AE5093 — Scientific Applications of Deep Learning

Homework 4

Daniel J. Pearson

Aerospace Engineering Department
Worcester Polytechnic Institute

1 Introduction

In this assigment, a Feed Forward Neural Network (FFNN) is trained and informed using a physical model of a system. The network will be trained on two PDEs: the 1D viscous Burgers equation and the 2D wave equation. The goal is to demonstrate the ability of the FFNN to learn the underlying physics of the system and make predictions based on the model. For both cases, it will be fed a series of boundary conditions and interior points to learn.

2 1D Burgers Equation

The 1D viscous Burgers equation is a fundamental partial differential equation (PDE) that describes the motion of a viscous fluid. It is given by:

$$u_t + u * u_x = \nu u_{xx} \tag{1}$$

where u is the velocity field, t is time, x is the spatial coordinate, and ν is the kinematic viscosity.

2.1 Problem Setup

For this problem, the domain is defined as $x \in [-1, 1]$ and $t \in [0, 1]$. With the following boundary conditions:

$$u(x, 0) = -\sin(\pi x) \tag{2}$$

$$u(-1, t) = u(1, t) = 0 \tag{3}$$

where,

$$\nu \in \{\frac{0.01}{\pi}, \frac{0.0001}{\pi}, 0.0\}$$

.

For each viscous case, the following parameters are used:

- Epochs = 5000
- LearningRate = 1000
- Neurons = 50
- $N_i = 5000$ (interior points)
- $N_b = 256$ (boundary points)
- $N_{ic} = 256$ (initial condition points)

The NN will be trained on the interior points, boundary points and initial conditions. The total loss function is defined by:

$$L = L_{pde} + L_{ic} + L_{bc} \tag{4}$$

where,

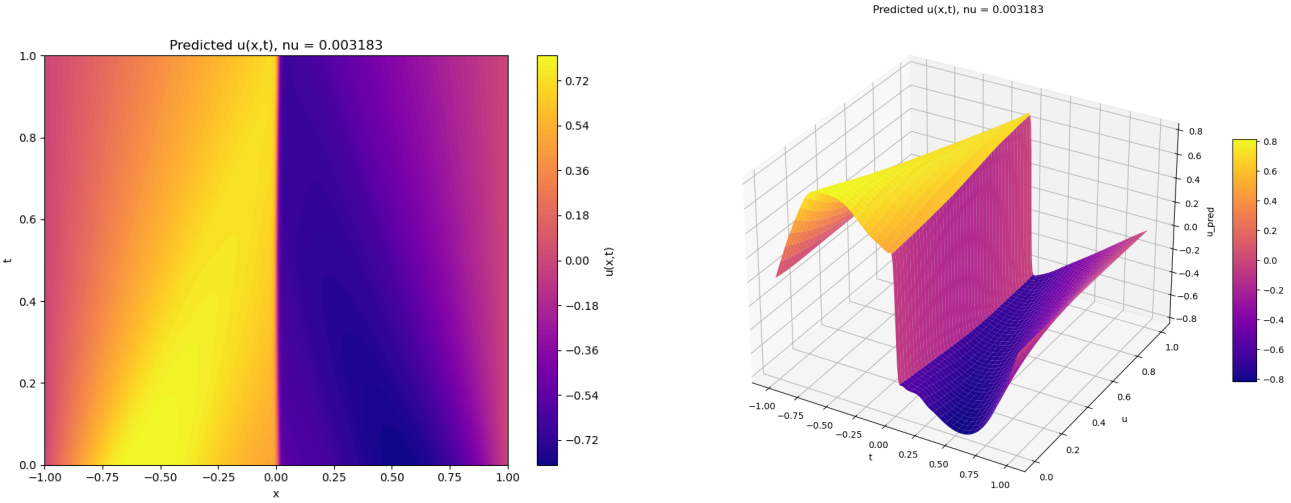
$$L_{pde} = u_t + uu_x - \nu u_{xx} \tag{5}$$

$$L_{ic} = \frac{1}{N}\sum_{i=1}^{N}(\theta(u|x_{ic}, t_{ic}) - u_{ic})^2 \tag{6}$$

$$L_{bc} = L_{ic} = \frac{1}{N}\sum_{i=1}^{N}(\theta(u|x_{bc}, t_{bc}) - u_{bc})^2 \tag{7}$$

2.2 Results - No LR Annealing

2.2.1 Case 1: High Viscosity



(a) Prediction Gradient for Case 1: High Viscosity

(b) Prediction for Case 1: High Viscosity

Figure 1: Prediction for Case 1: High Viscosity

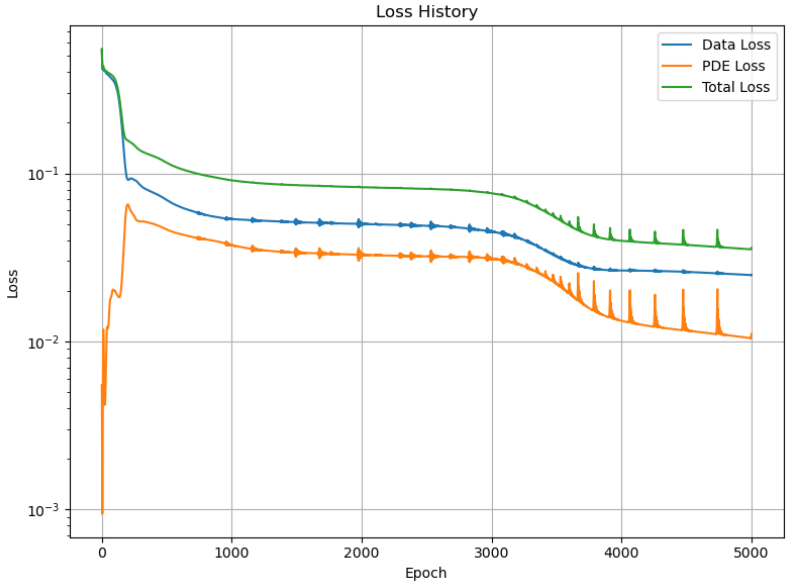
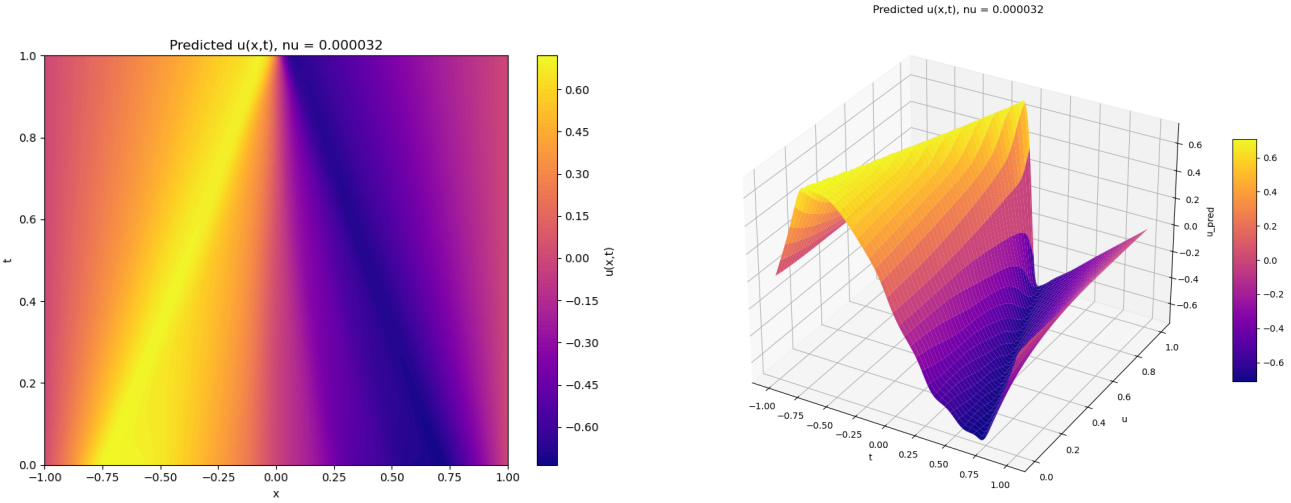


Figure 2: Loss for Case 1: High Viscosity

For the first case, the FFNN was trained on the highest viscocity case. The loss function converged to a value of $10^{-}1.4$ after approximately 4500 epochs. The prediction gradient and 3d plot of the prediction are shown in Figure 1. Since this function does not have a closed form solution, the prediction is compared to a numerical solution. Using the gradient, it is clear that the boundary and initial conditions are satisfied and is a smooth and continuous function expected for a viscous fluid. This case is the most difficult to learn, as the viscocity is high and there is a signfiicant jump discontinuity in the solution.

2.2.2 Case 2: Low Viscosity



(a) Prediction Gradient for Case 2: Low Viscosity (b) Prediction for Case 2: Low Viscosity

Figure 3: Prediction for Case 2: Low Viscosity

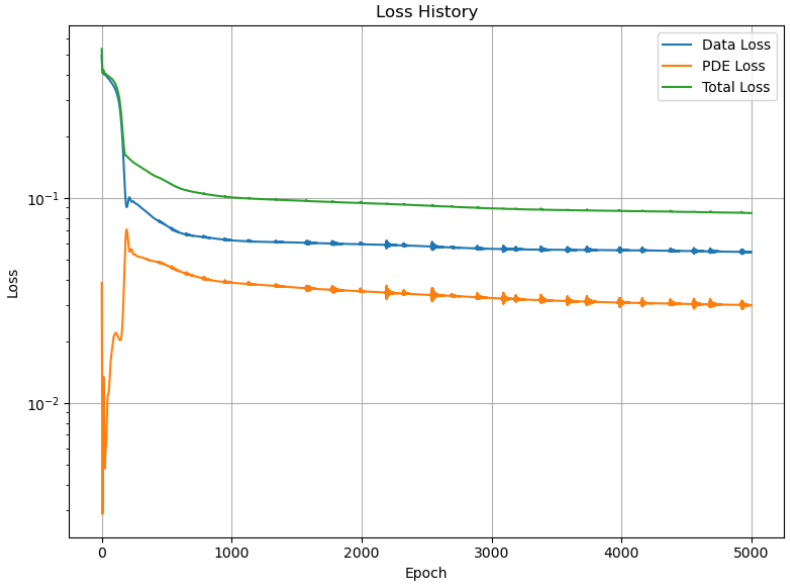
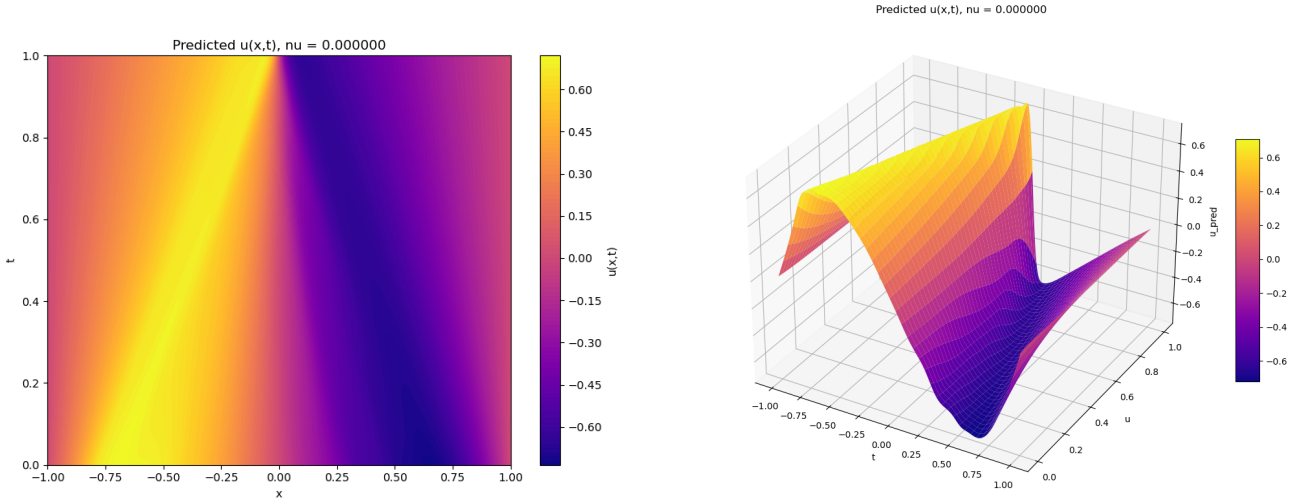


Figure 4: Loss for Case 2: Low Viscosity

For the second case, the FFNN was trained on the low viscocity case. The loss function converged to a value of $10^{-}1.2$ after approximately 3000 epochs. The prediction gradient and 3d plot of the prediction are shown in Figure 3. The prediction is much smoother than the first case and does not have a jump discontinuity. This case is expected to be easier to learn, despite a higher total loss.

2.2.3 Case 3: No Viscosity



(a) Prediction Gradient for Case 3: No Viscosity

(b) Prediction for Case 3: No Viscosity

Figure 5: Prediction for Case 3: No Viscosity

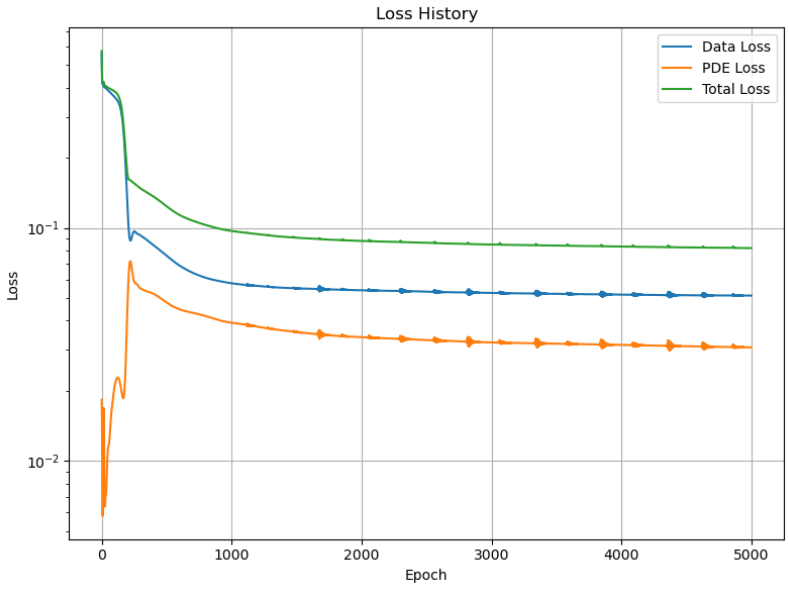


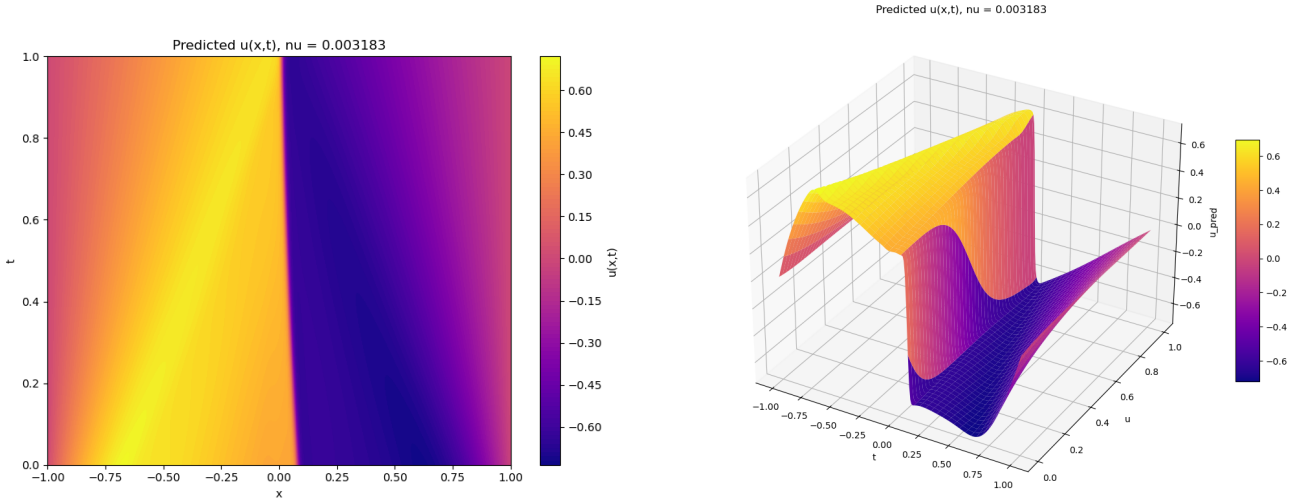
Figure 6: Loss for Case 3: No Viscosity

For the third case, the FFNN was trained on the no viscocity case. The loss function converged to a value of $10^{-}1.2$ after approximately 2000 epochs. The prediction gradient and 3d plot of the prediction are shown in Figure 5. The non-viscous case also matched well with numerical solutions. The prediction is smooth and continous, as expected.

2.3 Results - LR Annealing

For the second set of cases, the FFNN was trained with a learning rate annealing approach. Using, $\lambda = 0.9$ the following results are obtained.

2.3.1 Case 1: High Viscosity



(a) Prediction Gradient for Case 1: High Viscosity

(b) Prediction for Case 1: High Viscosity

Figure 7: Prediction for Case 1: High Viscosity

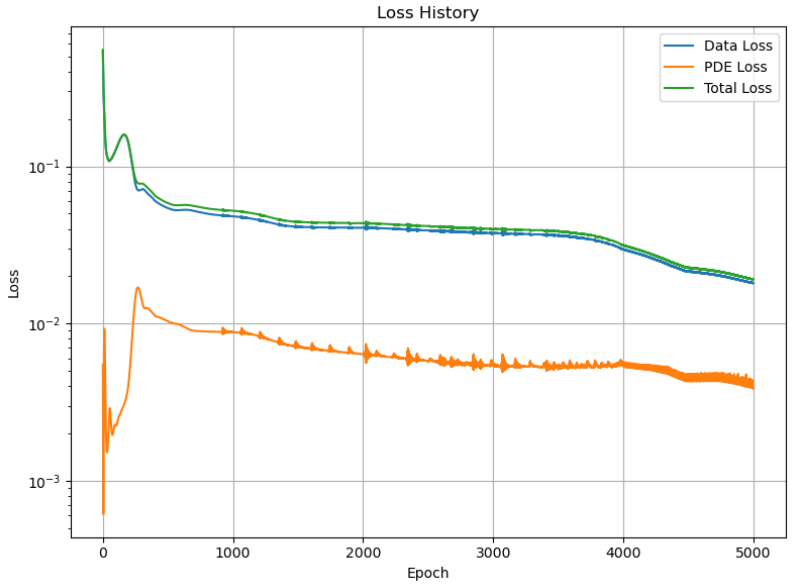
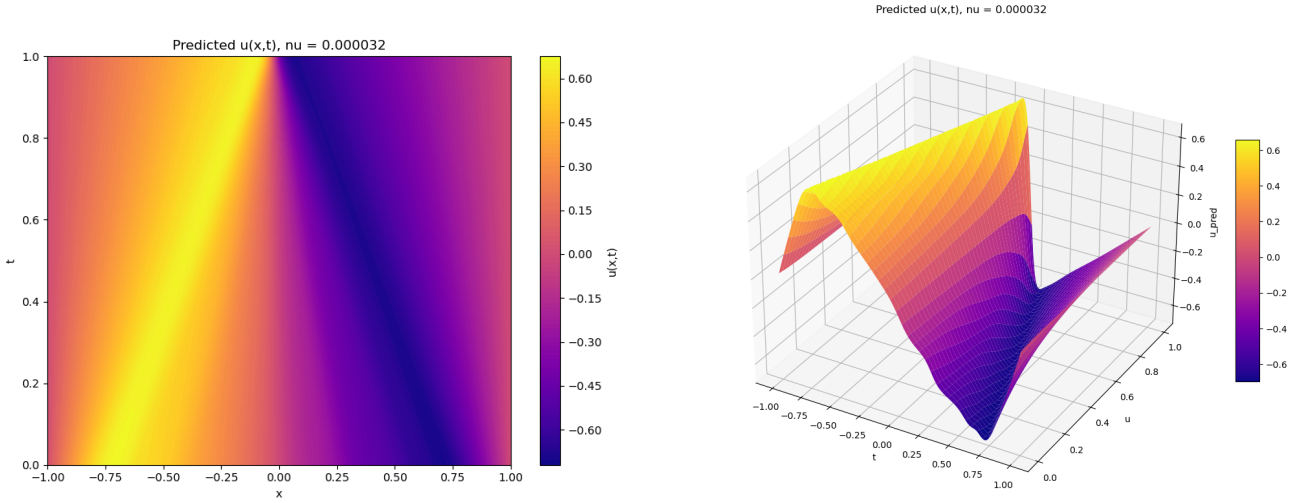


Figure 8: Loss for Case 1: High Viscosity

Compared to the case without annealing, the FFNN was able to converge to a smoother loss function with the data loss function converging significantly faster and closer to the total loss of $10^{-}1.7$. The prediction gradient and 3d plot of the prediction are shown in Figure 7. The prediction is much smoother than the first case and still captures the jump discontinuity.

2.3.2 Case 2: Low Viscosity



(a) Prediction Gradient for Case 2: Low Viscosity

(b) Prediction for Case 2: Low Viscosity

Figure 9: Prediction for Case 2: Low Viscosity

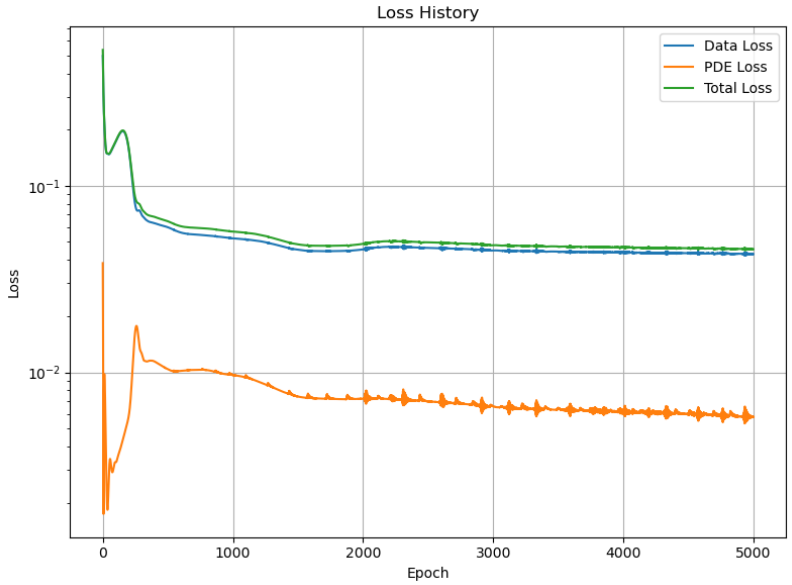
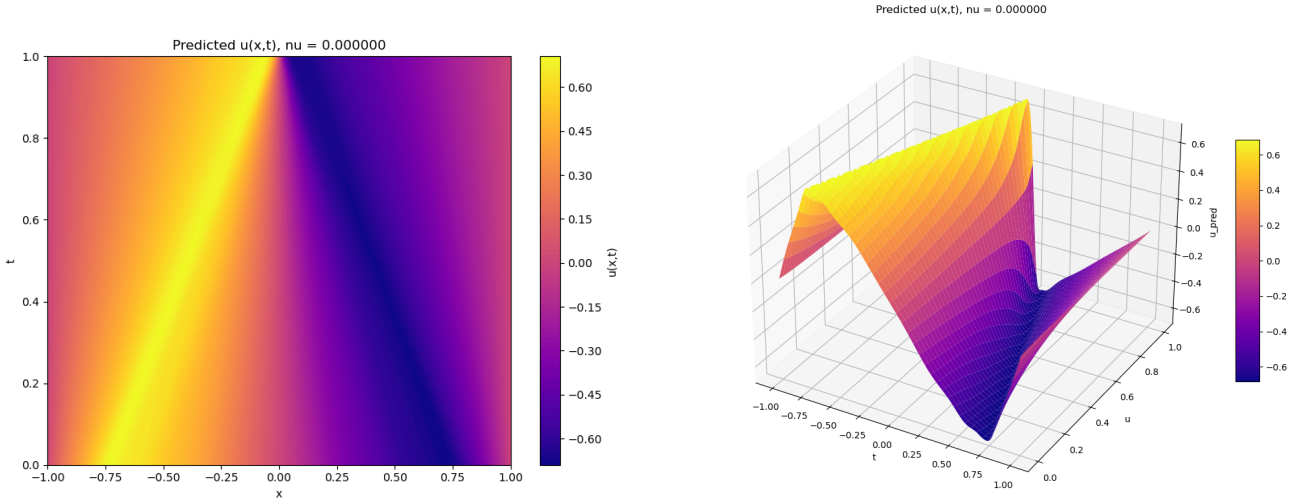


Figure 10: Loss for Case 2: Low Viscosity

Comparing the second case with and without learning rate annealing, the FFNN was yet again able to converge to a smoother loss function with less epochs. The data loss function converged to a value of $10^{-}1.5$ after approximately 2000 epochs.

2.3.3 Case 3: No Viscosity



(a) Prediction Gradient for Case 3: No Viscosity

(b) Prediction for Case 3: No Viscosity

Figure 11: Prediction for Case 3: No Viscosity

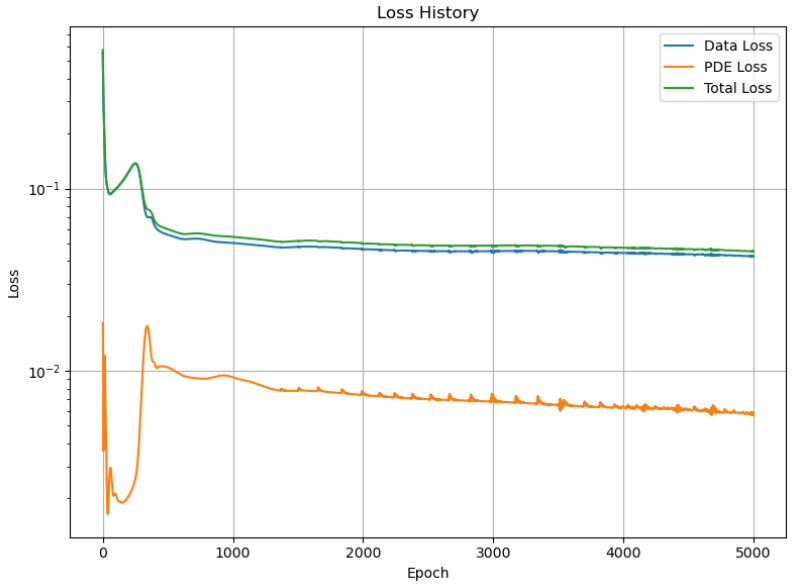


Figure 12: Loss for Case 3: No Viscosity

Comparing the third case with and without learning rate annealing, yet again the FFNN was able to converge to a smoother loss function with less epochs. The data function converged to a value of $10^{-}1.5$ after approximately 2500 epochs.

2.4 Conclusion

Overall, the FFNN was able to learn the underlying physics of the system and make accurate predictions based on the PDE and boundary conditions. The use of learning rate annealing significantly improved the speed of convergence and the smoothness of the loss function. Comparing to a numerical solution, the FFNN performed well and was able to capture the jump discontinuity in the viscous case.

3 2D Wave Equation

For this problem, the same domain as the previous case is used, but the wave equation is given by:

$$u_{tt} = c^2(u_{xx} + u_{yy}) \tag{8}$$

where c is the wave speed. The closed form solution is given by:

$$u(x, y, t) = \sin(k\pi x)\sin(k\pi y)\cos(\omega t) \tag{9}$$

where k is the wave number and ω is the angular frequency given by:

$$\omega = \sqrt{2}c\pi k \tag{10}$$

The following parameters are used for the 2D wave equation:

- k = $\{2, 10, 25\}$
- Epochs = 10000
- LearningRate = $5 \cdot 10^{-4}$
- Neurons = 50
- $N_i = 5000$ (interior points)
- $N_b = 256$ (boundary points)
- $N_{ic} = 256$ (initial condition points)
- $N_{bc} = 256$ (boundary condition points)

The NN will be trained on the interior points, boundary points and initial conditions. For the wave equation, since the closed form solution is known, the loss function is defined by:

$$L = L_{pde} + L_{ic} + L_{bc} \tag{11}$$

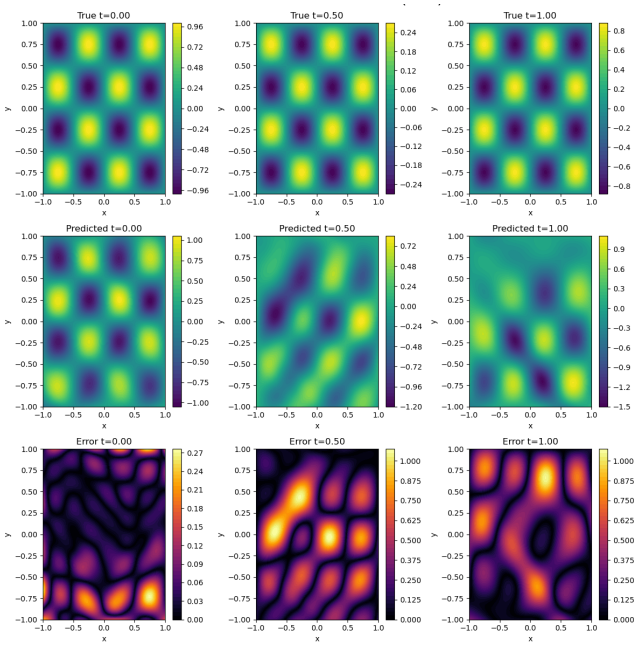
where,

$$L_{pde} = u_{tt} - c^2(u_{xx} + u_{yy}) \tag{12}$$

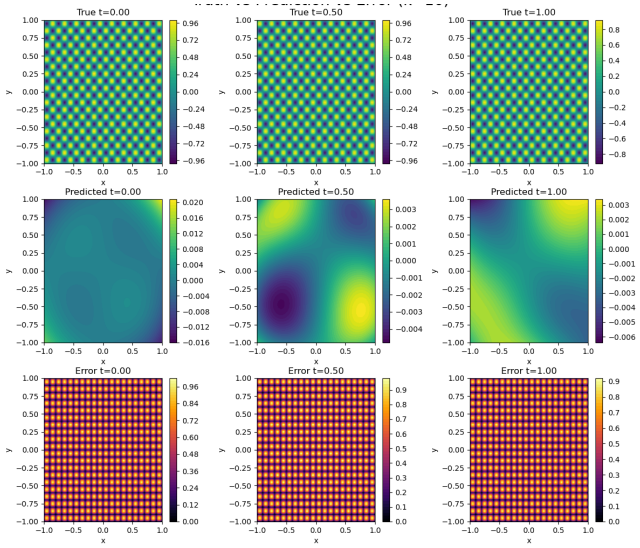
$$L_{ic} = \frac{1}{N}\sum_{i=1}^{N}(\theta(u|x_{ic}, y_{ic}, t_{ic}) - u_{ic})^2 \tag{13}$$

$$L_{bc} = \frac{1}{N}\sum_{i=1}^{N}(\theta(u|x_{bc}, y_{bc}, t_{bc}) - u_{bc})^2 \tag{14}$$

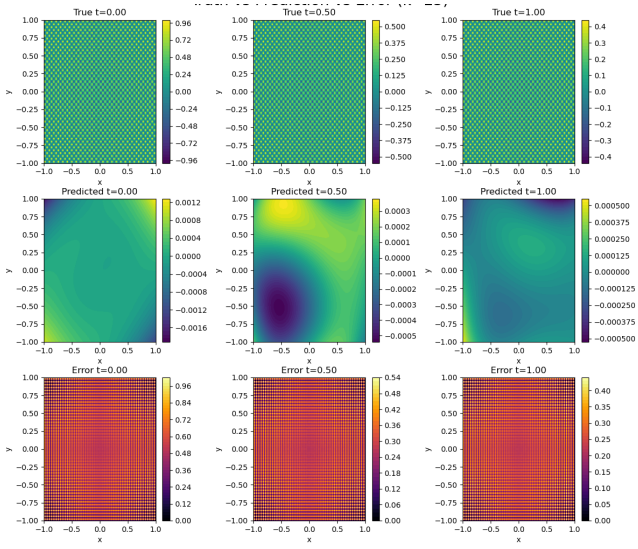
3.1 Results - Tanh Activation



(a) Error for Case 1: k = 2

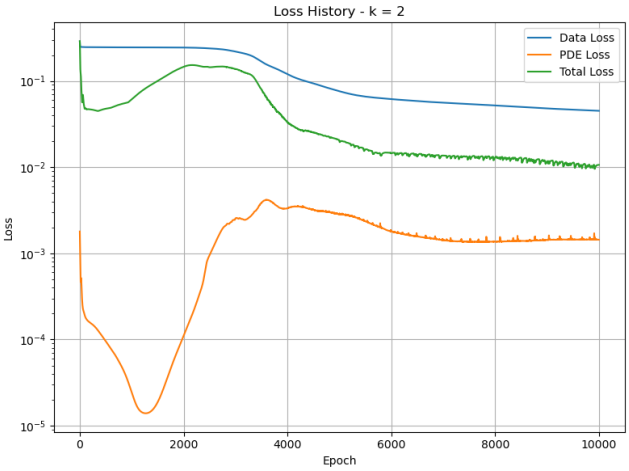


(b) Error for Case 2: k = 10

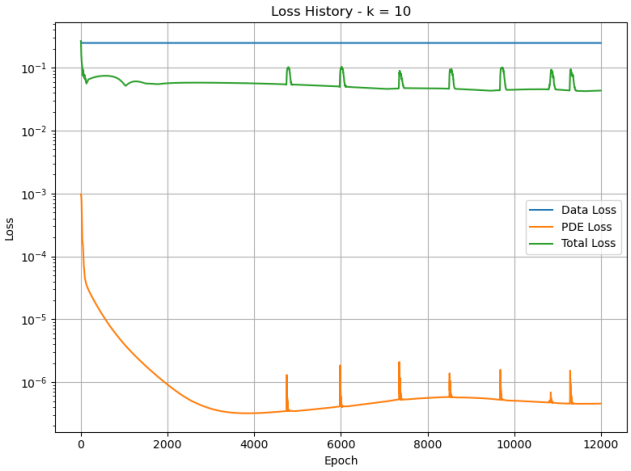


(c) Error for Case 3: k = 25

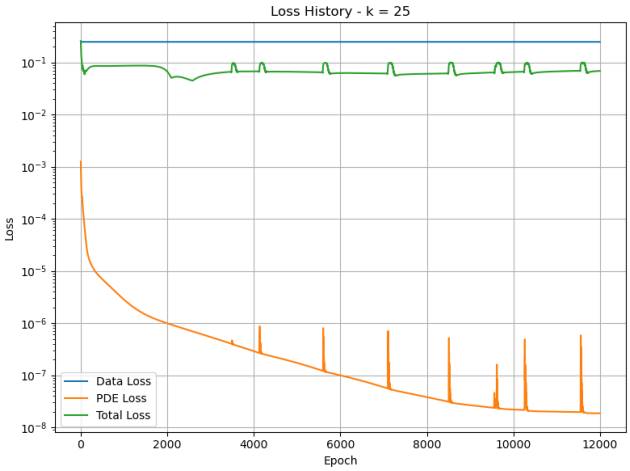
Figure 13: Error using Tanh Activation



(a) Loss for Case 1: k = 2

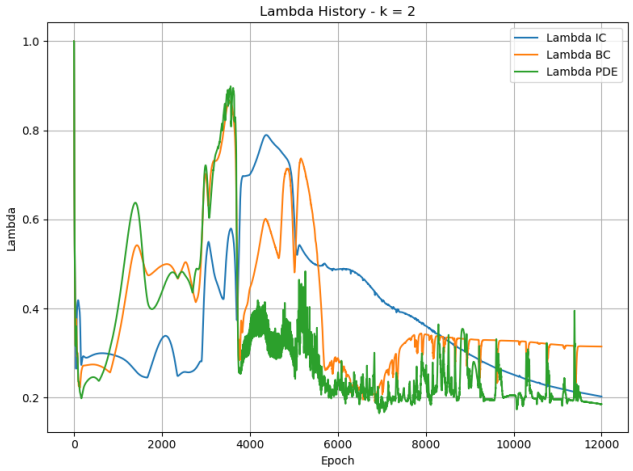


(b) Loss for Case 2: k = 10

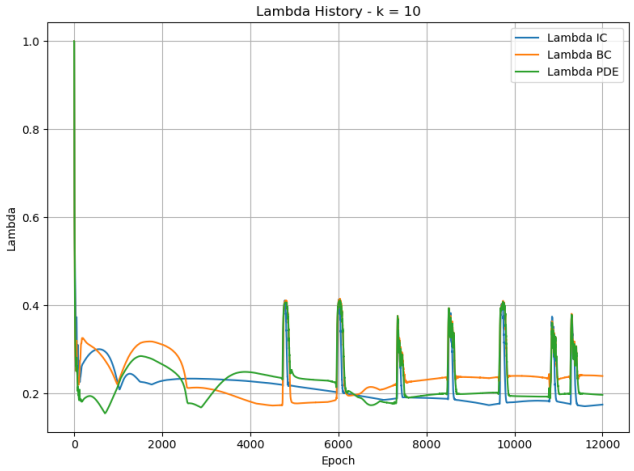


(c) Loss for Case 3: k = 25

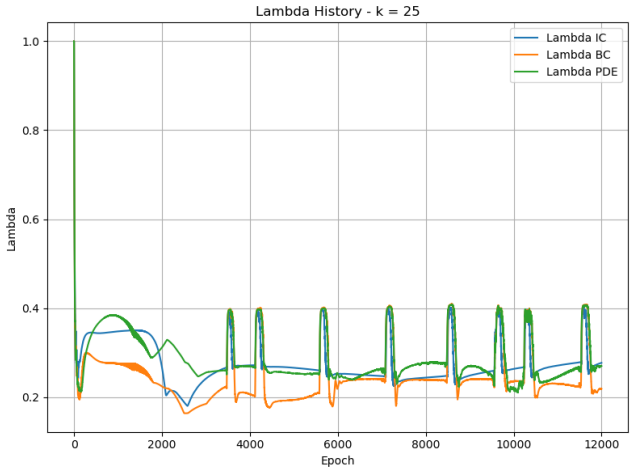
Figure 14: Loss Function using Tanh Activation



(a) Lambda for Case 1: k = 2



(b) Lambda for Case 2: k = 10



(c) Lambda for Case 3: k = 25

Figure 15: Lambda for Case 3: No Viscosity

For the 2D wave equation, the FFNN was able to learn the underlying physics of the first case where $k = 2$ decently well. The loss function converged to a value of 10^{-2}. The prediction gradient and error plot in Figure 13 show that the prediction matches the initial conditions and boundary conditions well. As time progresses, the prediction becomes less accurate but still captures the expected behavior of the wave equation.

As the wave number increases, the FFNN has a harder time learning the underlying physics of the system. For both the $k = 10$ and $k = 25$ cases, the FFNN converged to a loss of $10^{-1.2}$. The prediction gradient and error plot in Figure 13 show that the prediction does not match the initial conditions but does match the boundary conditions. With this, the PDE loss is very low compared to the data loss. The error seen in the prediction is likely due to the fact that the wave number is too high for the FFNN to learn the initial conditions well enough.

Without the initial conditions, the FFNN will not be able to learn the underlying physics of the system at all. The start conditions can be thought of as the initial velocity of the wave. That wave behaves differently depending on the initial velocity and cannot be learned without it. Shown in the loss function, the initial conditions loss converges to a value of exactly 0.2480 for both the $k = 10$ and $k = 25$ cases and the PDE loss converges to a value of $10^{-6.4}$ and $10^{-7.8}$ respectively.

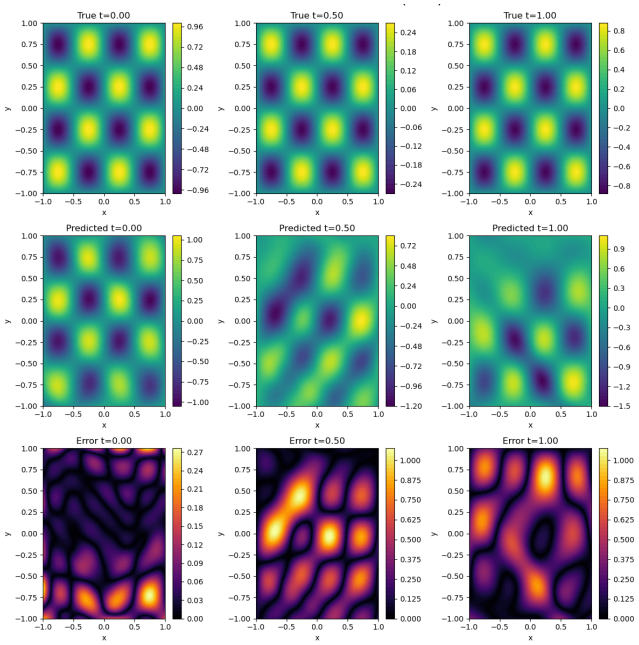
3.2 Results - Rowdy Activation

Since the traditional tanh activation function did not perform well for the 2D wave equation, a new activation function was considered. The Rowdy Activation Function (Deep Kronecker neural network) is given by:

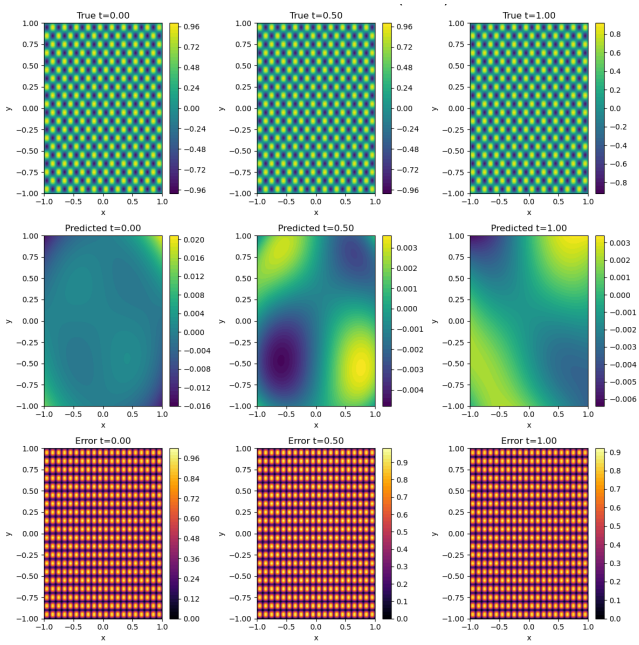
$$\phi_1 = \frac{e^z - e^{-z}}{e^z + e^{-z}} \equiv \tanh(z) \tag{15}$$

$$\phi_2 = na\sin((k-1)nx) \tag{16}$$

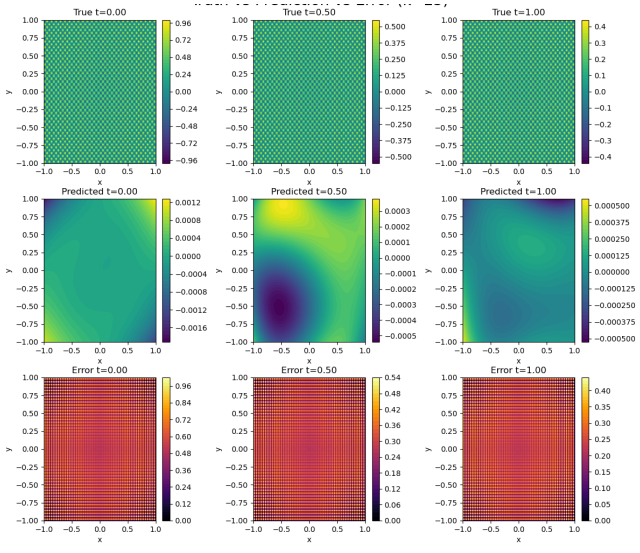
$$\phi_3 = nb\sin((k-1)nx) \tag{17}$$



(a) Error for Case 1: k = 2

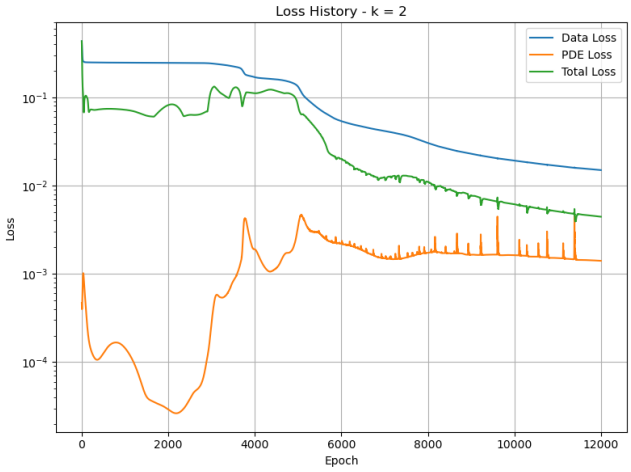


(b) Error for Case 2: k = 10

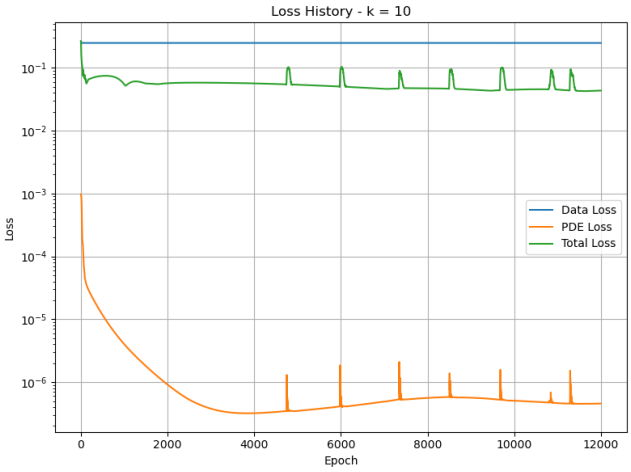


(c) Error for Case 3: k = 25

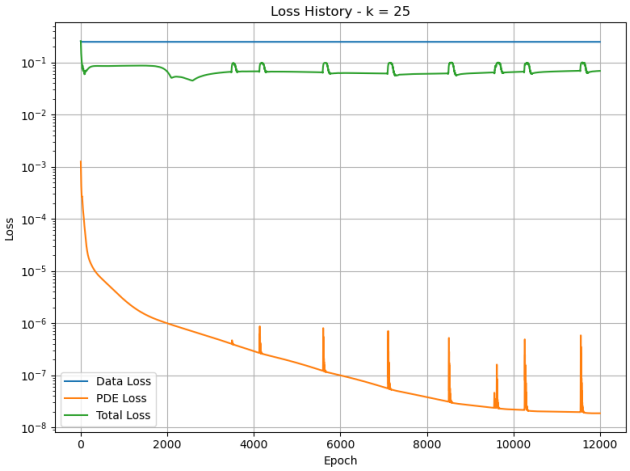
Figure 16: Error using Rowdy Activation



(a) Loss for Case 1: k = 2

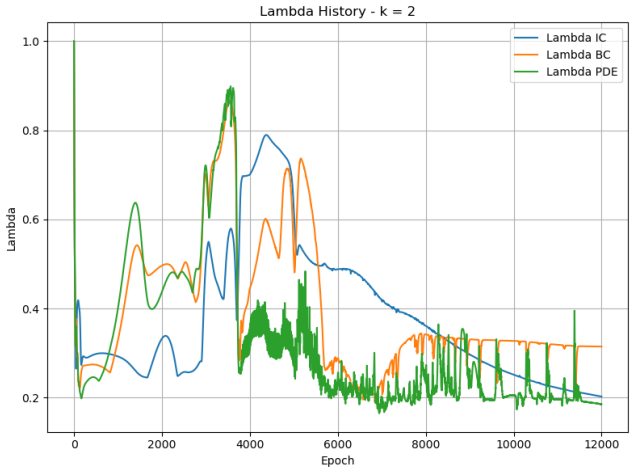


(b) Loss for Case 2: k = 10

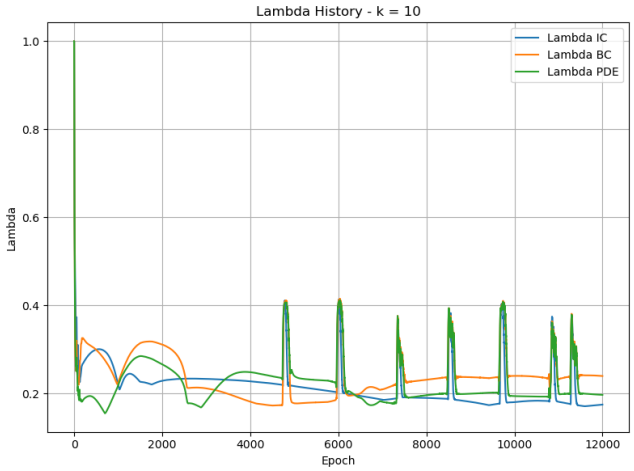


(c) Loss for Case 3: k = 25

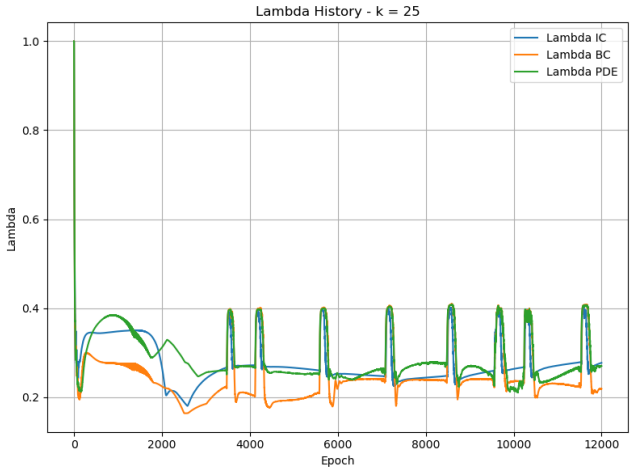
Figure 17: Loss Function using Rowdy Activation



(a) Loss for Case 1: k = 2



(b) Loss for Case 2: k = 10



(c) Loss for Case 3: k = 25

Figure 18: Lambda using Rowdy Activation

Using the Rowdy activation function, the FFNN was able to achieve slightly better results for case 1 with $k = 2$. The loss function reached a value of $10^{-2.5}$ and still descending. Given more epochs, the FFNN would likely converge to a lower loss function but from this point was a relatively small gradient. The prediction and error plot in Figure 16 show that the prediction matches the initial conditions and boundary conditions well. Unlike the tanh activation function, the Rowdy activation function was able to learn the initial conditions well and through the entire domain.

Despite the better results for case 1, the FFNN still was unable to learn the underlying physics of the system for the $k = 10$ and $k = 25$ cases. Yet again, the initial conditions are still converging to a value of eitherr 0.2480 or 0.2490. The PDE loss is still very low compared to the data loss. Using the Rowdy activation function, the PDE loss converged to a value less than that of the tanh activation function.

4 Conclusion

In this assignment, a FFNN was trained on the 1D viscous Burgers equation as well as the 2D wave equation. While the FFNN did a good job learning the 1D viscous Burgers equation, it clearly struggled with the higher wave numbers in the 2D wave function. This assignment was a great introduction to the use of PINNs and provided a great baseline for future work into the final project. Even for the 1D viscous Burgers equation, where the exact closed form solution is not known, the PINN was able to learn the

equation well and provide an approximation comparable to a traditional numerical solution. The use of learning rate annealing proved to be very useful in faster convergence and left all cases with a smoother loss function.

Despite the success of the PINN for the 1D viscous Burgers equation and Case 1 of the 2D wave equation, the PINN still struggled with the higher wave numbers. The use of the Rowdy activation function did help with the convergence of the loss function but still did not improve the solution for the higher wave numbers. Still the major issue with the PINN is the significant dependence on the initial conditions. Without the initial conditions, the PINN will not be able to learn the full phyiscs of the system. This is true for all PDEs and even effects numerical solutions, not just PINNs, but this particular case is very sensitive to the initial conditions - unveiling the limitations of the PINN approach. Potential future work could include the use of a different base activation function, a different loss function, or further tunning of the hyperparameters despite current efforts returning no significant improvements.

5 1dBurgers.py

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

# === Seed & Device ===
seed = 45
np.random.seed(seed)
torch.manual_seed(seed)

device = torch.device("mps" if torch.backends.mps.is_available()
                      else "cuda" if torch.cuda.is_available()
                      else "cpu")
print(f"Using device: {device}")

# === Hyperparameters ===
numEpochs = 5000
learningRate = 0.001
numInt = 5000
numIC = 256
numBC = 256

# === Model ===
class PINN1DBurgers(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 50),
            nn.Tanh(),
            nn.Linear(50, 50),
            nn.Tanh(),
            nn.Linear(50, 1)
        )

    def forward(self, x, t):
        return self.net(torch.cat([x, t], dim=1))

# === Dataset Generators ===
def generate_uniform_pts(num):
    x = torch.rand((num, 1)) * 2 - 1  # x -> [-1, 1]
    t = torch.rand((num, 1))          # t -> [0, 1]
    return x.to(device), t.to(device)

def generate_initial_conditions(num):
    x = torch.linspace(-1, 1, num).view(-1, 1)
    t = torch.zeros_like(x)
    u = -torch.sin(np.pi * x)
    return x.to(device), t.to(device), u.to(device)

def generate_boundary_conditions(num):
    t = torch.linspace(0, 1, num).view(-1, 1)
    x_left = -torch.ones_like(t)
    x_right = torch.ones_like(t)
    u = torch.zeros_like(t)

    return (
        torch.cat([x_left, x_right]).to(device),
```

```
        torch.cat([t, t]).to(device),
        torch.cat([u, u]).to(device)
    )

# === Loss Functions ===
def pde_loss(model, x, t, nu):
    x.requires_grad_(True)
    t.requires_grad_(True)

    u = model(x, t)
    u_t = torch.autograd.grad(u, t, grad_outputs=torch.ones_like(u),
        create_graph=True)[0]
    u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u),
        create_graph=True)[0]
    u_xx = torch.autograd.grad(u_x, x, grad_outputs=torch.ones_like(u_x),
        create_graph=True)[0]

    res = u_t + u * u_x - nu * u_xx
    return torch.mean(res**2)

# Initial Conditions
def initial_loss(model, x_ic, t_ic, u_ic):
    u_pred = model(x_ic, t_ic)
    return torch.mean((u_pred - u_ic) ** 2)

# Boundary Conditions
def boundary_loss(model, x_bc, t_bc, u_bc):
    u_pred = model(x_bc, t_bc)
    return torch.mean((u_pred - u_bc) ** 2)

# Learning Rate Annealing
def computeLambdaHat(loss):
    optimizer.zero_grad()
    loss.backward(retain_graph=True)

    grads = []
    for p in model.parameters():
        if p.grad is not None:
            grads.append(p.grad.view(-1).detach().abs())

    grads = torch.cat(grads)
    return grads.max() / grads.norm()

# Allocate memory for loss history
dataLossHist  = []
pdeLossHist   = []
totalLossHist = []

# Allocate memory for lambda history
lambda_ic_hist = []
lambda_bc_hist = []
lambda_pde_hist = []

# === Training ===
def train(model, optimizer, nu, x_int, t_int, x_ic, t_ic, u_ic, x_bc, t_bc, u_bc,
    useAnnealing=False):
    # === Initialize dynamic weights ===
    lambda_ic = torch.tensor(1.0).to(device)
    lambda_bc = torch.tensor(1.0).to(device)
    lambda_pde = torch.tensor(1.0).to(device)
    alpha = 0.9

```

```
    # === Initialize loss history ===
    dataLossHist.clear()
    pdeLossHist.clear()
    totalLossHist.clear()
    lambda_ic_hist.clear()
    lambda_bc_hist.clear()
    lambda_pde_hist.clear()

    for epoch in range(numEpochs):
        optimizer.zero_grad()

        loss_pde = pde_loss(model, x_int, t_int, nu)
        loss_ic = initial_loss(model, x_ic, t_ic, u_ic)
        loss_bc = boundary_loss(model, x_bc, t_bc, u_bc)

        if epoch > 1 and useAnnealing:
            lambda_ic_hat = computeLambdaHat(loss_ic)
            lambda_bc_hat = computeLambdaHat(loss_bc)
            lambda_pde_hat = computeLambdaHat(loss_pde)

            # Update lambda using moving average
            lambda_ic = alpha * lambda_ic + (1 - alpha) * lambda_ic_hat
            lambda_bc = alpha * lambda_bc + (1 - alpha) * lambda_bc_hat
            lambda_pde = alpha * lambda_pde + (1 - alpha) * lambda_pde_hat

        loss_data = lambda_ic * loss_ic + lambda_bc * loss_bc
        total_loss = lambda_pde * loss_pde +  loss_data

        # Store loss history
        dataLossHist.append(loss_data.item())
        pdeLossHist.append(loss_pde.item())
        totalLossHist.append(total_loss.item())

        # Store lambda history
        lambda_ic_hist.append(lambda_ic.item())
        lambda_bc_hist.append(lambda_bc.item())
        lambda_pde_hist.append(lambda_pde.item())

        total_loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print(f"[nu={nu:.5f}] Epoch {epoch:>4}: "
                  f"Total={total_loss.item():.6f}, PDE={loss_pde.item():.6f}, "
                  f"Data={loss_data.item():.6f}")

            print(f"lambda_ic: {lambda_ic.item():.6f}, "
                  f"lambda_bc: {lambda_bc.item():.6f}, "
                  f"lambda_pde: {lambda_pde.item():.6f}")

# === Inference & Plot ===
def plot_solution(model, title="Predicted u(x,t)"):
    x = torch.linspace(-1, 1, 256).view(-1, 1).to(device)
    t = torch.linspace(0, 1, 256).view(-1, 1).to(device)
    X, T = torch.meshgrid(x.squeeze(), t.squeeze(), indexing="ij")
    x_flat = X.reshape(-1, 1)
    t_flat = T.reshape(-1, 1)

    with torch.no_grad():
        u_pred = model(x_flat, t_flat).cpu().numpy().reshape(256, 256)

    # === Plot Prediction ===
```

```
    plt.figure(figsize=(8, 6))
    plt.contourf(X.cpu(), T.cpu(), u_pred, levels=100, cmap='plasma')
    plt.colorbar(label='u(x,t)')
    plt.xlabel("x")
    plt.ylabel("t")
    plt.title(title)
    plt.tight_layout()
    plt.show()

    # === Plot Prediction ===
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(X.cpu().numpy(), T.cpu().numpy(), u_pred, cmap='plasma',
        edgecolor='none')
    ax.set_xlabel("t")
    ax.set_ylabel("u")
    ax.set_zlabel("u_pred")
    ax.set_title(title)
    fig.colorbar(surf, ax=ax, shrink=0.5, aspect=10)
    plt.tight_layout()
    plt.show()

    # === Plot Loss History ===
    plt.figure(figsize=(8, 6))
    plt.semilogy(dataLossHist, label='Data Loss')
    plt.semilogy(pdeLossHist, label='PDE Loss')
    plt.semilogy(totalLossHist, label='Total Loss')
    plt.yscale('log')
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Loss History")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()

# === Main Loop ===
if __name__ == "__main__":
    nu_values = [0.01 / np.pi, 0.0001 / np.pi, 0.0]

    for nu in nu_values:
        print(f"\n--- Training for nu = {nu:.6f} ---")

        model = PINN1DBurgers().to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=learningRate)

        x_int, t_int = generate_uniform_pts(numInt)
        x_ic, t_ic, u_ic = generate_initial_conditions(numIC)
        x_bc, t_bc, u_bc = generate_boundary_conditions(numBC)

        train(model, optimizer, nu, x_int, t_int, x_ic, t_ic, u_ic, x_bc, t_bc, u_bc,
            useAnnealing=True)

        model_path = f"pinn_burgers_nu{nu:.6f}.pth"
        torch.save(model.state_dict(), model_path)

        plot_solution(model, title=f"Predicted u(x,t), nu = {nu:.6f}")
```

6 2dWave.py

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation

# === Seed & Device ===
seed = 45
np.random.seed(seed)
torch.manual_seed(seed)

device = torch.device("mps" if torch.backends.mps.is_available()
                      else "cuda" if torch.cuda.is_available()
                      else "cpu")
print(f"Using device: {device}")

# === Hyperparameters ===
numEpochs = 10000
learningRate = 0.0002
numInt = 5000
numIC = 512
numBC = 512

# === PDE Parameters ===
kCases = [25]
c = 1.0

# === Rowdy Activation ===
class Rowdy(nn.Module):
    def __init__(self, features):
        super().__init__()
        # Parameters for phi_2 and phi_3
        self.w = nn.Parameter(torch.randn(features))
        self.b = nn.Parameter(torch.randn(features))
        self.v = nn.Parameter(torch.randn(features))
        self.c = nn.Parameter(torch.randn(features))
        self.alpha = nn.Parameter(torch.ones(features) * 0.1)
        self.beta = nn.Parameter(torch.ones(features) * 0.1)

    def forward(self, x):
        # Base activation
        phi1 = torch.tanh(x)
        # Sinusoidal Rowdyness
        phi2 = self.alpha * torch.cos(self.w * x + self.b)
        phi3 = self.beta * torch.sin(self.v * x + self.c)
        return phi1 + phi2 + phi3

# === Model ===
class PINN2DWave(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 50),
            nn.Tanh(),
            nn.Linear(50, 50),
            nn.Tanh(),
            nn.Linear(50, 1)
```

```
        )

    def forward(self, x, y, t):
        return self.net(torch.cat([x,y,t], dim=1))

# === Model with Rowdy ===
class PINN2DWave_Rowdy(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 50),
            Rowdy(50),
            nn.Linear(50, 50),
            Rowdy(50),
            nn.Linear(50, 1)
        )

    def forward(self, x, y, t):
        return self.net(torch.cat([x, y, t], dim=1))

# === Dataset Generators ===
def generate_points(n_int, n_bdy, k):
    # Interior Points
    x_int = torch.rand((n_int, 1), device=device, requires_grad=True) * 2 - 1 # x -> [-1, 1]
    y_int = torch.rand((n_int, 1), device=device, requires_grad=True) * 2 - 1 # y -> [-1, 1]
    t_int = torch.rand((n_int, 1), device=device, requires_grad=True)          # t -> [0, 1]

    # Boundary (square edges)
    t_bdy = torch.rand((n_bdy // 4, 1))

    xb = torch.cat([
        torch.rand((n_bdy // 4, 1)) * 2 - 1,  # x on bottom edge
        torch.rand((n_bdy // 4, 1)) * 2 - 1,  # x on top edge
        -torch.ones((n_bdy // 4, 1)),         # x = -1 (left edge)
        torch.ones((n_bdy // 4, 1))           # x = 1 (right edge)
    ])

    yb = torch.cat([
        -torch.ones((n_bdy // 4, 1)),         # y = -1 (bottom)
        torch.ones((n_bdy // 4, 1)),          # y = 1 (top)
        torch.rand((n_bdy // 4, 1)) * 2 - 1,  # y left edge
        torch.rand((n_bdy // 4, 1)) * 2 - 1   # y right edge
    ])

    tb = torch.cat([t_bdy, t_bdy, t_bdy, t_bdy])

    omega = np.sqrt(2) * c * np.pi * k

    ub = torch.sin(k*np.pi*xb) * torch.sin(k*np.pi*yb) * torch.cos(omega*tb)

    return x_int.to(device), y_int.to(device), t_int.to(device), xb.to(device), yb.to(device), tb.to(device), ub.to(device)

def generate_initial_conditions(num_points_per_dim, k):
    x = torch.linspace(-1, 1, num_points_per_dim)
    y = torch.linspace(-1, 1, num_points_per_dim)
    X, Y = torch.meshgrid(x, y, indexing='ij')

    X_flat = X.reshape(-1, 1)
```

```
    Y_flat = Y.reshape(-1, 1)
    T_flat = torch.zeros_like(X_flat, requires_grad=True)

    # Initial condition: u(x, y, 0) = sin(k pi x) sin(k pi y) cos(omega t) (t = 0)
    U_flat = torch.sin(k * np.pi * X_flat) * torch.sin(k * np.pi * Y_flat)

    return X_flat.to(device), Y_flat.to(device), T_flat.to(device), U_flat.to(device)

# === Loss Functions ===
# == PDE Loss ==
def pdeLoss(model, x, y, t, k):
    u = model(x, y, t)

    u_t = torch.autograd.grad(u, t, grad_outputs=torch.ones_like(u), create_graph=True,
        retain_graph=True)[0]
    u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u), create_graph=True,
        retain_graph=True)[0]
    u_y = torch.autograd.grad(u, y, grad_outputs=torch.ones_like(u), create_graph=True,
        retain_graph=True)[0]

    u_xx = torch.autograd.grad(u_x, x, grad_outputs=torch.ones_like(u_x),
        create_graph=True, retain_graph=True)[0]
    u_yy = torch.autograd.grad(u_y, y, grad_outputs=torch.ones_like(u_y),
        create_graph=True, retain_graph=True)[0]
    u_tt = torch.autograd.grad(u_t, t, grad_outputs=torch.ones_like(u_t),
        create_graph=True, retain_graph=True)[0]

    u_pred = u_tt - np.pow(c, 2) * (u_xx + u_yy)

    return torch.mean((u_pred)**2)

# == Boundary Loss ==
def boundary_loss(model, x_b, y_b, t_b, u_b):
    ub_pred = model(x_b, y_b, t_b)

    return torch.mean((ub_pred - u_b)**2)

# == Initial Condition Loss ==
def intitial_loss(model, x_ic, y_ic, t_ic, u_ic):
    ui_pred = model(x_ic, y_ic, t_ic)

    return torch.mean((ui_pred - u_ic) ** 2)

# Allocate memory for loss history
dataLossHist  = []
pdeLossHist   = []
totalLossHist = []

# Allocate memory for lambda history
lambdaHistIC = []
lambdaHistBC = []
lambdaHistPDE = []

# Learning Rate Annealing
def computeLambdaHat(loss):
    loss.backward(retain_graph=True)

    grads = []
    for p in model.parameters():
        if p.grad is not None:
            grads.append(p.grad.view(-1).detach().abs())

```

```
    grads = torch.cat(grads)
    return grads.max() / grads.norm()

def train(model, optimizer, k, x_int, y_int, t_int,
          x_ic, y_ic, t_ic, u_ic,
          x_bc, y_bc, t_bc, u_bc,
          useAnnealing=False):

    dataLossHist.clear()
    pdeLossHist.clear()
    totalLossHist.clear()

    lambdaHistIC.clear()
    lambdaHistBC.clear()
    lambdaHistPDE.clear()

    lambda_ic = torch.tensor(1.0).to(device)
    lambda_bc = torch.tensor(0.5).to(device)
    lambda_pde = torch.tensor(0.5).to(device)
    alpha = 0.9

    model.train()

    for epoch in range(numEpochs):
        optimizer.zero_grad()

        # === PDE Loss ===
        loss_pde = pdeLoss(model, x_int, y_int, t_int, k)

        # === Initial Condition Loss ===
        loss_ic = intitial_loss(model, x_ic, y_ic, t_ic, u_ic)

        # === Boundary Condition Loss ===
        loss_bc = boundary_loss(model, x_bc, y_bc, t_bc, u_bc)

        # === Annealing ===
        if epoch > 1 and useAnnealing:
            lambda_ic_hat = computeLambdaHat(loss_ic)
            lambda_bc_hat = computeLambdaHat(loss_bc)
            lambda_pde_hat = computeLambdaHat(loss_pde)

            # Update lambda using moving average
            lambda_ic = alpha * lambda_ic + (1 - alpha) * lambda_ic_hat
            lambda_bc = alpha * lambda_bc + (1 - alpha) * lambda_bc_hat
            lambda_pde = alpha * lambda_pde + (1 - alpha) * lambda_pde_hat

        loss_data = lambda_ic * loss_ic + lambda_bc * loss_bc
        total_loss = lambda_pde * loss_pde + loss_data

        # Record loss history
        dataLossHist.append((loss_ic + loss_bc).item())
        pdeLossHist.append(loss_pde.item())
        totalLossHist.append(total_loss.item())

        # Record lambda history
        lambdaHistIC.append(lambda_ic.item())
        lambdaHistBC.append(lambda_bc.item())
        lambdaHistPDE.append(lambda_pde.item())

        # Backprop and step
        total_loss.backward(retain_graph=True)
        optimizer.step()
```

```
236
237         if epoch % 500 == 0:
238             print(f"Epoch {epoch}, Total Loss: {total_loss.item():.4f}, PDE Loss:
                    {loss_pde.item():.4f}, "
239                   f"IC Loss: {loss_ic.item():.4f}, BC Loss: {loss_bc.item():.4f}")
240
241             print(f"Lambda IC: {lambda_ic.item():.4f}, Lambda BC:
                    {lambda_bc.item():.4f}, Lambda PDE: {lambda_pde.item():.4f}")
242             print("========================================")
243
244 # === Inference & Plot ===
245 def plot_truth_pred_error(model, k, t_val=0.25):
246     model.eval()
247
248     x = torch.linspace(-1, 1, 100).reshape(-1, 1)
249     y = torch.linspace(-1, 1, 100).reshape(-1, 1)
250     X, Y = torch.meshgrid(x.squeeze(), y.squeeze(), indexing='ij')
251     X_flat = X.reshape(-1, 1).to(device)
252     Y_flat = Y.reshape(-1, 1).to(device)
253     T_flat = torch.full_like(X_flat, t_val).to(device)
254
255     omega = np.sqrt(2) * c * np.pi * k
256     u_true = torch.sin(k * np.pi * X_flat) * torch.sin(k * np.pi * Y_flat) *
            torch.cos(omega * T_flat)
257     with torch.no_grad():
258         u_pred = model(X_flat, Y_flat, T_flat)
259
260     u_true = u_true.cpu().numpy().reshape(100, 100)
261     u_pred = u_pred.cpu().numpy().reshape(100, 100)
262     error = np.abs(u_true - u_pred)
263
264     fig, axs = plt.subplots(1, 3, figsize=(18, 5))
265
266     c0 = axs[0].contourf(X.cpu().numpy(), Y.cpu().numpy(), u_true, levels=50,
            cmap='viridis')
267     axs[0].set_title(f'True Solution at t={t_val}')
268     axs[0].set_xlabel('x')
269     axs[0].set_ylabel('y')
270     fig.colorbar(c0, ax=axs[0])
271
272     c1 = axs[1].contourf(X.cpu().numpy(), Y.cpu().numpy(), u_pred, levels=50,
            cmap='viridis')
273     axs[1].set_title(f'Predicted Solution at t={t_val}')
274     axs[1].set_xlabel('x')
275     axs[1].set_ylabel('y')
276     fig.colorbar(c1, ax=axs[1])
277
278     c2 = axs[2].contourf(X.cpu().numpy(), Y.cpu().numpy(), error, levels=50,
            cmap='inferno')
279     axs[2].set_title(f'Absolute Error at t={t_val}')
280     axs[2].set_xlabel('x')
281     axs[2].set_ylabel('y')
282     fig.colorbar(c2, ax=axs[2])
283
284     plt.tight_layout()
285     plt.show()
286
287 def plot_solution(model, k):
288
289     # === Plot Loss History ===
290     plt.figure(figsize=(8, 6))
291     plt.semilogy(dataLossHist, label='Data Loss')
```

```
    plt.semilogy(pdeLossHist, label='PDE Loss')
    plt.semilogy(totalLossHist, label='Total Loss')
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title(f"Loss History - k = {k}")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()

    # === Plot Lambda History ===
    plt.figure(figsize=(8, 6))
    plt.plot(lambdaHistIC, label='Lambda IC')
    plt.plot(lambdaHistBC, label='Lambda BC')
    plt.plot(lambdaHistPDE, label='Lambda PDE')
    plt.xlabel("Epoch")
    plt.ylabel("Lambda")
    plt.title(f"Lambda History - k = {k}")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()

    plot_truth_pred_error_multiple_times(model, k, times=[0.0, 0.5, 1.0])

    animate_solution(model, k)

def animate_solution(model, k, num_frames=100):
    model.eval()

    x = torch.linspace(-1, 1, 100).reshape(-1, 1)
    y = torch.linspace(-1, 1, 100).reshape(-1, 1)
    X, Y = torch.meshgrid(x.squeeze(), y.squeeze(), indexing='ij')
    X_flat = X.reshape(-1, 1).to(device)
    Y_flat = Y.reshape(-1, 1).to(device)

    z_max = -float('inf')
    z_min = float('inf')
    for frame in range(num_frames):
        t_val = torch.full_like(X_flat, frame / num_frames).to(device)
        with torch.no_grad():
            u_pred = model(X_flat, Y_flat, t_val).cpu().numpy().reshape(100, 100)
        z_max = max(z_max, np.max(u_pred))
        z_min = min(z_min, np.min(u_pred))

    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')

    def update(frame):
        ax.clear()
        t_val = torch.full_like(X_flat, frame / num_frames).to(device)
        with torch.no_grad():
            u_pred = model(X_flat, Y_flat, t_val).cpu().numpy().reshape(100, 100)

        ax.plot_surface(X.cpu().numpy(), Y.cpu().numpy(), u_pred, cmap=cm.viridis)
        ax.set_zlim(z_min, z_max)
        ax.set_title(f't = {frame / num_frames:.2f} - k = {k}')
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('u(x, y, t)')

    ani = animation.FuncAnimation(fig, update, frames=num_frames, interval=100)
```

```
354     plt.show()
355 
356 def plot_truth_pred_error_multiple_times(model, k, times=[0.0, 0.25, 0.5, 0.75, 1.0]):
357     model.eval()
358 
359     x = torch.linspace(-1, 1, 100).reshape(-1, 1)
360     y = torch.linspace(-1, 1, 100).reshape(-1, 1)
361     X, Y = torch.meshgrid(x.squeeze(), y.squeeze(), indexing='ij')
362     X_flat = X.reshape(-1, 1).to(device)
363     Y_flat = Y.reshape(-1, 1).to(device)
364 
365     omega = np.sqrt(2) * c * np.pi * k
366 
367     n_times = len(times)
368     fig, axs = plt.subplots(3, n_times, figsize=(4 * n_times, 12))
369 
370     for idx, t_val in enumerate(times):
371         T_flat = torch.full_like(X_flat, t_val).to(device)
372 
373         # Exact solution
374         u_true = torch.sin(k * np.pi * X_flat) * torch.sin(k * np.pi * Y_flat) *
                torch.cos(omega * T_flat)
375         u_true = u_true.cpu().numpy().reshape(100, 100)
376 
377         # Prediction
378         with torch.no_grad():
379             u_pred = model(X_flat, Y_flat, T_flat).cpu().numpy().reshape(100, 100)
380 
381         # Error
382         error = np.abs(u_true - u_pred)
383 
384         # --- Plot True ---
385         c0 = axs[0, idx].contourf(X.cpu().numpy(), Y.cpu().numpy(), u_true, levels=50,
                cmap='viridis')
386         axs[0, idx].set_title(f'True t={t_val:.2f}')
387         axs[0, idx].set_xlabel('x')
388         axs[0, idx].set_ylabel('y')
389         fig.colorbar(c0, ax=axs[0, idx])
390 
391         # --- Plot Prediction ---
392         c1 = axs[1, idx].contourf(X.cpu().numpy(), Y.cpu().numpy(), u_pred, levels=50,
                cmap='viridis')
393         axs[1, idx].set_title(f'Predicted t={t_val:.2f}')
394         axs[1, idx].set_xlabel('x')
395         axs[1, idx].set_ylabel('y')
396         fig.colorbar(c1, ax=axs[1, idx])
397 
398         # --- Plot Error ---
399         c2 = axs[2, idx].contourf(X.cpu().numpy(), Y.cpu().numpy(), error, levels=50,
                cmap='inferno')
400         axs[2, idx].set_title(f'Error t={t_val:.2f}')
401         axs[2, idx].set_xlabel('x')
402         axs[2, idx].set_ylabel('y')
403         fig.colorbar(c2, ax=axs[2, idx])
404 
405     plt.tight_layout()
406     plt.suptitle(f"Truth vs Prediction vs Error (k={k})", fontsize=20, y=1.02)
407     plt.show()
408 
409 # === Main Loop ===
410 if __name__ == "__main__":
411 
```

```
412     # === Train for each k case ===
413     for k in kCases:
414         print(f"Training for k = {k}")
415 
416         # === Initialize Model and Optimizer ===
417         # model = PINN2DWave().to(device)
418         model = PINN2DWave_Rowdy().to(device)
419         optimizer = torch.optim.Adam(model.parameters(), lr=learningRate)
420 
421         # === Generate Initial and Boundary Conditions ===
422         x_int, y_int, t_int, x_bc, y_bc, t_bc, u_bc = generate_points(numInt, numBC, k)
423         x_ic, y_ic, t_ic, u_ic = generate_initial_conditions(numIC, k)
424 
425         train(model, optimizer, k,
426               x_int, y_int, t_int,
427               x_ic, y_ic, t_ic, u_ic,
428               x_bc, y_bc, t_bc, u_bc, useAnnealing=True)
429 
430         plot_solution(model, k)
```