Machine Learning and Shock Detection

Thesis submitted by

Zuhaib Ul Zamann
2018MT60798

under the guidance of

Prof. Harish Kumar, Indian Institute of Technology Delhi

in partial fulfilment of the requirements
for the award of the degree of

Bachelor and Master of Technology

Department Of Mathematics
INDIAN INSTITUTE OF TECHNOLOGY DELHI

July 2023

THESIS CERTIFICATE

This is to certify that the thesis titled **Machine Learning and Shock Detection**, submitted by **Zuhaib Ul Zamann (2018MT60798)**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Dual Degree(B.Tech + M.Tech)**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof.Harish Kumar
Professor
Dept. of Mathematics
IIT-Delhi

Place: New Delhi

Date: 5th July 2023

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Prof. Harish Kumar, who guided me throughout the course of this project. His guidance, expertise and support throughout this journey have been invaluable.

I wish to acknowledge the help provided by Prof. Ananta Kumar Majee and Prof. Kamana Porwal. They provided valuable suggestions during the project which improved the quality and depth of the research done. I would also like to thank my classmates and faculty members for their support and compassion

Last but not least, I would like to thank my parents for their endless love, support and sacrifices they made throughout my academic journey. I would like to thank my brother for being my confidant and always lending an ear when I needed it the most.

I am aware that words alone cannot express the depth of my gratitude, but please accept my heartfelt thanks. Each of you has played an integral part in my journey, and I am truly fortunate to have such wonderful individuals in my life.

Zuhaib Ul Zamann
2018MT60798
Department of Mathematics
IIT Delhi

ABSTRACT

KEYWORDS: Shock cells; Troubled cell Indicators; Minmod; Discontinuous Galerkin Methods; Limitter; Convolutional Neural Network(CNN); Graph Neural Network

We consider the problem of identifying troubled cells in the solution of a differential equation using a numerical method. We consider the various formulations of this problem and discuss the developed machine learning models and non-machine learning models developed for the same. We discuss a novel approach to the problem and the motivation behind the model. We discuss the advantage of the model over the already established models and also describe some of the improvements that can be made in the model to make it more robust.

The main objective of this thesis is to develop a universal model that can make predictions irrespective of the size and shape of a mesh and is easy to develop and train for each numerical method. The model should be universal enough not to need hyperparameter tuning and should not be problem specific. We provide a step forward toward the development of such a model and discuss the advantages of the model over the other current models. The findings of this thesis contribute to the advancement of shock detection and localization in numerical solutions and will pave the way for high-order accurate simulations in Computational Fluid Dynamics.

Contents

List of Tables

List of Figures

ABBREVIATIONS

IITD	Indian Institute of Technology, Delhi
DG	Discontinuous Galerkin
CNN	Convolutional Neural Network
GNN	Graph Neural Network
GCN	Graph Convolution Networks
GAT	Graph Attention Network
MPNN	Message passing Neural Network
FVM	Finite Volume Method
FDM	Finite Difference Method
FEM	Finite Element Method
LMVT	Lagrange Mean Value Theorem

NOTATION

h	Mesh size
L_m	Approximation function of polynomial annihilation
q_m	Normalization factor
u	Variable under consideration
$[f]$	Jump Function
P_n	Legendre polynomial of order n

Chapter 1

INTRODUCTION

Numerical solutions of hyperbolic conservation laws often lead to solutions having discontinuities(shocks) even if we start with smooth initial data. If not properly accounted for, shocks can lead to numerical errors, oscillations, or even divergence of the solution. This is particularly problematic in applications where high accuracy is required, such as fluid dynamics, aerodynamics, and structural mechanics. Moreover, shocks often represent important physical phenomena. For example, in fluid dynamics, shocks can represent sudden changes in pressure or velocity, such as those occurring in supersonic flows or hydraulic jumps. In structural mechanics, shocks can represent sudden changes in stress or strain, such as those occurring in impact or fracture problems. Therefore, accurate detection and handling of shocks are crucial for obtaining reliable numerical solutions. It allows for the application of appropriate numerical methods or algorithms that can handle discontinuities, such as shock-capturing methods or adaptive mesh refinement techniques. Furthermore, it enables the accurate representation and understanding of the underlying physical phenomena.
To localize the limiting to the relevant regions of the mesh, a two-step strategy is followed:

1. Identify the cells in which the solution loses regularity.
2. Limit the solution in the flagged cells.

Most of this thesis will focus on the first step, i.e. identifying the cells in which the solution is discontinuous.
The cells near the discontinuities are referred to as troubled cells, and the techniques used to capture the troubled cells are known as troubled cell indicators.
We will discuss some empirical detectors and some machine-learning detectors also. We will then introduce a novel detector and discuss how the model is superior to its contemporaries. The rest of this thesis is structured as follows:

In chapter 2, we present two distinct formulations of the problem statement. Each formulation provides a different perspective on the challenges of identifying troubled cells and sets the stage for developing the machine-learning models in the subsequent chapters.

Chapter 3 delves into the Polynomial Annihilation (PA) method as a potential solution for the problem. We describe the theoretical foundations of the PA method and its implementation as a TCI.

In chapter 4, we explore the application of Convolutional Neural Networks (CNN) for shock detection. We discuss the Convolutional neural networks in detail and give insight into why convolutional neural networks can act as potential TCIs. We discuss the architecture and training of the CNN model and evaluate its performance as a TCI in comparison to the PA method.

In chapter 5, we give a detailed description of Graph Neural Networks(GNN) and why GNNs can act as universal TCIs, depending only on the numerical method used and not on the mesh structures and problem to be solved. We discuss the architecture of the GNN model and the process of training data generation and evaluate its performance in comparison with convolutional neural networks.

The final chapter presents the results of the application of the novel model to various test cases and compares its performance with the PA method and CNN. We draw conclusions from the results, discuss the implications of the findings, and provide recommendations for future research.

Each chapter is designed to build upon the previous ones, providing a comprehensive and systematic exploration of the problem and its potential solutions. The thesis concludes with a summary of the key findings and their implications for the field of numerical analysis and machine learning.

Finally, in the appendix, we will give a detailed proof of the results mentioned in the previous chapters and definitions of the key terms discussed throughout the thesis.

Chapter 2

Problem Statement

Numerical methods for solving Partial Differential Equations(PDEs) can be broadly classified as, Finite Volume Methods(FVM), Finite Difference Methods(FDM) or Finite Element Methods(FEMs)(See A.1 A.2 A.3 for more details). The type of numerical solutions obtained from each method hence differ from each other. Hence the problem of identifying the troubled cells will have many variants. We can broadly classify the problem of Troubled Cell Identification into two main categories viz FDM/FVM Type Solution and FEM/DG Type Solution. The thesis will focus on only these two variants of the problem

2.1 FDM/FVM Type Solution

Consider a function $f : D \to \mathbb{R}$ where $D \subset \mathbb{R}^d, d \geq 1$ is a compact set.
Without loss of generality let D is a $d-$ dimensional rectangle given by
$D = I_1 \times I_2 \times \ldots \times I_d$ where $I_i = [a_i, b_i], b_i = a_i + n_i\delta$ where $n_i \in \mathbb{N}, \delta > 0$.
Consider a uniform grid over D of $\prod_{i=1}^{d}(n_i + 1)$ grid points

$$S = \{(a_1 + i_1\delta, a_2 + i_2\delta, \ldots, a_d + i_d\delta), i_k \in \{0, 1, 2, \ldots, n_k\} \forall k = 1, 2, \ldots, d\}$$

Given the value of f at each of the points in S, the problem is to find the grid cells which contain points of discontinuity of f, i.e. we are given S and $f(S) = \{f(x) : x \in S\}$ and we have to label each cell of the grid as 0/1 with 0 meaning that the cell is not a troubled cell and 1 meaning that the cell is a troubled cell.
As is evident that this type of functional data will arise from solving the PDE using a Finite Difference Method(FDM), or using a Finite Volume Method(FVM). In the context of Finite Volume Methods, it's important to note that the average value of the function within a cell can be considered as the value at the centroid of the cell. This assumption is valid when the cells are of small size. Consequently, in such a scenario, the number of data points and the number of cells will be the same. In our 2D models, we have employed this concept. By considering the cell average value as the value at the centroid of the cell, we have generated data and trained the models.

2.2 FEM/DG Type Solution

Consider a function f defined on domain $\Omega \subset \mathbb{R}^d$. The space Ω is discretized into simpilicial elements $\{\Delta_0, \Delta_1, \ldots, \Delta_n\}$. In each of the elements, an approximate solution(e.g. polynomial approximation of degree $\leq p$ or Fourier series approximation, etc) is given. The aim is that give the approximation on each of the elements to detect the troubled cells.
Mathematically ,
Given $\{\Delta_0, \Delta_1, \ldots, \Delta_n\}$ and $\{P_0, P_1, \ldots, P_n\}$ where P_i is the approximation of the function f in the simplicial element Δ_i, determine cells containing discontinuity points of the function f.
As is evident that this type of functional data will arise from solving the PDE using a Finite Element Method(FEM), e.g DGFEM.
In the implementations of this thesis, we implement the Discontinuous Galerkin method for the one-dimensional advection equation. The complete detail of the implementations done will be provided in C

Chapter 3

Polynomial Annihilation Detection

The goal of polynomial annihilation[2] is to construct a function $L_m f(x), m \in \mathbb{N}$, such that for x away from discontinuity points of $f(\cdot), L_m f(x) \approx 0$.
Hence the detection of discontinuities is based on $|L_m f(x)| > t$ where t is some threshold.
Suppose that f is known only on the discrete set S. Let Π_m be the space of all polynomials of degree $\leq m-1$ in d variables.
The value of $L_m f(x)$ at $x \in D$ is determined by the function values of f on a local set $S_x \subset S$ of $m_d = \begin{pmatrix} m+d \\ d \end{pmatrix}$ points around x.
Let $S_x = x_1, x_2, \ldots, x_{m_d}$, be the set of m_d nearest points to x.
For polynomial annihilation up to degree $m-1$, one solves the linear system for coefficients $\{c_j(x) : j = 1, 2, \ldots, m_d\}$

$$\sum_{x_j \in S_x} c_j(x) p_i(x_j) = \sum_{|\alpha|_1 = m} p_i^{\alpha}(x)$$

where $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d), \alpha_i \in \mathbb{N} \bigcup \{0\}, p_1, p_2, \ldots, p_{m_d}$ is a basis of Π_m
Since the solution of the above system exists and is unique, one can define

$$L_m f(x) = \frac{1}{q_{m,d}(x)} \sum_{x_j \in S_x} c_j(x) f(x_j) \tag{3.1}$$

where $q_{m,d}(x)$ is a suitably chosen normalization factor

3.1 One Dimensional Case

We consider the basis of the polynomial space as $\{1, x, x^2, \ldots, x^m\}$.
The grid is uniformly constructed as $[a, a+h], [a+h, a+2h], \ldots, [a+(n-1)h, b]$ and on this grid we define S_x as the set of nearest $m+1$ points in the grid to x.
We define $S_x^+ = \{i \in \mathbb{N} | x_i \in S_x, x_i \geq x\}$ and choose the normalization factor as

$$q_m(x) = \sum_{i \in S_x^+} c_i(x)$$

The system of equations to be solved for $L_m f(x)$ in this case becomes

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & 1\dots & 1 \\ x_1 & x_2 & x_3 & \dots & x_{m+1} \\ x_1^2 & x_2^2 & x_3^2 & \dots & x_{m+1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^m & x_2^m & x_3^m & \dots & x_{m+1}^m \end{bmatrix}}_{:=\mathbf{A}} \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_m \end{bmatrix}}_{:=\mathbf{c}} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ m! \end{bmatrix}}_{:=\mathbf{b}} \tag{3.2}$$

In the implementation, we take $m = 5$. Better results can be obtained by taking larger values of m.

Suppose that f has jump discontinues and $J = \{\xi : a \leq \xi \leq b\}$ be the set of jump discontinuities of f.Define

$$[f](x) := \lim_{t\to x^+} f(t) - \lim_{t\to x^-} f(t)$$

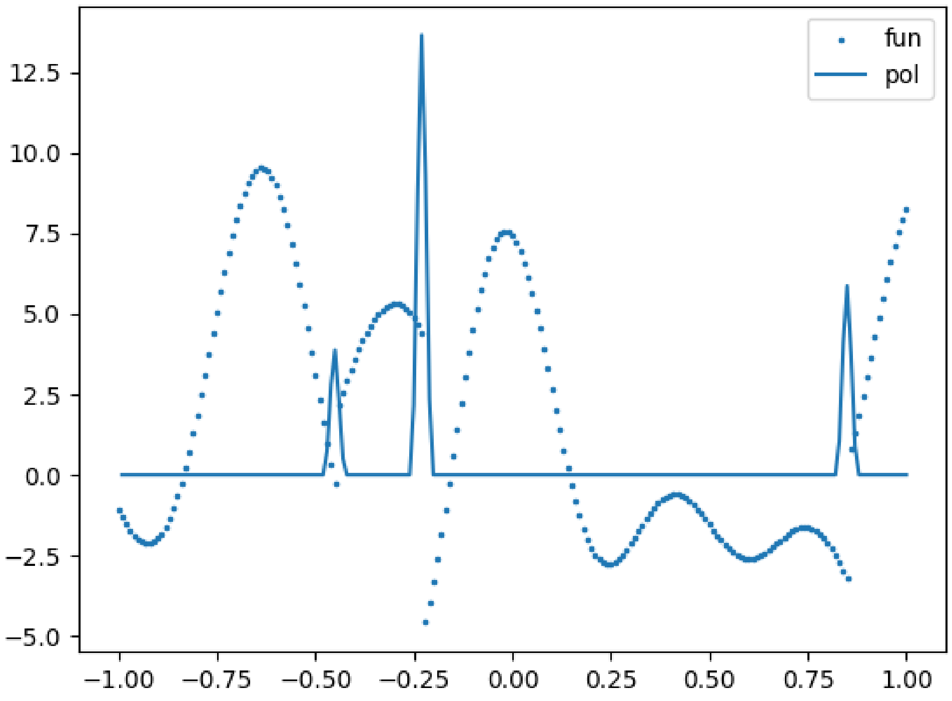
Clearly the better we approximate $[f]$ with $L_m f$, the better is our indicator function.

Theorem 1. *$L_m f$ satisfies the following relation*

$$L_m f(x) = \begin{cases} [f](x) + O(h) & \textit{if } x, \xi \in [x_{j-1}, x_j], \xi \in J \\ O(h^{\min(m,k)} & \textit{if } f \in C^k(I_x), k > 0 \end{cases} \tag{3.3}$$

where $I_x = \overline{S_x}$.

Proof of this result is given in Appendix B.



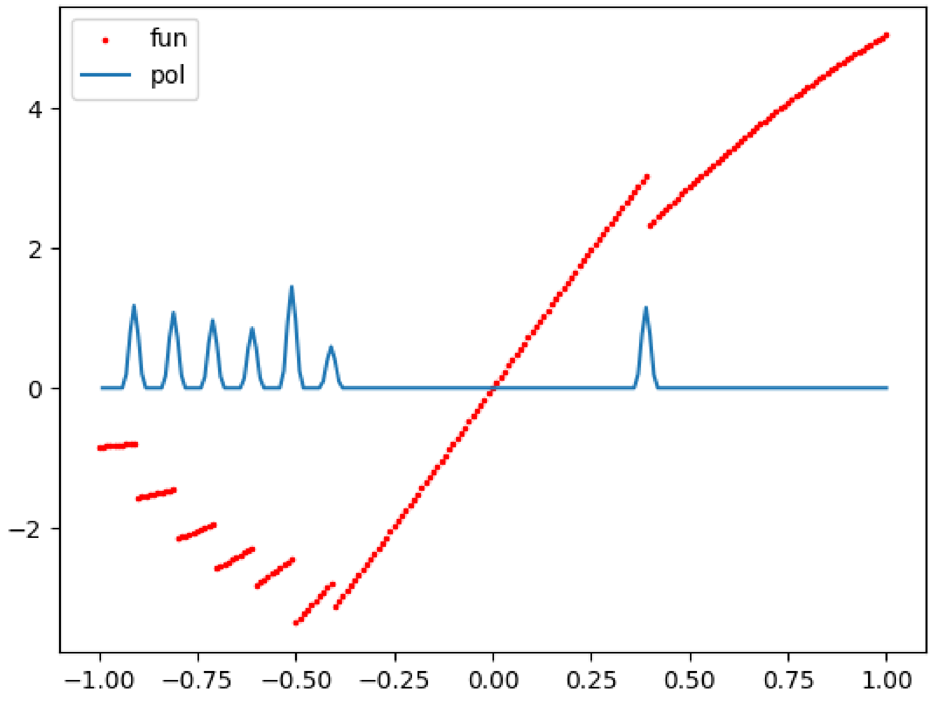


Figure 3.1: Examples of polynomial Annihilation detection. It can be clearly seen that at the points of discontinuity the function value peaks and rapidly decreases to zero as we move away from the points of discontinuity

3.2 Two Dimensional Case

We consider the basis of the space of two-dimensional polynomials with degree $\leq m$ as $\{x^i y^j | i+j \leq m\}$.
The entire Domain is divided into triangular elements.
Let $S = \{T_1, T_2, \ldots, T_N\}$ be the discretization of the space into triangular elements.
For an element T_j of S define $\bar{x_j} = \left(\frac{x_{j1}+x_{j2}+x_{3j}}{3}, \frac{y_{j1}+y_{j2}+y_{j3}}{3}\right)$ where the vertices of T_j are $(x_{j1}, y_{j1}), (x_{j2}, y_{j2}), (x_{j3}, y_{j3})$
For $x \in \Omega$, define $S_x = T_j \cup S_{T_j}$ where $x \in T_j$ and S_{T_j} is the collection of nearest $m_2 - 3$ points to x in $S \backslash T_j$
The detection algorithm can be described in the following 3-steps
Step 1: For element T_j we consider $S_{\bar{x_j}}$ and order $S_{\bar{x_j}} = \{x_1, x_2, \ldots, x_{m_2}\}$ in order such that $f(x_1) \leq f(x_2) \leq f(x_3) \ldots \leq f(x_{m_2})$.
Step 2: Solve the system

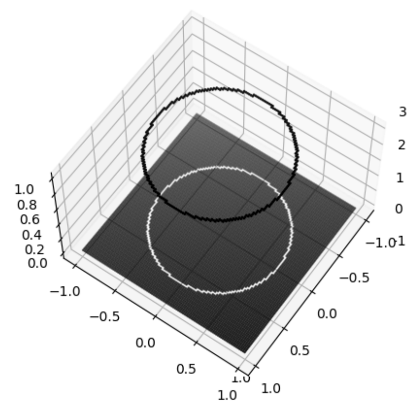
$$\sum_{x_i \in S_{\bar{x_j}}} c_i x_{i1}^{\alpha_1} x_{i2}^{\alpha_2} = \begin{cases} 0, \text{ if } \alpha_1 + \alpha_2 < m \\ \alpha_1! \alpha_2!, \text{ if } \alpha_1 + \alpha_2 = m \end{cases}$$

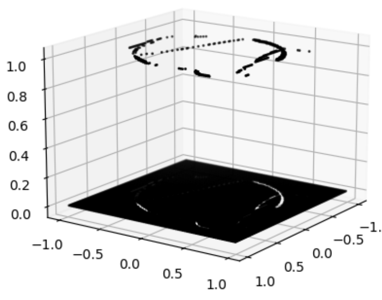
Step 3: Calculate $q_m(\bar{x_j}) = \sum_{x_i \in \mathcal{P}_{\bar{x_j}}} c_i$,
where $\mathcal{P}_x = \{x_1, x_2, x_3, \ldots, x_r\}$, and $|f(x_{r+1} - f(x_r)| = \max_{i=1,2,\ldots,m_2-1} |f(x_{i+1}) - f(x_i)|$
Step 4: Calculate $L_m f(\bar{x_j}) = \frac{1}{q_m(\bar{x_j})} \sum_{x_i \in S_{\bar{x_j}}} c_i f(x_i)$
Detect discontinuity if $|L_m f(\bar{x_j})| \geq t$ for some suitably chosen threshold t

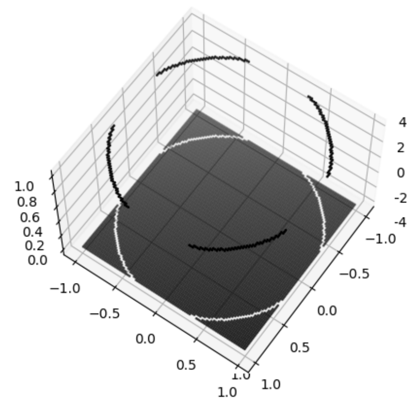
Theorem 2. *For two dimensional case, if $f \in C^k(K_{S_x})$, then*

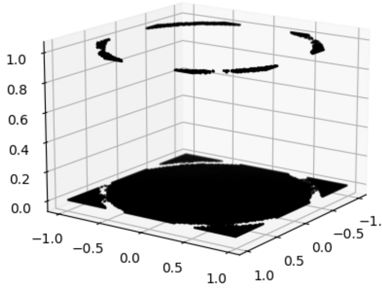
$$L_m f(x) = O(h^{\min(k,m)})(x)$$

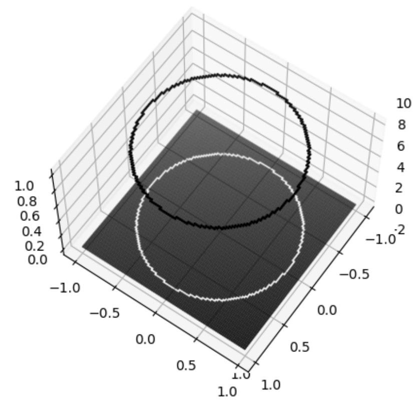
See Appendix B for proof


(a) Ground Truth


(b) Model prediction with $m = 2$


(c) Ground Truth


(d) Model prediction with $m = 5$


(e) Ground Truth

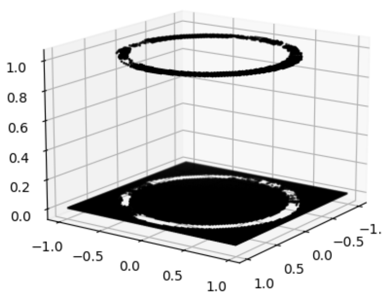

(f) Model prediction with $m = 7$

Figure 3.2: Polynomial Annihilation model predictions with $m = 2, 5, 7$. Notice the improvement in indicator quality as m increases

3.3 Coupling with Minmod

We obtain far superior results if we couple the polynomial annihilation method with minmod function. The procedure is as follows
Define the set $\mathcal{M} = \{1, 2, \dots, M\}$
Define $L_{\mathcal{M}} f = \{L_m f : \mathbb{R} \to \mathbb{R} | m \in \mathcal{M}\}$
The *minmod* function is hence given by

$$MM(L_{\mathcal{M}} f(x)) = \begin{cases} \min\limits_{m \in \mathcal{M}} L_m f(x) \text{ if } L_m f(x) > 0 \text{ for all } m \in \mathcal{M} \\ \max\limits_{m \in \mathcal{M}} L_m f(x) \text{ if } L_m f(x) < 0 \text{ for all } m \in \mathcal{M} \\ 0 \text{ Otherwise} \end{cases}$$

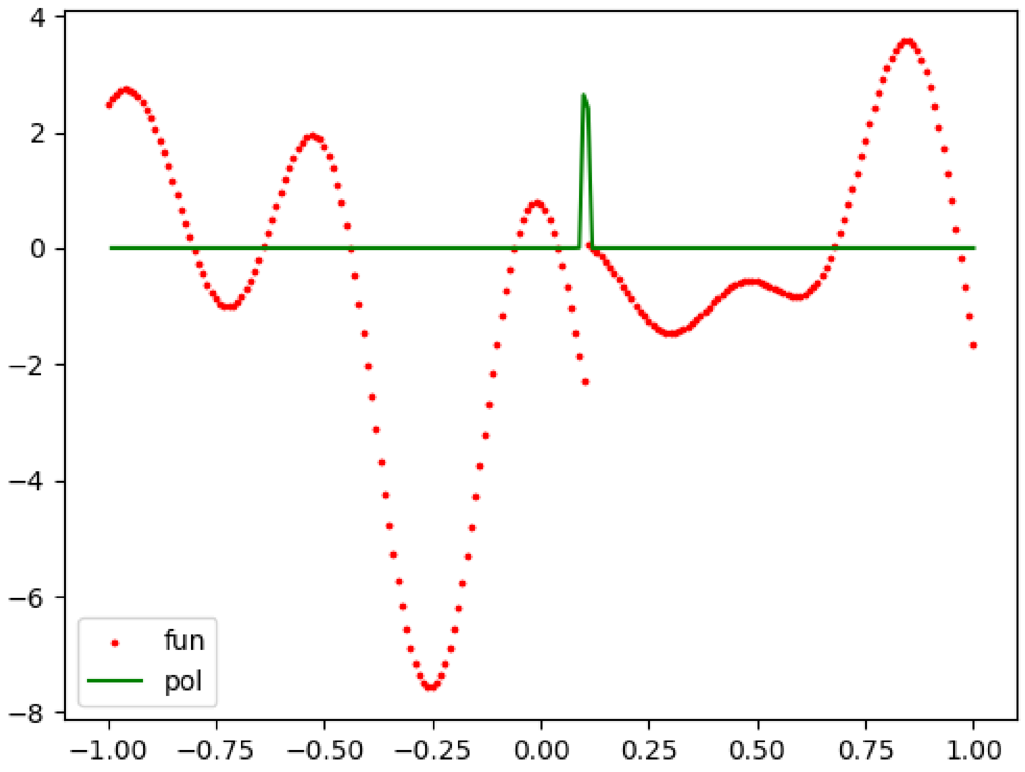
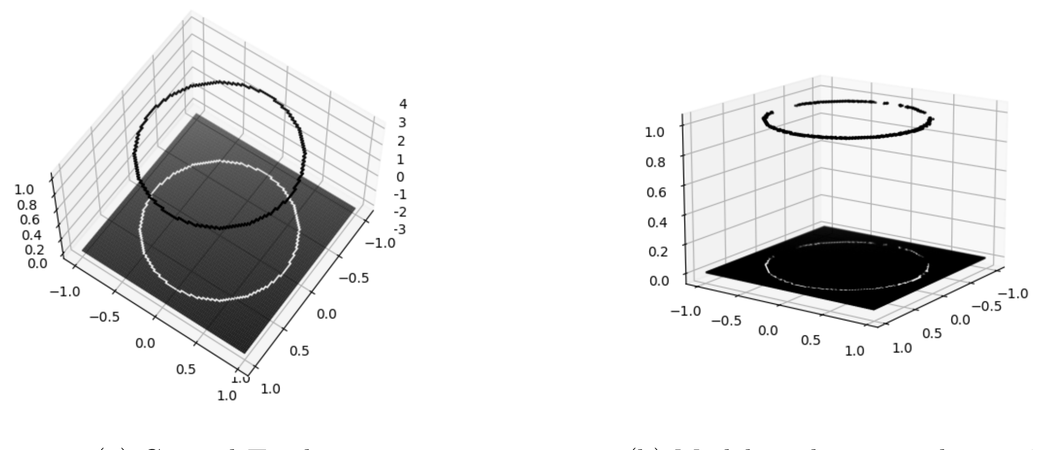


Figure 3.3: Minmod coupled polynomial annihilation method(1D). Notice the thinner peak line, which indicates a rapid decrease to 0 away from points of discontinuity of f



(a) Ground Truth (b) Model prediction with $m = 7$

Figure 3.4: Minmod coupled with polynomial annihilation(2D). Notice the thinner boundary lines. This indicates that coupling with minmod allows the method to decrease more rapidly towards zero away from points of discontinuity

Theorem 3. *For one dimensional case , $\mathcal{M} = \{1, 2, \dots, \mu\}$, we have*

$$MM(L_{\mathcal{M}} f(x)) = \begin{cases} [f](\xi) + O(h(x)) & \text{if } x, \xi \in [x_{j-1}, x_j], x \in J \\ O(h^{\min(\mathcal{M}_x, k)}(x)) & \text{if } f \in C^k(I_x) \end{cases}$$

where $I_x = \overline{S_x}, \#S_x \leq \mathcal{M}_x + 1$ and $\mathcal{M}_x$ is defined as

$$\mathcal{M}_x = \max\{m \in \mathcal{M} | \#S_x = m + 1, I_x \cap J = \emptyset\}$$

Proof of this result is given in Appendix B.
We note that even though we have verified the results only for uniform mesh, the results can be extended to non-uniform meshes also. The implementation provided in the GitHub link of this project can easily be extended to non-uniform meshes also, details of which will be provided in the README file of the project.

Chapter 4

Convolutional Neural Networks

Convolutional Neural Networks (CNNs), also known as Convnets, are a specific category of neural networks that excel in processing data with a grid-like topology, such as images and videos. A typical CNN is generally composed of convolutional layers, pooling layers, and a few fully connected layers.

CNNs are particularly adept at computer vision tasks such as edge detection, image segmentation, and image classification. They have brought about a revolution in the field of computer vision, achieving results that surpass human-level classification and nearing perfection in segmentation and object detection tasks.

In addition to their high accuracy, CNNs are straightforward to implement and extremely fast, making them highly suitable for tasks that require high precision and accuracy. Figure 4.2 illustrates an example of a CNN model, highlighting the relative positions of each type of layer. A CNN generally consists of the following layers:

1. **Convolutional Layers**: These are the primary building blocks of CNNs. They perform a mathematical operation known as convolution on the input data. This operation involves the application of a filter or kernel to the input data to produce a feature map. The convolutional layer learns the values of these filters on its own during the training process.

2. **Pooling Layers**: These layers are used to reduce the spatial dimensions (width and height) of the input volume. This is done to decrease the computational complexity, control overfitting, and progressively aggregate the spatial information.

3. **Fully Connected Layers:** These layers are typically placed at the end of the network. They take the output of the previous layers (which represent high-level features of the input data) and transform them into a form suitable for final classification.

CNNs have proven to be incredibly effective in the field of image recognition and classification, and their use continues to expand into new areas as the technology evolves. The following sections provide a brief description of the various layers of CNNs.

4.1 Convolutional Layer

Convolutional layers are the core components of Convolutional Neural Networks (CNNs). These layers make up the majority of a CNN, and their primary function is to perform a

mathematical operation known as convolution. The convolution operation in these layers is performed between two matrices. One of these matrices is the input to the convolutional layer, which could be an image or the output from a previous layer. The other matrix is a set of learnable parameters, often referred to as a filter or kernel. This filter is a small matrix that is used to modify the input. The filter is moved across the input matrix, performing element-wise multiplication and summing the results to produce a new matrix, known as a feature map or convolved feature. This process is repeated for different filters, each designed to detect a specific feature in the input, such as edges, corners, or textures.
One of the key characteristics of convolutional layers, and by extension CNNs, is their ability to share parameters across different parts of the image. This means that the same filter is used to scan the entire image, making the model's predictions robust to translational changes. In other words, regardless of where a particular feature appears in the image, the model will be able to recognize it because it uses the same filter throughout the image. This property is particularly useful in computer vision tasks, where the location of a feature in an image can vary.
In addition to this, convolutional layers also help in reducing the complexity of the model by significantly reducing the number of parameters, making the model more efficient. This is because each filter in a convolutional layer is typically much smaller than the input, and the same filter is used across the entire input
A convolutional layer consists of a set of matrices of learnable parameters (known as kernels) stacked on each other. Each kernel through convolution operation produces one channel of the output.
Consider a 4×4 matrix(A) and a kernel of size 3×3(K)

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

The output of convolution of K and A denoted as $K * A$ is given by

$$K * A = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$$

$(i, j)^{th}$ element of $K * A$ by doing element wise product with the matrices

$A[s(i-1)+1 : s(i-1)+k, s(j-1)+1 : s(j-1)+k]$ and K and then summing the elements of this matrix, where s is the stride length(typically taken as 1) and k is the kernel size (typically taken as 3).

Hence $(K * A)_{1,1} = sum\left(\begin{bmatrix}1 & 1 & 0\\1 & 1 & 0\\1 & 1 & 0\end{bmatrix} \odot \begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}\right) = sum\left(\begin{bmatrix}1 & 0 & 0\\1 & 0 & 0\\1 & 0 & 0\end{bmatrix}\right) = 3$

Similarly, other elements can be obtained.

A convolutional layer enables sparse connections between outputs and inputs as an output value is calculated using only a few numbers of inputs. This leads to a dramatic decrease in the number of parameters. Furthermore, due to the nature of convolution operation, multiple outputs share parameters. This essentially helps the model to generalize any learning to the whole of the image. Thus the model develops equivariance to translation.

4.2 Pooling Layer

Pooling is a crucial operation in Convolutional Neural Networks (CNNs) that involves the process of spatial downsampling of the feature maps (outputs from the convolutional layers). This is achieved by sliding a window, typically of size 2×2, across each channel in the feature map.

The window, also known as a filter or a kernel, moves across the feature map, covering different sections according to a specified stride (the amount by which the window moves). For each position of the window, a summarizing operation is performed on the features within that window. This operation could be calculating the mean (average pooling), the maximum value (max pooling), etc.

The result of this operation is a pooled feature map, which has a reduced spatial size compared to the original feature map. For instance, if a 2×2 window is used with a stride of 2, the spatial dimensions (width and height) of the feature map are effectively halved.

One of the key benefits of pooling is that it introduces a form of spatial invariance to the model, making it more robust to variations or noise in the input. This means that even if the position of features in the input changes slightly, the pooled feature map remains relatively unchanged. This property is particularly useful in image recognition tasks where the exact location of features isn't as important as their presence.

In addition, by reducing the spatial dimensions of the feature maps, pooling also helps to decrease the computational load for subsequent layers in the CNN and helps to prevent overfitting by providing an abstracted form of the input.

Consider the following feature map

$$A = \begin{bmatrix}2 & 2 & 7 & 3\\9 & 4 & 6 & 1\\8 & 5 & 2 & 4\\3 & 1 & 2 & 6\end{bmatrix}$$

We consider a window size of 2×2 and stride length of 2 and use max as our summary statistic. The output hence is given by

$$B = \begin{bmatrix} 9 & 7 \\ 8 & 6 \end{bmatrix}$$

as shown in the diagram4.1

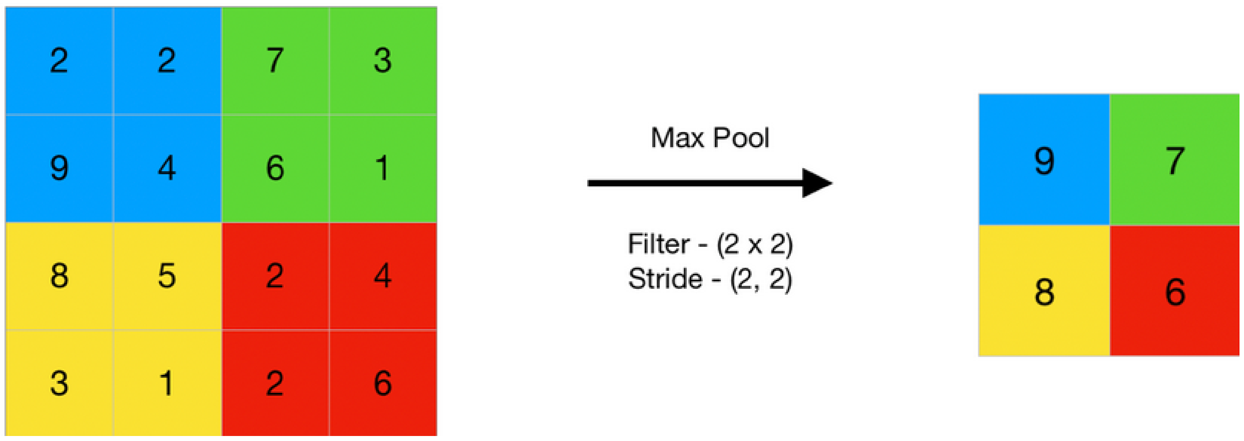


Figure 4.1: Max Pooling Example: Each colour depicts the window in which max is taken and corresponds to one element in the output

4.3 Fully Connected layer

In a Convolutional Neural Network (CNN), a fully connected layer, also known as a dense layer, is a layer in which each node (or neuron) is connected to every neuron in the previous layer. This is in contrast to the convolutional and pooling layers, where neurons are connected to only a small subset of neurons in the previous layer, corresponding to their receptive field.

The fully connected layer plays a crucial role in the network. It takes the high-level features extracted by the convolutional layers and uses them to classify the input image into various classes based on those features. The neurons in the fully connected layer perform complex non-linear transformations of the input features to map them to the output space.

Due to the all-to-all connections between the neurons, the fully connected layer often has the most parameters in the network. For instance, if a fully connected layer has n inputs and m outputs, the total number of weights (parameters) in this layer would be nm. Additionally, there is a bias term associated with each output neuron, adding m more parameters. This results in a total of $(n+1) * m$ parameters.

The fully connected layer essentially learns a possibly non-linear function in the high-level feature space that was output by the convolutional layers. This function is used to classify the input image into one of the many classes, which is the final output of the network. The learning process involves adjusting the weights and biases of the neurons in the fully connected layer using a process called backpropagation and an optimization algorithm, such as stochastic gradient descent.

In summary, the fully connected layer in a CNN is responsible for taking the high-level features extracted by the convolutional layers and mapping them to the final output classes. It does this by learning a possibly non-linear function in the feature space, which is used to classify the input image.

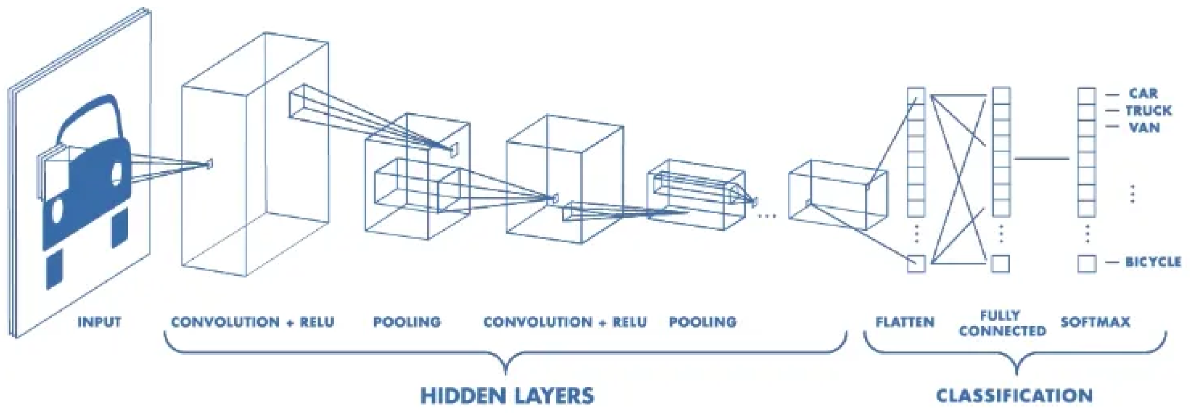


Figure 4.2: An example of a CNN model. Notice that pooling layers follow after a few convolutional layers, and Fully connected layers are present at the last of the model. This pattern is followed in almost all of the CNN models

The output of each layer in a CNN is passed through a non-linear function(Known as activation function), e.g. ReLU, LeakyRelu, sigmoid, tanh etc.

- **Rectified Linear Unit(ReLU)**: $ReLU(x) = \max(x, 0)$
- **Leaky ReLU**: $LeakyReLU(x) = \max(x, 0) + \alpha \min(x, 0)$, where α is known as negative slope parameter.
- **Paramter ReLU**: $ParamReLU(x) = \max(x, 0) + \beta \min(x, 0)$, where β is a learnable parameter.
- **Sigmoid**: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$
- **Softmax**

$$softmax \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \frac{\exp x_1}{\sum_{i=1}^{n} \exp x_i} \\ \frac{\exp x_2}{\sum_{i=1}^{n} \exp x_i} \\ \vdots \\ \frac{\exp x_n}{\sum_{i=1}^{n} \exp x_i} \end{bmatrix}$$

- **tanh**: $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

4.4 CNN as Troubled Cell Indicator

The tasks of discontinuity identification and edge detection share a common goal: to identify abrupt changes or breaks in data. In the context of image processing, edge detection is essentially the process of identifying points in a digital image where the brightness of pixel intensities changes distinctly, marking the boundaries of objects within an image. This is fundamentally a process of discontinuity identification but applied to the pixel values of an image.

Convolutional Neural Networks (CNNs) have demonstrated remarkable efficiency in edge detection tasks. They do this by learning and applying a series of filters that can identify features such as edges in input data. Given their success in this area, it is reasonable to suggest that CNNs could also be effective in identifying discontinuities in functional data, which is essentially another form of edge detection but in a different context.

This idea leads to the proposition of training CNNs to act as troubled cell indicators[16], where they could identify cells (or data points) that represent discontinuities. The performance of such a detector would, however, depend on the quality and diversity of the training data used. The more varied and comprehensive the training data, the better the CNN would likely be at identifying a range of different discontinuities. In the following sections of this chapter, we will delve into various CNN models for one-dimensional and two-dimensional cases. We will also discuss the process of generating training data for these models.

4.5 One Dimensional CNN Detector

Consider $D = [a, b]$ and $S = \{x_i = a + (i-1)\delta, i = 1, 2, \ldots, N+1\}$
This set S forms a uniform grid with N cells, where each cell is represented by the intervals $[x_i, x_{i+1}), i = 1, 2, \ldots, N$.
We then define a binary vector $y = (y_1, y_2, \ldots, y_n) \in \{0, 1\}^N$ which indicates the ground truth values for each cell $i = 1, 2, \ldots, N$. In this vector, $y_i = 1$ indicates that the i^{th} cell is a trouble cell, and $y_i = 0$ indicates that it is not. Let $v_f = \{f(x) : x \in S\}$ be the set of observed function values on S.
Before feeding these observed function values into a Convolutional Neural Network (CNN), we standardize them. This is done by subtracting the mean μ_f and dividing by the standard deviation σ_f of the observed function values v_f. This gives us the standardized

observed function values

$$\tilde{v}_f = \left\{ \frac{f(x) - \mu_f}{\sigma_f} : x \in S \right\}$$

.

The CNN then processes these standardized observed function values and outputs a vector of N real values, denoted as $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N) = \mathcal{N}(\tilde{v}_f)$. This output vector $\hat{y}$ serves as an estimate of the ground truth vector y.
Finally, we choose a threshold value t. For each cell i, if the corresponding value $\hat{y}_i$ in the output vector is greater than the threshold t, the detector labels the i^{th} cell as a trouble cell.

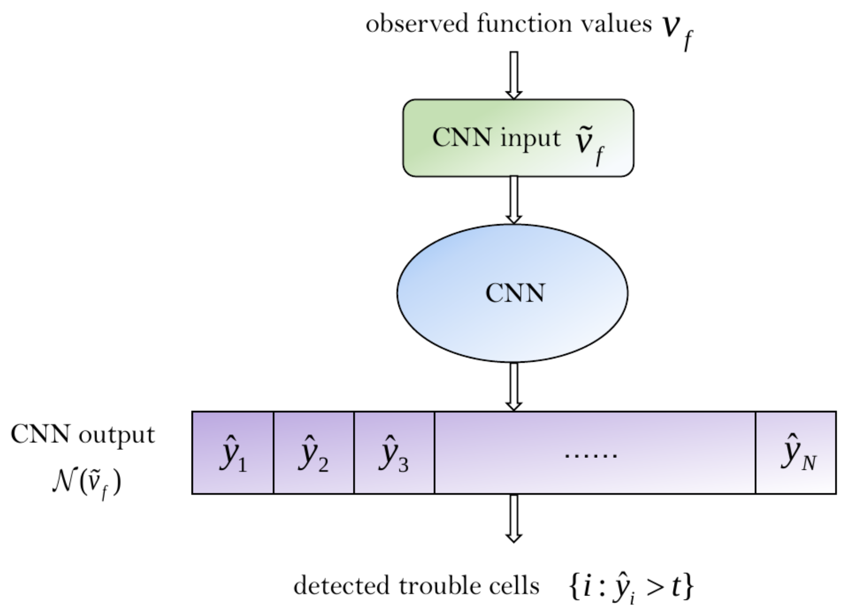

observed function values v_f
CNN input $\tilde{v}_f$
CNN
CNN output $\mathcal{N}(\tilde{v}_f)$
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
......
$\hat{y}_N$
detected trouble cells $\{i : \hat{y}_i > t\}$

Figure 4.3: Workflow of One dimensional CNN Detector. Function values are first normalized and then passed through CNN which outputs the labels corresponding to each cell

The architecture of the CNN model used in the implementation is given in the table4.1

One dimensional CNN Architecture				
Layer type	Input Size	Kernel Size	Kernel Number	Output size
Conv1D	201×1	2	24	200×24
Conv1D	200×24	2	24	199×24
Conv1D	199×24	2	24	198×24
Conv1D	198×24	2	24	197×24
Flatten	197×24	NA	NA	4728
Dense	4728	NA	NA	200

Table 4.1: One dimensional CNN model architecture details. The CNN model essentially consists of a few one-dimensional convolutional layers followed by a Fully connected layer

We will describe two methods of construction of training data for the CNN model.The second method is an improvement on the first, as it allows for the accumulation of numerical errors in the functional data. This is more realistic when we obtain functional data from a numerical method.
The first method we'll discuss is the randomized construction of training data. This method does not include any numerical errors in the functional values, meaning there are no errors due to time propagation using a numerical method. This dataset will serve as a benchmark to evaluate the predictive performance of the CNN model. Later, we will introduce numerical errors into the dataset by propagating it through a numerical method.

4.5.1 Randomized construction of Training Data

The first type of dataset generation method that we will discuss has no numerical errors in functional values(i.e. no errors due to time propagation using a numerical method). This dataset will serve as an indicator of how well we are able to predict using the CNN model. Then we will introduce numerical errors in the dataset by propagating through a numerical method.
Construction of Exact (No numerical errors) Dataset

1. Randomly select an integer N_d from the set $0, 1, \ldots, M$. In the experiments, we take $M = 5$, meaning that there can be, at most, 5 discontinuities.

2. Using a uniform distribution on D, generate N_d random numbers. These numbers represent the locations of the discontinuities and partition the domain D into $N_d + 1$ subdomains. In the experiments, we take $D = [-1, 1]$ and partition the domain into 200 elements.

3. Within each subdomain, create a Fourier series of the form

$$\tilde{a}_0 + \sum_{n=1}^{N_F} \left(\tilde{a}_n \cos nx + \tilde{b}_n \sin nx \right),$$

where $\tilde{a}_n, b_n \sim N(0,1)$ are independent and identically distributed Gaussian random variables and $N_F = 15$. This Fourier series represents the functional values within each subdomain.

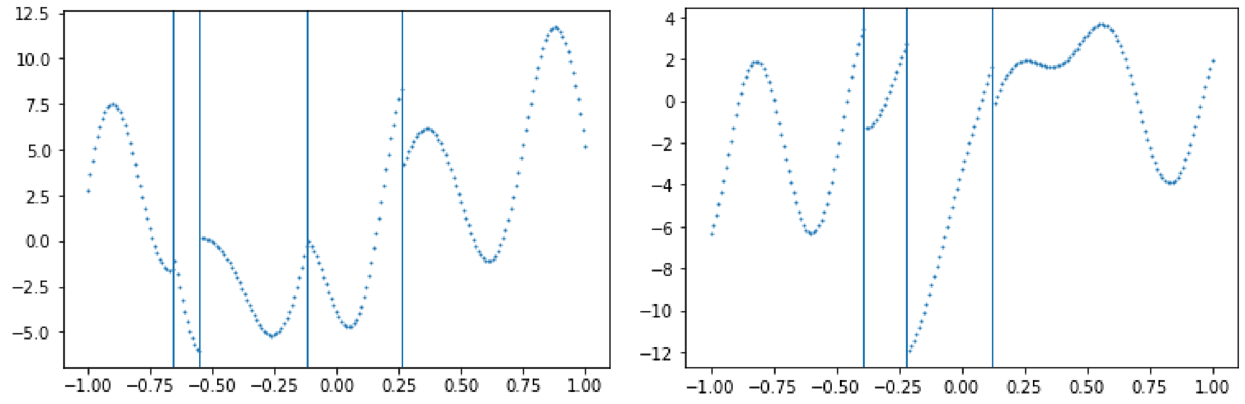


Figure 4.4: Examples of Data generated using the exact method for data generation.

Construction of Numerical Dataset

We consider the advection differential equation

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{4.1}$$

with initial condition as

$u_0 = u(x,0)$

It is well known that for constant $a > 0$, the solution is given by

$$u(x,t) = u_0(x - at) \tag{4.2}$$

We use this result to generate the data for our training. The data generation process is as described below

1. Randomly select an integer N_d from the set $0, 1, \ldots, M$. In the experiments, we take $M = 5$, meaning that there can be, at most, 5 discontinuities.

2. Using a uniform distribution on D, generate N_d random numbers. These numbers represent the locations of the discontinuities and partition the domain D into $N_d + 1$ subdomains. In the experiments, we take $D = [-1, 1]$ and partition the domain into 200 elements.

3. Within each subdomain, create a Fourier series of the form

$$\tilde{a}_0 + \sum_{n=1}^{N_F} \Big(\tilde{a}_n \cos nx + \tilde{b}_n \sin nx\Big),$$

where $\tilde{a}_n, b_n \sim N(0,1)$ are independent and identically distributed Gaussian random variables and $N_F = 15$. This Fourier series represents the functional values within each subdomain.

4. Set this obtained function as the initial condition for 4.1

5. Using a numerical method solve equation (4.1) with appropriate limiting, e.g. min-mod limiting. Choose $a \sim U([0.5, 1])$ and solve for time $t = T$ where $T \sim U([0.1, 0.3])$. Call the obtained solution as $\tilde{u}(x,t)$

6. Label a cell as a troubled cell if $u_0(x - ct)$ is discontinuous within the cell and hence obtain the label vector y.

7. The generated data point is $(\tilde{u}(x,t), y)$

Due to propagation through the numerical method, the dataset henceforth generated is relatively closer to the data that the model will have to predict when coupled with a numerical method(see Figure 4.5). Hence the model trained on this data will be more robust(see figures 4.6, 4.7).
In the experiments, we used the DG method as a numerical solver(See Appendix A and B for more details) and obtained a 9×100 feature vector which was then fed into the CNN, and predictions were made.

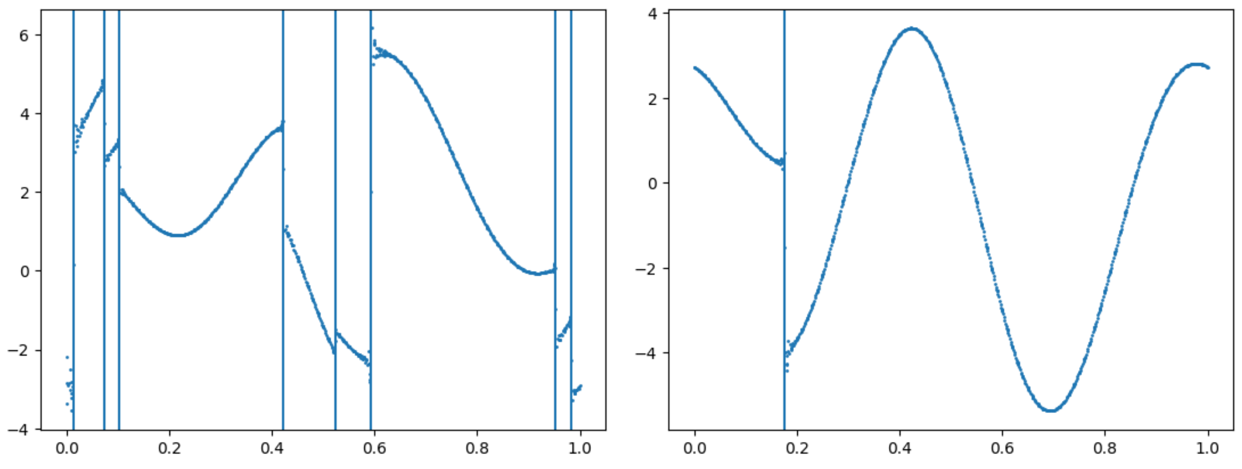


Figure 4.5: Examples of Data generated using the numerical method for data generation. Notice the oscillations in the data near points of discontinuity

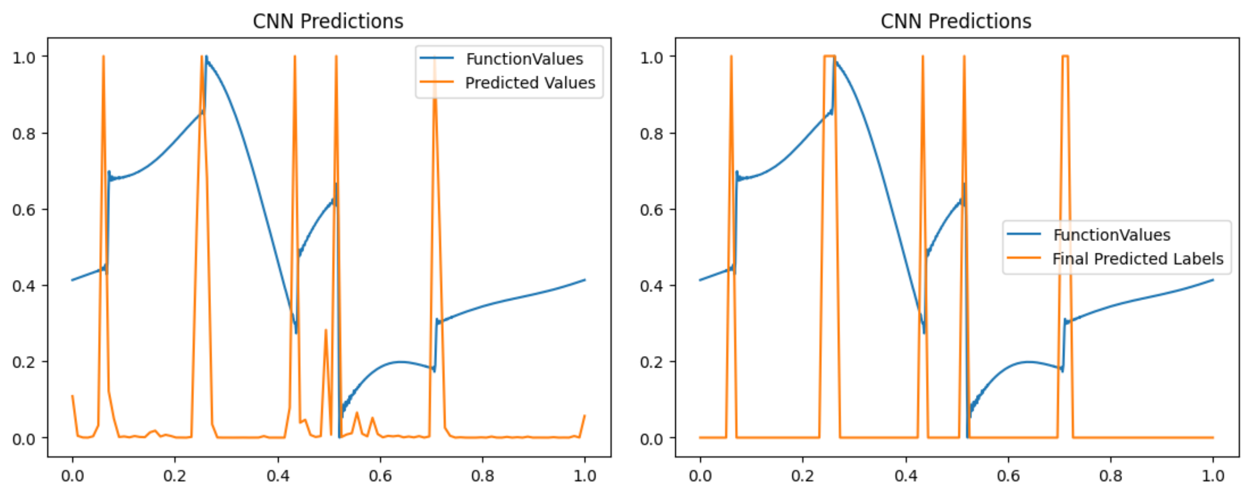


Figure 4.6: CNN model on a test example with actual predicted values(probabilities) plotted on left and predicted labels(thresholding, t = 0.5) plotted on right. Notice the small peaks around points of discontinuity in the predicted values

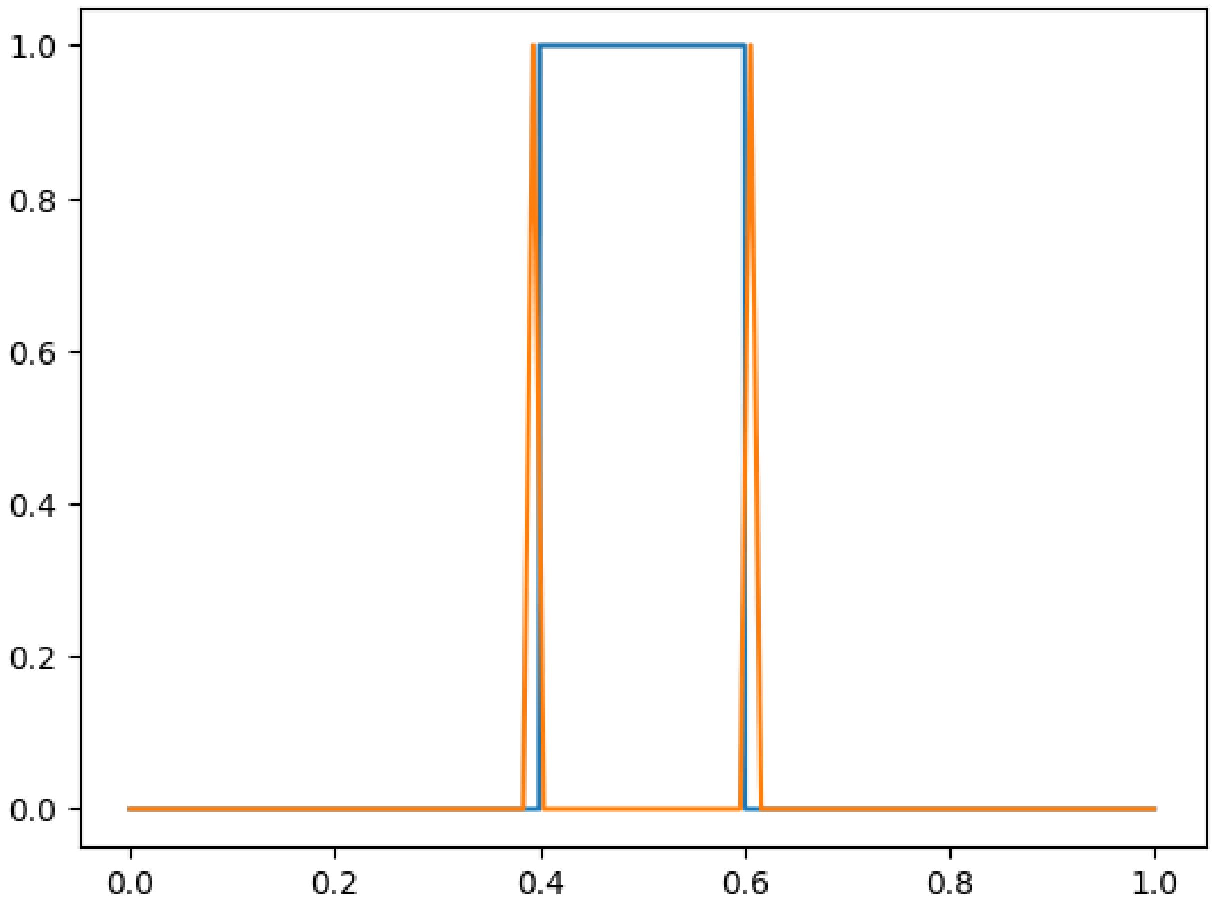


Figure 4.7: CNN Model prediction on exact functional data of the function $f(x)$

$$f(x) = \begin{cases} 1 & 0.4 \leq x \leq 0.6 \\ 0 & \text{Otherwise} \end{cases}$$

4.6 Two Dimensional Detector

Consider the domain $D = [a, b]^2$ which is a square in the two-dimensional plane, with corners at (a, a), (b, a), (a, b), and (b, b). Create a uniform grid of points S within this domain. Each point x_{ij} in the grid is determined by the formula
$x_{ij} = (a + (i-1)\delta, a + (j-1)\delta)$, where $1 \leq i, j \leq N + 1$. This results in a grid of N^2 cells, denoted as C_{ij}.
Define a binary matrix $y = (y_{ij}) \in 0, 1^{N\times N}$, where each entry y_{ij} corresponds to a ground truth value for the cell C_{ij} in the grid.
Let $v_f = \{f(x) : x \in S\}$ be the set of observed function values on S.
We normalize the function values to create $\tilde{v}_f$, which is then used as input to a detector. This detector outputs a binary matrix that predicts the values of y.

We will discuss two types of models for prediction viz One level detection model and Two level detection model. This nomenclature will become clear in the next few sections

Two dimensional CNN Model Architecture: One level Detction				
Layer type	Input Size	Kernel Size	Kernel Number	Output size
Conv2D	$101 \times 101 \times 1$	3	16	$99 \times 99 \times 16$
Conv2D	$99 \times 99 \times 16$	3	16	$97 \times 97 \times 16$
MaxPooling2D	$7 \times 97 \times 16$	2	NA	$48 \times 48 \times 16$
Conv2D	$48 \times 48 \times 16$	3	32	$46 \times 46 \times 32$
Conv2D	$46 \times 46 \times 32$	3	32	$44 \times 44 \times 32$
MaxPooling2D	$44 \times 44 \times 32$	2	NA	$20 \times 20 \times 32$
Conv2D	$22 \times 22 \times 32$	3	64	$20 \times 20 \times 64$
Conv2D	$20 \times 20 \times 64$	3	64	$18 \times 18 \times 64$
MaxPooling2D	$18 \times 18 \times 64$	2	NA	$9 \times 9 \times 64$
Conv2D	$9 \times 9 \times 64$	3	128	$7 \times 7 \times 128$
Conv2D	$7 \times 7 \times 128$	3	128	$5 \times 5 \times 128$
Flatten	$5 \times 5 \times 128$	NA	NA	32000
Dense	32000	NA	NA	10000

Table 4.2: One level Two dimensional CNN model architecture details. The CNN model essentially consists of a few two-dimensional convolutional layers followed by a Fully connected layer

4.6.1 One Level Detection Model

The One Level Detection Model is a type of model where only one CNN is trained and used for prediction. Due to the increase in dimensions from 1D to 2D, the number of parameters in this model dramatically increases (from 949,400 in the 1D model to 32,303,232 in the 2D model). The architecture of the CNN model used in the One Level Detection Model is described in table 4.2

We first perform min-max normalization to the input

$$\tilde{f}(S) = \left\{ \frac{f(x) - \min(f(S))}{\max(f(S)) - \min(f(S))} \right\}$$

Then this normalized data is fed to a CNN, which then predicts the probabilities of discontinuities in each cell.
Finally, we predict a cell as a troubled cell if the probability is greater than threshold t(We take the threshold as 0.1 since we do not want to miss any troubled cell)(see figure

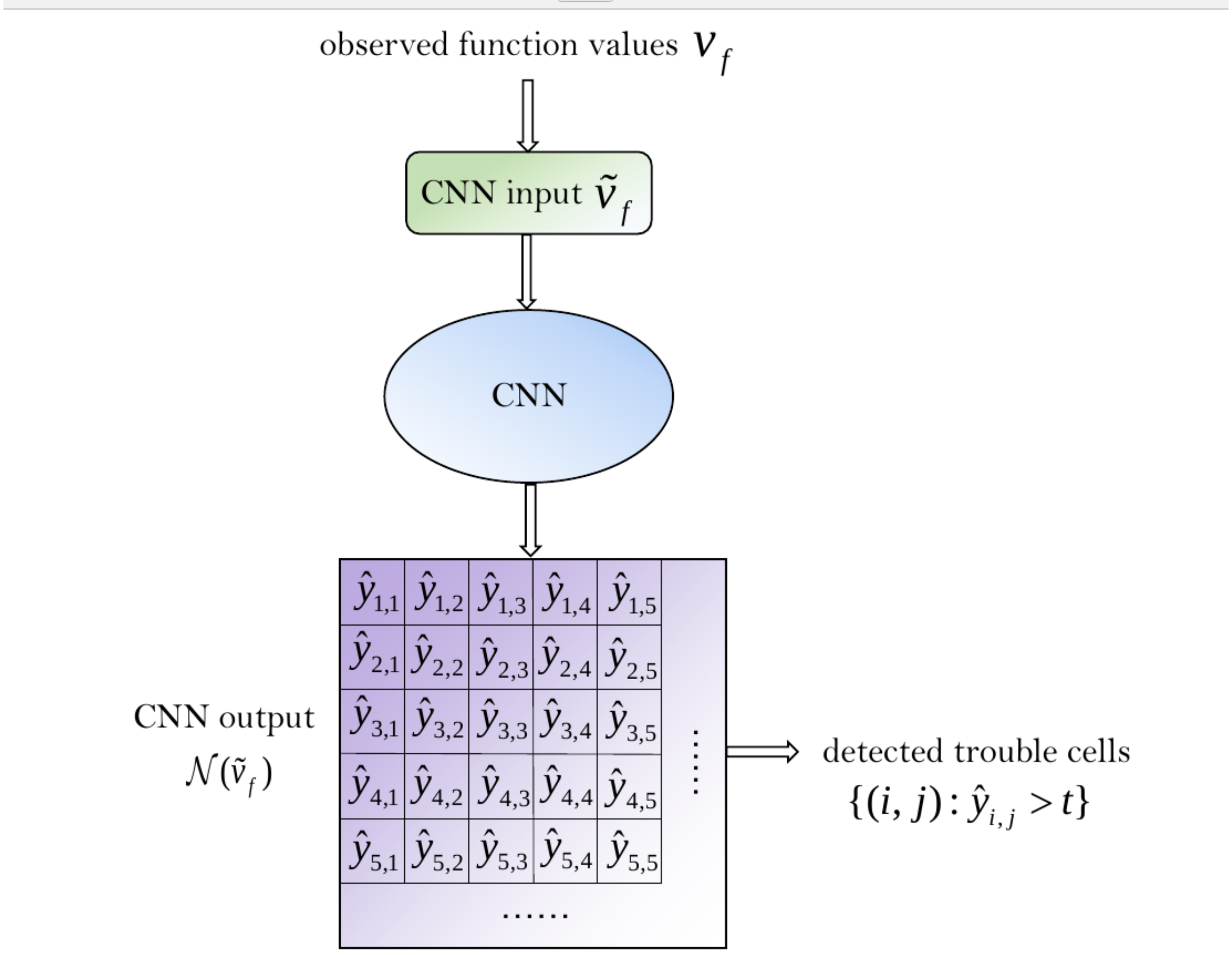

observed function values v_f
CNN input $\tilde{v}_f$
CNN
CNN output $\mathcal{N}(\tilde{v}_f)$
$\hat{y}_{1,1}$ $\hat{y}_{1,2}$ $\hat{y}_{1,3}$ $\hat{y}_{1,4}$ $\hat{y}_{1,5}$
$\hat{y}_{2,1}$ $\hat{y}_{2,2}$ $\hat{y}_{2,3}$ $\hat{y}_{2,4}$ $\hat{y}_{2,5}$
$\hat{y}_{3,1}$ $\hat{y}_{3,2}$ $\hat{y}_{3,3}$ $\hat{y}_{3,4}$ $\hat{y}_{3,5}$
$\hat{y}_{4,1}$ $\hat{y}_{4,2}$ $\hat{y}_{4,3}$ $\hat{y}_{4,4}$ $\hat{y}_{4,5}$
$\hat{y}_{5,1}$ $\hat{y}_{5,2}$ $\hat{y}_{5,3}$ $\hat{y}_{5,4}$ $\hat{y}_{5,5}$
......
detected trouble cells
$\{(i,j) : \hat{y}_{i,j} > t\}$

Figure 4.8: Workflow of one level two dimensional CNN detector. Function values are first normalized(Min-Max Normalization) and then passed through CNN, which outputs the label probabilities corresponding to each cell. Finally, thresholding is done to predict the labels

4.6.2 Two Level Detection Method

The problem with the single-level detection approach is that it experiences a significant surge in the number of parameters when the dimension is incremented by one. This can lead to computational inefficiencies and difficulties in managing the increased complexity. To mitigate this issue, we can employ a two-level detection method. In this approach, we initially perceive the entire grid as a coarse or broad grid. The first level of detection is applied to this coarse grid, with the aim of identifying the cells that are potentially problematic or 'troubled'.
Each cell within this coarse grid contains a finer, more detailed sub-grid. If a cell in the coarse grid was flagged as 'troubled' by the first level detector, a second level detector is then applied to the sub-grid within that particular cell.
This two-level detection method allows for a more efficient and targeted approach to identifying and dealing with troubled cells. It reduces the computational load by focusing on problematic areas rather than applying a high-resolution detection process to the entire grid. This way, the increase in parameters is managed more effectively, making the process more efficient and manageable.

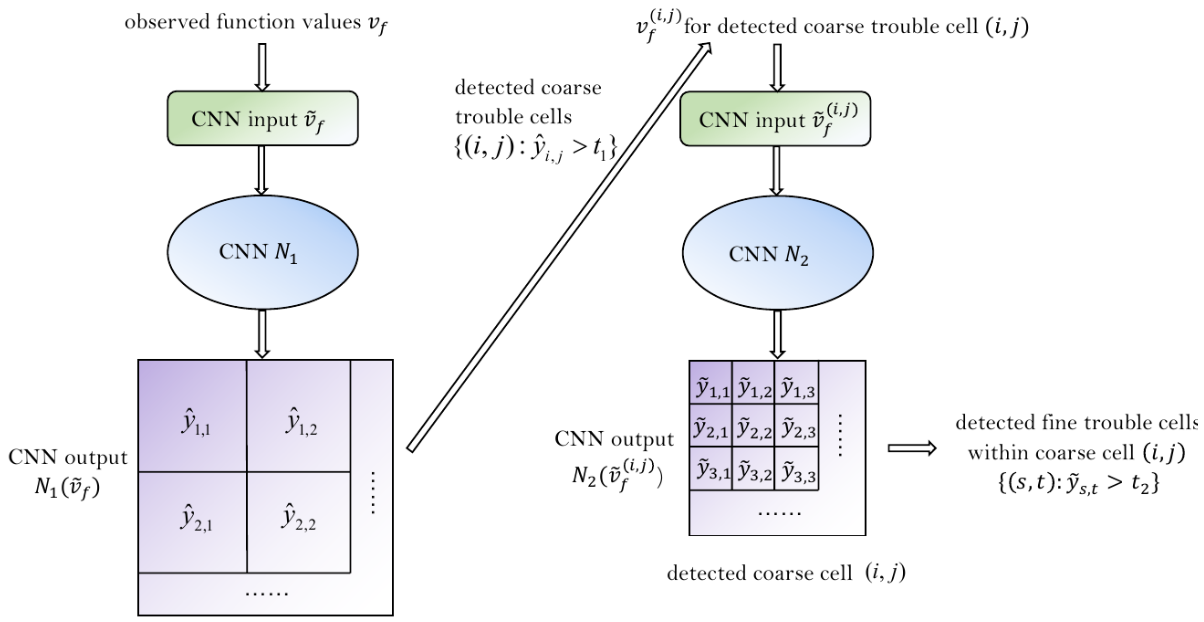

observed function values v_f
CNN input $\tilde{v}_f$
CNN N_1
CNN output $N_1(\tilde{v}_f)$
$\hat{y}_{1,1}$ $\hat{y}_{1,2}$ $\hat{y}_{2,1}$ $\hat{y}_{2,2}$
detected coarse trouble cells $\{(i,j): \hat{y}_{i,j} > t_1\}$
$v_f^{(i,j)}$ for detected coarse trouble cell (i,j)
CNN input $\tilde{v}_f^{(i,j)}$
CNN N_2
CNN output $N_2(\tilde{v}_f^{(i,j)})$
$\tilde{y}_{1,1}$ $\tilde{y}_{1,2}$ $\tilde{y}_{1,3}$ $\tilde{y}_{2,1}$ $\tilde{y}_{2,2}$ $\tilde{y}_{2,3}$ $\tilde{y}_{3,1}$ $\tilde{y}_{3,2}$ $\tilde{y}_{3,3}$
detected coarse cell (i,j)
detected fine trouble cells within coarse cell (i,j) $\{(s,t): \tilde{y}_{s,t} > t_2\}$

Figure 4.9: Workflow diagram of a two-level detector. Inputs are first normalized and then passed to the coarse-level detection model. The detected coarse-level cells are then passed to the fine-level detection model, where they are again normalized and then passed to the fine-level model, which then makes the final predictions

We implement this model on a grid of size 101×101. Each coarse cell contains 10×10 fine cells. Hence there are 10×10 coarse cells in the grid.
We mark a coarse cell as a troubled cell if any of the fine cells within the coarse cell is a

troubled cell.
The second level detector is then applied to coarse cells to do the final prediction. The model architecture dimensions for the coarse and fine models chosen are described as follows

Fine Model Architecture		
Layer type	Output Shape	Param #
Conv2D	(None, 10, 10, 32)	160
Conv2D	(None, 9, 9, 32)	4128
Conv2D	(None, 8, 8, 32)	4128
Conv2D	(None, 46, 46, 32)	4128
Flatten	(None, 2048)	0
Dropout	(None, 2048)	0
Dense	(None, 100)	**204900**

Table 4.3: Fine Model Architecture Details

Coarse Model Architecture		
Layer type	Output Shape	Param #
Conv2D	(None, 98, 98, 32)	544
Conv2D	(None, 48, 48, 32)	4128
Conv2D	(None, 47, 47, 32)	4128
Conv2D	(None, 46, 46, 32)	4128
Dropout	(None, 46, 46, 32)	0
Flatten	(None, 67712)	0
Dense	(None, 100)	**6771300**

Table 4.4: Coarse Model architecture details

4.6.3 Randomized Construction of Training Dataset

We only discuss the construction of the exact dataset. The dataset with numerical errors, analogous to the one-dimensional case, can be generated by using the current generation technique to form the initial condition surface for a two-dimensional advection equation and then labelling the final points of discontinuity accordingly.

1. Domain D is divided into two sub-regions by a *random curve*
2. Inside each sub-region a smooth function is generated given by

$$f_i(x, y) = \sum_{m+n \leq N_p} a^{(i)}_{m,n} P_m(x) P_n(y) \;\; i = 1, 2$$

   where $P'_n s$ are the standard Legendre polynomials and $a_{m,n}$ are randomly sampled from $N(0, 10)$, $N_p = 4$

3. For the random curve serving as an interface between the two sub-regions and is also the location of the discontinuity curve, the following two cases are employed.
   - Line Cut: Defined by the random straight line

$$\cos\theta(x - x_0) + \sin\theta(y - y_0) = 0$$

     where $\theta \sim U(0, 2\pi), (x_0, y_0) \sim U(D)$
   - Circular Cut: Defined as

$$(x - x_0)^2 + (y - y_0)^2 = r$$

     where $r \sim U(0, 3), (x_0, y_0) \sim U(D)$

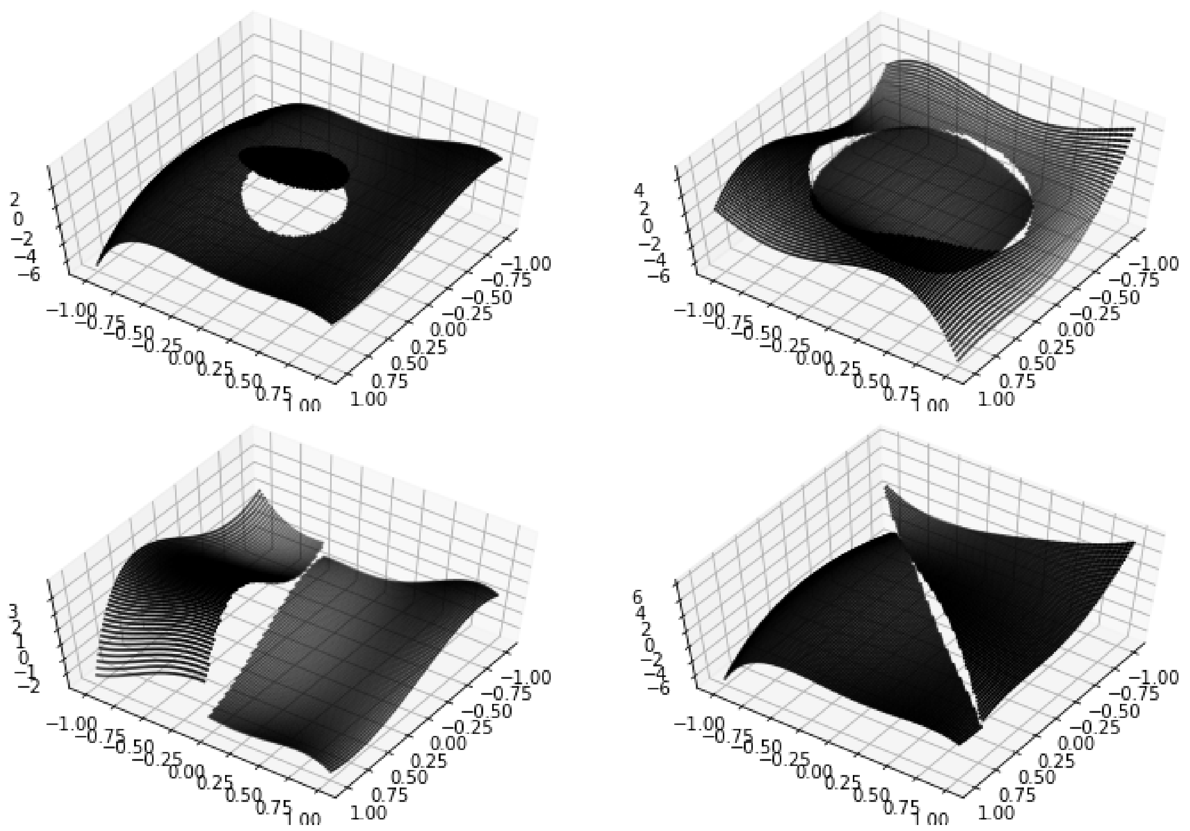


Figure 4.10: Examples of two-dimensional data generated using exact method.

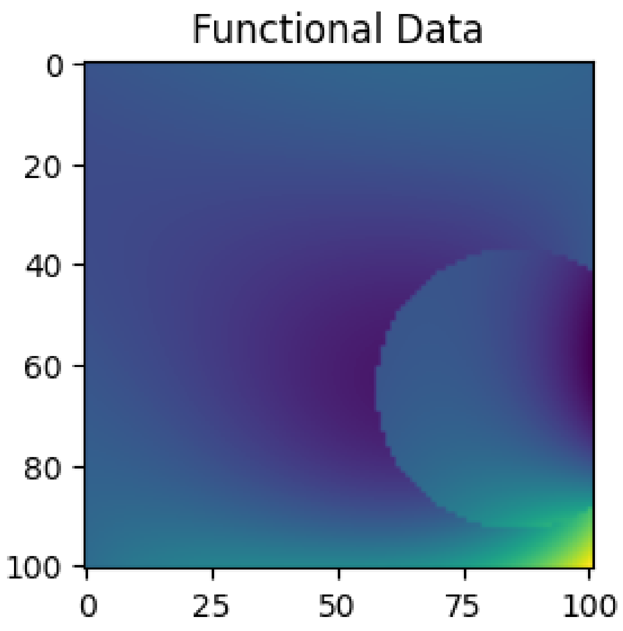


Figure 4.11: Heatmap plot of a data point using the above method of generation

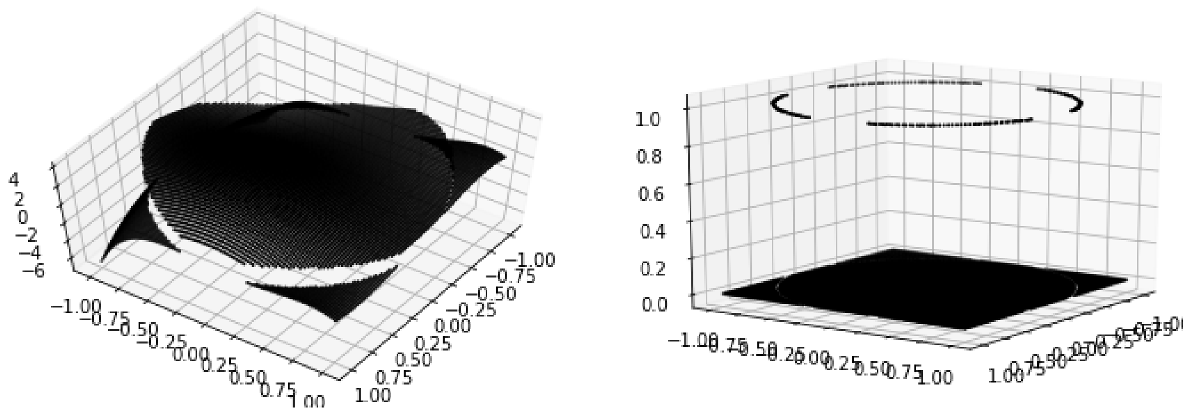
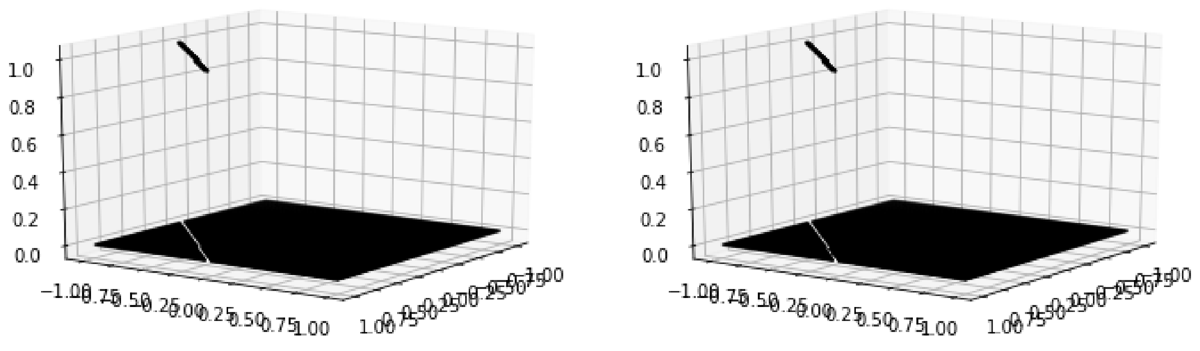


Figure 4.12: One Level CNN model prediction on a test example, with functional data(on left) and label prediction(on right)



(a) Ground Truth (b) Predicted Labels

Figure 4.13: Two level Model on a test example with ground truth labels(on left) and predicted labels(on right)

4.7 Results

4.7.1 Comparison of Model accuracies

1D Models		
Model Name	Number of parameters	Accuracy
CNN Model	949,400	**99.92%**
Polynomial Annihilation($m = 5$)	0	97.29%
Polynomial Annihilation($m = 7$)	0	96.95%
Polynomial + MinMod($m = 5$)	0	**99.01%**
Polynomial + MinMod($m = 7$)	0	98.94%

Table 4.5: One dimensional CNN model and PA method accuracies on synthetic exact functional data

1D Models	
Model Name	Accuracy
CNN Model	**95.78%**
Polynomial Annihilation	96.31%
Polynomial + MinMod	**95.34%**

Table 4.6: One dimensional CNN model and PA method accuracies on Numerical data

2D Models		
Model Name	Number of parameters	Accuracy
1 Level CNN Model	721,111,033	99.45%
2 Level CNN Model	6,784,228 + 213,316 = 6,997,544	99.82%

Table 4.7: Two-Dimensional CNN model accuracies on synthetic exact functional data

The accuracy of 2D polynomial annihilation models was not calculated as they take approximately $1-2$ minutes per example for calculation.

4.7.2 Comparison of Model Recall and Precision

1D Models		
Model Name	Recall	Precision
CNN Model	**92.94%**	**98.91%**
Polynomial Annihilation($m = 5$)	85.63%	24.73%
Polynomial Annihilation($m = 7$)	86.59%	22.03 %
Polynomial + MinMod($m = 5$)	84.60%	50.31 %
Polynomial + MinMod($m = 7$)	82.35%	50.14%

Table 4.8: One dimensional CNN model and PA method precision and recall on synthetic exact functional data

1D Models		
Model Name	Recall	Precision
CNN Model	**98.11%**	55.85%
Polynomial Annihilation	42.58%	**76.56%**
Polynomial + MinMod	21.06%	73.93 %

Table 4.9: One dimensional CNN model and PA method precision and recall on Numerical Data

4.8 Comparison of limiting by CNN vs standard minmod limiting

We consider the advection equation 5.4 with initial condition as

$$u_0(x) = \begin{cases} 1 & x \in [0.4, 0.6] \\ 0 & x \in [0, 0.4) \cup (0.6, 1] \end{cases}$$

We choose the advection velocity as $a = 1$ and final time $T = 0.5$. The final solution obtained by using minmod(as limitter and TCI) and GNN(as TCI and minmod as limitter) is shown in Figure 4.14 and the graph of the cells predicted during the course of numerical simulation is shown in Figure **??**

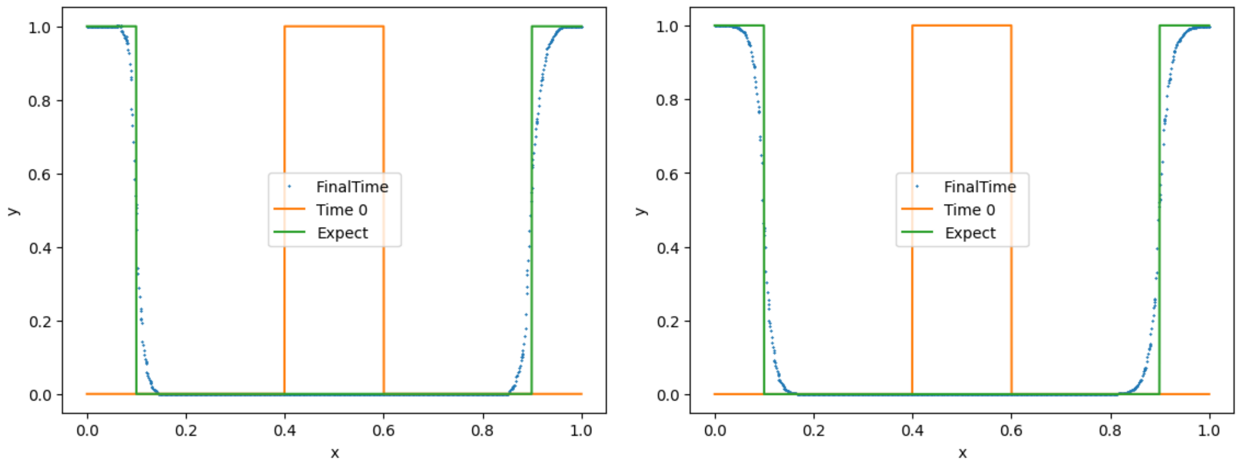


Figure 4.14: Solution obtained by using CNN(left) as TCI vs using minmod(right) as TCI. It can be seen clearly that the minmod over limits the solution and CNN also overlimits the solution, but the extent of overlimitting is greater for minmod.

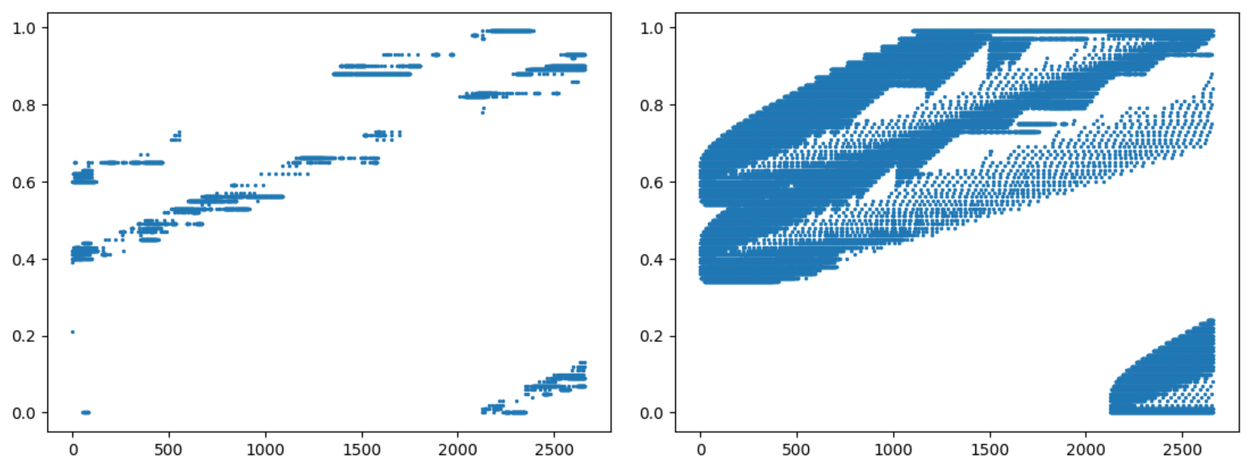


Figure 4.15: Trajectory of the predicted troubled cells CNN(left) vs minmod(right). From the graph it is evident that both CNN and minmod overlimit the solution, but the extent of overlimitting is far greater for minmod case than that of CNN

Chapter 5

Graph Neural Networks

Graph Neural Networks (GNNs) have emerged as a transformative tool in the realm of machine learning and artificial intelligence, specifically engineered to manage and interpret data that is structured in the form of graphs. These networks are particularly adept at handling tasks that involve data with relational or networked characteristics, such as those found in social networks, molecular chemistry, and recommendation systems.

The popularity of GNNs has seen a significant surge in recent years, largely due to their diverse range of applications. For instance, in the pharmaceutical industry, GNNs have been instrumental in the discovery of new drugs. A notable example is the prediction of the antibiotic Halicin [14], which was identified using a GNN-based approach. This demonstrates the potential of GNNs in revolutionizing drug discovery processes by predicting the properties of potential drug candidates. In addition to drug discovery, GNNs have also found applications in areas such as fraud detection and recommendation systems. By analyzing the complex relationships and patterns within data, GNNs can identify unusual patterns that may indicate fraudulent activity. Similarly, in recommendation systems, GNNs can analyze the relationships between users and items to provide personalized recommendations, enhancing the user experience and increasing engagement.

GNNs have also been utilized by tech giants like Google to predict arrival times[4], demonstrating their potential in the field of transportation. By treating transportation networks as graphs, GNNs can optimize routes and predict arrival times more accurately. Moreover, recent research by DeepMind has shown that GNNs can be used to predict protein folding structures with a high degree of accuracy[9]. This breakthrough has significant implications for biological research and could potentially accelerate the development of new treatments and therapies. In the next sections, we will discuss some examples of graph data and some of the various types of graph neural networks, and how graph neural networks work.

5.1 Graph Data

A graph ($G(V, E)$) consists of a set of vertices/nodes (denoted as V) and a set of edges(denoted as E) depicting connections between the vertices. Graph data can be a collection of graphs or a single large graph in which nodes and edges may have features associated with them. The features associated with nodes are known as node features, and the features associated with edges are known as edge features.

5.2 Graph Machine Learning Tasks

Predictions on graph data can be made at various levels. Depending on the level at which classification occurs, graph machine learning tasks can be broadly classified as

1. **Node Classification** Given a graph, predict a property of nodes of the graph, e.g. In a social network graph, predict which of the nodes are fake accounts/bots.

2. **Link prediction** Given a graph, predict whether there are missing links between two nodes, e.g. In a recommendation system, we predict whether a link exists between a node(person) and another node (an item).

3. **Graph Classification** Given graph data, classify the graph into one of the categories, e.g. predicting human toxicity of a drug molecule

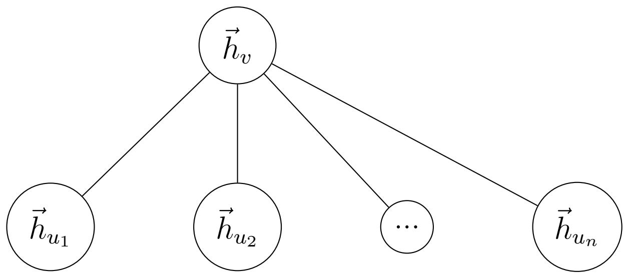
4. **Graph Generation** Given a collection of graphs, generate a graph likely to be among the collection, e.g. Antibiotic Discovery

5. **Graph Evolution** Given the current state of the graph determine the time evolution structure of the graph, e.g. physical simulation in gas dynamics, etc.

The above list is not exhaustive, and there exist many other types of graph machine learning tasks, e.g. clustering etc

5.3 Message Passing Layer

A graph neural network typically consists of two main components: a message-passing layer and an aggregation mechanism. These components allow information to flow between nodes in a graph, enabling effective learning and representation of graph-structured data. Each node receives a message from its neighbours. This message is a function of the attributes of the current node, the neighbouring node, and, optionally, the edge attributes of the edge between the two nodes. Consider a graph $G = (V, E)$. Let us look at a node v and its adjacent nodes $[u_1, u_2, ..., u_n]$.



Here

$$\vec{h}_v : node\ embedding\ vector, v \in V$$

Message from each neighbour u_i of v is calculated as

$$m_v = AGG\left(\{\phi(h_u, h_v), \forall u \in \mathcal{N}(v)\}\right), \tag{5.1}$$

where $\mathcal{N}(v)$ is the neighbourhood of node v, and AGG is an appropriate aggregation operation, e.g. $\min, \max$ sum etc., and ϕ is an appropriate differentiable function, e.g. a multilayer perceptron. This message is then used to update the embeddings of the node

$$\tilde{h}_v = f(m_v, h_v) \tag{5.2}$$

where f is an appropriate differentiable function, e.g a multilayer perceptron. Equations 5.1 and 5.2 together comprise one propagation step. Before making final predictions, we can have multiple propagation steps. Each propagation steps increases the influence of neighbours(neighbours of neighbours and so on) in the final node embeddings. Based on the aggregation operation, we discuss some of the common types of GNN models

5.4 Graph Convolution Network(GCN)

One step propagation in GCN is defined as follows[10]

$$H^{l+1} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^lW^l\right) \tag{5.3}$$

where H^{l+1} are the updated note embeddings,H^l are the previous node embeddings, $\tilde{A}$ is the adjacency matrix of the graph with added self-loops, $\tilde{D}$ is a diagonal matrix containing degrees of each node with the addition of one self-loop, i.e. $\tilde{D}_{ij} = \sum_j \tilde{A}_{ij}$, W^l is a layer-specific trainable matrix and σ is an activation function e.g. ReLU.
In terms of notations described in section 5.3 above, for GCN we have

$$\phi(h_u, h_v) = \frac{W^T h_u}{\sqrt{\tilde{d}_u \tilde{d}_v}}$$

where $\tilde{d}_v = 1 + |\mathcal{N}(v)|$
AGG operation for GCN is simple addition operation and

$$f(m_v, h_v) = \sigma\left(m_v + W^T \frac{h_v}{\tilde{d}_v}\right)$$

Hence equations 5.1 and 5.2 for GCN(for l^{th} propagation step) become

$$m_v^{l+1} = \sum_{u \in \mathcal{N}(v)} \frac{W^{lT} h_u^l}{\sqrt{\tilde{d_v} \tilde{d_u}}}$$

$$h_v^{l+1} = \sigma \left(m_v^{l+1} + \frac{W^{lT} h_v^l}{\tilde{d_v}} \right)$$

5.5 Graph SAGE

Graph SAGE replaces the aggregation operation of GCN with a general aggregation function e.g. mean, max-pool etc. and uses independent weight matrices in m_v and h_v[7]. The equations 5.1 and 5.2 hence become

$$m_v^{l+1} = AGG_l \left(\{ W^{lT} h_u, u \in \mathcal{N}(v) \} \right)$$
$$h_v^{l+1} = \sigma \left(m_v^{l+1} + B^{lT} h_v \right)$$

where W^l and B^l are independent layer specific learnable weight matrices.

5.6 Graph Attention Networks(GATs)

Graph Attention Networks (GATs) are a type of neural network designed to handle graph-structured data. They were introduced by Petar Veličković et al. in their 2018 paper "Graph Attention Networks"[15]. GATs are particularly useful for tasks where data is represented as a graph, such as social network analysis, biological network analysis, and citation network analysis.
The key innovation of GATs is the use of an attention mechanism that allows the model to focus on different parts of the graph when making predictions. This is in contrast to other graph neural networks, which typically use a fixed aggregation function to combine information from neighbouring nodes.
In a GAT, each node in the graph is associated with an embedding, which is a vector representation of the node's features. The model learns to update these embeddings based on the features of neighbouring nodes. The attention mechanism determines how much weight to give to each neighbour's features when updating a node's embedding.
The attention mechanism in a GAT is a learnable function, which means it can adapt to the data during training. This allows the model to learn complex patterns in the graph structure. The attention weights are computed using a softmax function, which ensures they sum to one and can be interpreted as probabilities.
One propagation of a GAT involves the following steps:

1. **Message Generation:** This step involves using the node embeddings of the current node and its neighbour to generate a message to the current node. $m_{vu} = W^T h_u$
2. **Attention Mechanism** $e_{vu} = a(m_v, m_u)$ where a is a learnable functoin e.g. an MLP. The attention mechanism computes an attention score for each neighbour, indicating how important that neighbour's features are for updating the current node's embedding.
3. **Attention Weights** The attention scores are normalized using a softmax function to produce attention weights
$$\alpha_{vu} = \frac{\exp e_{vu}}{\sum\limits_{w \in \mathcal{N}(v) \cup \{v\}} \exp e_{vw}}$$
4. **Aggregate and Update Node embeddings** The node embeddings are updated by aggregating the messages from neighbours, weighted by the attention weights.
$$\tilde{h}_v = \sigma\left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{vu} m_v\right)$$

To make the model more robust and capture different types of dependencies in the data, GATs often use multiple attention heads. Each attention head learns a different attention function, and their outputs are concatenated to form the final node embeddings. GATs have been shown to achieve state-of-the-art performance on a variety of tasks involving graph-structured data. They are also highly flexible and can be easily integrated into other neural network architectures.

5.7 GNN as Troubled Cell Indicator

Graph neural networks perform particularly well in situations in which the categorization of nodes depends on the classification of nodes that are geographically adjacent to the node under consideration (i.e., neighbours, neighbours of neighbours, etc.). Many troubled cell indicators, such as the KXCRF[11] and the Fu and Shu[6] limiter, among others, employ the simplicies (cells) that share a face with a simplex (cell), also known as neighbouring cells, to determine whether or not the cell in question is a troubled cell. In light of this, a natural extension is that "neighbours of neighbours" and so on should also be included in the prediction process, in addition to a cell's immediate neighbours. This suggests that Graph Neural Networks might potentially be used as Troubled cell indicators.
The job of classifying cells in a mesh may be modelled as the classification of nodes in a network. Each individual cell that makes up the mesh is modelled as a node in the network, and an edge is inserted between any two nodes in which the cells that make up those nodes share a face. The features that are assigned to each node include the cell's

size, factors portraying the cell's shape (such as transformation factors between the cell and the standard element), and solution attributes (such as the coefficients of the polynomial solution in the cell, value at the corners, etc.). It is possible to train a graph neural network to categorise the nodes based on the features of the nodes themselves. In the following sections, we will provide the specifics of the model that was trained and also outline the strategy that was utilised to generate the training data. Only the one-dimensional detector will be covered in this article. We were unable to gather adequate data for the training of a two-dimensional model since we did not have access to sufficient computer resources. We will provide a brief overview of the architecture of the 2D model that may be constructed as well as how the training data for the model can be obtained in Appendix B.

5.7.1 GNN model architecture

The workflow of a one-dimensional GNN detector is similar to that of CNN. We first normalize the data(min-max Normalization). This normalized data is then fed into a GNN detector which outputs the probabilities corresponding to each class. Then we predict class label 1 if the probability is greater than threshold t(We take $t = 0.1$ in our model). The GNN model architecture details are given in Table 5.1

One dimensional GNN Model				
Layer type	Input size	Attention Heads	Dropout rate	Output size
GATConv	9	8	0.2	32×8
GATConv	32×8	1	0	1

Table 5.1: One-dimensional GNN detector architecture. The GNN model essentially consists of two GAT layers. The last layer of GAT is applied with the sigmoid activation function to output the class probabilities, and the output of the first convolution layer is applied with Exponential Linear unit

In one of the layers of the GNN model, we use Exponential Linear Unit(ELU) as our activation function

$$\mathrm{ELU}(x) = \begin{cases} x & x \geq 0 \\ \alpha(\exp x - 1) & x < 0 \end{cases}$$

We take $\alpha = 1$ in our implementation.

5.7.2 One-dimensional Data Generation for GNN

In the following paragraphs, we will talk about the procedure of generating training data for the GNN model. The procedure of training and generating new data is analogous to that of the CNN model. On the other hand, for the GNN example, we add a randomization procedure to the start and end points of the meshes, which results in the mesh size (the number of cells in the mesh) being variable rather than constant. During the implementation process (Appendix C), we take precautions to limit the generation of an excessive number of data points that are continuous. Consider the advection equation

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{5.4}$$

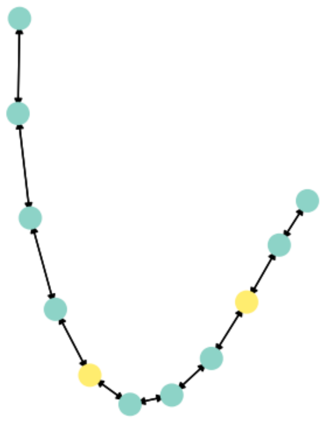


Figure 5.1: Example of a one-dimensional graph data point. The yellow nodes indicate cells with discontinuities, and the blue nodes indicate cells without discontinuities

with initial condition as $u_0 = u(x, 0)$

1. Randomly select an integer N_d from the set $\{0, 1, ..., M\}$. In the experiments, we take $M = 5$(i.e. at most 5 discontinuities).

2. Using uniform distribution on D, generate N_d random numbers as the locations of the discontinuities. This partitions D into $N_d + 1$ subdomains. In the experiments, we take $D = [0, 1]$ and partition the domain in 100 elements.

3. Divide the domain into $N_d + 1$ sub intervals and create function

   $$\tilde{a}_0 + \sum_{n=1}^{N_F}(\tilde{a}_n \cos nx + \tilde{b}_n \sin nx)$$

   as described in the above data generation process for 1D in each of the sub-intervals. This will form the piece-wise definition of the initial condition $u_0(x)$

4. Using a numerical method solve equation (5.4) with appropriate limiting, e.g min-mod limiting. Choose $a \sim U([0.5, 1])$ and solve for time $t = T$ where $T \sim U([0.1, 0.3])$. Call the obtained solution as $\tilde{u}(x, t)$

5. Label a cell as a troubled cell if $u_0(x - ct)$ is discontinuous within the cell and hence obtain the label vector y.

6. The generated data point is $(\tilde{u}(x, t), y)$

7. Randomly select a continuous portion $u[x_l, x_r]$ and the corresponding label from this generated example. (Make sure that not too many cases in which no discontinuity is present are formed).5.1

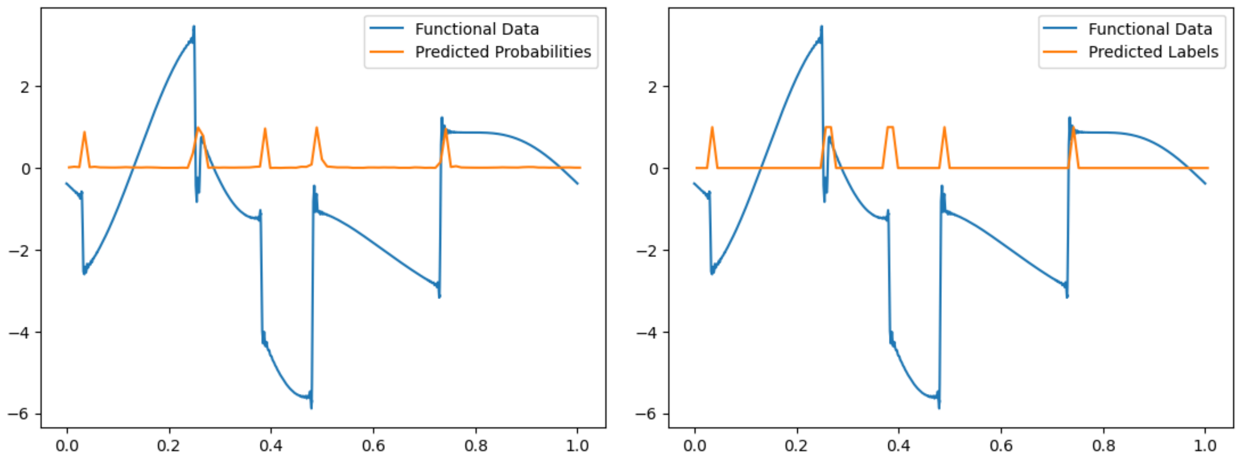


Figure 5.2: GNN model predictions on a test example. Predicted probabilities are plotted on the left, and predicted labels are plotted on the right

5.8 Comparison of Model Accuracy, Precision and Recall

One dimensional Models			
Model Name	Accuracy	Recall	Precision
CNN Model	95.78%	98.11%	55.85%
GNN Model(Randomized mesh Length)	97.35%	90.0%	74.31%
GNN Modle(Fixed Mesh Length)	97.97%	90.27%	75.99%

Table 5.2: CNN vs GNN model on numerical data. Note that the GNN model lags behind the CNN model in the recall metric. The GNN model however has a relatively difficult task on which prediction is done compared that to of a CNN model. The GNN model predicts on variable mesh lengths(number of cells) compared to that of the CNN model on a fixed mesh length. This adaptability of the GNN model to meshes of various sizes and lengths makes it an attractive option as a general TCI

5.9 Comparison of limiting by GNN vs standard minmod limiting

We consider the advection equation 5.4 with the initial condition as

$$u_0(x) = \begin{cases} 1 & x \in [0.4, 0.6] \\ 0 & x \in [0, 0.4) \cup (0.6, 1] \end{cases}$$

We choose the advection velocity as $a = 1$ and final time $T = 0.5$. The final solution obtained by using minmod(as limitter and TCI) and GNN(as TCI and minmod as limitter) is shown in Figure 5.3 and the graph of the cells predicted during the course of numerical simulation is shown in Figure 5.4

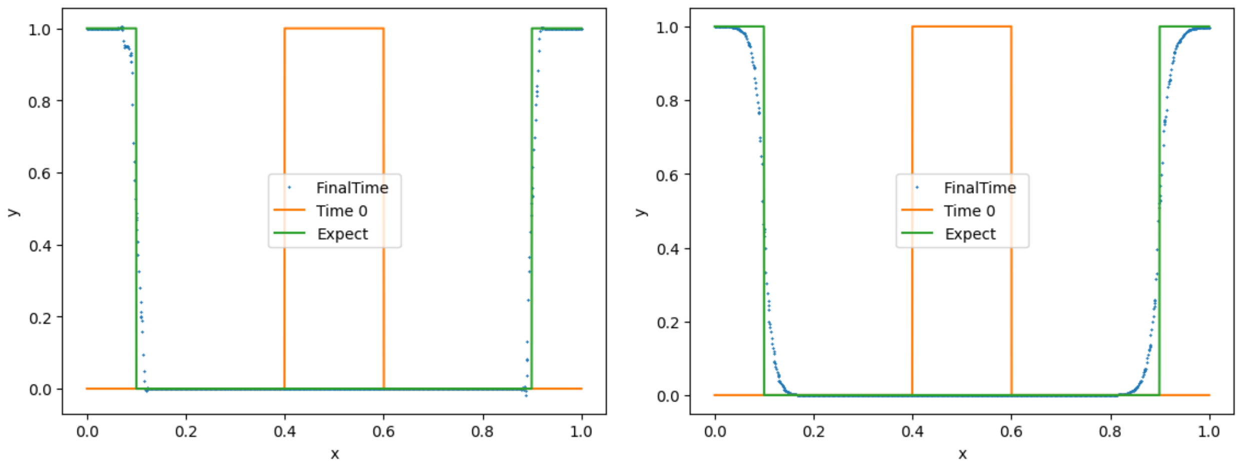


Figure 5.3: Solution obtained by using GNN(left) as TCI vs using minmod(right) as TCI. It can be seen clearly that the minmod over limits the solution, whereas GNN does not.

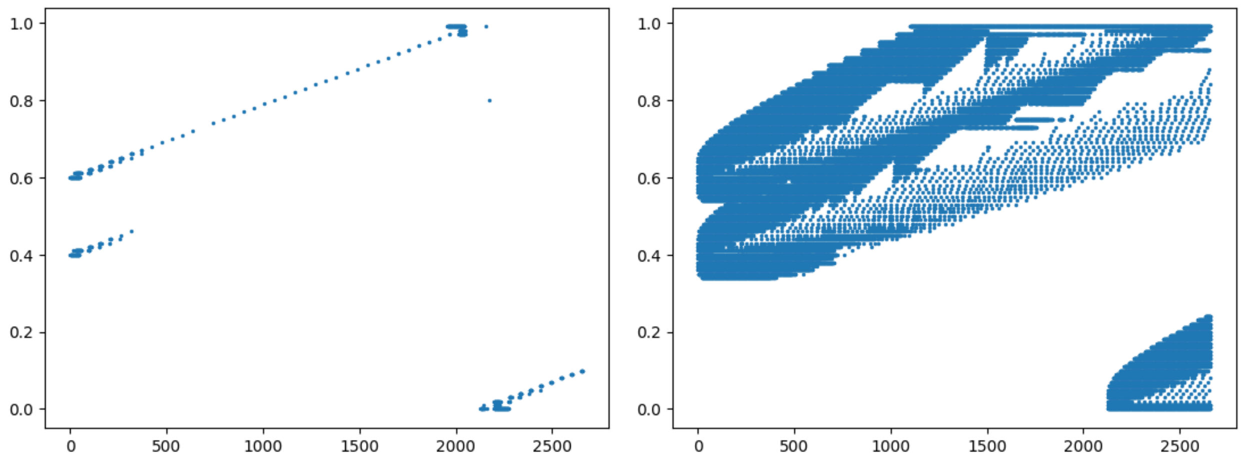


Figure 5.4: Trajectory of the predicted troubled cells GNN(left) vs minmod(right). From the graph it is evident that minmod over limits the solution, whereas the GNN model does not.

In this thesis, we have implemented the GNN model only for the uniform meshes in one dimension and have randomized the location of the start and end points of the meshes only. The GNN model however can be potentially trained for non-uniform meshes also. We suggest the introduction of new node features(e.g. length of the cell etc) for efficient training of the network in those cases. Also, the GNN model can be implemented for uniform meshes but with different refinements. In those cases also new node features must be introduced.

Chapter 6

Conclusion and Future works

The correctness of the Polynomial Annihilation (PA) approach has been established in circumstances in which functional data is accurate. It is remarkable how well it can suit a large variety of data while maintaining its accuracy. Despite this, the performance of the approach suffers a severe setback whenever there are numerical mistakes present in the functional data. This constraint may provide substantial issues in systems that are used in the real world, where data is frequently inaccurate. Additionally, the computing speed of the PA technique is another area of concern because it has a tendency to be slow. This can be a significant disadvantage in the context of applications that are time-sensitive. In addition, the method may be considerably difficult to adopt for procedures that do not provide functional data, such as DG methods. On the other hand, the Convolutional Neural Network (CNN) model offers an option that is both reliable and effective. It is particularly well-known for both its speed and its precision, which makes it a formidable rival for the processing of numerical and accurate data. The CNN model, on the other hand, comes with its share of complications and difficulties. Because of the complexity of the model, which is exemplified by a large number of parameters, it may be challenging to manage and optimise. In addition, it can only be applied to structured meshes, which severely limits its utility in circumstances that call for the processing of unstructured meshes.

The GNN model that is developed in this thesis offers a potentially useful solution in light of the restrictions discussed above. It is fast and accurate, just like the CNN model; but, in contrast to the CNN model, it runs with a substantially fewer amount of parameters to work with. This makes the model easier to work with and less likely to overfit.

In addition to this, the novel model has a great degree of flexibility because it is able to function successfully with meshes of variable sizes. Because of this property, its usefulness may be greatly extended to a far wider range of situations. The fact that the model may work with meshes of any size is another feature that contributes to its adaptability as well as its potential for General TCI.

In terms of accuracy, and precision, the GNN model performs much better than both the CNN model and the PA approach and in terms of recall (Most Important), the model performs considerably well(about 90%)

In conclusion, the GNN model is the most promising candidate for the role of General Troubled Cell Indicator, i.e. the same model may be applied to meshes of various sizes and refinements as well as for different PDEs. The only thing that the model is dependent on is the kind of approach that was used to generate the data that was used to train the model.

We were limited in the amount of processing power available, therefore we were unable to conduct in-depth experiments with the GNN model. The results of such tests will be considered for future development. The GNN model may be constructed and trained for two-dimensional meshes to be used in many applications. In the appendix C we have provided a brief description of how the model can be extended to the two-dimensional case and how the training data can be created for it. The one-dimensional model may also be applied to non-uniform meshes if desired for the application. AppendixC contains a discussion of the approach for the production of unstructured mesh, as well as implementation and training for the GNN model for such circumstances.

Appendix A

Mathematical Fundamentals and Some more limitters

A.1 Runge Kutta Discontinuous Galerkin Scheme

Consider the problem

$$\begin{cases} \partial_t u + \partial_x f(u) = 0, (x,t) \in \Omega \times [0,\infty] \\ u(x,0) = u_0(x) \\ u_{\partial\Omega} = g \end{cases} \tag{A.1}$$

Let $\Omega \in \mathbb{R}$ be a regular domain discretized by N elements $K_p = \left[x_{p-1/2}, x_{p+1/2}\right]$ for $p = 1, 2, \ldots, N$.
Consider the local space $\mathcal{V}$ given by the set $\{\phi_i\}_{i=0}^{n}$ of one-dimensional Legendre polynomials of degree at most n in x.
The numerical solution for any element K is written as

$$u^K(x,t) = \sum_{i=0}^{n} \hat{u}_i^K(t)\phi_i(x)$$

where the coefficient $u_i^K(t)$ is chosen to minimize L^2 error between the approximation and the exact solution, i.e. an L^2 projection
The scheme then reads as

$$\frac{d\hat{u}_i^K}{dt} + [\hat{f}(u^K(x,t))\phi_i(x)]_{x_{p-1/2}}^{x_{p+1/2}} - \int_K f(u^K(x,t))\partial_x\phi_i(x)dx = 0, i = 0, 1, 2, \ldots, n$$

After this, discretization in space is obtained by using Strong Stability preserving Runge-Kutta method.
To avoid the problem of getting oscillatory solutions, due to strong discontinuities, the method is coupled with a slope limiter.

$$\tilde{u}_1^K = \frac{1}{\sqrt{3}}\text{minmod}\left(\sqrt{3}\hat{u}_1^K, \frac{1}{2}(\hat{u}_0^K - \hat{u}_0^{K_l}), \frac{1}{2}(\hat{u}_0^{K_r} - \hat{u}_0^K)\right)$$

where K_l, K_r are the left and the right cells corresponding to a given cell and

$$\text{minmod}(a,b,c) = \begin{cases} s\min(|a|,|b|,|c|), s = sgn(a) = sgn(b) = sgn(c) \\ 0, \text{ Otherwise} \end{cases}$$

The numerical solution then becomes

$$\tilde{u}^K = \hat{u}_0^K + \tilde{u}_1^K \phi_1^K$$

if the limited weights are not the same as the unlimited weights(i.e. $u_1^K = \tilde{u}_1^K$), otherwise, the solution is taken as the unlimited one.
The codes implemented for the DG method of advection equation in the project have been developed from Matlab codes of [8] and translated to Python

A.2 Finite Difference Methods

In finite difference methods, we approximate derivatives as differences in function values. Consider the differential equation

$$u_t = u_{xx} \tag{A.2}$$

with boundary condition as $u(0,t) = 0 = u(1,t)$ and initial condition as $u(x,0) = g(x)$
Approximating $u_t(x,t) = \dfrac{u(x,t+\Delta t) - u(x,t)}{\Delta t}$ and
$u_{xx}(x,t) = \dfrac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}$ we can write the finite difference scheme as

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^{n-1}}{h^2} \tag{A.3}$$

where $u_j^n = u(j\Delta x, n\Delta t), k = \Delta t, h = \Delta x$
Defining $\mathbf{u}^n = [u_1, u_2, u_3, \ldots, u_{N-1}]^T$, the above system of equations can be written as $A^{(n)}\mathbf{u}^n = \mathbf{u}^{n+1}$. This method of solving PDEs is known as the finite difference method

A.3 Finite Volume Methods

We convert the PDE into algebraic equations of cell averages over a finite volume in finite volume methods. An important property of Finite Volume methods is that they satisfy the conservation of quantities like mass, energy etc. This makes finite volume methods highly attractive for Computational Fluid dynamics. Consider the advection equation in one dimension

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = 0 \tag{A.4}$$

Define the cell average for the cell i in the mesh generated as

$$\overline{u}_i(t) = \frac{1}{x_{i+1/2} - x_{i-1/2}} \int_{x_{i-1/2}}^{x_{i+1/2}} u(x,t)dx \tag{A.5}$$

Integrating A.4, we get

$$u(x,t_2) = u(x,t_1) - \int_{t_1}^{t_2} f_x dt$$

Integrating w.r.t x within the limits $x_{i-1/2}, x_{i+1/2}$, and dividing by $\Delta x_i = x_{i+1/2} - x_{i-1/2}$ and using A.5 and using divergence theorem, we get

$$\overline{u}_i(t_2) = \overline{u}_i(t_1) - \frac{1}{\Delta x_i}\left(\int_{t_1}^{t_2} f_{i+1/2}dt - \int_{t_1}^{t_2} f_{i+1/2}dt\right)$$

where $f_{i\pm1/2} = f(x_{i\pm1/2}, t)$
Hence, we can derive a semi-discrete scheme that is

$$\frac{\partial \overline{u}_i}{\partial t} + \frac{1}{\Delta x_i}\left[f_{i+1/2} - f_{i-1/2}\right] = 0$$

This is an example of a finite volume scheme for the advection equation

A.4 KXCRF Limitter

We divide the boundary of each simplex $\partial\Delta_j$ into two regions viz $\partial\Delta_j^+$ and $\partial\Delta_j^-$ where respectively $\vec{v}\cdot\vec{n} > 0$ and $\vec{v}\cdot\vec{n} < 0$
For smooth regions, solutions of polynomials of order p satisfy

$$\frac{1}{|\partial\Delta_j^+|}\int_{\Delta_j^+}(Q_j - q)ds = O(h^{2p+1})$$

where Q_j is the approximate solution and q is the exact solution. Consider the integral

$$I_j = \int_{\Delta_j^-}(Q_j - Q_{nbj})ds$$

We know that $\Delta_j^- = \Delta_{nbj}^+$ Hence, we have

$$I_j = \int_{\Delta_j^-}(Q_j - q)ds + \int_{\Delta_{nbj}^+}(q - Q_{nbj})ds$$

The second integral is $O(h^{2p+1})$ while the first is $O(h^{p+2})$ in smooth regions
Hence in smooth regions $I_j = O(h^{p+2})$. If the solution is discontinuous, then either one of the two integrals is $O(1)$ and hence I_j is $O(1)$

$$I_j = \begin{cases} O(h^{p+2}), & \text{if q is smooth} \\ O(1), & \text{if q is discontinuous} \end{cases}$$

Hence a discontinuity identifier can be chosen as [11]

$$\mathcal{I}_j = \frac{|\int_{\partial\Delta_j^-(Q_j - Q_{nbj})ds}|}{h^{(p+1)/2}|\partial\Delta_j^-|\|Q_j\|}$$

The discontinuity identification scheme is hence taken as

$$\begin{cases} \mathcal{I}_j > 1, \text{ q is discontinuous} \\ \mathcal{I}_j < 1, \text{ q is continuous} \end{cases}$$

A.5 Fu and Shu Limitter

For the sake of simplicity, we relabel the indicator stencil as $S = \{\Delta_0, \Delta_1, \Delta_2, \Delta_3\}$ (see Figure A.1)
We define two types of cell averages as $\bar{\bar{p_j}} = \frac{1}{\Delta_0}\int_{\Delta_0} p_j(x,y)dxdy$ and
$\bar{p_j} = \frac{1}{\Delta_j}\int_{\Delta_j} p_j(x,y)dxdy$ Over the smooth solution regions, the quantity $\bar{\bar{p_0}} - \bar{\bar{p_j}}$ represents the same cell average and hence will be of $O(h^{p+1})$
With this observation, we define the quantity.

$$I_{\Delta_0} = \frac{\sum_{j=1}^{3}|\bar{\bar{p_0}} - \bar{\bar{p_j}}|}{\max_{j=0,1,2,3}|\bar{p_j}|}$$

Near the points of discontinuity, the quantity I_{Δ_0} behaves wildly resulting in $O(1)$ value[6].
Hence a limiting procedure is

$$\begin{cases} I_{\Delta_0} > C_k \text{ The solution is discontinuous in this cell} \\ I_{\Delta_0} < C_k \text{ The solution is continuous in this cell} \end{cases}$$

where C_k is a threshold constant depending only on the degree k of the piecewise polynomial approximation solution

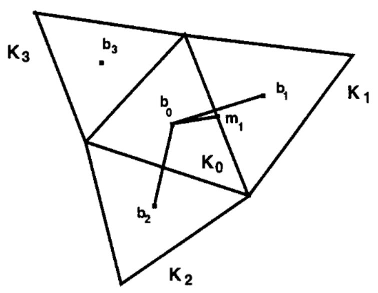


Figure A.1: Triangular Stencil

A.6 ANN as Limiter

Both "KXRCF" and "Fu and shu" limiters attempt to detect discontinuity in the cell of interest by using cell averages in neighbouring cells and the cell of interest. We attempt to make use of this observation by training a neural network that predicts whether a cell is a distressed cell based on the cell's size (h), the averages of its neighbouring cells, and the cell of interest.
Given that the task is a classification assignment, we use binary cross-entropy loss to train the model. The details of the training data generation are given in the paper[13].

Appendix B

Proofs

Proof of Theorem 1: If $f \in C^k(I_x), k > 0$, denote $k_m = \min(m, k)$
By Taylor series expansion, we have

$$f(x) = T_{k_m-1}f(x) + R_{k_m-1}f(x) \tag{B.1}$$

where $T_{k_m-1}f$ is Taylor series expansion polynomial of f of degree $k_m - 1$ and $R_{k_m-1}f(x)$ is the corresponding remainder term.
Since $k_m - 1 < m$, by the definition of $c_j(x)$, we have

$$\sum_{x_j \in S_x} c_j(x) T_{k_m-1}f(x_j) = 0 \tag{B.2}$$

Using equation B.2 and B.1, we have

$$L_m f(x) = \frac{1}{q_m(x)} \sum_{x_j \in S_x} c_j(x) \left(T_{k_m-1}f(x_j) + R_{k_m-1}f(x_j)\right) = \frac{1}{q_m(x)} \sum_{x_j \in S_x} c_j(x) R_{k_m-1}f(x_j)$$

Hence, we have

$$|L_m f(x)| = \frac{1}{|q_m(x)|} \sum_{x_j \in S_x} c_j(x)(x_j - x)^{k_m} f^{(k_m)}(\xi_j)/k_m!$$

Since $x_j - x \leq mh$ and $f \in C^k(I_x) \Rightarrow \exists C \in \mathbb{R}^+$ such that $|f^k(t)| \leq C \forall t \in I_x$, we have

$$|L_m f(x)| \leq A h^{k_m} \frac{1}{q_m(x)} \sum_{x_j \in S_x} c_j(x) = O(h^{k_m})$$

Next, consider the case in which $x, \xi \in [x_{j-1}, x_j]$ and $\xi \in J, x_{j-1}, x_j \in S_x$
WLOG assume that ξ is the only discontinuity of f in a neighborhood I_ξ and $S_x \subset I_\xi$
Analogous to the definition of S_x^+, we define $S_x^- := \{i \in \mathbb{N} | x_i \in S_x, x_i < x\} = S_x \backslash S_x^+$
From the above definitions, it is clear that $S_x = S_x^+ \cup S_x^- S$. Hence, we have

$$L_m f(x) = \frac{1}{q_m(x)} \left[\sum_{x_j \in S_x^+} c_j(x) f(x_j) + \sum_{x_j \in S_x^+} c_j(x) f(x_j) \right]$$

From LMVT, we have

$$L_m f(x) = \frac{1}{q_m(x)} \sum_{x_j \in S_x^+} (f(\xi^+) + (x_j - \xi)f'(\zeta_j^+)) + \frac{1}{q_m(x)} \sum_{x_j \in S_x^-} (f(\xi^-) + (x_j - \xi)f'(\zeta_j^-))$$

We know that $\sum_{x_j \in S_x} c_j(x) = 0$. Hence, we have $\sum_{x_j \in S_x^-} c_j(x) = -\sum_{x_j \in S_x^+} c_j(x) = -q_m(x)$
Hence, we have

$$L_m f(x) = (f(\xi^+) - f(\xi^-)) + O(h) = [f](\xi) + O(h)$$

which completes the proof ■
Proof of Theorem 2: Let $f \in C^k(K_{S_X}, k > 0$, defined $k_m = \min(k, m)$
By Taylor series expansion, we can write

$$f = T_{k_m-1} f + R_{k_m-1} f \tag{B.3}$$

where $T_{k_m-1} f$ is Taylor series polynomial expansion of f of degree $k_m - 1$ around x and $R_{k_m-1} f$ is the corresponding residual term.

$$T_{k_m-1} f(y) = \sum_{|\alpha| \leq k_m - 1} (y - x)^\alpha D^\alpha f(x)/\alpha! \tag{B.4}$$

$$R_{k_m-1} f(y) = \sum_{|\alpha| = k_m} (y - x)^\alpha D^\alpha f(\xi) \tag{B.5}$$

By definition of $c_j(x)$, we have $\sum_{x_j} \in S_x c_j(x) T_{k_m-1} f(x_j) = 0$. Hence, we have

$$L_m f(x) = \frac{1}{q_m(x)} \sum_{x_j \in S_x} c_j(x) \sum_{|\alpha| = k_m} (x_j - x)^\alpha D^\alpha f(\xi_j)$$

Hence, we have $L_m f(x) = O(h^{k_m})$ which completes the proof ■

Proof of Theorem 3: For $x \in [a, b]$, WLOG $x_{j-1} \leq x \leq x_j$, $x_{j-1}, x_j \in S$. If there exists $\xi \in [x_{j-1}, x_j]$, then by Theorem 1, $L_m f(x) = [f](\xi) + O(h) \forall m \in M$.
Therefore $MM(L_m f(x)) = [f](\xi) + O(h)$
Now if $J \cap [x_{j-1}, x_j] = \emptyset$, then by definition, $\mathcal{M}_x \geq 1$. Also from the definition of $\mathcal{M}_x, \forall m \in \mathcal{M}$ such that $m \leq \mathcal{M}_x$ and $\#S_x = m + 1$, we have $I_x \cap J = \emptyset$ Theorem by Theorem 1, we have $L_m f(x) = O(h^{\min(k,m)})$.
Hence, we have $MM(L_{\mathcal{M}} f(x)) = O(h^{\min(k, \mathcal{M}_x})$.■

Appendix C

Grid Generation and Implementational Details

Throughout the course of this project, we have worked only on uniform grids. The GNN method described in the thesis can be extended easily to unstructured meshes with just a need to add some more feature information to the node embeddings. We give a basic description of how the code of the project works and a basic reference of individual frameworks used for the implementation. We will provide the code snippets in Python. We also provide some details on how the tasks mentioned in future work can be implemented and executed.

C.1 Polynomial Annihilation Method, One dimensional

Code Snippet C.1 for calculation of $m!$

Listing C.1: Calculates the value of m! for given m

```python
def factorial(m):
    ans = 1
    for i in range(1, m + 1):
        ans *= i
    return ans
```

Code Snippet C.2 for calculation of S_x *matrix. Only calculates the indices of the points in* S_x*, where* $x = \frac{x_{index} + x_{index-1}}{2}$*. In this code, we calculate the indices left and right such that that the term* $\max(index - left + 1, right - index)$ *is minimized and both left and right are within the range* $0 \leq left, right \leq n - 1$

Listing C.2: Returns indices of the points that lie in S_x

```python
def getIndices(self, index, n):
    l = index
    r = index + 1
    id = 1
    while(r - l < self.m + 1):
        if id == 0 and l -1 >=0:
            l -=1
        elif id == 1 and r < n :
            r += 1
            id ^=1
        return list(range(l, r))
```

Code snippet C.3 for calculation of matrix $\boldsymbol{A}$ *in 3.2*

Listing C.3: Calculates the matrix A

```
    def createMatrix(self, indices, data):
        arr = []
        for i in range(self.m + 1):
            temp = []
            for index in indices:
                temp.append(pow(data[index], i))
            arr.append(temp)
        return np.array(arr)
```

Code snippet C.4 for calculation of matrix $\boldsymbol{b}$ *in 3.2*

Listing C.4: Calculates the matrix b

```
    def getLoadMatrix(self):
        arr = np.zeros((self.m + 1, 1))
        arr[self.m] = factorial(self.m)
        return arr
```

Code snippet C.5 for solving the system $\boldsymbol{Ax} = \boldsymbol{b}$ *in 3.2. Uses np.linalg.solve which is unstable. For a better stability np.linalg.lstsq*

Listing C.5: Returns the solution of the equation 3.2

```
    def getConstants(self, indices, data):
        mat = self.createMatrix(indices, data)
        load = self.getLoadMatrix()
        return np.linalg.solve(mat, load)
```

Code snippet C.6 for getting the label corresponding to cell i

Listing C.6: Returns the values of $L_m f$ corresponding to midpoint of cell index

```
def label(self, index,data, fvalues):
        indices = self.getIndices(index, len(data))
        constants = self.getConstants(indices, data)
        normFactor = 0
        sm = 0
        l_index =indices[0]
        for i in indices:
            if i>= index:
                normFactor += constants[i - l_index]
            sm += constants[i - l_index] * fvalues[i]
        sm = sm/normFactor
        return sm
```

C.2 One-dimensional Data generation

Code snippet C.7 One-dimensional configuration class. Sets the values of M, K, N and N_f. The number of cells generated in the mesh is given by $K + 1$ and the polynomial degree in each cell is given by N

Listing C.7: Sets the parameters for data generation

```
class Config:
    M = 10
    K = 99
    N = 8
    NF = 15
```

Code snippet C.8 for the data generation. A description of each method is given in the docstrings of the methods. For the generation of exact functional data, we make use of getFunc function, and for the generation of numerical data, we make use of getDiscontData function. Note that getDiscontData makes use of SolveAdvection subroutine, which will be described in (Insert reference here)

Listing C.8: Discontinuity data generator class

```
import pandas as pd
class Generator:
  @staticmethod
  def getAdvectionSpeed():
    '''
    Returns a number in range(0.1, 1) uniformly at random, serving as
        advection speed
    '''
    return np.random.uniform(0.1, 1)
  @staticmethod
  def getFinalTime():
    '''
    Returns a number in the range(0.1, 1) uniformly at random, serving as the
        final time in the advection equation numerical solution
    '''
    return np.random.uniform(0.1, 0.5)

  @staticmethod
  def numJumps():
    '''
    Return an integer in the range [0, 10] uniformly at random, describing the
        number of discontinuities in the functional data generated
    '''
    return np.random.randint(0, Config.M )
  @staticmethod
  def getDisconts(n):
    '''
    Return "n" points in the range of [0, K + 1] uniformly at random, serving
        as the points at which discontinuity is present
    '''
    return np.unique(np.random.choice(Config.K + 1, n))
  @staticmethod
  def getSinFunc():
    '''
    Returns the function f(x) = \sum_{n = 1}^NF a_n sin(nx) where a_n's are
        chosen uniformly at random in the range (-1, 1)
    '''
    data = [np.random.uniform(-1, 1) for i in range(Config.NF)]
    return lambda x : sum([data[n] * np.sin(n * x) for n in range(Config.NF)])
  @staticmethod
  def getCosFunc():
    '''
    Returns the function f(x) = \sum_{n = 1}^NF a_n cos(nx) where a_n's are
        chosen uniformly at random in the range (-1, 1)
    '''
    data = [np.random.uniform(-1, 1) for i in range(Config.NF)]
    return lambda x : sum([ data[n] * np.cos(n * x) for n in range(Config.NF)])
  @staticmethod
  def getFunc(cosFList, sinFList, labels, x):
    '''
    Returns the final piecewise continuous function. cosFList and sinFlist
        contain the Fourier series cos and sin expansions, respectively, in
        each subdomain.
    '''
    x = np.divmod(x, 1)[1]
    h  = 1/(Config.K + 1)
    lst = []
```

```
      for i in range(len(labels) + 1):
        cosF = cosFList[i]
        sinF = sinFList[i]
        f = lambda x, cosF_ = cosF, sinF_= sinF : cosF_(x) + sinF_(x)
        lst.append(lambda x , fn = f: fn(x))
      if len(labels) == 0:
        return cosFList[0](x) + sinFList[0](x)
      if len(labels) == 1:
        return np.piecewise(x, [x < labels[0] * h, x>= labels[0] * h ], lst)
      return np.piecewise(x, [x < labels[0] * h] + [(x< labels[i] * h )& (x>=
          labels[i - 1] * h) for i in range(1, len(labels))] + [x>= labels[- 1]
          * h ], lst)
    @staticmethod
    def getDiscontData():
      '''
      Returns data with numerical errors present
      '''
      a = Generator.getAdvectionSpeed()
      finalTime = Generator.getFinalTime()
      numJumps = Generator.numJumps()
      labels = Generator.getDisconts(numJumps)
      labels = labels.sort()
      cosFList = [Generator.getCosFunc() for _ in range(len(labels) + 1)]
      sinFList = [Generator.getSinFunc() for _ in range(len(labels) + 1)]
      x = np.linspace(0,1, Config.K + 1)
      f = lambda x: Generator.getFunc(cosFlist, sinFlist, labels, x)
      u = SolveAdvection(f, finalTime, a)
      return u
```

C.3 Two Dimensional Data generation

Code snippet C.9 Configuration class for Two Dimensional case

Listing C.9: 2D configuration class

```
  class Config:
    N = 50
    N_P = 5
    L = 0.5
```

Code sinppet C.10 for two dimensional data generation. Description of each method is *given in the dosctrings of the methods*

Listing C.10: Generator class for two dimensional exact data generatoion

```
import numpy as np
import matplotlib.pyplot as plt
class DataGenerator:
    @staticmethod
    def Legendre(x, N):
      '''
      Eavluates Legendre polynomial of degree N at points in x
      '''
      prev = 0
      cur = 0
      if N == 0:
        return np.ones(x.shape)
      if N == 1:
        return x
      prev = np.ones(x.shape)
      cur = x
      for n in range(1, N):
        temp = 1/(n + 1) *((2*n + 1)*x * cur - n * prev)
        prev = cur
        cur = temp
      return cur
    @staticmethod
    def getLegendList():
        '''
        Returns list of legendre polynomial functions of degree up to N_P
        '''
        return [lambda x, i_ = i : DataGenerator.Legendre(x, i_) for i in
            range(Config.N_P)]

    @staticmethod
    def getFunc():
        '''
        Returns the function f(x, y) = sum_{i = 0}^N_p sum_{j = 1}^N_P
            a_{ij}P_i(x) P_j(y) where a_{ij} are chosen in the range (-1, 1)
            and P_i is legendre polynomial of degree i
        '''
        LegendXList = DataGenerator.getLegendList()
        data = [np.random.uniform(-1, 1) for _ in range(Config.N_P ** 2)]
        return lambda x, y: sum([data[ Config.N_P* i + j] * LegendXList[i](x)
            * LegendXList[j](y) for i in range(Config.N_P) for j in
            range(Config.N_P)])


    @staticmethod
    def getLineDiscontData():
        '''
        Returns data in which discontinuity is present along a line cut
        '''
        theta = np.random.uniform(0, 2 * np.pi)
        x_0, y_0 = np.random.uniform(0, Config.L), np.random.uniform(0,
            Config.L)

        # cos(theta) * (x - x_0) + sin(theta)(y- y_0) = 0
        h = 1/Config.N
        fun1 = DataGenerator.getFunc()
        fun2 = DataGenerator.getFunc()
        x = np.linspace(0, Config.L, Config.N + 1)
        y = np.linspace(0, Config.L, Config.N + 1)
        xx,yy = np.meshgrid(x, y)
        # xx = xx[:-1, :-1]
        # yy = yy[:-1, :-1]
        val1 = fun1(xx, yy).reshape(xx.shape)
        val2 = fun2(xx,yy).reshape(yy.shape)
        val  = np.where((xx - x_0)*np.cos(theta) + (yy -
            y_0)*np.sin(theta)<=0,val1, val2)
        xx = xx[:-1, :-1]
        yy = yy[:-1, :-1]
        calc1 = (xx - x_0)*np.cos(theta) + (yy - y_0)*np.sin(theta)>0
        calc2 = (xx + h - x_0)*np.cos(theta) + (yy - y_0)*np.sin(theta)>0
        calc3 = (xx - x_0)*np.cos(theta) + (yy + h - y_0)*np.sin(theta)>0
        calc4 = (xx + h - x_0)*np.cos(theta) + (yy + h - y_0)*np.sin(theta)>0
        return val, (calc1 | calc2 | calc3 |calc4) & (np.logical_not(calc1 &
            calc2 & calc3 & calc4))


    @staticmethod
    def getCircDiscontData():
        '''
        Returns data in which disocntinuity is present along a circular cut
        '''
        r = np.random.uniform(1e-2, 7/9 * np.sqrt(Config.L))
        x_0 , y_0 = np.random.uniform(0, Config.L) , np.random.uniform(0,
            Config.L)
        # x_0, y_0 = 0.5, 0.5
        h = 1/Config.N
        fun1 = DataGenerator.getFunc()
        fun2 = DataGenerator.getFunc()
        x = np.linspace(0, Config.L, Config.N + 1)
        y = np.linspace(0, Config.L, Config.N + 1)
        xx, yy = np.meshgrid(x, y)

        val1 = fun1(xx, yy).reshape(xx.shape)
        val2 = fun2(xx, yy).reshape(xx.shape)

        val = np.where(((xx - x_0)**2) + ((yy-y_0)**2) - (r)<=0, val1, val2)
        xx = xx[:-1, :-1]
        yy = yy[:-1, :-1]
        calc1 = ((xx - x_0)**2) + ((yy-y_0)**2) - (r)>0
        calc2 = ((xx + h - x_0)**2) + ((yy-y_0)**2) - (r)>0
        calc3 = ((xx - x_0)**2) + ((yy + h-y_0)**2) - (r)>0
        calc4 = ((xx + h - x_0)**2) + ((yy + h -y_0)**2) - (r)>0
        label = (calc1 | calc2 | calc3 |calc4) & (np.logical_not(calc1 & calc2
            & calc3 & calc4))
        if np.sum(label) == 0:
          return DataGenerator.getCircDiscontData()
        return val, label
```

C.4 One dimensional Models

Code snippet C.11 for one dimensional CNN model. The implementation is done in TensorFlow [1] and keras [3] frameworks

Listing C.11: Implementation of one dimensional CNN model in TensorFlow and Keras

```
import tensorflow as tf
from keras.layers import Conv1D, Dense, Input, Flatten
from keras.models import Sequential
def create_model(input_size= 201):
  '''
  Returns One dimensional CNN model
  '''
  inp = Input(shape = (input_size, 1))
  X = Conv1D(kernel_size = 2, filters = 24,activation = 'relu' )(inp)
  X = Conv1D(kernel_size = 2, filters = 24, activation = 'relu')(X)
```

```
    X = Conv1D(kernel_size = 2, filters = 24, activation = 'relu')(X)
    X = Conv1D(kernel_size = 2, filters = 24, activation = 'relu')(X)
    X = Flatten()(X)
    X = Dense(input_size- 1, activation = 'sigmoid')(X)
    model  = tf.keras.Model(inputs = inp, outputs = X)
    return model
```

Code snippet C.12 for one dimensional GNN model. The implementation is done in PyTorch-geometric [5] and PyTorch [12] frameworks

Listing C.12: Implementation of one dimensional GNN model in PyTorch and PyTorch-geometric

```
from torch_geometric.nn import GATConv
class GATModel(nn.Module):
  def __init__(self):
    '''
    Two GATConv layers, viz self.conv1 and self.conv2 are used. Multihead
        attention with 10 heads are used in the first GAT convolution layer
    '''
    super(GATModel, self).__init__()
    self.conv1 = GATConv(
        in_channels = 9,
        out_channels = 32,
        heads = 10,
        dropout = 0.2
    )
    self.conv2 = GATConv(
        in_channels = 32 * 10,
        out_channels = 1,
        concat = False
    )

  def forward(self, data):
    x, edge_index = data.x, data.edge_index
    x = self.conv1(x, edge_index)
    x = torch.nn.ELU()(x)
    x = self.conv2(x, edge_index = edge_index)
    x = nn.Sigmoid()(x)
    return x
```

Code Snippet C.13 for converting numerical data of C.8 to data for the graph neural network with randomization on start and end points of mesh

Listing C.13: Code for converting u of DG method to graph data for GNN

```
import networkx as nx
from random import shuffle
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from torch_geometric.utils import from_networkx, to_networkx
from torch_geometric.data import Data, DataLoader
def DataItem(u, labels, advectionSpeed, finalTime):
  labels = np.concatenate([[0], labels])
  finalTime = data['finalTime']
  h = 1/100
  labels = labels * h
  labels = labels + advectionSpeed*finalTime
  labels = np.divmod(labels, 1)[1]
  y = np.zeros((100, ))
  labels = np.floor(labels/h)
  labels = list(map(int, labels))
  for label in labels:
    y[label] = 1
  y = torch.tensor(y).long()
  lis = [random.sample(range(max(min(labels) - 1, 0) + 1), 1)[0],
      random.sample(range(min(max(labels) + 1, 99), 100), 1)[0]]
  left, right = lis
  u = u[:, left : right + 1]
  u = u.T
  y = y[left : right + 1]
  rows = list(range(0, right - left)) + list(range(1 , right - left + 1))
  cols = list(range(1 , right - left + 1)) + list(range(0, right - left))
  edges = torch.tensor([rows, cols])
  u = torch.tensor(u.astype(np.float64)).double()
  y = torch.tensor(y).double()
  graph = Data(x = u, edge_index = edges, y = y)
  return graph
```

C.5 DG Numerical solver for Advection equation

The implementational details described in this section have been taken from the book [8]. The implementations of this book are in Matlab and are merely translated into Python. We also follow the book closely in explaining each code snippet.
Jacobi polynomial $P_n^{\alpha,\beta}$(x) of order n is a solution of the equation

$$\frac{d}{dx}(1-x^2)w(x)\frac{d}{dx}y + n(n+\alpha+\beta+1)w(x)y = 0 \tag{C.1}$$

for $x \in [-1, 1]$, where $w(x) = (1-x)^\alpha(1+x)^\beta$ The polynomials are orthogonal to each other in the inner product $< f, g >= \int_{-1}^{1} f(x)g(x)w(x)$.

$$\int_{-1}^{1} P_i^{\alpha,\beta} P_j^{\alpha,\beta} w(x) = \|P_i\|\delta_{ij} \tag{C.2}$$

The polynomials are normalized to ensure get an orthonormal basis. The normalized polynomials follow the recurrence

$$xP_n^{\alpha,\beta}(x) = a_n P_{n-1}^{\alpha,\beta}(x) + b_n P_n^{\alpha,\beta}(x) + a_{n+1}P_{n+1}^{\alpha,\beta}(x) \tag{C.3}$$

where the coefficients are given by

$$a_n = \frac{2}{2n+\alpha+\beta}\sqrt{\frac{n(n+\alpha+\beta)(n+\alpha)(n+\beta)}{(2n+\alpha+\beta-1)(2n+\alpha+\beta+1)}}$$

$$b_n = -\frac{\alpha^2-\beta^2}{(2n+\alpha+\beta)(2n+\alpha+\beta+2)}$$

The initial values are

$$P_0^{\alpha,\beta}(x) = \sqrt{2^{-\alpha-\beta-1}\frac{\Gamma(\alpha+\beta+2)}{\Gamma(\alpha+1)\Gamma(\beta+1)}}$$

$$P_1^{\alpha,\beta}(x) = \frac{1}{2}P_0^{\alpha,\beta}\sqrt{\frac{\alpha+\beta+3}{(\alpha+1)(\beta+1)}}((\alpha+\beta+2)x+(\alpha-\beta))$$

where $\Gamma(\alpha) = \int_0^\infty e^{-x}x^{\alpha-1}$ is the classic Gamma function.
Code SnippetC.14 for evaluating Jacobi polynomial of order N and type α, β at points x

Listing C.14: Evaluates jacobi polynomial of order N for given points x

```
def JacobiP(x, alpha, beta, N):
    '''
    Calculates the value of jacobi polynomial of type alpha, beta at points x
    for  order N.
    x: Numpy array
    '''
    xp = x.reshape(1, -1)
    n = xp.shape[1]
    PL = np.zeros((N+ 1, n))

    gamma0 = np.power(2, alpha + beta + 1)/(alpha + beta + 1) *gamma(alpha +
        1) * gamma(beta + 1)/gamma(alpha + beta + 1)
    PL[0, :] = 1.0/np.sqrt(gamma0)
    if N == 0:
        return PL.T

    gamma1 = (alpha + 1)*(beta + 1)/(alpha + beta +3)* gamma0

    PL[1, :] = ((alpha + beta + 2) * xp/2 + (alpha - beta)/2)/np.sqrt(gamma1)

    if N == 1:
        return PL[N, :]

    aold = 2/(2 + alpha + beta) * np.sqrt((alpha + 1) *(beta + 1)/(alpha +
        beta + 3))
    for i in range(N - 1):
        h1 = 2*(i + 1) + alpha + beta
        anew = 2/(h1 + 2) * np.sqrt((i + 2) * (i+2 + alpha + beta) * (i + 2 +
            alpha) *(i + 2 + beta)/(h1 + 1)/(h1 + 3))
        bnew = -(alpha ** 2 - beta**2)/h1/(h1 + 2)
        PL[i + 2, : ] = 1/(anew) *(-aold * PL[i, :] + (xp - bnew) *PL[i + 1,
            :])
        aold = anew
    return PL[N, :]
```

Approximation of integrals using gaussian quadrature is done as

$$\int_{-1}^{1} f(x)w(x)dx = \sum_{i=0}^{N} = f(x_i)w_i$$

where x_i are the quadrature nodes and w_i are the quadrature weights. Choosing the points x_i as the roots of the polynomial $P_{N+1}^{\alpha,\beta}$ and the weights w_i to ensure that the integration is exact for polynomials of degree $\leq N$ leads to an exact quadrature for all polynomials of degree $\leq 2N + 1$. The code snippet C.15 returns the points x_i and the corresponding weights w_i.

Listing C.15: Calculate gaussian quadrature nodes and weights

```
def JacobiGQ(alpha, beta, N):
    '''
    Computes the Nth order Guass Quadrature points(x) and the
    associated weights(w), associated with the Jacobi polynomial of type
        (alpha, beta)
    '''
    if N == 0:
        zer = (alpha - beta)/(alpha + beta + 2)
        wt = 2
        return np.array([zer]), np.array([wt])


    J = np.zeros((N + 1, N + 1))
    brr = np.array(range(1, N + 1))
    h1 = 2 * np.array(range(N + 1)) + alpha + beta
    J = np.diag(-1/2 * (alpha ** 2 - beta ** 2)/(h1 + 2)/h1) + np.diag(
        2/(h1[:N] + 2) * np.sqrt(brr * (brr + alpha + beta )*(brr + alpha) *
            (brr + beta)/(h1[:N] + 1)/(h1[:N] + 3)), 1
        )
    eps = 1e-5
    if alpha + beta < 10 * eps:

        J[0, 0] = 0
    J = J + J.T

    D, V = np.linalg.eig(J)
    x = D
    w = np.power(V[0, :].T, 2) * (2**(alpha + beta + 1))/(alpha + beta + 1) *
        gamma(alpha + 1) * gamma(beta + 1)/gamma(alpha + beta + 1)
    return x, w
```

The nodes obtained using Gaussian quadrature do not include the endpoints of the intervals. It is often convenient to use these points to impose boundary conditions. Hence we use Guasss-Lobato quadrature points, which are given as the roots of the polynomial $(1-x^2)\frac{d}{dx}P_N^{\alpha,\beta}(x)$
The Jacobi polynomials satisfy

$$\frac{d}{dx}P_n^{\alpha,\beta}(x) = \sqrt{n(n+\alpha+\beta+1)}P_{n-1}^{\alpha+1,\beta+1}(x) \tag{C.4}$$

From equation C.4 we can see that the Guass-Lobato points are the $(N-2)^{th}$ order Guass points of $P_{N-2}^{\alpha+1,\beta+1}(x)$ with $\{-1,1\}$ added. Code snippet C.16 computes the Guass-Lobato nodes

Listing C.16: Computes Guass-Lobato nodes

```python
def JacobiGL(alpha, beta, N):
    '''
    Computes the Nth order Gauss Lobato quadrature points(x), associated
    with the Jacobi polynomial of type (alpha, beta)
    '''
    x = np.zeros((N + 1, 1))
    if N == 1:
        return np.array([-1, 1])
    xint, _ = JacobiGQ(alpha + 1, beta + 1, N - 2)
    ind = xint.argsort(axis = None)
    xint = xint[ind]
    x = np.concatenate(([-1], xint, [1]))

    return x
```

The nodal formulation of a polynomial solution in each cell is of the form

$$u(\xi_i) = \sum_{n=1}^{n} \hat{u}_n \tilde{P}_{n-1}(\xi_i) \tag{C.5}$$

where ξ_i are the nodal points(grid points) and $\hat{u}_n$ are the modal coefficients. The equation C.5 can be written as

$$\mathcal{V}\hat{\mathbf{u}} = \mathbf{u} \tag{C.6}$$

where $\mathcal{V}_{ij} = \tilde{P}_{j-1}(\xi_i), \quad \hat{\mathbf{u}}_i = \hat{u}_i, \quad \mathbf{u}_i = u(\xi_i)$ The matrix $\mathcal{V}$ is known as the Vandermonde matrix. In code snippet C.17, we calculate the Vandermonde matrix.

Listing C.17: Calculates Vandermonde1D matrix

```python
def Vandermonde1D(N, r):
    '''
    Function to calculate the vandermonde martrix
    V_{ij} = phi_j(r_i)
    '''
    n = max(r.shape)
    V1D = np.zeros((n, N + 1))
    for j in range(N + 1):
        V1D[:, j] = JacobiP(r, 0, 0,j ).reshape((n, ))
    return V1D
```

A consequence of equation C.5 and uniqueness of polynomial interpolation is

$$\mathcal{V}^T \ell(r) = \tilde{P}(r) \tag{C.7}$$

where $\ell(r) = [\ell_1(r), \ell_2(r), \ldots, \ell_{N_p}(r)]^T$ are the lagrange interpolation polynomials defined on nodal points $\xi_i s$, i.e. $\ell_i(r_j) = \delta_{ij}$ Defining $\mathcal{D}_{r,(i,j)} = \frac{d\ell_j}{dr}|_{r_i}$ and using equation C.7, we have

$$\mathcal{V}^T \mathcal{D}_r^T = (\mathcal{V}_r)^T$$

where $\mathcal{V}_{r,(i,j)} = \dfrac{d\tilde{P}_j}{dr}|_{r_i}$. The code snippet C.18 calculates the derivative of $P_N^{\alpha,\beta}$ at points of r, C.19 calculates the matrix $\mathcal{V}_r$ and the code snippet C.20 calculates $\mathcal{D}_r$

Listing C.18: Code for calculating the derivative of jacobi polynomial

```python
def GradJacobiP(r, alpha, beta, N):
    '''
    Function to calculate the derivative of the jacobi polynomial
    of type (alpha, beta) at points r for order N and returns
    '''
    r = r.reshape(-1, 1)
    dP = np.zeros(r.shape)
    if N == 0:
        dP[:, :] = 0.0
    else:
        dP = np.sqrt(N*(N + alpha + beta + 1)) * JacobiP(r, alpha + 1, beta +
            1, N - 1)

    return dP
```

Listing C.19: Calculates the matrix of derivatives of lagrange polynomials(Vr)

```python
def GradVandermonde1D(N, r):
    '''
    Initialize the gradient of the modal basis (i) at (r) at order N
    '''
    n = max(r.shape)
    DVr = np.zeros((n, N+ 1))
    for i in range(N + 1):
        DVr[:, i ] = GradJacobiP(r, 0, 0, i).reshape((n,))
    return DVr
```

Listing C.20: Calculates the matrix Dr

```python
def Dmatrix1D(N, r, V):
    '''
    Function to calculate the differentiation matrix on the interval,
    evaluated at (r) at order N
    '''
    Vr = GradVandermonde1D(N, r)
    Dr = Vr @ np.linalg.pinv(V)
    return Dr
```

Defining for the k^{th} element $\mathcal{M}_{ij}^k = \int_{x_l^k}^{x_r^k} \ell_i^k(x)\ell_j^k(x)$, where the kth element is the interval $[x_l, x_r]$, it can be show that $\mathcal{M}_{ij}^k = \frac{h^k}{2}\mathcal{M}_{ij}$ where $\mathcal{M}_{ij} = \int_{-1}^{1} \ell_i(x)\ell_j(x)$
Substitution of C.7 quickly yields

$$\mathcal{M} = (\mathcal{V}\mathcal{V}^T)^{-1} \tag{C.8}$$

Defining $\mathcal{S}_{ij}^k = \displaystyle\int_{x_l^k}^{x_j^k} \ell_i^k(x)\frac{d\ell_j^k(x)}{dx} = \int_{-1}^{1} \ell_i(r)\frac{d\ell_j(r)}{dr}dr = \mathcal{S}_{ij}$ It can be shown that $\mathcal{M}\mathcal{D}_r = \mathcal{S}$
To calculate the surface integrals in the DG formulation

$$\oint_{-1}^{1} \hat{\mathbf{n}} \cdot (u_h - u*)l_i(r)dr = (u_h - u*)|_{r_{N_p}}\mathbf{e}_{N_p} - (u_h - u*)_{r_1}\mathbf{e}_1$$

where $\mathbf{e}_i$ is a N_p sized array with 1 at position i and 0 at all other places. Defining $\mathcal{E} = [\mathbf{e}_1, \mathbf{e}_2]$, in code snippet C.21, we calculate the operator $\mathcal{M}^{-1}\mathcal{E}$

Listing C.21: Lift operator for calculating the surface integral

```
def Lift1D():
    '''
    Compute surface integral term in dg-formulation
    '''
    global Np, Nfaces, Nfp
    global V
    Emat = np.zeros((Np, Nfaces * Nfp))

    Emat[0, 0] = 1.0
    Emat[Np -1, 1] = 1.0

    Lift = V@(V.T @ Emat)
    return Lift
```

We assume that the grid generator provides us with the following two sets of data

1. A row vector VX of $N_v = K + 1$ vertex coordinates, which correspond to the endpoints of the intervals

2. A matrix $EToV$ of size $K \times 2$ in which each row k corresponds to the two endpoints of element k

With this notation, we evaluate in C.22 inverse of the Affine mapping from each element to the standard element

Listing C.22: Computes metric of local mappings

```
def GeometricFactors1D(x, Dr):
    '''
    Computes the metric elements for the local mappings
    of the 1D elements
    '''
    xr = Dr@x
    J = xr
    rx = 1/J
    return rx, J
```

In code snippet C.23, we calculate the normals pointing outwards from each element face

Listing C.23: Computes outward pointing normals

```
def Normals1D():
    global Nfp, Nfaces, K
    nx = np.zeros((Nfp*Nfaces, K))
    nx[0, :] = -1.0
    nx[1, :] = 1.0
    return nx
```

Using $VX, EToV$, we recover the connectivity information in code snippet **??**

```
def Connect1D(EToV):
    '''
    Build Global connectivity arrays for 1D grid based on standard
    EToV input array from grid generator
    '''
    global Nfaces
    K = EToV.shape[0]
    TotalFaces = Nfaces*K
    Nv = K + 1
    vn = np.array([0, 1])

    sk = 1
    rows = []
    cols = []
    sk = 0
    for k in range(K):
        for face in range(Nfaces):
            rows.append(sk)
            cols.append(EToV[k, vn[face]])
            sk += 1
    data = [1]*len(cols)
    SpFToV = csr_matrix((data, (rows, cols)), shape = (TotalFaces, Nv))

    SpFToF = SpFToV@(SpFToV.T) - identity(TotalFaces)

    faces1 , faces2, _ = find(SpFToF == 1)
    element1 = faces1//Nfaces
    face1    = faces1%Nfaces
    element2 = faces2//Nfaces
    face2    = faces2 % Nfaces
    ind = sub2ind([K, Nfaces], element1, face1)
    ind = np.array(ind)
    EToE = np.array(range(K)).reshape(-1, 1)@np.ones((1, Nfaces))
    EToF = np.ones((K, 1))@np.array(range(Nfaces)).reshape(1, -1)
    EToE[ind%K, ind//K] = element2
    EToF[ind%K, ind//K] = face2
    return EToE, EToF
```

In code snippet C.24, we form connectivity and boundary tables(negative face and positive face maps for each element)

Listing C.24: returns connectivity and boundary tables

```
def BuildMaps1D():
    global K, Np, Nfaces, Nfp, NODETOL
    global EToE, EToV, EToF
    global Fmask
    global x



    nodeids = np.array(range(K*Np)).reshape(Np, K, order = 'F')
    vmapM = np.zeros((Nfp, Nfaces, K))
    vmapP = np.zeros((Nfp, Nfaces, K))
    for k1 in range(K):
        for f1 in range(Nfaces):
            vmapM[:, f1, k1] = nodeids[Fmask[f1], k1]




    for k1 in range(K):
        for f1 in range(Nfaces):
            k2 = int(EToE[k1, f1])
```

```
                f2 = int(EToF[k1, f1])

                vidM = vmapM[:, f1, k1]
                vidP = vmapM[:, f2, k2]
                vidM = vidM.astype(np.uint16)
                vidP = vidP.astype(np.uint16)
                vidM_ = np.unravel_index(vidP, x.shape, 'F')
                vidP_ = np.unravel_index(vidP , x.shape, 'F')
                x1 = x[vidM_]
                x2 = x[vidP_]
                D = (x1 -x2)**2
                if D < NODETOL:
                    vmapP[:, f1, k1] = vidP

        vmapP = vmapP.ravel('F')
        vmapM = vmapM.ravel('F')
        mapB = np.where(vmapP == vmapM)
        global vmapB, mapI, map0, vmapI, vmap0
        vmapB = vmapM[mapB]
        mapI = 0
        map0 = K*Nfaces - 1
        vmapI = 0
        vmap0 = K*Np - 1

        return vmapM, vmapP, vmapB, mapB
```

In C.25, we develop a subroutine to form a uniform mesh of K elements

Listing C.25: Generates uniform mesh

```
    def MeshGen1D(xmin, xmax, K):
        '''
        Function to generate mesh with K elements which are equidistant
        '''
        Nv = K + 1
        VX = np.array(range(Nv), dtype = np.float128)
        for i in range(Nv):
            VX[i] = (xmax - xmin) *(i)/(Nv - 1) + xmin
        EToV = np.zeros((K, 2))
        for k in range(K):
            EToV[k, 0] = k
            EToV[k, 1] = k + 1
        return Nv, VX, K, EToV
```

In code snippet C.26, we assemble all the operators required in DG formulation

Listing C.26: Set up script

```
def StartUp1D():
    Globals1D()
    global N, Nfp, Np, K
    global r , x , VX
    global Dr, LIFT
    global nx, Fx, Fscale
    global vmapM, vmapP, vmapB, mapB, Fmask
    global vmapI, vmapO, mapI, mapO
    global rx, J
    global rk4a, rk4b, rk4c
    global Nfaces, EToE, EToF, EToV
    global V, invV
    global NODETOL
    Np = N + 1
    Nfp = 1
    Nfaces = 2
    r = JacobiGL(0, 0, N)
    V = Vandermonde1D(N, r)
    invV = np.linalg.pinv(V)
    Dr = Dmatrix1D(N, r, V)
    LIFT = Lift1D()
    va = EToV[:, 0].reshape(1, -1)
    va = va.ravel()
    vb  = EToV[:, 1].reshape(1, -1)
    vb = vb.ravel()

    fmask1 = np.where(abs(1 + r)< NODETOL)
    fmask2 = np.where(abs(1 - r) < NODETOL)
    Fmask = [fmask1[0][0], fmask2[0][0]]

    r = r.reshape(-1, 1)
    x = np.ones((N + 1, 1)) @( VX[va.astype(np.uint8)].reshape(1, -1)) + 0.5 *
        ( 1 + r) @ ((VX[vb.astype(np.uint8)] -
       VX[va.astype(np.uint8)]).reshape(1, - 1))

    rx, J = GeometricFactors1D(x, Dr)

    EToE, EToF = Connect1D(EToV)
    Fx = x[np.ravel(np.array(Fmask)), :]
```

```
        nx = Normals1D()

        Fscale = 1/J[np.array(Fmask), :]
        vmapM, vmapP, vmapB, mapB = BuildMaps1D()
```

In code snippet C.27, we define the global variables needed in DG computation

Listing C.27: Definig Global variables

```
def Globals1D():
    global N, Nfp, Np, K
    global r , x , VX
    global Dr, LIFT
    global nx, Fx, Fscale
    global vmapM, vmapP, vmapB, mapB, Fmask
    global vmapI, vmapO, mapI, mapO
    global rx, J
    global rk4a, rk4b, rk4c
    global Nfaces, EToE, EToF
    global V, invV
    global NODETOL

    rk4a = np.array([
        0.0,
        -567301805773.0/1357537059087.0,
        -2404267990393.0/2016746695238.0,
        -3550918686646.0/2091501179385.0,
        -1275806237668.0/842570457699.0
    ])
    rk4b = np.array([
        1432997174477.0/9575080441755.0,
        5161836677717.0/13612068292357.0,
        1720146321549.0/2090206949498.0,
        3134564353537.0/4481467310338.0,
        2277821191437.0/14882151754819.0
    ])
    rk4c = np.array([
         0.0,
         1432997174477.0/9575080441755.0,
         2526269341429.0/6820363962896.0,
         2006345519317.0/3224310063776.0,
         2802321613138.0/2924317926251.0
    ])
```

```
NODETOL = 1e-10
```

Code snippet C.28 calculates the numerical flux for the advection equation given by $(au)* = \{\{au\}\} + a\frac{1-\alpha}{2}[\![u]\!]$ where $\{\{u\}\} = \frac{u^{+}+u^{-}}{2}$ and $[\![u]\!] = \hat{\mathbf{n}}^{-}u^{-} + \hat{\mathbf{n}}^{+}u^{+}$. For our case, we choose $\alpha = 0.5$

Listing C.28: calculates numerical flux for advection equation

```
def AdvecRHS1D(u, time, a):
    '''
    Evaluate the RHS flux in 1D Advection
    '''
    global Nfp, Nfaces, K
    global vmapP, vmapM, vmapB, mapB
    global mapI, vmapI, map0, vmap0
    global Dr
    global nx
    global bc





    alpha = 1
    vmapM = vmapM.astype(np.int16)
    vmapP = vmapP.astype(np.int16)
    # mapI = mapI.astype(np.int16)
    # map0 = map0.astype(np.int16)
    vmapM_ = np.unravel_index(vmapM, u.shape, 'F')
    vmapP_ = np.unravel_index(vmapP, u.shape, 'F')

    du = ((u[vmapM_] - u[vmapP_]).reshape(nx.shape, order = 'F')) * (a * nx -
        (1 - alpha)* np.abs(a * nx))/2


    uin = bc(-a * time)
    mapI_ = np.unravel_index(mapI, du.shape, 'F')
    map0_ = np.unravel_index(map0, du.shape, 'F')
    vmapI_ = np.unravel_index(vmapI, du.shape, 'F')
    du[mapI_] = (u[vmapI_] - uin)*(a* nx[mapI_] - (a - alpha) * np.abs(a *
        nx[mapI_]))/2

    map0_ = np.unravel_index(map0, du.shape, 'F')
    du[map0_] = 0
```

```
        rhsu = - a* rx * (Dr@u) + LIFT@(Fscale * du)
        # print(f"Fmask : {Fmask}")
        # print(f"Fscale : {Fscale}")
        # print(f'J : {J}')
        # print(f"r : {r}")
        return rhsu
```

Code snippet C.5 solves the advection equation

```
    def Advec1D(u, a, finalTime):
        time = 0
        global Nfp, Nfaces, K
        global vmapP, vmapM, vmapB, mapB
        global mapI, vmapI, mapO, vmapO
        global Dr
        global nx
        global bc
        global rk4c
        global N


        resu = np.zeros((Np, K))
        xmin = min(abs(x[0, :] - x[1, :]))

        CFL  = 0.75
        dt = (CFL/a)*xmin
        dt = 0.5 * dt
        Nsteps = np.ceil(finalTime/dt)
        dt = finalTime/Nsteps
        Nsteps = int(Nsteps)
        for tstep in range(Nsteps):
            for INTRK in range(5):
                timelocal = time + rk4c[INTRK]*dt


                rhsu = AdvecRHS1D(u, timelocal, a)
                resu = rk4a[INTRK] * resu + dt * rhsu

                u = u + rk4b[INTRK] * resu

            time = time + dt
    return u
```

Code snippet C.29 is the driver code for the advection equation

Listing C.29: Driver code

```
def SolveAdvection(f, finalTime, advectionSpeed, plot = False):
  Globals1D()
  global N, Nfp, Np, K
  global r , x , VX
  global Dr, LIFT
  global nx, Fx, Fscale
  global vmapM, vmapP, vmapB, mapB, Fmask
  global vmapI, vmapO, mapI, mapO
  global rx, J
  global rk4a, rk4b, rk4c
  global Nfaces, EToE, EToF, EToV
  global V, invV
  global NODETOL
  global bc
  bc = f
  Nv, VX, K, EToV = MeshGen1D(0.0, 1, 100)
  StartUp1D()
  u = f(x)
  u = Advec1D(u, advectionSpeed, finalTime)
  if plot:

    from matplotlib import pyplot as plt
    plt.plot(x.ravel('F'), u.ravel('F'), 'x', markersize = 1)
    plt.plot(x.ravel('F'), f(x).ravel('F'))
    plt.plot(x.ravel('F'), f( x-advectionSpeed * finalTime ).ravel('F'))
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(['FinalTime ', 'Time 0', 'Expect'])
    plt.show()
  return u
```

C.6 High-level implementation details for GNN in two dimensions

To generate the data for the two-dimensional case, we follow the data generation process for the two-dimensional CNN models. Once the data is generated, we then randomize the mesh size as follows:

1. Choose $(x_1, y_1) \in D$ uniformly at random
2. Now choose $(x_2, y_2) \in D$ uniformly at random
3. Choose the rectangular patch with top left corner as $(\min(x_1, x_2), \max(y_1, y_2))$ and bottom right corner as $(\max(x_1, x_2), \min(y_1, y_2))$
4. Form the graph on this rectangular patch by considering each cell in the patch as a node and edges between cells sharing a face.

In code snippet C.30, we provide a GNN model architecture that can be trained on the above-generated data. Note that each node contains 9 attributes which are the values at the four corners of the cell, 4 values at midpoints of the faces of the cell and one value at the centroid of the cell

Listing C.30: Two dimensional GNN detector

```
from torch_geometric.nn import GATConv
class GATModel(nn.Module):
  def __init__(self):
    '''
    Three GATConv layers, viz self.conv1, self.conv2, self.conv3 are used.
        Multihead attention with 8 heads are used in the first GAT convolution
        layer and multihead attention with 4 heads in second GAT convolution
        layer are used
    '''
    super(GATModel, self).__init__()
    self.conv1 = GATConv(
        in_channels = 9,
        out_channels = 32,
        heads = 8,
        dropout = 0.2
    )
    self.conv2 = GATConv(
        in_channels = 32 * 8,
        out_channels = 32,
        heads = 4,
        dropout = 0.2
        )
        self.conv3 = GATConv(
            in_channels = 32*4,
            out_channels = 1,
            concat = False
        )

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = torch.nn.ELU()(x)
        x = self.conv2(x, edge_index = edge_index)
        x = torch.nn.ELU()(x)
        x = self.conv3(x, edge_index = edge_index)
        x = nn.Sigmoid()(x)
        return x
```

C.7 Algorithm for iterative modification of GNN

We describe an algorithm (1) to iteratively train the GNN so as to make the model more robust and accurate. We have not tested the algorithm, but we expect the algorithm to help in increasing the quality of limiting by the GNN.

Algorithm 1 Algorithm to iteratively train GNN model improving the limiting quality of GNN

```
1: $Input \leftarrow GNN, NumIterations$
2: $i \leftarrow 0$
3: while $i < NumIterations$ do
4:     Generate Data using the GNN as limiter as described in 5.7.2
5:     $GNN \leftarrow$ Trained new GNN on the data generated
6:     $i \leftarrow i + 1$
7: end while
```

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] R. Archibald, A. Gelb, and J. Yoon. Polynomial fitting for edge detection in irregularly sampled signals and images. *SIAM J. Numerical Analysis*, 43:259–279, 01 2005.

[3] F. Chollet et al. Keras. `https://keras.io`, 2015.

[4] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire, P. W. Battaglia, V. Gupta, A. Li, Z. Xu, A. Sanchez-Gonzalez, Y. Li, and P. Velickovic. ETA prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. ACM, oct 2021.

[5] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[6] G. Fu and C.-W. Shu. A new troubled-cell indicator for discontinuous galerkin methods for hyperbolic conservation laws. *Journal of Computational Physics*, 347:305–327, 2017.

[7] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs, 2018.

[8] J. S. Hesthaven and T. Warburton. *Nodal discontinuous galerkin methods: Algorithms, analysis, and applications.* Springer, 2011.

[9] J. M. Jumper, R. Evans, A. Pritzel, T. J. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. a. A. Kohl, A. J. Ballard, A. M. Cowie, B. Romera-Paredes, S. Nikolov, R. D. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zieliński, J. Söding, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, S. W, Andrew, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, Jul 2021.

[10] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks, 2017.

[11] L. Krivodonova, J. Xin, J.-F. Remacle, N. Chevaugeon, and J. Flaherty. Shock detection and limiting with discontinuous galerkin methods for hyperbolic conservation laws. *Applied Numerical Mathematics*, 48(3):323–338, 2004. Workshop on Innovative Time Integrators for PDEs.

[12] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[13] D. Ray and J. S. Hesthaven. Detecting troubled-cells on two-dimensional unstructured grids using a neural network. *Journal of Computational Physics*, 397:108845, 2019.

[14] J. M. Stokes, K. Yang, K. Swanson, W. Jin, A. Cubillos-Ruiz, N. M. Donghia, C. R. MacNair, S. French, L. A. Carfrae, Z. Bloom-Ackermann, V. M. Tran, A. Chiappino-Pepe, A. H. Badran, I. W. Andrews, E. J. Chory, G. M. Church, E. D. Brown, T. S. Jaakkola, R. Barzilay, and J. J. Collins. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702.e13, 2020.

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2018.

[16] S. Wang, Z. Zhou, L.-B. Chang, and D. Xiu. Construction of discontinuity detectors using convolutional neural networks. *J. Sci. Comput.*, 91(2), may 2022.