Smart Reactor : Machine Learning (V2)

Undergraduate project - Semester 2024-25-I

Mentees:

Ankit Kumar (220159)
Komal Kumari (220540)

Contents

1 Overview

The main objective of our project was to develop a machine learning model capable of predicting the kinetic behavior of a reaction using experimental data.

- Throughout the project, we explored the fundamentals of neural networks
- Gained experience in solving forward PINNS
- Learned implementation of inverse problems with PINNs.
- We applied this knowledge to the Belousov-Zhabotinsky (BZ) reaction, which was provided to us as a case study.

However, the actual task presented a considerable challenge, as it involved a higher degree of stiffness and complex equations beyond what we had practiced. This required us to adapt our approach and deepen our understanding of working with PINNs in handling stiff and intricate systems.

2 PINNs

Physics-Informed Neural Networks (PINNs) solve differential equations by embedding physical laws into a neural network's loss function. The loss combines residuals from the governing equations and boundary/initial conditions, ensuring adherence to the physics. PINNs are efficient, requiring minimal data, and versatile, handling forward and inverse problems across fields like fluid dynamics, biology, and quantum mechanics. They excel in solving complex, high-dimensional systems where traditional methods may struggle.

3 Regression Problem using TensorFlow

We have used Artificial Neural Network (ANN) to perform regression on a synthetic dataset. The goal is to model a quadratic relationship between the input variable x and the output y, given by the equation:

$$y = 3x2 + 2x + 5 + \epsilon \tag{1}$$

where:

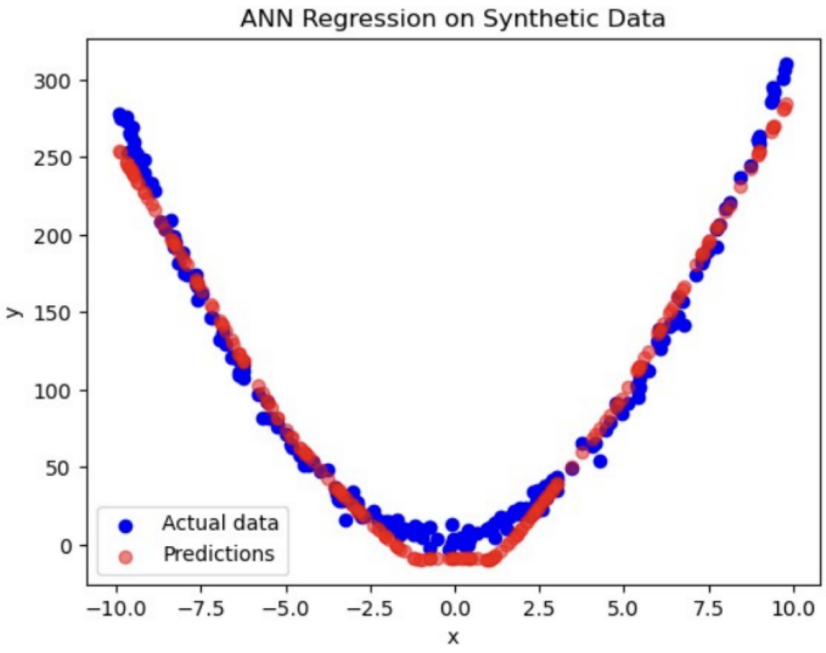
- x is randomly sampled from a uniform distribution in the range $[-10, 10]$.

- ϵ is random noise sampled from a normal distribution $N(0,5)$ to simulate real-world data variability.

3.1 Code

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Step 1: Generate the dataset
np.random.seed(42)  # Set a seed for reproducibility
x = np.random.uniform(-10, 10, 1000)  # Generate 1000 random samples
    uniformly from [-10, 10]
y = 3 * x**2 + 2 * x + 5 + np.random.normal(0, 5, x.shape)  # Quadratic
     equation with added Gaussian noise

# Reshape the data to make it compatible with the neural network input/
    output
x = x.reshape(-1, 1)  # x as a column vector
y = y.reshape(-1, 1)  # y as a column vector

# Split the dataset into training and testing sets (80% training, 20%
    testing)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size
    =0.2, random_state=42)

# Step 2: Define the ANN model for regression
# Sequential model consists of layers stacked sequentially
model = Sequential([
    Dense(64, input_dim=1, activation='relu'),   # First hidden layer
        with 64 neurons and ReLU activation
    Dense(32, activation='relu'),                # Second hidden layer
        with 32 neurons and ReLU activation
    Dense(1)                                     # Output layer with 1
        neuron (no activation for regression)
])

# Step 3: Compile the model
# Define the optimizer, loss function, and evaluation metrics
model.compile(
    optimizer='adam',       # Adam optimizer for adaptive learning
    loss='mse',             # Mean Squared Error as the loss function
        for regression
    metrics=['mae']         # Mean Absolute Error as an additional
        metric for evaluation
)

# Step 4: Train the model
# Fit the model on the training data, using the test data for
    validation
history = model.fit(
    x_train, y_train,                   # Training data
    validation_data=(x_test, y_test),   # Validation data
```

```
    epochs=100,                          # Number of training epochs
    batch_size=32,                       # Number of samples per
        gradient update
    verbose=1                            # Print training progress
)

# Step 5: Evaluate the model
# Evaluate the trained model on the test data
loss, mae = model.evaluate(x_test, y_test, verbose=0)
print(f"Mean Absolute Error on Test Data: {mae:.2f}")  # Print the test
     MAE for interpretation

# Step 6: Plotting the model's predictions
# Use the trained model to predict the output for test data
predictions = model.predict(x_test)

# Create a scatter plot to visualize actual vs predicted values
plt.scatter(x_test, y_test, color="blue", label="Actual data")  # Blue
    points for actual data
plt.scatter(x_test, predictions, color="red", alpha=0.5, label="
    Predictions")  # Red points for predictions
plt.xlabel("x")        # Label for x-axis
plt.ylabel("y")        # Label for y-axis
plt.legend()           # Add a legend to distinguish actual vs
    predicted points
plt.title("ANN Regression on Synthetic Data")  # Title for the plot
plt.show()             # Display the plot
```



4 Forward Problem Using PINNs

PINNs solve forward problems of differential equations by embedding the governing physics directly into the neural network's loss function. In a forward problem, the objective is to find unknown functions that satisfy given differential equations with specific initial and boundary conditions.

- **Define the Domain and Differential Equations:** Specify the problem's spatial/temporal domain and the differential equations that govern it.
- **Neural Network Setup:** Create a neural network that takes domain points (e.g., time or spatial coordinates) as inputs and outputs the approximated solution.
- **Physics-Based Loss Function:** solution. Physics-Based Loss Function: Incorporate three loss terms|differential equation residuals, initial conditions, and boundary conditions|into a single loss function. Automatic differentiation is used to compute derivatives for the differential equation residuals.
- **Training:** Optimize the network's weights to minimize the total loss, ensuring that the network output satisfies the differential equations and conditions.

- **Solution Extraction** After training, the network approximates the solution across the domain.

4.1 Example: Sine and Cosine Functions

Functions:

$$y_1(x) = \sin(x)$$
$$y_2(x) = \cos(x)$$

ODE System:

$$\frac{dy_1}{dx} = y_2$$
$$\frac{dy_2}{dx} = -y_1$$

Description:

In this problem, we aim to solve a system of ordinary differential equations (ODEs) representing simple harmonic oscillators. The functions involved are sine and cosine, which are solutions to the ODE system. Specifically, we are trying to model two functions that satisfy the above differential equations.

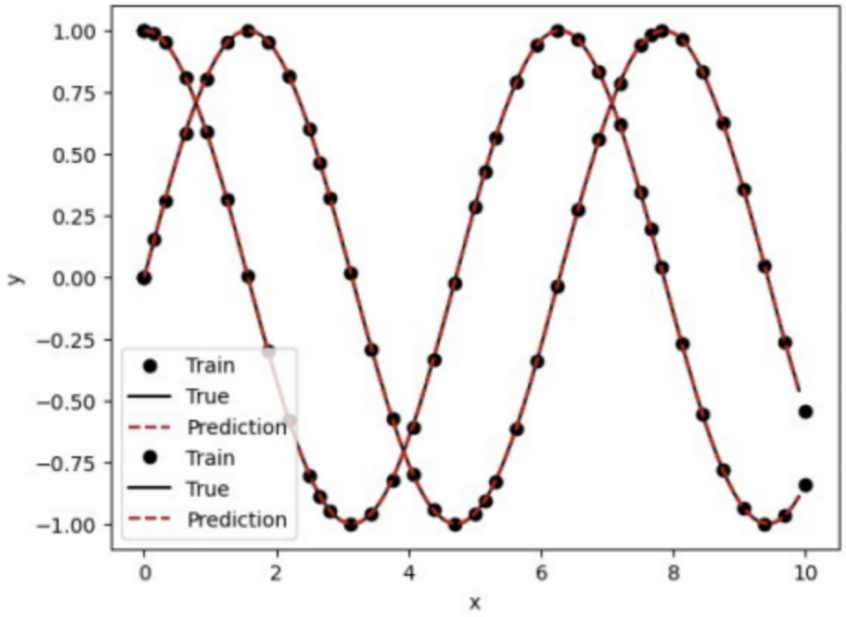
The initial conditions at $x = 0$ are:

$$y_1(0) = 0, \quad y_2(0) = 1$$

Code

```
import deepxde as dde
import numpy as np
# Define the time domain for the problem, which ranges from 0 to 10
geom = dde.geometry.TimeDomain(0, 10)
# Define the ODE system representing the relationships between y1 and
    y2
def ode_system(x, y):
    # Separate y into its components y1 and y2
    y1, y2 = y[:, 0:1], y[:, 1:]
    # Compute the derivatives of y1 and y2 with respect to x
    dy1_x = dde.grad.jacobian(y, x, i=0)
    dy2_x = dde.grad.jacobian(y, x, i=1)
    # Return the ODEs as a list of expressions
    return [dy1_x - y2, dy2_x + y1]
# Define the boundary conditions
# The boundary condition function checks if x is at the initial point (
    x=0)
def boundary(_, on_initial):
    return on_initial
# Define initial conditions for y1 and y2 at x = 0
```

```
ic1 = dde.icbc.IC(geom, lambda x: 0, boundary, component=0)  # y1(0) =
    0
ic2 = dde.icbc.IC(geom, lambda x: 1, boundary, component=1)  # y2(0) =
    1
# Define the true solution for comparison and error calculation during
    training
def func(x):
    return np.hstack((np.sin(x), np.cos(x)))
# Set up the data for the problem using DeepXDE's PDE data structure
data = dde.data.PDE(
    geom,               # Geometry (time domain)
    ode_system,         # The ODE system function
    [ic1, ic2],         # Initial conditions
    num_domain=35,      # Number of training points in the domain
    num_boundary=2,     # Number of training points on the boundary
    solution=func,      # True solution (optional)
    num_test=100        # Number of test points for validation
)
# Define the neural network structure
layer_size = [1] + [50] * 3 + [2]  # Input layer + three hidden layers
    of 50 nodes + output layer
activation = "tanh"                # Activation function
initializer = "Glorot uniform"     # Weight initializer
net = dde.nn.FNN(layer_size, activation, initializer)
# Set up the model using the data and neural network
model = dde.Model(data, net)
# Compile the model with the Adam optimizer, learning rate, and a
    metric for error
model.compile("adam", lr=0.001, metrics=["l2 relative error"])
# Train the model for 20,000 iterations and save the loss history and
    training state
losshistory, train_state = model.train(iterations=20000)
# Plot and save the loss history and training results
dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0
2
4
6
8
10
x
Train
True
Prediction
Train
True
Prediction

4.2 Example: Damped Harmonic Oscillator

System Description

The damped harmonic oscillator is described by the following second-order differential equation:

$$\frac{d^2y}{dt^2} + 2\gamma\frac{dy}{dt} + \omega^2 y = 0$$

Where:

- (y(t)) is the displacement as a function of time.
- (\gamma) is the damping coefficient, which represents the strength of the damping (friction).
- (\omega) is the natural frequency of the system.

In this example, we rewrite the second-order ODE as a system of two first-order ODEs by defining:

$$y_1(t) = y(t)$$

$$y_2(t) = \frac{dy}{dt}$$

The system of ODEs then becomes:

$$\frac{dy_1}{dt} = y_2$$

$$\frac{dy_2}{dt} = -2\gamma y_2 - \omega^2 y_1$$

Initial Conditions

For this example, we set the initial conditions as:

$$y_1(0) = 1 \quad \text{(initial displacement)}$$

$$y_2(0) = 0 \quad \text{(initial velocity)}$$

Target Solution (Damped Oscillations)

The analytical solution to this system is given by:

$$y(t) = e^{-\gamma t}\cos(\omega t)$$

Where:

$$y_1(t) = e^{-\gamma t}\cos(\omega t)$$

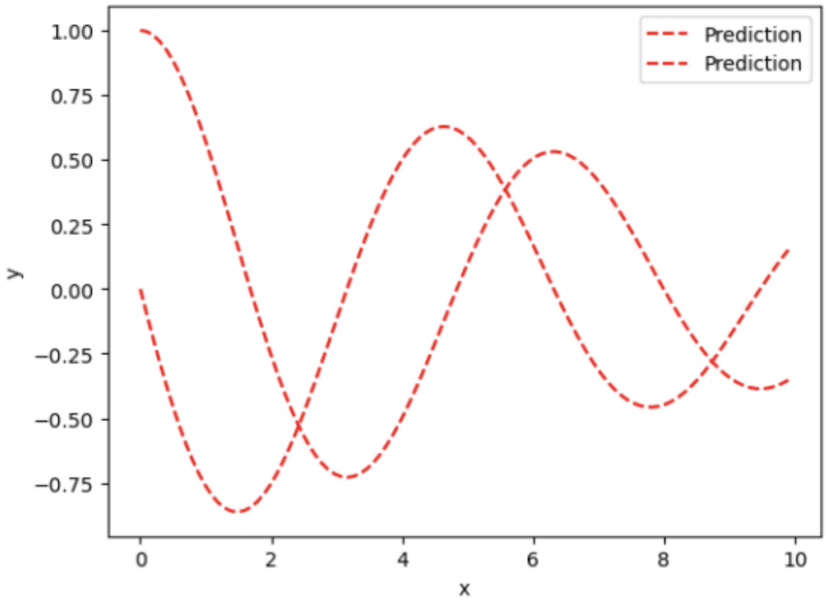
$$y_2(t) = -\gamma e^{-\gamma t}\cos(\omega t) - \omega e^{-\gamma t}\sin(\omega t)$$

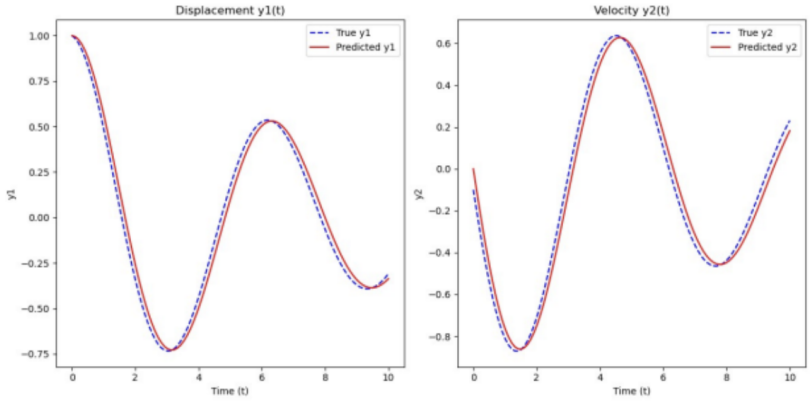
Code

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Set parameters for the damped oscillator
gamma = 0.1  # Damping coefficient
omega = 1.0  # Natural frequency

# Define the time domain from t=0 to t=10
geom = dde.geometry.TimeDomain(0, 10)

# Define the system of ODEs for the damped harmonic oscillator
def ode_system(x, y):
    # Separate y into its components y1 and y2
    y1, y2 = y[:, 0:1], y[:, 1:]
    # Compute the derivatives of y1 and y2 with respect to t
    dy1_t = dde.grad.jacobian(y, x, i=0)
    dy2_t = dde.grad.jacobian(y, x, i=1)
    # Return the ODEs as a list of expressions
    return [dy1_t - y2, dy2_t + 2 * gamma * y2 + omega**2 * y1]

# Define the boundary condition at t=0 (initial conditions)
def boundary(_, on_initial):
    return on_initial

# Initial conditions for y1(0) = 1 and y2(0) = 0
ic1 = dde.icbc.IC(geom, lambda x: 1, boundary, component=0)  # y1(0) =
    1
ic2 = dde.icbc.IC(geom, lambda x: 0, boundary, component=1)  # y2(0) =
    0

# Set up the data for the problem using DeepXDE's PDE data structure
data = dde.data.PDE(
    geom,               # Geometry (time domain)
    ode_system,         # The ODE system function
    [ic1, ic2],         # Initial conditions
    num_domain=35,      # Number of training points in the domain
    num_boundary=2,     # Number of training points on the boundary
    num_test=100        # Number of test points for validation
)

# Define the neural network structure
layer_size = [1] + [50] * 3 + [2]  # Input layer + three hidden layers
    of 50 nodes + output layer
activation = "tanh"                # Activation function
initializer = "Glorot uniform"     # Weight initializer
net = dde.nn.FNN(layer_size, activation, initializer)

# Set up the model using the data and neural network
model = dde.Model(data, net)

# Compile the model with the Adam optimizer, learning rate, and a
    metric for error
model.compile("adam", lr=0.001)

# Train the model for 20,000 iterations
losshistory, train_state = model.train(iterations=20000)

# Plot and save the loss history and training results
dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```

```
# Define the true solution for y1 and y2
def true_solution(x):
    y1 = np.exp(-gamma * x) * np.cos(omega * x)
    y2 = -gamma * np.exp(-gamma * x) * np.cos(omega * x) - omega * np.
        exp(-gamma * x) * np.sin(omega * x)
    return np.hstack((y1[:, None], y2[:, None]))  # Reshape for
        consistency

# Generate test points and evaluate predictions
x_test = geom.uniform_points(100, True)  # Generate 100 points in the
    time domain
y_pred = model.predict(x_test)           # Predicted solution
y_true = true_solution(x_test)           # True solution

# Plot the predicted and true solutions
plt.figure(figsize=(12, 6))

# Plot y1 (displacement)
plt.subplot(1, 2, 1)
plt.plot(x_test, y_true[:, 0], label="True y1", linestyle="--", color="
    blue")
plt.plot(x_test, y_pred[:, 0], label="Predicted y1", linestyle="-",
    color="red")
plt.title("Displacement y1(t)")
plt.xlabel("Time (t)")
plt.ylabel("y1")
plt.legend()

# Plot y2 (velocity)
plt.subplot(1, 2, 2)
plt.plot(x_test, y_true[:, 1], label="True y2", linestyle="--", color="
    blue")
plt.plot(x_test, y_pred[:, 1], label="Predicted y2", linestyle="-",
    color="red")
plt.title("Velocity y2(t)")
plt.xlabel("Time (t)")
plt.ylabel("y2")
plt.legend()

plt.tight_layout()
plt.show()
```





5 Forward PINNs on BZ Reaction

Description

The Belousov–Zhabotinskii (BZ) reaction is a well-known oscillating chemical reaction, where periodic changes in concentration cause visible color changes. This behavior arises from the periodic oscillation of reactants between different oxidation states. Due to its complexity, the reaction is often modeled using the Oregonator, a simplified system of nonlinear differential equations. These equations are stiff, making analysis difficult. By nondimensionalizing and reducing the system, researchers

study a simplified 2x2 model that exhibits relaxation oscillations. The goal is to solve this reduced system using Physics-Informed Neural Networks (PINNs) to analyze its oscillatory behavior and compare it with experimental data.

$$\varepsilon \frac{dx}{d\tau} = x(1-x) + \frac{f(q-x)}{q+x} z \equiv g(x,z),$$
$$\frac{dz}{d\tau} = x - z \equiv h(x,z).$$

Figure 1: Oregonator Model for BZ Reaction

The reduced set of equations are (f=0.67, ϵ=0.04, q=8e-04).

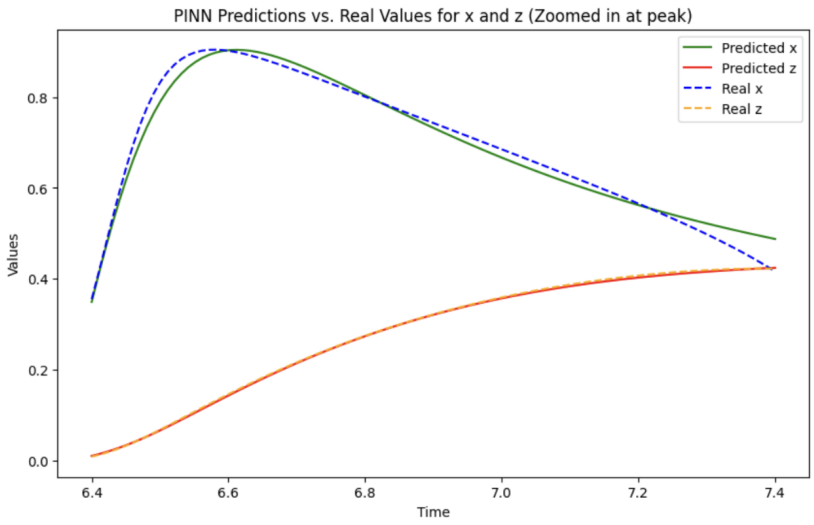
Code

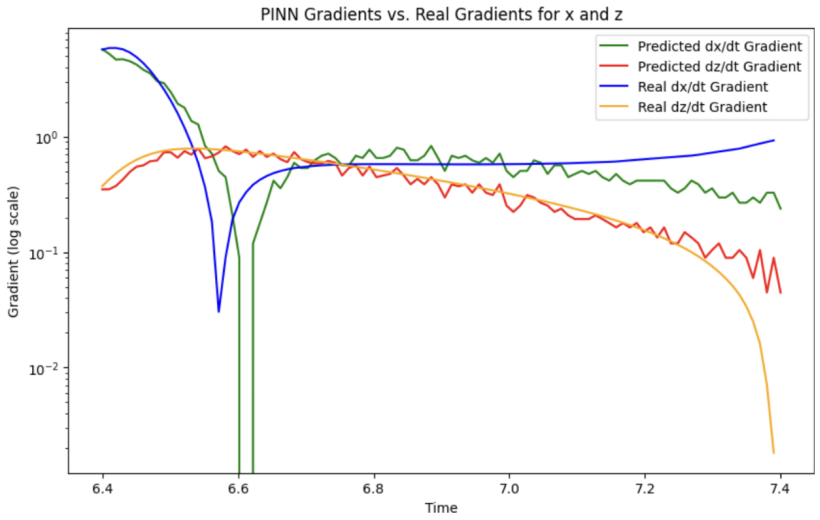
```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp


# Define the PINN class
class PINN(nn.Module):
    def __init__(self, hidden_layers=3, hidden_units=20):
        super(PINN, self).__init__()
        layers = [nn.Linear(1, hidden_units), nn.Tanh()]
        for _ in range(hidden_layers):
            layers.append(nn.Linear(hidden_units, hidden_units))
            layers.append(nn.Tanh())
        layers.append(nn.Linear(hidden_units, 2))
        self.model = nn.Sequential(*layers)


    def forward(self, t):
        return self.model(t)


# Physics loss function for PINN
def physics_loss(pinn, t, x, z, epsilon=0.04, q=0.0008, f=2/3):
    x_t = torch.autograd.grad(x.sum(), t, create_graph=True,
        retain_graph=True)[0]
    z_t = torch.autograd.grad(z.sum(), t, create_graph=True,
        retain_graph=True)[0]
    residual_x = x_t * epsilon - (x * (1 - x) + f * z * (q - x) / (q + x
        ))
```

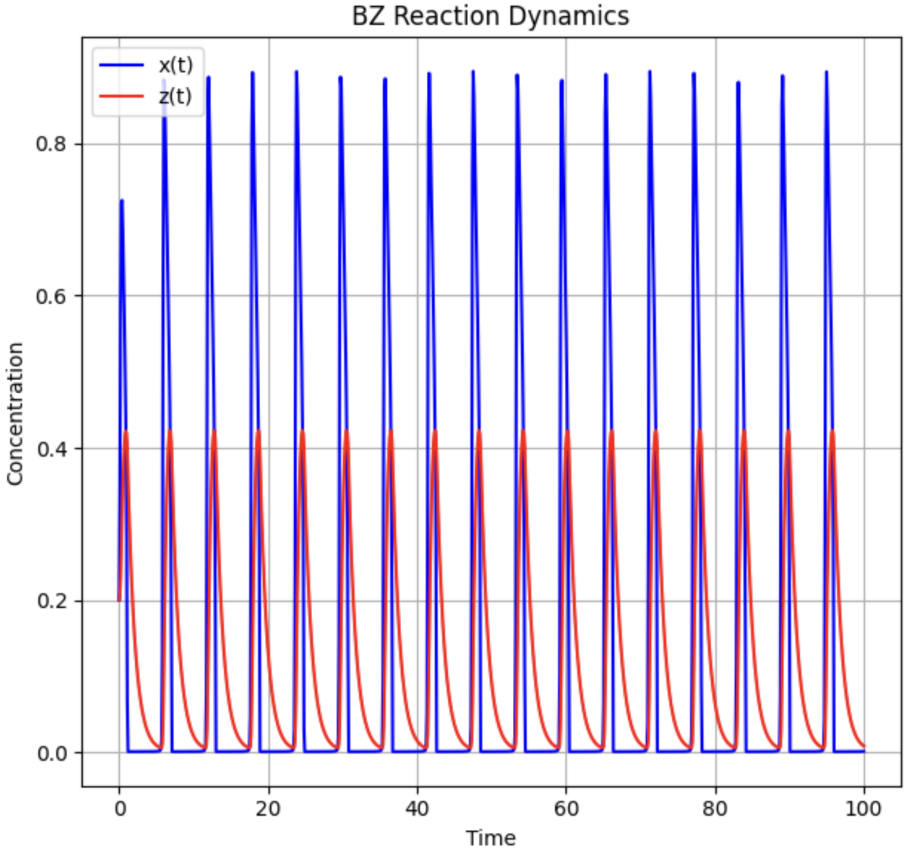
```
    residual_z = z_t - (x - z)
    return torch.mean(residual_x ** 2) + torch.mean(residual_z ** 2)


# Training function for PINN
def train_pinn(pinn, t_init, initial_conditions, t_span, n_epochs=5000,
     lr=0.001):
    optimizer = optim.Adam(pinn.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    loss_history = []


    for epoch in range(n_epochs):
        t = torch.linspace(t_span[0], t_span[1], 100, requires_grad=True
            ).reshape(-1, 1)
        x_pred, z_pred = pinn(t).chunk(2, dim=1)
        boundary_loss = loss_fn(pinn(t_init), initial_conditions)
        phys_loss = physics_loss(pinn, t, x_pred, z_pred)
        total_loss = boundary_loss + phys_loss
        optimizer.zero_grad()
        total_loss.backward()
        optimizer.step()
        loss_history.append(total_loss.item())


        if epoch % 500 == 0:
            print(f"Epoch [{epoch}/{n_epochs}], Loss: {total_loss.item()
                :.8f}")


    return pinn, loss_history


# Initialize and train the PINN model
pinn = PINN(hidden_layers=3, hidden_units=20)
initial_conditions = torch.tensor([[0.355729241909736,
    0.009507989778190 52]], dtype=torch.float32)
t_init = torch.tensor([[6.4]], dtype=torch.float32)
t_span = (6.4, 7.4)
trained_pinn, loss_history = train_pinn(pinn, t_init,
    initial_conditions, t_span)


# Generate real values using SciPy's LSODA solver
def bz_reaction(t, y):
    x, z = y
    dx_dt = (1 / 0.04) * (x * (1 - x) + (2 / 3) * z * (0.0008 - x) /
        (0.0008 + x))  # Adjusted dx/dt
    dz_dt = x - z                               # dz/dt based on BZ system
    return [dx_dt, dz_dt]


# Solve the system using `solve_ivp` with the LSODA method
initial_conditions_real = [0.355729241909736, 0.009507989778190 52]
t_eval_real = np.linspace(t_span[0], t_span[1], 100)  # Higher
    resolution for smooth plotting
```

```
sol = solve_ivp(bz_reaction, t_span, initial_conditions_real, method='
    LSODA', t_eval=t_eval_real)


# Extract the real solution
t_real = sol.t
x_real = sol.y[0]
z_real = sol.y[1]


# Plot the predictions vs. real values
with torch.no_grad():
   t_test = torch.linspace(t_span[0], t_span[1], 100, dtype=torch.
       float32).reshape(-1, 1)
   x_pred, z_pred = trained_pinn(t_test).chunk(2, dim=1)


plt.figure(figsize=(10, 6))
plt.plot(t_test.numpy(), x_pred.numpy(), label="Predicted x", color="
    green")
plt.plot(t_test.numpy(), z_pred.numpy(), label="Predicted z", color="
    red")
plt.plot(t_real, x_real, label="Real x", linestyle="--", color="blue")
plt.plot(t_real, z_real, label="Real z", linestyle="--", color="orange"
    )
plt.xlabel("Time")
plt.ylabel("Values")
plt.legend()
plt.title("PINN Predictions vs. Real Values for x and z")
plt.show()


# Calculate gradients using central difference method
dx_dt_pred = []
dz_dt_pred = []
delta_t = 1e-6  # Small time delta for central difference


for t in t_test:
   t_forward = t + delta_t
   t_backward = t - delta_t


   # Forward and backward predictions for x and z
   x_forward, z_forward = trained_pinn(t_forward.reshape(-1, 1)).chunk
       (2, dim=1)
   x_backward, z_backward = trained_pinn(t_backward.reshape(-1, 1)).
       chunk(2, dim=1)


   # Calculate central differences
   dx_dt = (x_forward - x_backward) / (2 * delta_t)
   dz_dt = (z_forward - z_backward) / (2 * delta_t)


   dx_dt_pred.append(dx_dt.item())
   dz_dt_pred.append(dz_dt.item())
```

```
# Plot the predicted gradients vs real gradients in log scale
plt.figure(figsize=(10, 6))
plt.semilogy(t_test.numpy(), np.abs(dx_dt_pred), label="Predicted dx/dt
     Gradient", color="green")
plt.semilogy(t_test.numpy(), np.abs(dz_dt_pred), label="Predicted dz/dt
     Gradient", color="red")
plt.semilogy(t_real[:-1], np.abs(np.diff(x_real) / np.diff(t_real)),
    label="Real dx/dt Gradient", color="blue")
plt.semilogy(t_real[:-1], np.abs(np.diff(z_real) / np.diff(t_real)),
    label="Real dz/dt Gradient", color="orange")
plt.xlabel("Time")
plt.ylabel("Gradient (log scale)")
plt.legend()
plt.title("PINN Gradients vs. Real Gradients for x and z")
plt.show()
```




PINN Gradients vs. Real Gradients for x and z
Predicted dx/dt Gradient
Predicted dz/dt Gradient
Real dx/dt Gradient
Real dz/dt Gradient
Gradient (log scale)
10^{0}
10^{-1}
10^{-2}
6.4
6.6
6.8
7.0
7.2
7.4
Time


BZ Reaction Dynamics
x(t)
z(t)
Concentration
0.0
0.2
0.4
0.6
0.8
0
20
40
60
80
100
Time

6 Inverse Problem on PINNs

Physics-Informed Neural Networks (PINNs) solve inverse problems by combining physical laws (encoded as PDEs/ODEs) with observed data to infer unknown parameters or conditions. A neural network approximates the solution and unknowns, while a loss function enforces both the governing equations (physics loss) and data consistency (data loss). During training, the network learns to satisfy the physical model and match observed data, enabling the estimation of unknowns and the reconstruction of the system's state. PINNs are efficient, flexible, and handle sparse or noisy data effectively.

Steps in Solving Inverse Problems with PINNs:

- **Define the Domain and Observations**: Specify the spatial/temporal domain and use available data points.
- **Neural Network Setup**: Build a neural network that incorporates the unknown parameters.
- **Loss Function for Parameter Estimation**: Include terms for equation residuals and fit to observational data.
- **Training**: Optimize both network weights and parameters to minimize the loss.

6.1 Example: Diffusion Equation

Consider the following diffusion equation with a source term:

$$\frac{\partial u}{\partial t} = C\frac{\partial^2 u}{\partial x^2} + e^{-t}\left(\sin(\pi x) - \pi^2 \sin(\pi x)\right) \tag{2}$$

where $x \in [-1, 1]$, $t \in [0, 1]$, and C is the unknown diffusion coefficient.

The boundary conditions are given as:

$$u(\pm 1, t) = \sin(\pi x)e^{-t} \tag{3}$$

The initial condition is:

$$u(x, 0) = \sin(\pi x) \tag{4}$$

Additionally, there are measurements of $u(x, 1)$ at 10 points for the time $t = 1$.

The exact solution to the equation is:

$$u(x, t) = \sin(\pi x)e^{-t} \tag{5}$$

The goal is to recover the unknown diffusion coefficient C and the complete solution field $u(x,t)$ using a Physics-Informed Neural Network (PINN) approach. We will try to predict C using an initial guess of $C = 2$, while the true value of C is known to be $C = 1$.

Code

```
import torch
import torch.nn as nn
import numpy as np
from torch.autograd import grad
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

class PINN(nn.Module):
    def __init__(self, layers=[2, 32, 32, 32, 1]):
        super().__init__()
        self.C = nn.Parameter(torch.tensor([2.0]))

        modules = []
        for i in range(len(layers)-1):
            modules.append(nn.Linear(layers[i], layers[i+1]))
            if i < len(layers)-2:
                modules.append(nn.Tanh())

        self.net = nn.Sequential(*modules)

    def forward(self, x):
        return self.net(x)

    def compute_gradients(self, x, y):
        dy_t = grad(y, x, grad_outputs=torch.ones_like(y),
                    create_graph=True)[0][:, 1:2]

        dy_x = grad(y, x, grad_outputs=torch.ones_like(y),
                    create_graph=True)[0][:, 0:1]

        dy_xx = grad(dy_x, x, grad_outputs=torch.ones_like(dy_x),
                     create_graph=True)[0][:, 0:1]

        return dy_t, dy_xx

    def pde_residual(self, x):
        y = self.forward(x)
        dy_t, dy_xx = self.compute_gradients(x, y)

        source = torch.exp(-x[:, 1:2]) * (torch.sin(np.pi * x[:, 0:1]) -
                                          np.pi**2 * torch.sin(np.pi * x
                                              [:, 0:1]))

        return dy_t - self.C * dy_xx + source

def exact_solution(x):
    return np.sin(np.pi * x[:, 0:1]) * np.exp(-x[:, 1:2])
```

```
def generate_training_data(num_domain=40, num_boundary=20, num_initial
    =10):
   x_domain = torch.rand(num_domain, 2) * torch.tensor([2., 1.]) +
       torch.tensor([-1., 0.])
   x_domain.requires_grad = True

   x_boundary = torch.zeros(num_boundary, 2)
   x_boundary[:, 0] = torch.rand(num_boundary) * 2 - 1
   x_boundary[:, 1] = torch.rand(num_boundary)
   x_boundary.requires_grad = True

   x_initial = torch.zeros(num_initial, 2)
   x_initial[:, 0] = torch.rand(num_initial) * 2 - 1
   x_initial.requires_grad = True

   observe_x = torch.zeros(10, 2)
   observe_x[:, 0] = torch.linspace(-1, 1, 10)
   observe_x[:, 1] = 1
   observe_x.requires_grad = True

   y_boundary = torch.tensor(exact_solution(x_boundary.detach().numpy()
       ))
   y_initial = torch.tensor(exact_solution(x_initial.detach().numpy()))
   y_observe = torch.tensor(exact_solution(observe_x.detach().numpy()))

   return (x_domain, x_boundary, x_initial, observe_x,
           y_boundary, y_initial, y_observe)

def train_model(model, num_epochs=50000, learning_rate=0.001):
   optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

   data = generate_training_data()
   x_domain, x_boundary, x_initial, observe_x, y_boundary, y_initial,
       y_observe = data

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

       pde_loss = torch.mean(model.pde_residual(x_domain)**2)

       bc_loss = torch.mean((model(x_boundary) - y_boundary)**2)
       ic_loss = torch.mean((model(x_initial) - y_initial)**2)
       observe_loss = torch.mean((model(observe_x) - y_observe)**2)

       total_loss = pde_loss + bc_loss + ic_loss + observe_loss

       if epoch % 1000 == 0:
           print(f'Epoch {epoch}, Loss: {total_loss.item():.6f}, C: {
               model.C.item():.6f}')

       total_loss.backward()
       optimizer.step()

   return model

def evaluate_solution(model):
```

```
    x = torch.linspace(-1, 1, 100)
    t = torch.linspace(0, 1, 100)
    X, T = torch.meshgrid(x, t)
    points = torch.stack([X.flatten(), T.flatten()], dim=1)

    with torch.no_grad():
        predicted = model(points).reshape(100, 100)
        exact = torch.tensor(exact_solution(points.numpy())).reshape
            (100, 100)

    error = torch.mean((predicted - exact)**2).sqrt()
    print(f'L2 Error: {error.item():.6f}')

    plt.figure(figsize=(15, 5))
    plt.subplot(131)
    plt.contourf(X.numpy(), T.numpy(), predicted.numpy())
    plt.colorbar()
    plt.title('Predicted Solution')

    plt.subplot(132)
    plt.contourf(X.numpy(), T.numpy(), exact.numpy())
    plt.colorbar()
    plt.title('Exact Solution')

    plt.subplot(133)
    plt.contourf(X.numpy(), T.numpy(), (predicted - exact).numpy())
    plt.colorbar()
    plt.title('Error')
    plt.show()

model = PINN()
trained_model = train_model(model)
evaluate_solution(trained_model)
```

The physics-informed neural network successfully recovered both the unknown parameter C and the solution field $u(x,t)$ with high accuracy. The network achieved a relative L_2 error of order 10^{-3}, demonstrating that PINNs can effectively solve inverse problems in partial differential equations.

7 Inverse Problem on BZ Reaction

Building upon the forward problem for BZ reaction we solved above, we now address the inverse problem for the reduced Oregonator system. We would solve this by using synthetic data by solving using a numerical solver for stiff equations-'Radau'. We will take only a small data range because the stiffness of the equation highly increases the computational power needed to solve the system for a bigger range and more number of oscillations. We would limit ourselves to just a single oscillation.

7.1 Code

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp


class InversePINN(nn.Module):
   def __init__(self, hidden_layers=3, hidden_units=20):
       super(InversePINN, self).__init__()
       # Neural network for solution
       layers = [nn.Linear(1, hidden_units), nn.Tanh()]
       for _ in range(hidden_layers):
           layers.append(nn.Linear(hidden_units, hidden_units))
           layers.append(nn.Tanh())
       layers.append(nn.Linear(hidden_units, 2))
       self.solution_net = nn.Sequential(*layers)

       # Learnable parameters (initialized with random guesses)
       self.epsilon = nn.Parameter(torch.tensor([0.1], dtype=torch.
           float32))
       self.q = nn.Parameter(torch.tensor([0.0001], dtype=torch.float32
           ))
       self.f = nn.Parameter(torch.tensor([0.5], dtype=torch.float32))


   def forward(self, t):
       return self.solution_net(t)


def physics_loss(model, t, x, z):
   # Get current predicted parameter values
   epsilon = torch.abs(model.epsilon)  # Ensure positive
   q = torch.abs(model.q)  # Ensure positive
   f = torch.abs(model.f)  # Ensure positive

   # Calculate derivatives
   x_t = torch.autograd.grad(x.sum(), t, create_graph=True,
       retain_graph=True)[0]
   z_t = torch.autograd.grad(z.sum(), t, create_graph=True,
       retain_graph=True)[0]

   # Physics residuals using learned parameters
   residual_x = x_t * epsilon - (x * (1 - x) + f * z * (q - x) / (q + x
       ))
   residual_z = z_t - (x - z)

   return torch.mean(residual_x ** 2) + torch.mean(residual_z ** 2)


def train_inverse_pinn(model, t_data, x_data, z_data, t_span, n_epochs
    =30000, lr=0.001):
   optimizer = optim.Adam(model.parameters(), lr=lr)
   loss_fn = nn.MSELoss()
```

```
    loss_history = []
    param_history = []


    for epoch in range(n_epochs):
        # Forward pass through the network
        t = torch.linspace(t_span[0], t_span[1], 100, requires_grad=True
            ).reshape(-1, 1)
        x_pred, z_pred = model(t).chunk(2, dim=1)

        # Data loss
        output_data = model(t_data)
        x_data_pred, z_data_pred = output_data.chunk(2, dim=1)
        data_loss = loss_fn(x_data_pred, x_data) + loss_fn(z_data_pred,
            z_data)

        # Physics-informed loss
        phys_loss = physics_loss(model, t, x_pred, z_pred)

        # Total loss
        total_loss = data_loss + phys_loss

        # Backpropagation
        optimizer.zero_grad()
        total_loss.backward()
        optimizer.step()

        # Store loss and parameters
        loss_history.append(total_loss.item())
        param_history.append([model.epsilon.item(), model.q.item(),
            model.f.item()])

        if epoch % 500 == 0:
            print(f"Epoch [{epoch}/{n_epochs}]")
            print(f"Loss: {total_loss.item():.8f}")
            print(f"Estimated parameters:   ={model.epsilon.item():.6f},
                 q={model.q.item():.6f}, f={model.f.item():.6f}")


    return model, loss_history, param_history


# Generate synthetic data with known parameters for training
def generate_synthetic_data(t_span, n_points=10):
    def bz_reaction(t, y, epsilon=0.04, q=0.0008, f=2/3):
        x, z = y
        dx_dt = (1/epsilon) * (x * (1 - x) + f * z * (q - x) / (q + x))
        dz_dt = x - z
        return [dx_dt, dz_dt]


    # Generate solution
    t_eval = np.linspace(t_span[0], t_span[1], n_points)
    initial_conditions = [0.355729241909736, 0.00950798977819052]
    sol = solve_ivp(bz_reaction, t_span, initial_conditions, method='
        LSODA', t_eval=t_eval)
```

```
    # Convert to torch tensors
    t_data = torch.tensor(sol.t.reshape(-1, 1), dtype=torch.float32)
    x_data = torch.tensor(sol.y[0].reshape(-1, 1), dtype=torch.float32)
    z_data = torch.tensor(sol.y[1].reshape(-1, 1), dtype=torch.float32)

    return t_data, x_data, z_data


# Main execution
t_span = (6.4, 7.4)
t_data, x_data, z_data = generate_synthetic_data(t_span)


# Initialize and train the inverse PINN
inverse_pinn = InversePINN()
trained_model, loss_history, param_history = train_inverse_pinn(
    inverse_pinn, t_data, x_data, z_data, t_span)


# Plot results
with torch.no_grad():
    t_test = torch.linspace(t_span[0], t_span[1], 100, dtype=torch.
        float32).reshape(-1, 1)
    x_pred, z_pred = trained_model(t_test).chunk(2, dim=1)


    # Solution plot
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(t_test.numpy(), x_pred.numpy(), label="Predicted x")
    plt.plot(t_test.numpy(), z_pred.numpy(), label="Predicted z")
    plt.plot(t_data.numpy(), x_data.numpy(), 'o', label="Data x")
    plt.plot(t_data.numpy(), z_data.numpy(), 'o', label="Data z")
    plt.xlabel("Time")
    plt.ylabel("Values")
    plt.legend()
    plt.title("Solution Comparison")


    # Parameter convergence plot
    param_history = np.array(param_history)
    plt.subplot(1, 2, 2)
    plt.plot(param_history[:, 0], label='  ')
    plt.plot(param_history[:, 1], label='q')
    plt.plot(param_history[:, 2], label='f')
    plt.axhline(y=0.04, color='r', linestyle='--', label='True   ')
    plt.axhline(y=0.0008, color='g', linestyle='--', label='True q')
    plt.axhline(y=2/3, color='b', linestyle='--', label='True f')
    plt.xlabel("Epoch")
    plt.ylabel("Parameter Value")
    plt.legend()
    plt.title("Parameter Convergence")

    plt.tight_layout()
    plt.show()
```

```
# Print final parameter estimates
print("\nFinal Parameter Estimates:")
print(f"   (epsilon) = {trained_model.epsilon.item():.6f} (true:
    0.040000)")
print(f"q = {trained_model.q.item():.6f} (true: 0.000800)")
print(f"f = {trained_model.f.item():.6f} (true: 0.666667)")
```

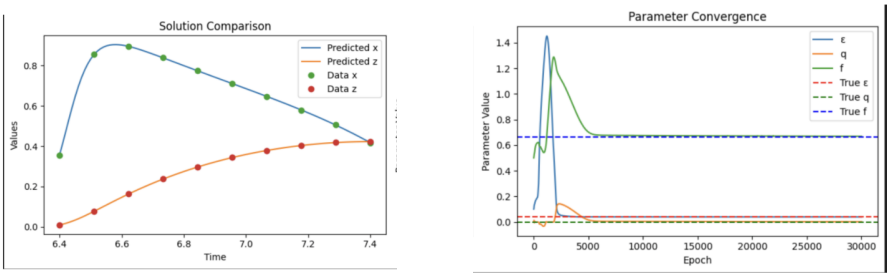


Figure 2: performances of inverse pinn

7.2 Model Performance and Parameter Estimation

For a small number of epochs, the model successfully estimates the parameters $f = 0.669743$, $\epsilon = 0.039910$. However, it struggles to solve for q. For lower to medium numbers of epochs, the estimated value of q ranges from 0.002 to 0.0002, whereas the real value is 0.0008. This discrepancy can be resolved by increasing the computational capacity of the machine being used and performing a higher number of epochs to get values as close to the true value as 0.0007993.

8 References

- https://scholar.rose-hulman.edu/cgi/viewcontent.cgi?article=1286&context=rhumj
- https:https://arxiv.org/pdf/2011.04520
- https://arxiv.org/pdf/2407.10836
- https://amses-journal.springeropen.com/articles/10.1186/s40323-024-00265-3