Physics Informed Neural Networks for solving Partial Differential Equations

Report submitted to

Indian Institute of Technology Kharagpur

in partial fulfilment for the award of the degree of

Master of Science

in

Physics

By

Diganta Samanta

(21PH40020)

Under the supervision of

Professor Vishwanath Shukla

Department of Physics

Indian Institute of Technology Kharagpur

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

Diganta Samanta
(21PH40020)

In my capacity as supervisor of the candidate's thesis, I certify that the above statements are true to the best of my knowledge.

Dr. Vishwanath Shukla

Date: November 26, 2022

Place: Kharagpur

Acknowledgements

I thank Professor Vishwanath Shukla for giving me an opportunity to work, despite my lacking of the detailed theoretical nuances needed. This project helped me to practice and sharpen my skills in the field of physics and computing.

I especially thank Dr. Abhay Kumar Tiwari, a Data Scientist at More Retail Private Ltd., who has given me valuable advice in this Deep Learning field.

I was also helped by my fellow members of StatFluid Lab who have listened to my problems and helped me when I was struck. I am grateful for their generous aid.

I want to thank also my parents, friends, roommates who have continuously motivated me.

Contents

Chapter 1

Introduction

For the last few decades, Machine Learning has been performing an outstanding job in diverse scientific disciplines, including image recognition, natural language processing, cognitive science, and genomics. In most Artificial Intelligence applications, we use Deep Neural Networks(DNN). DNNs are increasingly being used to solve classical mathematical problems such as partial differential equations of Fluid Dynamics, Heat Flow, Diffusion, Waves, Quantum Mechanics, etc. DNNs have shown very good performance in predicting the relation between input and output in a very complex system. A neural network can predict any function up to a certain approximation due to the universal approximation theorem. Using this property, we find out the underlying properties of such a system that we have some physical knowledge (governing equations, boundary and initial conditions, dependencies, etc.) of the system. PINNs are scientific machine learning techniques with an artificial network that knows the physics of the specified system.

We have introduced a PINN framework using DeepXDE and solved Burgers' equation for different viscosity and heat equations for two different initial functions.

Chapter 2

Physics Informed Neural Networks

2.1 Physics Informed Neural Networks (PINNs)

The basic concept behind PINN training is that it can be thought of as an unsupervised strategy that does not require labelled data, such as results from prior simulations or experiments. PINN approximate PDE solutions by training a neural network to minimize a loss function, which includes terms reflecting the initial and boundary conditions. The PINN algorithm is essentially a mesh-free technique that finds PDE solutions by converting the problem of directly solving the governing equations into an optimization problem. PINNs are mainly built by

- Deep Neural Networks
- Optimizer
- Automatic Differentiation
- Loss Function

After building a network we have to train it by specified optimizer to minimize defined loss function. Sometimes if we have some data generated by noisy experiment observations, simulations and any process which gives the physical intuition of the system, we can use it by adding it to the network and defining a loss function according to it.

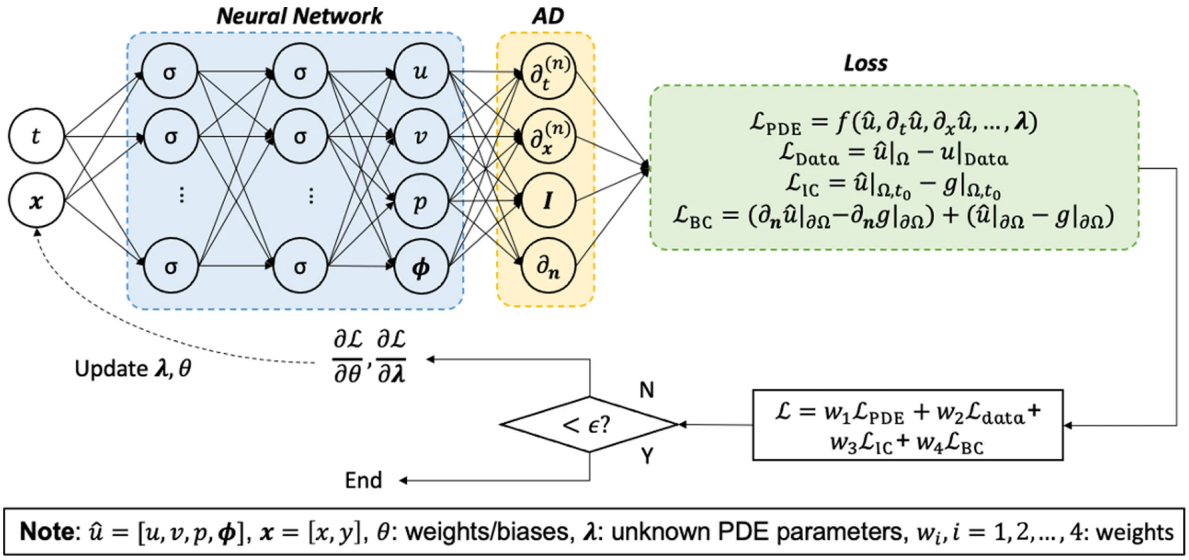

FIGURE 2.1: Building Blocks of PINNs. Taken from: Cai, S., Mao, Z., Wang, Z. et al. Physics-informed neural networks (PINNs) for fluid mechanics: a review. Acta Mech. Sin. 37, 1727–1738 (2021). https://doi.org/10.1007/s10409-021-01148-1

2.1.1 Deep Neural Networks

A neural network is an obvious part of PINNs. There are several types of neural networks. Here we have only used Multi-layer perceptron (MLP) (also known as Feed Forward Neural Network (FFNN)). FFNN is a collection of neurons organised in layers which can perform sequential calculations. Consider a M -layered neural network, which has Ni neurons in the i^{th} layer. For a given nonlinear activation function which is applied element by element, the Feed Forward Neural Network can be defined as

$$\begin{aligned}
&\text{input layer :} \quad \mathcal{N}^0(\mathrm{x}) = \mathrm{x} \in \mathbb{R}^{d_{in}},\\
&\text{hidden layers :} \ \mathcal{N}^i(\mathrm{x}) = \sigma(\mathbf{W}^i\mathcal{N}^{i-1}(\mathrm{x})) + \mathbf{b}^i \in \mathbb{R}^{N_i}, \ \text{ for } 1 \leq i \leq M-1,\\
&\text{output layer :} \ \mathcal{N}^M(\mathrm{x}) = \sigma(\mathbf{W}^M\mathcal{N}^{M-1}(\mathrm{x})) + \mathbf{b}^M \in \mathbb{R}^{d_{out}}
\end{aligned}$$

where the weight matrix in the $i-th$ layer is given by $\mathbf{W}^i \in \mathbb{R}^{N_i \times N_{i-1}}$ and the bias vector for the same is given by $\mathbf{b}^i \in \mathbb{R}^{N_i}$, respectively.

2.1.2 Optimizer

Optimizer is an integral part of a neural network. To train a network we need a process or algorithm to find the parameters for which the loss function is minimum or equal to expected value. Here we have used 'Adam' and 'L-BFGS' optimizers. Adam is a variant of Gradient Descent. L-BFGS is a variant of quasi newton method.

2.1.3 Automatic/Algorithmic Differentiation

In PINNs, one needs to compute the derivatives of the network outputs $(u(x,t))$ with respect to the network inputs (x,t). Backpropagation, a specialized technique of AD, is usually used In deep learning to calculate the derivatives. AD has two steps: firstly, a forward pass to compute the values of all the variables, and secondly, a backward pass to compute all the derivatives. For example, consider a FNN with one hidden layer that has two inputs x_1, x_2 along with one output u:

$$\begin{aligned}
y &= -4x_1 + x_2 + 0.5,\\
h &= \tanh y,\\
u &= 4h + 1
\end{aligned}$$

The forward and backward passes of AD to compute the partial derivatives $\frac{\partial u}{\partial x_1}$ and $\frac{\partial u}{\partial x_2}$ at $(x_1, x_2) = (0.5,1)$ are shown in Table 1.1.

Forward Pass	Backward Pass
$x_1 = 0.5,\ \ x_2 = 1$	$\frac{\partial u}{\partial u} = 1$
$y = -4x_1 + x_2 + 0.5 = -0.5$	$\frac{\partial u}{\partial h} = \frac{\partial 4h+1}{\partial h} = 4$
$h = \tanh(y) \approx -0.462$	$\frac{\partial u}{\partial y} = \frac{\partial u}{\partial h}\frac{\partial h}{\partial y} \approx 3.145$
$u = 4h + 1 = -0.848$	$\frac{\partial u}{\partial x_1} = \frac{\partial u}{\partial y}\frac{\partial y}{\partial x_1} = -12.58$
	$\frac{\partial u}{\partial x_2} = \frac{\partial u}{\partial y}\frac{\partial y}{\partial x_2} = 3.145$

TABLE 2.1: Example of AD computation for the partial derivatives at $(x_1, x_2) =$ (0.5,1)

Unlike the finite differences in which d-dimensional input requires $d_{in} + 1$ forward passes to calculate each partial derivative $\frac{\partial u}{\partial x_i}$, AD requires only one forward pass and one backward pass to compute all the partial derivatives, regardless of the input dimension. Thus, AD is more efficient than finite differences, especially in cases with high input dimensions.

2.1.4 Loss Function

When we solve a partial differential equation using FFNN we have to define proper loss function considering the equation, boundary condition, initial value and data loss. Actually, any physical knowledge known about the system is generally implemented by the loss function. There are mainly three loss functions. The first one comes from the partial differential equation. If the partial differential equation is

$$\mathcal{F}(u(x); \gamma) = f(x)$$

Then the first loss function is

$$\mathcal{L}_{\mathcal{F}} = \frac{1}{N}\sum_{x_i} |\mathcal{F}(u(x_i), \gamma) - f(x_i)|^2$$

We define second loss function from initial conditions or boundary conditions. If there are both initial and boundary conditions we have to consider both separately ($\mathcal{L}_B \quad and \quad \mathcal{L}_I$) The third loss function comes from if there is any known data of the system.

$$\mathcal{L}_{data} = \frac{1}{N}\sum_{i} |u(\mathbf{x_i} - y_i)|^2$$

The total loss function is a weighted sum of all loss functions.

$$\mathcal{L}_{total} = \mathcal{W}_{\mathcal{F}}\mathcal{L}_{\mathcal{F}} + \mathcal{W}_{\mathcal{B}}\mathcal{L}_{\mathcal{B}} + \mathcal{W}_{\mathcal{I}}\mathcal{L}_{\mathcal{I}} + \mathcal{W}_{data}\mathcal{L}_{data}$$

2.1.5 Training

After all, we have to train the network. Training means changing the network's parameters towards the loss function being minimum or some expected error limit. The optimizer does it efficiently in an iterative process.

$$\theta_i^* = \arg\min_{\theta_i} \left[\mathcal{W}\mathcal{L}_{\mathcal{F}}(\theta) + \mathcal{W}_{\mathcal{B}}\mathcal{L}_{\mathcal{B}}(\theta) + \mathcal{W}_{\mathcal{I}}\mathcal{L}_{\mathcal{I}}(\theta) + \mathcal{W}_{data}\mathcal{L}_{data}(\theta)\right]$$

where $\theta_i \in \{\omega_i, b_i\}$. ω_i and b_i is the weights and biases of the neurons of neural network. In a simple gradient descent optimization

$$\theta_i = \theta_{i-1} - \eta\frac{\partial\mathcal{L}}{\partial\theta_{i-1}}$$

η is learning rate. The partial derivative in the right hand side is calculated by backpropagation algorithm.

2.2 Implementations

This section will discuss how we can implement PINN in the python framework. We have chosen the python library DeepXDE to solve the partial differential equations.

2.2.1 DeepXDE

DeepXDE is a library for for physics-informed machine learning. To solve a PDE, we have to specify the PDE, the computational domain(both space and time domain), the initial and boundary conditions, constraints, Neural network architecture, training data and optimizer.
Following is a step-by-step example of implementation:

1. Import the DeepXDE library and declaring the parameters in the PDE.

```
import deepxde as dde
import numpy as np
from deepxde.backend import tf
# Problem parameters
alpha, L, n = 0.4, 1, 1
```

2. We will define the computational domain using DeepXDE built-in classes. Time and space domain is specified by `class TimeDomain` and `class Interval` respectively.

```
# Defining geometry
geom = dde.geometry.Interval(0,L)
```

```
timedomain = dde.geometry.TimeDomain(0,1)
geotime = dde.geometry.GeometryXTime(geom, timedomain)
```

3. We Define the PDE using the `dde.grad` module which can compute the first-order derivatives(`dde.grad.jacobian`) and second-order derivatives(`dde.grad.hessian`).

```
# Defining PDE
def pde(x,y):
    dy_t = dde.grad.jacobian(y,x,j=1)
    dy_xx = dde.grad.hessian(y,x,j=0)
    return dy_t - alpha * dy_xx
```

4. DeepXDE not only supports the four standard boundary conditions: Dirichlet, Neumann, Periodic and Robin but also supports handling general boundary conditions using OperatorBC.

```
# Defining Initial and Boundary conditions
bc = dde.icbc.DirichletBC(geotime, lambda x: 0, lambda _,
   on_boundary: on_boundary)
ic = dde.icbc.IC(geotime, lambda x: tf.sin(n*tf.pi*x[:, 0:1]/L),
   lambda _, on_initial: on_initial)
```

5. Combine all the information about the geometry, PDE and initial/boundary conditions together into data.TimePDE or data.PDE depending on the kind of the PDE. Also, specify the number of training point at specific point locations like initial and boundary.

```
# Combining information into data
data = dde.data.TimePDE(geotime, pde, [bc, ic],
                        num_domain=2540, num_boundary=80,
```

```
                        num_initial=160, num_test=2540)
```

6. Construct the neural network and define the model by combining the neural network with the data combined in the previous step. DeepXDE has two types of neural network: feed-forward and residual neural network. And, finally compile the model to set the optimization parameters like optimizers, learning rates, loss types.

```
# Defining the network:
net = dde.nn.FNN([2] + [32] * 3 + [1], "tanh", "Glorot normal")
model = dde.Model(data, net)
model.compile('adam', lr=1e-2)
```

The network used here has 2 inputs, 3 hidden layers with 32 neurons each and 1 output neuron with 'tanh' activation and a learning rate of 0.01.

7. Train the network model (either with random initialization or a pretrained model using argument *model_restore_path*).

```
# Training the model
model.train(epochs=20000)
```

Here, the model is trained for 20,000 iterations.

```
model.compile("L-BFGS")
losshistory, train_state = model.train()
```

This is followed by another training process with 'L-BFGS' optimizer. 'L-BFGS' does not need learning rate and number of iterations.

8. Make PDE solution prediction using Model.predict.

```
# Making Prediction
y_pred = model.predict(X)
# Saving Prediction
np.savetxt("DeepXDE_sol.dat", np.hstack((X, y_pred)))
```

Chapter 3

Application on Partial Differential Equations

3.1 Solving Burgers' equation

3.1.1 Burgers' equation

In many physical systems like fluid dynamics, nonlinear acoustics, gas dynamics, etc., Burgers' equation is a fundamental partial differential equation. The general form of 1d Burgers' equation(viscous Burgers' equation) is

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

Where ν is viscosity of fluid in fluid mechanics.

3.1.2 Results

We have solved this equation by taking four values of viscosity ($\nu = 0.1$,0.01,0.001,0.0001. The space domain is [-1,1] and the time domain is [0,1]. The initial condition at t=0 is

$$u(x, 0) = -\sin \pi x$$

and the boundary condition is

$$u(-1, t) = u(1, t) = 0.0$$

Here we have used FFNN made with three layers of 50 neurons and chosen L-BFGS as the optimizer. The advantage of L-BFGS optimizer is

- No need to define learning rate(lr)
- No need to define number of iteration
- faster than usual gradient descent optimizer

For viscosity 0.001 and 0.0001 we have used Adam optimizer with learning rate 0.01 for 20000 iteration and after that we've used L-BFGS.

From figure 3.1 we can see that the PINN can capture the non-linearity of the solution of burger equation. The non-linearity is the cause of shock formation in wave of low viscosity. The time of training and final train loss has been provided in Table:3.1. We say that solution is nicely approximated if the final loss function has a very small value($\ll 1$). For viscosity 0.0001, the final loss function has a large value compared to other loss function values. So the solution for the viscosity of 0.0001 has yet to be fairly approximated.

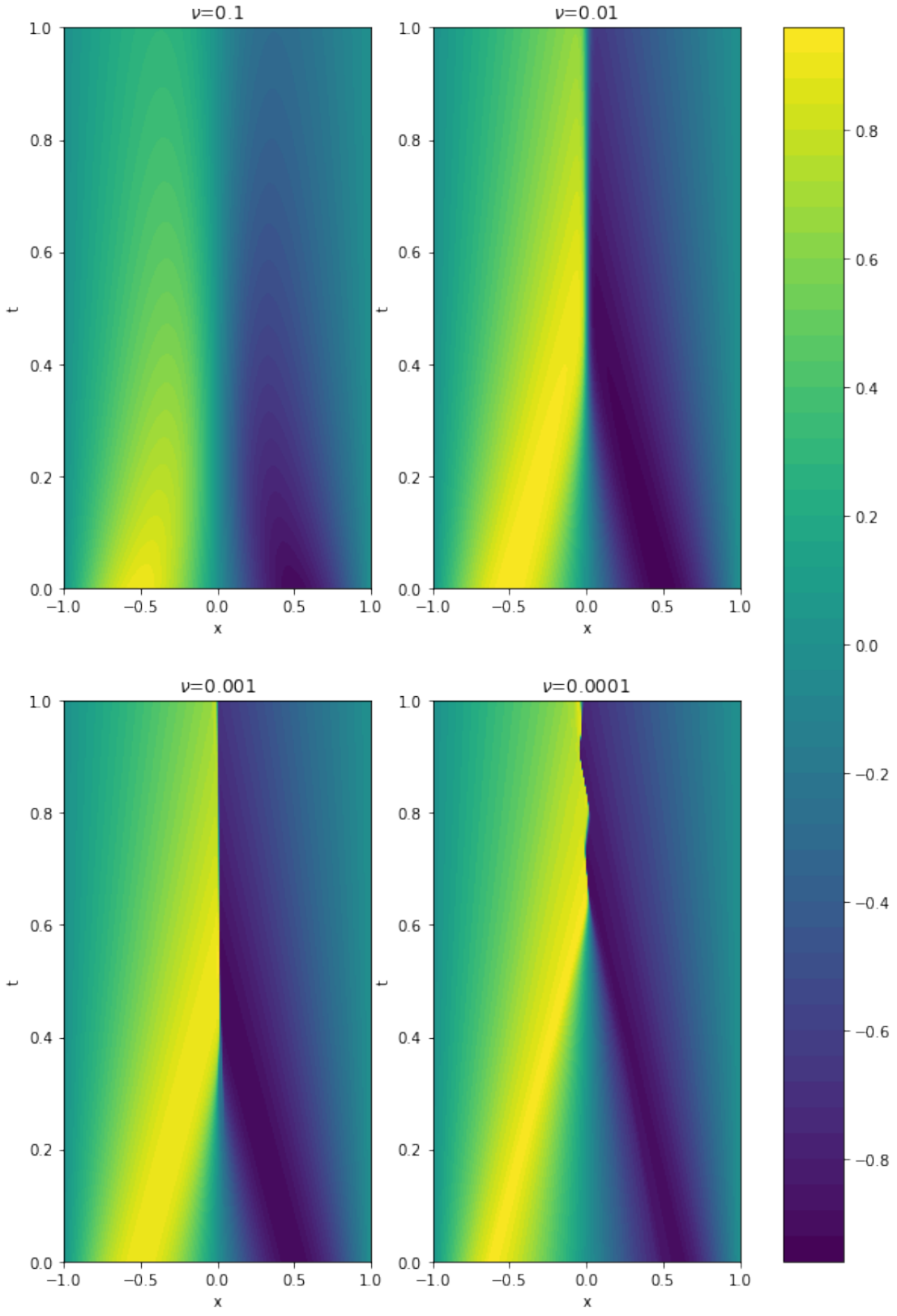

FIGURE 3.1: Solution of 1d Burgers' equation for four viscosity(ν = 0.1,0.01,0.001,0.0001)

ν	Optimizer	Training time (second)	final train loss
0.1	L-BFGS	128.997	2.05×10^{-06}
0.01	L-BFGS	214.848	4.50×10^{-06}
0.001	adam(lr=0.001) + L-BFGS	1417.553	2.05×10^{-06}
0.0001	adam(lr=0.001) + L-BFGS	1166.165	2.89×10^{-02}

TABLE 3.1: Training times and final losses for different viscosity

3.2 Solving Heat Equation

3.2.1 Heat Equation

The heat equation shows how heat flows in a homogeneous and isotropic medium. If u(x,y,z,t) is the temperature at the point (x,y,z) and time t, the equation is

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

α, a positive coefficient, is called thermal diffusivity of the medium. α decides how fast heat will flow in medium.

In our case we have considered the heat flow in an 1d rod of length L. So now the heat equation is

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

We have solved heat equation using PINN for two initial function. One is $\sin \frac{n\pi x}{L}$ and the other is Dirac Delta function

3.2.2 Results for First Initial Condition

We have solved

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

where $x \in [0, 1]$ and $t \in [0, 1]$. We have taken $\alpha = 0.4$ with Dirichlet boundary conditions

$$u(0, t) = u(1, t) = 0$$

and periodic initial condition

$$u(x, 0) = \sin \frac{n\pi x}{L}$$

where L=1. The exact solution is

$$u(x, t) = e^{-\frac{n^2\pi^2\alpha t}{L^2}} \sin \frac{n\pi x}{L}$$

In this case, we have used FFNN of 3 hidden layers of 20 neurons and tanh as activation function. At first we have used Adam optimizer for 20000 iteration and after this we have used L-BFGS. Total training has taken 145.807 second and final loss value is 7.50×10^{-07}.

Here are plotted results of the PDE solution

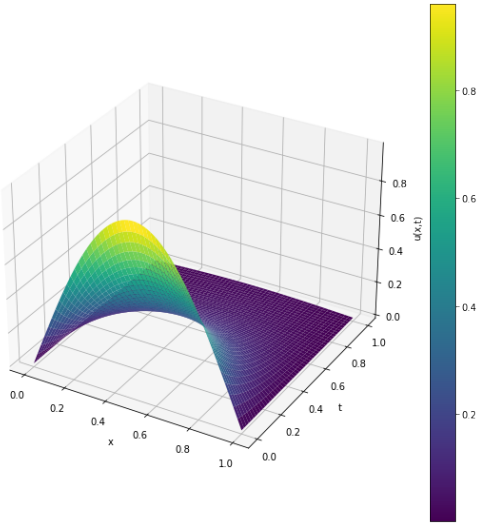

FIGURE 3.2: 3D plot of solution

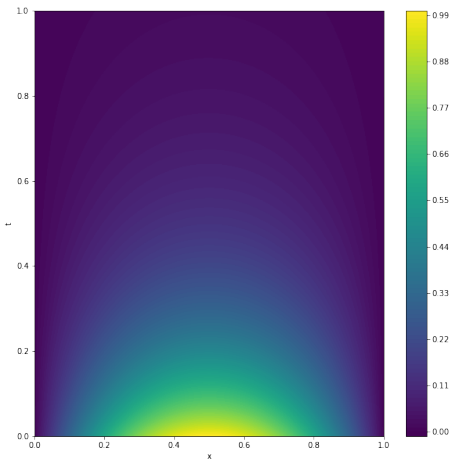

FIGURE 3.3: 2D surface plot of solution

FIGURE 3.4: Plots of solution of Heat equation for periodic initial function

3.2.3 Results for second Initial Condition

Here we have used Dirac delta function as initial condition

$$u(x, 0) = \delta(x - \frac{L}{2})$$

We have generated Dirac function by using Gaussian distribution with variance very less than one. In this case, viscosity is 0.008. We have used the same NN architecture as earlier defined.

Here are the results

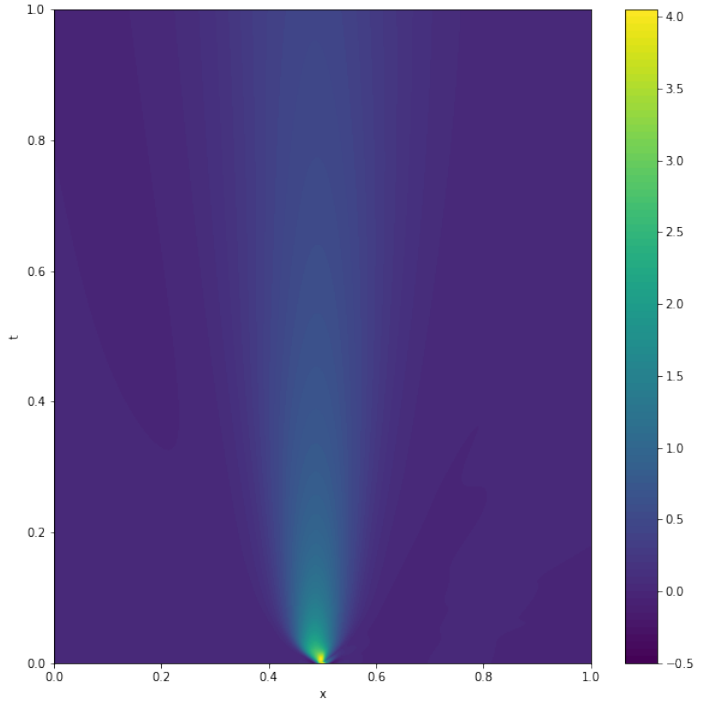

FIGURE 3.5: 2D contour plot of solution of Heat equation for Dirac delta initial function

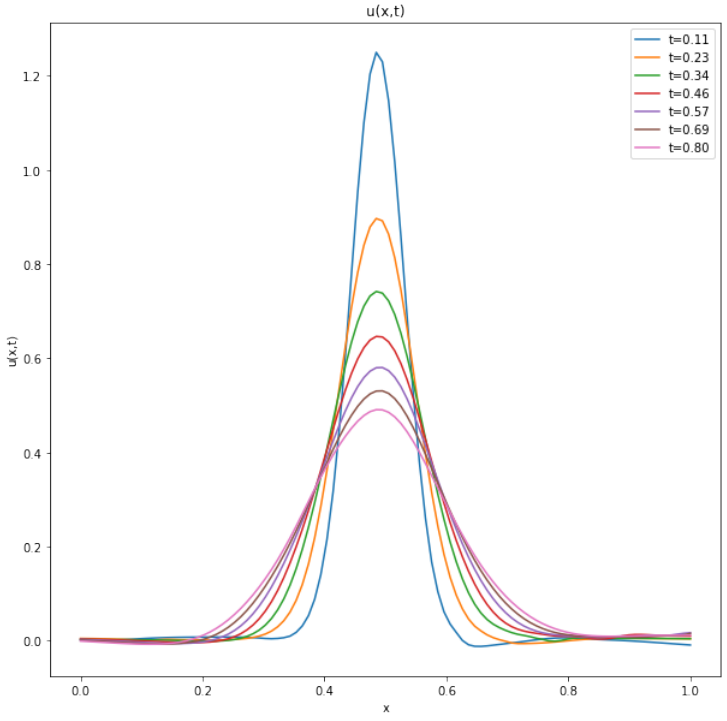

FIGURE 3.6: Time evaluation of temperature distribution

From the results, we can see that the temperature distribution is flattening with time.

Initial function	Optimizer	Training time (second)	final train loss
Periodic	adam(lr=0.001) + L-BFGS	145.807	6.72×10^{-07}
Dirac delta	adam(lr=0.001) + L-BFGS	145.332	3.60×10^{-02}

TABLE 3.2: Training times and final losses for different initial function

Chapter 4

Future Plan

In this semester's project, we have acquired a general idea of PINN for different physical systems. Next semester, we will try to find out different aspects of PINNs and apply them to different physical systems. We will use different functionalities of DeepXDE, NeuroDiffEq, and PyDens.

Bibliography

[1] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, 2019. A high-bias, low-variance introduction to Machine Learning for physicists.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[3] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, Aug 1989.

[4] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[5] Feiyu Chen, David Sondak, Pavlos Protopapas, Marios Mattheakis, Shuheng Liu, Devansh Agarwal, and Marco Di Giovanni. Neurodiffeq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46):1931, 2020.

[6] Khudorozhkov R., Tsimfer S., and Koryagin. A. Pydens framework for solving differential equations with deep learning, 2019.

[7] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[8] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[9] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.