(d+1)$ and is then lifted to a n-dimensional vector field by a local transformation P which is often parameterized by a shallow fully-connected neural network. The representation of the "dimension-lifting" process is written as $v_0(x) = P(a(x))$, and the operation could be viewed as the first layer of the neural network. Then this functions as an initialization part to the kernel neural network, and we apply T iterations of updates $v_t \to v_{t+1}$ defined above. In the last layer, we project v_T back to the scalar field of interest

using another neural network layer, which is also a local transformation $Q : \mathbb{R}^n \to \mathbb{R}^{d+1}$. The "dimension-lowering" process is expressed as $u(x) = Q(v_T(x))$. To summarize, in each iteration, the update $v_t \to v_{t+1}$ contains a (non-local) neural network κ parameterized by $\phi \in \Theta$ and a non-linear local activation function σ.

Example: For instance, we consider the problem of approximation of the second-order elliptic PDE and transform the problem using the graph neural operator:

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in D \tag{9}$$

$$u(x) = 0, \quad x \in \partial D \tag{10}$$

for some bounded, open set $D \subset \mathbb{R}^d$ and a parameter function $f(x)$. For a given $a \in \mathcal{A} = L^\infty(D; \mathbb{R}^+) \cap L^2(D; \mathbb{R}^+)$, the equations above have a unique weak solution $u \in \mathcal{U} = H_0^1(D; \mathbb{R})$ [6]. As a result, we could define the solution operator $G^\dagger$ as the map from a to u. Then we can apply the proposed neural operator iterations to this example. Additionally, we apply the initialization $(x, a(x))$ with a Gaussian smoothed version of the coefficients $a_\epsilon(x)$ and their gradient $\nabla a_\epsilon(x)$ because of the smoothing effect of the inverse elliptic operator in the equation (9) (10) regarding the input data a (and actually f when we consider this as input in experiments). Thus, we initialize the problem with a $2(d+1)$-dimensional vector field. And the PDE problem is thus formulated

using the graph neural network as follows [15]:

$$v_0(x) = P(x, a(x), a_\epsilon(x), \nabla a_\epsilon(x)) + p \tag{11}$$

$$v_{t+1} = \sigma(W v_t(x) + \int_{B(x,r)} \kappa_\phi(x, y, a(x), a(y)) v_t(y) \nu_x(dy)) \tag{12}$$

$$u(x) = Q v_T(x) + q \tag{13}$$

where $P : \mathbb{R}^{2(d+1)} \to \mathbb{R}^n$, $p \in \mathbb{R}^n$, $v_t(x) \in \mathbb{R}^n$, $Q \in \mathbb{R}^{1\times n}$ and $q \in \mathbb{R}$. The integration in equation (12) can be seen as approximated by a Monte-Carlo sum (the basic idea behind Monte Carlo sum is to approximate the integral of a function $f(x)$ over a domain D by generating a large number of random points within the domain and using these points to estimate the integral) via a message-passing graph network with edge feature $(x, y, a(x), a(y))$ in equation (7).

Remark: Gaussian smoothing, also known as Gaussian blur, is to blur an image by a Gaussian function. It is widely used in graphics software, to reduce image noise and reduce the image's high-frequency components [25]. In the paper, the Gaussian smoothing is conducted with a centered isotropic Gaussian with variance 5, and the Borel measure ν_x is selected as the Lebesgue measure supported on a ball of center x and radius r.

2.3 Fourier Neural Operator

Fast Fourier Transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse, converting a signal from its original domain to a representation in the frequency domain and vice versa. By factorizing the discrete Fourier transform matrix into a product of sparse components, an FFT quickly computes the transformations. The most widely

used FFT algorithms are based on the factorization of N, where N is the data size, and in the fact that $e^{-2\pi I/N}$ is an N-th primitive root of unity, thus could be applied to analogous transforms over any finite field. Consequently, FFT manages to reduce the complexity of computing the DFT matrix from $O(N^2)$ to $O(NlogN)$, which is quasi-linearity [22]. Let $x_0, \ldots, x_{N-1}$ be complex numbers, the FFT is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}, \quad k = 0, \ldots, N-1$$

where $e^{i2\pi/N}$ is a primitive N-th root of 1. Evaluating this definition directly requires $O(N^2)$ operations since there are N outputs X_k, and each output requires a sum of N terms. An FFT is any method that can compute the same results in $O(NlogN)$ operations. Apart from that, by calculating the different frequency components in time-varying signals, FFT is able to reconstruct such signals from a set of frequency components, thus reducing the time complexity. Examples of FFT algorithms include Cooley-Tukey algorithm, Prime-factor FFT algorithm, and Bruun's FFT algorithm [22].

Though the graph neural operator is good at approximating the solutions to PDE systems, the process of training the kernel κ_ϕ and the linear transformation W directly from the data provided is time-consuming and complicated due to the large size of the training parameters. So based on the graph neural network and the graph neural operator, we introduce the Fourier neural operator. We parameterize κ_ϕ directly in Fourier space and use the Fast Fourier Transform (FFT) to compute the kernel efficiently. According to Li et al. [14], we choose $\mathcal{K}(a;\phi)$ to be a kernel integral transformation parameterized by a neural network and define the kernel integral operator mapping in equations

(8) and (12) by:

$$(\mathcal{K}(a;\phi)v_t)(x) := \int_D \kappa_\phi(x, y, a(x), a(y))v_t(y)dy, \quad \forall x \in D \tag{14}$$

where κ_ϕ is the neural network parameterized by $\phi \in \Theta_\kappa$. And the neural operator iterative updates could be written as:

$$v_{t+1}(x) := \sigma(Wv_t(x) + (\mathcal{K}(a;\phi)v_t)(x)), \quad \forall x \in D \tag{15}$$

Our task is to train (14) using FFT at a high speed in the Fourier space. According to Li et al. [14], we first propose replacing the kernel integral operator in (14) with a convolution operator defined in Fourier space. Let $\mathcal{F}$ denote the Fourier transform of a function $f : D \to \mathbb{R}^n$ and $\mathcal{F}^{-1}$ its inverse. Then:

$$(\mathcal{F}f)_j(k) = \int_D f_j(x)e^{-2i\pi\langle x,k\rangle}dx, \quad (\mathcal{F}^{-1}f)_j(x) = \int_D f_j(k)e^{2i\pi\langle x,k\rangle}dk \tag{16}$$

for $j = 1, \ldots, n$, where $i = \sqrt{-1}$ is the imaginary unit [14]. We assume $\kappa_\phi(x, y, a(x), a(y)) = \kappa_\phi(x - y)$ in (14) and then apply the convolution theorem below. We get:

$$(\mathcal{K}(a;\phi)v_t)(x) = \mathcal{F}^{-1}(\mathcal{F}(\kappa_\phi) \cdot \mathcal{F}(v_t))(x), \quad \forall x \in D \tag{17}$$

As a consequence, we propose to parameterize κ_ϕ in Fourier space directly.

Definition 1 (Fourier integral operator): *We define the Fourier integral operator as:*

$$(\mathcal{K}(\phi)v_t)(x) = \mathcal{F}^{-1}(R_\phi \cdot (\mathcal{F}v_t))(x), \quad \forall x \in D \tag{18}$$

where R_ϕ is the Fourier transform of a periodic function $\kappa : \bar{D} \to \mathbb{R}^{n\times n}$ which is parameterized by $\phi \in \Theta_{\mathcal{K}}$.

Definition 2 (Convolution theorem): *The convolution theorem is a fundamental result in mathematics and signal processing. Under suitable conditions, the Fourier transform of a convolution of two functions equals to the pointwise product of their Fourier transforms. Let f and g be two functions in $L^1(\mathbb{R})$, the space of integrable functions on the real line.Then the Fourier transform of f and g are defined respectively as:*

$$F(\omega) = \int f(x)e^{-i\omega x}dx \tag{19}$$

$$G(\omega) = \int g(x)e^{-i\omega x}dx \tag{20}$$

for all ω in $\mathbb{R}$. Then, the convolution theorem states that the Fourier transform of the convolution $f * g$ is given by the pointwise product of the Fourier transforms F and G:

$$F(\omega) \cdot G(\omega) = \int (f * g)(x)e^{-i\omega x}dx \tag{21}$$

for all ω in $\mathbb{R}$ [21].

According to Li et al. [14], for frequency mode $k \in D$, we have $(\mathcal{F}v_t)(k) \in \mathbb{C}^n$ and $R_\phi(k) \in \mathbb{C}^{n\times n}$. Since κ has a Fourier series expansion as we assume the κ is periodic, we can work with the discrete

modes $k \in \mathbb{Z}^d$. We first choose a finite-dimensional parameterization by truncating the Fourier series at a maximal number of modes k_{max}, and we define the set $Z_{k_{max}} = \{k \in \mathbb{Z}^d : |k_j| \leq k_{max,j}, \text{for } i = 1, \cdots, d\}$. Thus we directly parameterize R_ϕ as a complex-valued tensor of size $(k_{max} \times n \times n)$ which contains a collection of truncated Fourier modes. Due to the real-valued κ, we impose conjugate symmetry and notice that the set $Z_{k_{max}}$ is not the canonical choice for the low-frequency modes of v_t. Actually, we often define the low-frequency modes by placing an upper bound on the l_1 norm of $k \in \mathbb{Z}^d$. To make the calculations and implementation more quickly, we choose $Z_{k_{max}}$ as defined above.

In such a discrete mode, we assume the domain D is discretized with $N \in \mathbb{N}$ points, then we obtain $v_t \in \mathbb{R}^{N \times n}$ and $\mathcal{F}(v_t) \in \mathbb{C}^{N \times n}$. Since we convolve v_t using a function that only has k_{max} Fourier modes, we just simply truncate the higher modes to get $\mathcal{F}(v_t) \in \mathbb{C}^{k_{max} \times n}$. This makes the training more efficient and fast. Then we multiply $\mathcal{F}(v_t)$ using the weight tensor $R \in \mathbb{C}^{k_{max} \times n \times n}$ and obtain [14]:

$$(R \cdot (\mathcal{F}(v_t)))_{k,l} = \sum_{j=1}^{n} R_{k,l,j} (\mathcal{F}(v_t))_{k,j} \tag{22}$$

where $k = 1, \cdots, k_{max}, \ j = 1, \cdots, n$. Then $\mathcal{F}$ could be replaced by the Fast Fourier Transform when the discretization is uniform with resolution $s_1 \times \cdots \times s_d = N$ to boost speed. For $f \in \mathbb{R}^{N \times n}$, $k = (k_1, \cdots, k_d) \in \mathbb{Z}_{s_1} \times \cdots \times \mathbb{Z}_{s_d}$, and $x = (x_1, \cdots, x_d) \in D$, the Fast Fourier Transform $\hat{\mathcal{F}}$ and

its inverse $\hat{\mathcal{F}}^{-1}$ are expressed as follows:

$$(\hat{\mathcal{F}}f)_l(k) = \sum_{x_1=0}^{s_1-1} \cdots \sum_{x_d=0}^{s_d-1} f_l(x_1, \cdots, x_d) e^{-2i\pi \sum_{j=1}^{d} \frac{x_j k_j}{s_j}}, \tag{23}$$

$$(\hat{\mathcal{F}}^{-1}f)_l(x) = \sum_{k_1=0}^{s_1-1} \cdots \sum_{k_d=0}^{s_d-1} f_l(k_1, \cdots, k_d) e^{2i\pi \sum_{j=1}^{d} \frac{x_j k_j}{s_j}} \tag{24}$$

where $l = 1, \cdots, n$. In this scenario, the set of truncated modes becomes:

$$Z_{k_{max}} = \{(k_1, \cdots, k_d) \in \mathbb{Z}_{s_1} \times \cdots \times \mathbb{Z}_{s_d} | k_j \leq k_{max,j} \text{ or } s_j - k_j \leq k_{max,j}, \text{ for } j = 1, \cdots, d\} \tag{25}$$

R is treated like a $(s_1 \times \cdots \times s_d \times n \times n)$-sized tensor when it is implemented, and according to Li et al. [14], the above definition of $Z_{k_{max}}$ is consistent with the "corners" of R, allowing for the matrix-vector multiplication to produce a simple parallel implementation of equation (22). Via real experiments, we select $k_{max,j} = 12$ as the ideal value, resulting in $k_{max} = 12^d$ channel parameters that are effective for all the scenarios we want. In practice, we build the neural network consisting of multiple blocks which combine spectral convolution with regular linear convolution. The convolution in Fourier space filters out higher-order oscillations in the solution, while the linear convolution learns local correlations. We set four layers and two input channels, and in each iteration, we compute loss and gradients and update model parameters using the stochastic descent gradient update rule [4].

Through the detailed analysis of the Fourier neural operator, we conclude the several major features of FNO as below:

1. **Invariance to discretization:** The Fourier layer can train and evaluate functions that

are discretized in an arbitrary way. According to Li et al. [14], solving the functions in the physical space means directly projecting on the basis $e^{2\pi i\langle x,k\rangle}$ which are well-defined everywhere on $\mathbb{R}^d$, as parameters are learned directly in Fourier spaces. This leads to "zero-shot super-resolution" (zero-shot super-resolution is a type of image super-resolution that aims to enhance the resolution of an image without any specific training data for the specific image) and the formulation has a consistent (almost the same) error at any resolution of inputs and outputs. In comparison, other neural-network-based methods such as CNN have an error growing along with the resolution.

2. **Invariance to resolution:** For some resolution-invariant operators, they have consistent error rates among diverse resolutions and are independent of the ways its data is discretized as long as all relevant information is resolved. Such operators also fulfill zero-shot super-resolution. In contrast, the traditional PDE solvers such as FEM and FDM only solve a single case of function each time and thus the error decreases as the resolution increases [14].

3. **Parameterizations of R:** According to Li et al. [14], R could be generally defined to depend on $(\mathcal{F}a)$ to parallel (14). Indeed we can define $R_\phi : \mathbb{Z}^d \times \mathbb{R}^n \to \mathbb{R}^{n\times n}$ as a parametric function that maps $(k, (\mathcal{F}a)(k))$ to the values of appropriate Fourier modes. In their experiments, they observed both linear and neural network parameterizations of R_ϕ, and they conclude that the linear parameterization and the previously described direct parameterization have similar performance. However, the neural networks exhibit worse performances than them, resulting from the discrete structure of the space $\mathbb{Z}^d$.

4. **Quasi-linear complexity:** Since the weight tensor R contains $k_{max} < N$ modes, the inner multiplication only has $O(k_{max})$ time complexity. Consequently, the computational costs mainly depend on the computation of the Fourier transform $\mathcal{F}(v_t)$ and its inverse, thus fol-

lowing the time complexity of FFT, $O(NlogN)$. In general cases, the Fourier transforms have complexity $O(N^2)$, however, since we truncate the modes, the complexity is then $O(Nk_{max})$ in our cases [14]. To summarize, under uniform discretization cases, the Fast Fourier Transform here usually has complexity $O(NlogN)$, which is very fast and efficient.

3 Numerical Experiments

3.1 Darcy Flow

3.1.1 Darcy Flow Problem Formulation

Darcy flow typically represents the flow of a fluid through a porous medium, such as groundwater through soil or oil through rock. Flow data usually consists of measurements of pressure, velocity, or other properties at different points in the medium, which are then used to model the flow. The Darcy flow partial differential equation is a mathematical expression that describes the flow of a fluid through a porous medium under steady-state conditions. The equation is based on the principle that the flow velocity of a fluid through a porous medium is proportional to the pressure gradient, and inversely proportional to the viscosity and the porosity of the medium. It is a type of elliptic PDE that involves the Laplace operator, and we now consider the steady-state of the 2-d Darcy Flow equation on the unit box which is the second order, linear, elliptic PDE:

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x) \quad x \in (0,1)^2 \tag{26}$$

$$u(x) = 0 \quad x \in \partial(0,1)^2 \tag{27}$$

with a Dirichlet boundary where $a \in L^\infty((0,1)^2;\mathbb{R}_+)$ is the diffusion coefficient and $f \in L^2((0,1)^2;\mathbb{R})$ is the forcing function. We are trying to learn the operator mapping the diffusion coefficient to the solution, $G^\dagger : L^\infty((0,1)^2;\mathbb{R}_+) \to H_0^1((0,1)^2;\mathbb{R}_+)$ defined by $a \mapsto u$ [14].

3.1.2 Numerical Experiments for Darcy Flow

We generate the Darcy Flow datasets through traditional solvers. The data consists of grayscale images that represent the pressure field of the fluid flow through a porous medium. Each image in the data represents a 2-d slice through the medium, with darker areas indicating lower pressure and lighter areas indicating higher pressure. Each dataset is loaded as a 3-d tensor. The first dimension is the sample index, and the rest of the indices are the discretization. In our experiments, we generate $1000 \times 16 \times 16$ (16×16 resolution), $1000 \times 32 \times 32$ (32×32 resolution), and $1000 \times 64 \times 64$ (64×64 resolution) Darcy flow tensors to train the operator, and then test our operators on both 16×16 and 32×32 resolutions of 50 samples (same). In the training and testing process, the data we are using is the sample of coefficient-value pairs. The coefficients are samples of mesh data in a Gaussian random field on $[0,1]^2$ with zero mean and covariance operator $C = (-\Delta + \tau^2)^{(-\alpha)}$, where Δ is the Laplacian with zero Neumann boundary conditions, and α and τ control smoothness (the bigger they are, the smoother the function). Then we derive the value of the PDE by solving the equation $-d(a(x) * dp) = f(x)$. The input data Figure 2 and Figure 3 and operator training results Figure 4 and Figure 5 for both the 16×16 and 32×32 resolutions are shown below. We could find that the training results are close to the ground truth, with average loss 1.1172, training error 0.0223 and average loss 1.1729, training error 0.0235 respectively, and they are almost close to each other. This phenomenon lays a solid foundation for the resolution error invariance property.

Remark: The Figure 2 and Figure 3 in the first line are data after taking log function to ensure

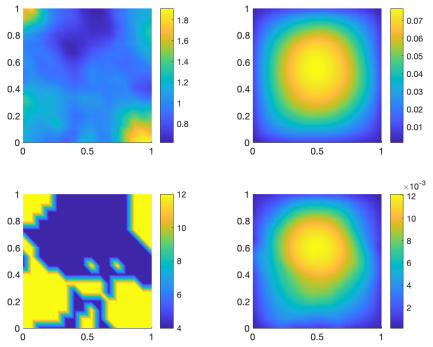

Figure 2: Input data of 16×16 resolutions: left: coefficient; right: value

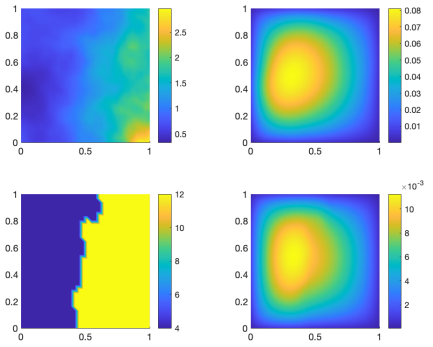

Figure 3: Input data of 32×32 resolutions: left: coefficient; right: value

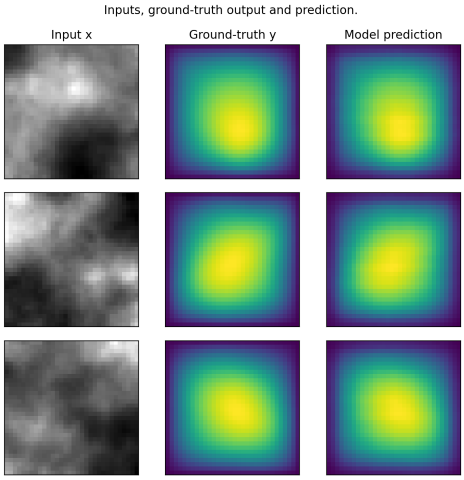

Figure 4: Testing result of training the operator using 16×16 resolutions (Gaussian) data

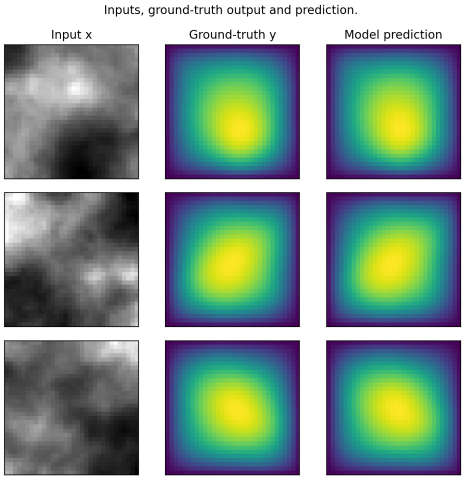

Figure 5: Testing result of training the operator using 32×32 resolutions (Gaussian) data

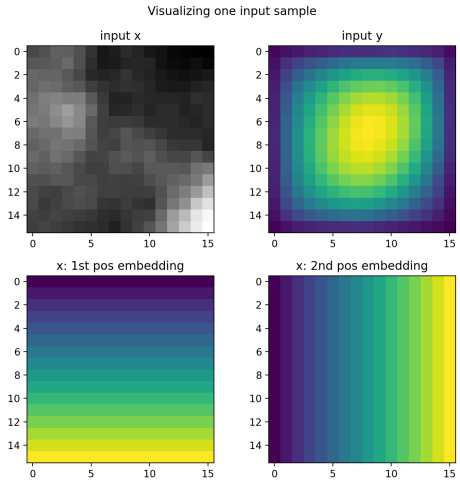

Figure 6: Training input for 16×16 resolutions (Gaussian)

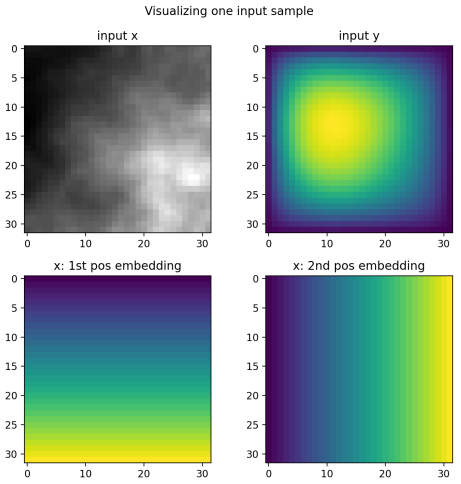

Figure 7: Training input for 32×32 resolutions (Gaussian)

ellipticity; the figures in the second line are data after thresholding to ensure ellipticity.

Aside from testing on the original Gaussian field, we can test on other datasets with a different structure. Similarly, the input training data is the same as the above, which can be displayed in Figure 6 and Figure 7. However, the testing set is no longer random Gaussian coefficient data, and the testing results from nine individual experiments (16×16, 32×32 and 64×64 resolutions) are shown in Figures 8 9 10 11 12 13 14 15 16. The experiments on the first line are three operators trained on the same 16×16 resolution data, while the experiments on the second line are three operators trained on the same 32×32 resolution data, and the same for 64×64 resolution data on the third line. All of them are tested on the same testing data, and their relevant average loss, training error, and training time are reflected in the table 3.1.2, to illustrate some properties of Fourier neural operator.

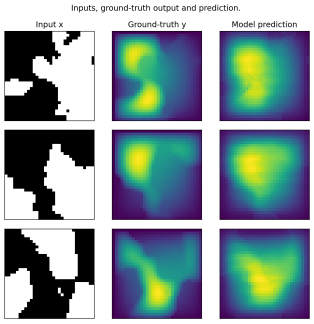

Figure 8: First testing output for 16×16 resolutions

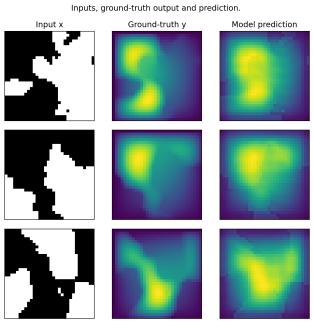

Figure 9: Second testing output for 16×16 resolutions

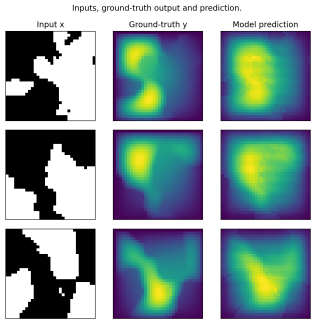

Figure 10: Third testing output for 16×16 resolutions

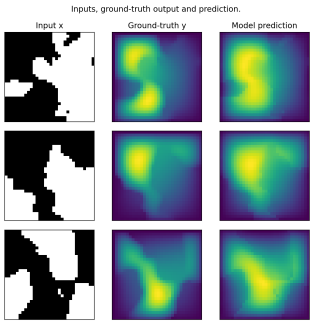

Figure 11: First testing output for 32×32 resolutions

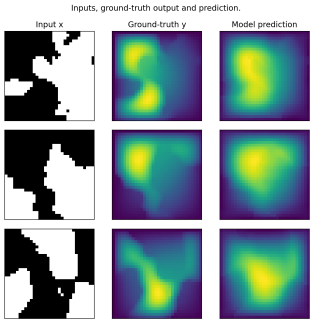

Figure 12: Second testing output for 32×32 resolutions

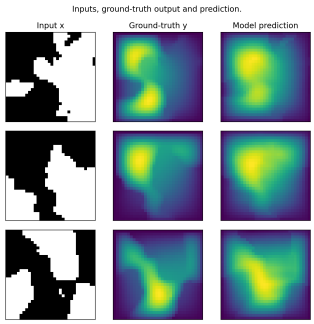

Figure 13: Third testing output for 32×32 resolutions

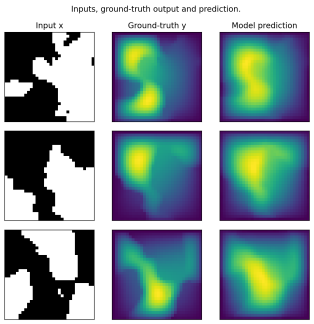

Figure 14: First testing output for 64×64 resolutions

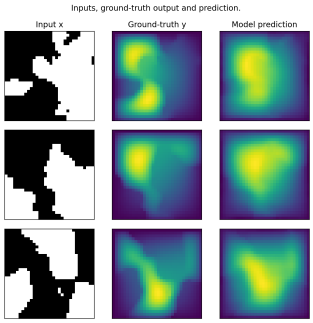

Figure 15: Second testing output for 64×64 resolutions

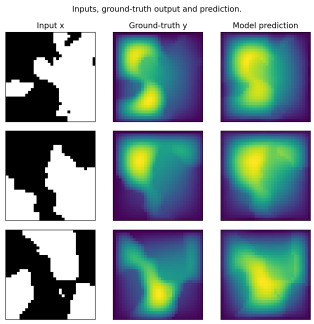

Figure 16: Third testing output for 64×64 resolutions

Table 3.1.2: Training statistics for Darcy flow experiments			
Resolution	Average Loss	Training Error	Training Time (s)
16×16	0.9870	0.0197	3.57
16×16	1.1531	0.0231	3.59
16×16	1.0098	0.0202	3.62
32×32	1.0440	0.0209	7.72
32×32	1.0428	0.0209	7.64
32×32	0.9268	0.0185	7.60
64×64	1.1056	0.0221	25.30
64×64	1.1160	0.0223	25.28
64×64	1.2202	0.0244	25.12

From Table 3.1.2, we can find that the training error and average loss are nearly the same for different resolutions, which proves that the resolution-invariant operator has consistent error rates among different resolutions. Additionally, we plot the training time for different input resolutions with the same big O notation constant C. We find that rather than increasing quadratically (16×16 resolution to 32×32 resolution, 32×32 resolution to 64×64 resolution), the training time follows quasi-linear complexity, in other words, $O(NlogN)$. This property is also reflected in Figure 17. The numerical experiments demonstrate such properties of Fourier neural operators. In the future, we will train more examples with diverse resolutions to better prove the quasi-linear time complexity.

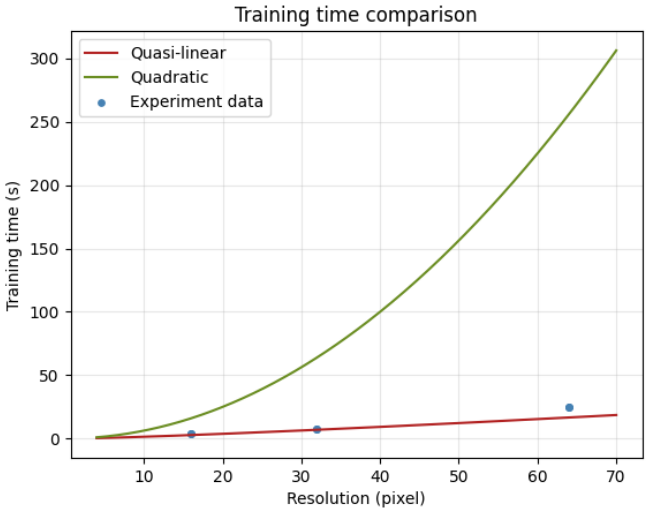

Figure 17: Training time for the Darcy Flow Model

3.2 Fokker-Planck (Kolmogorov Forward) Equation

3.2.1 Fokker-Planck Problem Formulation

The Fokker-Planck equation, also known as the Kolmogorov forward equation, is a partial differential equation that describes the time evolution of a probability density function (PDF) associated with a stochastic process. It was first introduced by Adriaan Fokker and Max Planck in the early 20th century to describe the diffusion of Brownian particles under the influence of random forces [23].

The Fokker-Planck equation is typically written in the form:

$$\frac{\partial P(x,t)}{\partial t} = -\frac{\partial}{\partial x}\left[a(x,t)P(x,t)\right] + \frac{1}{2}\frac{\partial^2}{\partial x^2}\left[b(x,t)P(x,t)\right] \tag{28}$$

where $P(x,t)$ is the probability density function of the stochastic process at time t, $a(x,t)$ is the drift coefficient which describes the average rate of change of the process at location x and time t,

and $b(x,t)$ is the diffusion coefficient which describes the randomness of the process.

The Fokker-Planck equation is a powerful tool in studying stochastic processes as it allows us to calculate the evolution of the probability density function over time. In particular, it is often used to model diffusion processes, such as the diffusion of molecules in a solution, and to study the behavior of stochastic systems, such as the dynamics of financial markets or the behavior of biological systems.

3.2.2 Numerical Experiments for Fokker-Planck

Here we consider the 1-d Fokker Planck equation problem on a unit torus,

$$\frac{\partial P(x,t)}{\partial t} = -a\frac{\partial P(x,t)}{\partial x} + \frac{b}{2}\frac{\partial^2 P(x,t)}{\partial x^2}, \quad x \in (0,1),\ t \in (0,1] \tag{29}$$

$$u(x,0) = u_0(x), \quad x \in (0,1) \tag{30}$$

with periodic boundary conditions where $u_0 \in L^2_{per}((0,1);\mathbb{R})$ is the initial condition. a is the drift coefficient and b is the diffusion coefficient. In our numerical experiments, we set them as constant to simplify the problem. We consider two settings: $a = 0.005$ and $b = 0.002$, and $a = 0.5$ and $b = 0.2$, respectively. We aim to learn the operator mapping the initial condition to the solution at the time one, $G^\dagger : L^2_{per}((0,1);\mathbb{R}) \to H^r_{per}((0,1);\mathbb{R})$ defined by $u_0 \mapsto u(\cdot,1)$ for any $r > 0$.

In the experiment, the initial condition $u_0(x)$ is generated with respect to $u_0 \sim \mu$ and $\mu = \mathcal{N}(0, 625(-\nabla + 25\tau)^{-\alpha})$ with periodic boundary conditions (truncate between zero and one). We solve the linear part of the equation (we only have the linear part as well) using the split-step method in the Fourier space. We calculate the equation value on a spatial mesh with resolution

$2^{13} = 8192$. We then train the Fourier neural operator using this dataset of size 2048 (initial and final value pairs) in 20 epochs and test the operator on other different resolutions [4]. Figures 18 19 20 21 are the results of our training and testing of the Fokker-Planck equation with a learning rate 1×10^{-3} (the testing results have been scaled to have summation 1). Generally, during the training process, they have a training loss and validation loss around 0.01, and a validation mean square error around 1×10^{-7}, which shows that the Fourier neural operator being trained is a good solver of the proposed Fokker-Planck equation and a good predictor of the input data. Though the testing case in Figure 19 seems not perfect, the loss is actually small due to the scaling of the Y-axis. Additionally, Figure 22 and Figure 23 are testing examples that compare the change from the initial stage to the terminal time between large drift and diffusion coefficient and small drift and diffusion coefficient. It is obvious that a small drift and diffusion coefficient leads to small change during the whole process, while a large drift and diffusion coefficient leads to a dramatic change from $t = 0$ to $t = 1$, moving and stretching the whole shape in the process.

One drawback of the self-designed experiment is that a tiny part of the output values of the Fokker-Planck equation may be negative or over 1, which is invalid as the values should be the probability density function. According to Harrison [12], when we use traditional solvers like the finite difference method to solve the equation to generate experiment data, it produces erroneous oscillations and negative values if the drift is large compared with the diffusion. However, Harrison also mentions that small negative values could be tolerated in some cases. As a result, the training data we generated using the software may not be reliable enough, and we will work further to solve the problem in later research.

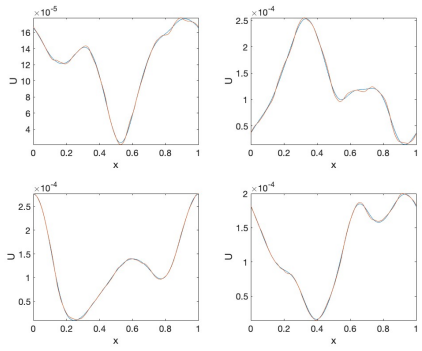

Figure 18: The testing output and ground truth of FP equation when a=0.005 and b=0.002

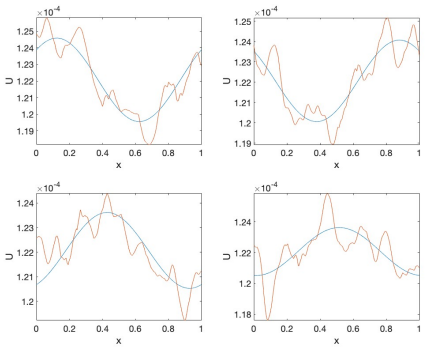

Figure 19: The testing output and ground truth of FP equation when a=0.5 and b=0.2

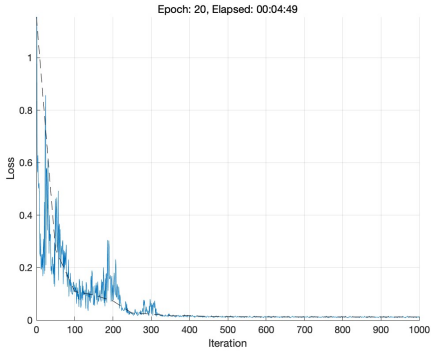

Figure 20: The training loss for FP equation when a=0.005 and b=0.002

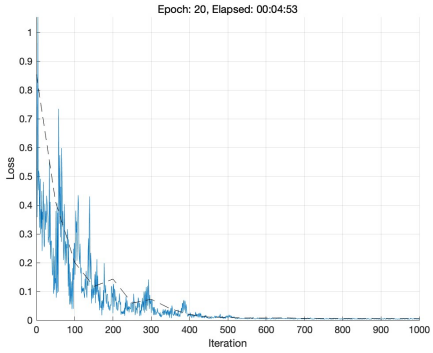

Figure 21: The training loss for FP equation when a=0.5 and b=0.2

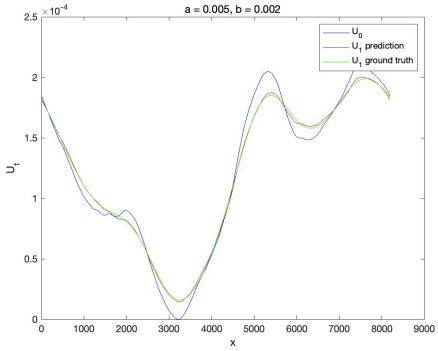

Figure 22: Comparison of the initial stage and terminal time when a=0.005 and b=0.002

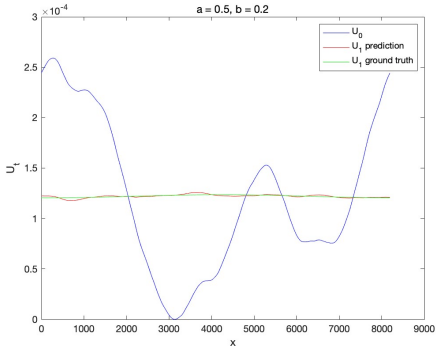

Figure 23: Comparison of the initial stage and terminal time when a=0.5 and b=0.2

4 Discussion and Conclusion

In conclusion, in the thesis, we first conducted detailed research and analysis of the Fourier neural operator. Generally speaking, the theoretical part of the thesis is a new and ordered illustration of the Fourier neural operator starting from scratch. We started with the graph construction, then set up the message-passing graph network, and finally built up the graph neural operator by transforming the discrete problem into a continuous format. We then introduced the iterative algorithm for solving the neural operator, and put such algorithm calculation in the Fourier spaces to boost the solution speed and efficiency. Moreover, we summarized the advantages of the Fourier neural operator such as resolution invariance, mesh independence, and quasi-linearity, and proved such properties through numerical experiments. Apart from the Darcy Flow experiments containing three different resolutions, we also performed experiments on a brand-new instance: the Fokker-Planck equation, by means of the Fourier neural operator. This is also the innovation part of the thesis. Our experiments proved that the Fourier neural operator trained is a good solver of the proposed PDEs, and has numerous advantages compared with other solvers.

References

[1] Adler, J., and Öktem, O. Solving ill-posed inverse problems using iterative deep neural networks. *Inverse Problems 33*, 12 (nov 2017), 124007.

[2] Bar, L., and Sochen, N. Unsupervised Deep Learning Algorithm for PDE-based Forward and Inverse Problems. *arXiv e-prints* (Apr. 2019), arXiv:1904.05417.

[3] Bhatnagar, S., Afshar, Y., Pan, S., Duraisamy, K., and Kaushik, S. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics 64*, 2 (2019), 525–545.

[4] Daly, C. fourier-neural-operator. https://github.com/matlab-deep-learning/fourier-neural-operator, April 16 2023. GitHub.

[5] E, W., and Yu, B. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics 6*, 1 (2018), 1–12.

[6] Evans, L. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010.

[7] Evans, L., and Society, A. M. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 1998.

[8] Gilbarg, D., and Trudinger, N. *Elliptic Partial Differential Equations of Second Order*. Classics in Mathematics. Springer Berlin Heidelberg, 2001.

[9] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *CoRR abs/1704.01212* (2017).

[10] Guibas, J., Mardani, M., Li, Z., Tao, A., Anandkumar, A., and Catanzaro, B. Adaptive fourier neural operators: Efficient token mixers for transformers, 2022.

[11] Guo, X., Li, W., and Iorio, F. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, Association for Computing Machinery, p. 481–490.

[12] Harrison, G. W. Numerical solution of the fokker planck equation using moving finite elements. *Numerical Methods for Partial Differential Equations 4*, 3 (1988), 219–232.

[13] KHOO, Y., LU, J., and YING, L. Solving parametric PDE problems with artificial neural networks. *European Journal of Applied Mathematics 32*, 3 (jul 2020), 421–435.

[14] Li, Z., Kovachki, N. B., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A. M., and Anandkumar, A. Fourier neural operator for parametric partial differential equations. *CoRR abs/2010.08895* (2020).

[15] Li, Z., Kovachki, N. B., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A. M., and Anandkumar, A. Neural operator: Graph kernel network for partial differential equations. *CoRR abs/2003.03485* (2020).

[16] Li, Z., Zheng, H., Kovachki, N., Jin, D., Chen, H., Liu, B., Azizzadenesheli, K., and Anandkumar, A. Physics-informed neural operator for learning partial differential equations, 2023.

[17] Pan, S., and Duraisamy, K. Physics-informed probabilistic learning of linear embeddings of nonlinear dynamics with guaranteed stability. *SIAM Journal on Applied Dynamical Systems 19*, 1 (jan 2020), 480–509.

[18] Raissi, M., Perdikaris, P., and Karniadakis, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics 378* (Feb. 2019), 686–707.

[19] Smith, J. D., Azizzadenesheli, K., and Ross, Z. E. Eikonet: Solving the eikonal equation with deep neural networks. *IEEE Transactions on Geoscience and Remote Sensing 59*, 12 (2021), 10685–10696.

[20] Vapnik, V. N. *Statistical Learning Theory*. Wiley-Interscience, 1998.

[21] Wikipedia contributors. Convolution theorem — Wikipedia, the free encyclopedia, 2023. [Online; accessed 16-April-2023].

[22] Wikipedia contributors. Fast fourier transform — Wikipedia, the free encyclopedia, 2023. [Online; accessed 16-April-2023].

[23] Wikipedia contributors. Fokker–planck equation — Wikipedia, the free encyclopedia, 2023. [Online; accessed 16-April-2023].

[24] Wikipedia contributors. Fourier transform — Wikipedia, the free encyclopedia, 2023. [Online; accessed 15-April-2023].

[25] Wikipedia contributors. Gaussian blur — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Gaussian_blur&oldid=1139408690, 2023. [Online; accessed 28-April-2023].

[26] Zhu, Y., and Zabaras, N. Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification. *Journal of Computational Physics 366* (2018), 415–447.