MA211(ODE)

Solving Differential Equations with Neural Networks

By-

Raj Maurya(B23406)

Under the supervision of-

Dr.Muslim Malik

muslim@iitmandi.ac.in

Contents

1 Introduction **4**
- 1.1 Differential Equations and Their Importance 4
- 1.2 Limitations of Traditional Analytical Methods 4
- 1.3 Neural Networks as a Tool for Solving DEs 5
  - 1.3.1 Advantages of Using Neural Networks 5

2 Background **7**
- 2.1 Neural Network Architecture: Feedforward Networks 7
- 2.2 Neurons, Activation Functions, and Layers 7
- 2.3 Neural Networks for Solving Differential Equations 8
  - 2.3.1 Advantages and Limitations of PINNs 9

3 Methodology(Legendre's) **10**
- 3.1 Specific Differential Equation: Legendre's Differential Equation . 10
- 3.2 PINN Architecture 10
- 3.3 Training Data Generation Process 11
- 3.4 Loss Function for Training the PINN 12
  - 3.4.1 Training Procedure 13
- 3.5 Final Legendre's Solution 13

4 Methodology(Bessel's) **14**
- 4.1 Specific Differential Equation: Bessel's Differential Equation . . . 14
- 4.2 PINN Architecture 15
- 4.3 Training Data Generation Process 15
- 4.4 Loss Function for Training the PINN 16
  - 4.4.1 Training Procedure 17
- 4.5 Final Bessel's Solution 17

5 Implementation Details **18**
- 5.1 Python Libraries Used 18
- 5.2 Training Process 19
- 5.3 Visualization and Animation of Training Progress 20
- 5.4 Final Code Snippet(Legendre's) 21
- 5.5 Final Code Snippet(Bessel's) 23

6 Results and Discussion(Legendre) **25**
- 6.1 Final Trained PINN Model 25
- 6.2 Predicted Solution Evolution During Training 26
- 6.3 Convergence of the Loss Function 26
- 6.4 Comparison with Analytical Solution 27
- 6.5 Accuracy and Limitations 27

7 Results and Discussion(Bessel) 28
7.1 Final Trained PINN Model 28
7.2 Predicted Solution Evolution During Training 28
7.3 Convergence of the Loss Function 28
7.4 Comparison with Analytical Solution 29
7.5 Accuracy and Limitations 29

8 Conclusion 30
8.1 Summary of Key Takeaways 30
8.2 Feasibility of Neural Networks for Solving DEs 31
8.3 Future Research and Improvements 32
8.4 Bessel's Equation Insights 32

9 References 34

10 Collab Notebooks 34

11 Words to be followed 35

1 Introduction

This section will introduce you to the basic Differential Equations and also the Traditional methods of solving along with the Modern approach.

1.1 Differential Equations and Their Importance

Differential equations (DE) are equations involving a function and its derivatives. They are used extensively to model real-world phenomena in fields such as physics, engineering, biology, and economics. For example, the motion of a spring-mass system can be modeled by a second-order differential equation:

Example of Second-Order DE

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0 \tag{1}$$

This represents the motion of a damped harmonic oscillator, where m is the mass, c is the damping coefficient, and k is the spring constant.

DEs also appear in population dynamics. The exponential growth of a population can be described by the following first-order differential equation:

Example of First-Order DE

$$\frac{dP}{dt} = rP \tag{2}$$

where P is the population at time t, and r is the growth rate.

1.2 Limitations of Traditional Analytical Methods

While there are many well-established methods to solve DEs, such as separation of variables, the Laplace transform, and power series solutions, these traditional techniques have limitations. For instance, when dealing with non-linear DEs or complex boundary conditions, finding an exact solution is often difficult or impossible.

For example, Bessel's equation, commonly encountered in problems of wave propagation and static potentials, can have solutions that are highly non-trivial:

Example: Bessel's Differential Equation

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - n^2)y = 0 \tag{3}$$

where n is the order of the Bessel function.

In cases like these, numerical methods such as finite difference or finite element methods can be applied, but they are computationally expensive and might struggle with high-dimensional problems.

1.3 Neural Networks as a Tool for Solving DEs

Neural networks (NNs) have recently gained traction as a powerful tool for solving complex problems, including DEs. Unlike traditional methods, neural networks can approximate solutions to DEs by learning from data and do not require an explicit analytical solution. A notable technique is Physics-Informed Neural Networks (PINNs), which incorporate the physical laws governing the system directly into the neural network's loss function.

For example, consider the following DE:

Example of a DE Solved with PINNs

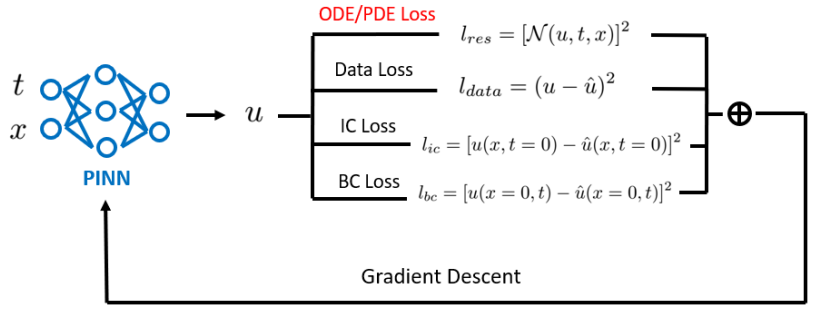
$$\frac{d^2u(x)}{dx^2} = f(x), \quad u(0) = 0, \quad u(1) = 0 \tag{4}$$

A neural network can approximate the solution $u(x)$, while minimizing the error in both the DE and boundary conditions.

Neural networks have shown promise in overcoming the challenges faced by traditional methods, particularly when dealing with high-dimensional or nonlinear DEs. They are also adaptable across different types of DEs, offering a flexible approach to solving equations that are otherwise intractable.

1.3.1 Advantages of Using Neural Networks

- **Generalization:** Neural networks can solve a wide range of DEs without the need for specific analytical methods.
- **Scalability:** NNs are particularly effective in handling high-dimensional problems that are computationally expensive for traditional methods.
- **Efficiency:** PINNs can be trained efficiently by embedding physical laws into the optimization process, improving the quality of the solution.



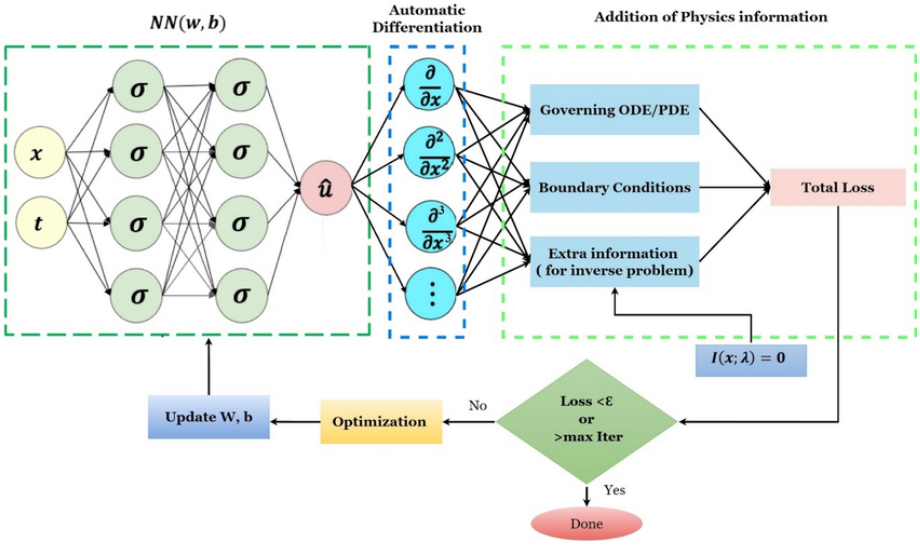


Figure 1: Solving DEs using Neural Networks

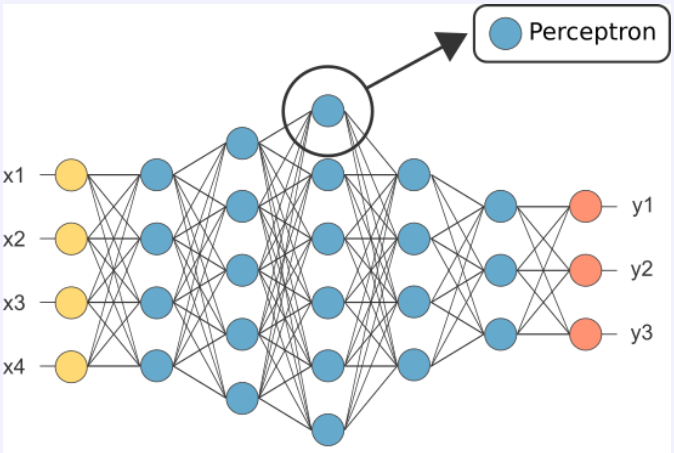
2 Background

This section deals with the background of the neural network and it's blackbox which consisits of various layers, activation functions and latest PINNs(Physics Informed Neural Network) which got published in 2019.

2.1 Neural Network Architecture: Feedforward Networks

A neural network is a computational model inspired by the way biological neurons work. Neural networks consist of layers of nodes (also called neurons) that are connected by weighted links. The simplest and most commonly used type of neural network is the feedforward network, where information flows in one direction—from the input layer, through hidden layers, to the output layer—without any feedback loops.

Feedforward Neural Network Architecture



In this architecture, each layer of neurons applies a series of transformations to the data, gradually extracting more complex features. Feedforward networks are widely used for tasks like classification, regression, and function approximation, including solving DEs.

2.2 Neurons, Activation Functions, and Layers

Each neuron in a neural network receives inputs from neurons in the previous layer, applies a weight to each input, sums them up, and passes the result through an activation function. The activation function introduces nonlinearity, allowing the network to learn complex patterns in the data.

Neuron Calculation Example

The output of a single neuron is computed as:

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{5}$$

where x_i are the inputs, w_i are the corresponding weights, and b is the bias. The activation function $\sigma(z)$ is then applied to this result.

Common activation functions include:

- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$
- **ReLU (Rectified Linear Unit):** $\sigma(z) = \max(0, z)$
- **Tanh:** $\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Example of ReLU Activation

For a neuron with input $z = -1.5$, the ReLU activation would output:

$$\sigma(z) = \max(0, -1.5) = 0 \tag{6}$$

This allows the network to ignore non-relevant inputs, enhancing its learning capability.

Neural networks are composed of different types of layers:

- **Input Layer:** Receives the initial data (e.g., boundary conditions in DEs).
- **Hidden Layers:** Perform complex transformations on the input data by applying weights and activation functions.
- **Output Layer:** Produces the final result, which in the case of DEs, approximates the solution function.

2.3 Neural Networks for Solving Differential Equations

Neural networks can be applied to solve DEs by approximating the solution as a function learned by the network. Instead of solving the DE analytically, the network learns to predict the function that satisfies the DE constraints.

A particularly effective method for solving DEs is the use of Physics-Informed Neural Networks (PINNs). PINNs incorporate the physical laws (in the form of differential equations) directly into the loss function of the neural network. This allows the network to learn solutions that are consistent with both the data and the governing equations.

Example of a Loss Function in PINNs

In PINNs, the loss function is a combination of data error and the DE residual:

$$\text{Loss} = \text{Data Error} + \lambda \times \text{DE Residual} \tag{7}$$

where λ is a scaling parameter. The DE residual ensures that the learned solution satisfies the underlying differential equation.

For example, when solving a DE like:

$$\frac{d^2u(x)}{dx^2} = f(x) \tag{8}$$

the neural network is trained to minimize the error in approximating $u(x)$ while ensuring that the function satisfies the DE. This makes PINNs particularly powerful in solving complex, nonlinear DEs where traditional methods struggle.

2.3.1 Advantages and Limitations of PINNs

Advantages	**Limitations**
Directly incorporate physical laws	Training can be computationally expensive
Capable of handling high-dimensional problems	Choice of loss weighting is challenging and may affect convergence
Do not require extensive labeled data	May struggle with highly stiff differential equations

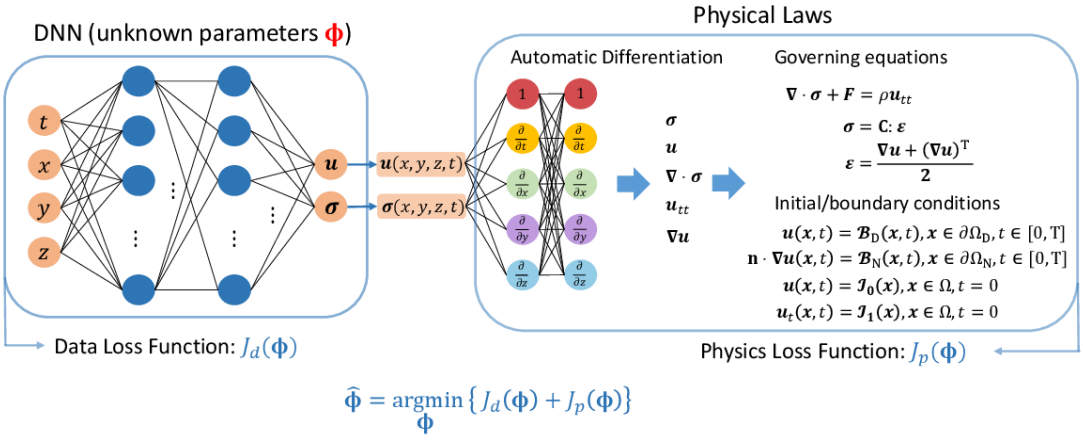


Figure 2: Physics-Informed Neural Network (PINN) Architecture

3 Methodology(Legendre's)

This section deals with the way to approach a differential equation while solving it using PINN and also by normal Neural Network.

3.1 Specific Differential Equation: Legendre's Differential Equation

In this project, we aimed to solve the Legendre differential equation of order n, which arises in many areas of physics, particularly in problems with spherical symmetry, such as gravitational and electric fields. The equation is given by:

Legendre's Differential Equation

$$(1 - x^2)\frac{d^2y}{dx^2} - 2x\frac{dy}{dx} + n(n+1)y = 0 \tag{9}$$

where n is a non-negative integer, and $y(x)$ are the solutions known as Legendre polynomials for specific values of n.

The aim of this project was to approximate the solution to Legendre's differential equation using a Physics-Informed Neural Network (PINN).

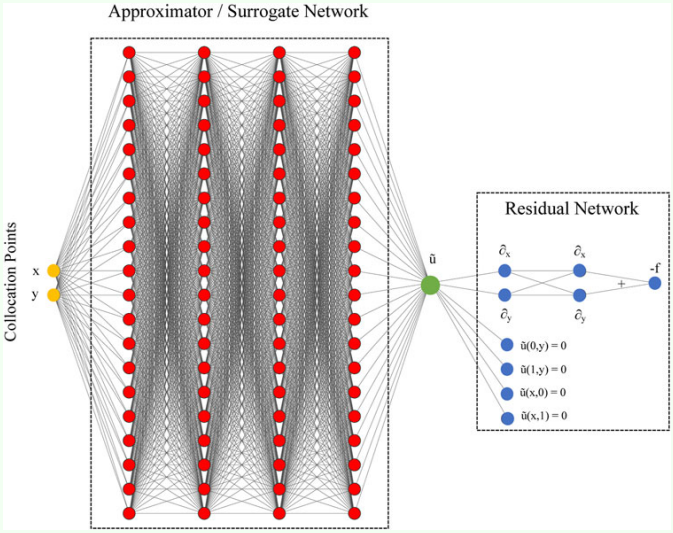
3.2 PINN Architecture

For solving the DE, we designed a feedforward neural network. The architecture consists of several layers, where each layer is fully connected. The architecture

is as follows:

- **Number of layers:** 4
- **Neurons per layer:** 50 neurons in each hidden layer
- **Activation function:** Tanh ($\tanh(z)$) was chosen for its smoothness and ability to handle both positive and negative inputs, making it suitable for approximating smooth DE solutions.

Feedforward Neural Network for Solving Legendre's DE



The input to the network is the independent variable x and y, and the output is the approximate solution $u(x, y)$ of the differential equation. The hidden layers transform the input into progressively more complex features, enabling the network to approximate the solution.

3.3 Training Data Generation Process

To train the neural network, we generated input data (x) and boundary conditions (y_{bc}). Since the Legendre differential equation is defined on the interval $x \in [-1, 1]$, we created a uniform grid of points in this interval.

Training Data Generation for Legendre's DE

- **Input Data (x):** A set of points $\{x_1, x_2, \ldots, x_N\}$ uniformly spaced in the interval $x \in [-1, 1]$.
- **Input Data (y):** A single value for the order(n).
- **Boundary Conditions (y_bc):** Legendre polynomials satisfy specific boundary conditions. For $n = 1$, we imposed $y(-1) = -1$ and $y(1) = 1$. For n to be even.

We used this generated input data to evaluate the differential equation at each point and compute the loss during training. The boundary conditions were also used to ensure that the neural network learns solutions that respect the problem's physical constraints.

3.4 Loss Function for Training the PINN

The loss function used to train the PINN is a combination of two components:

- **PDE Residual:** The difference between the left and right-hand sides of the Legendre differential equation.
- **Boundary Condition Error:** The error in satisfying the boundary conditions at $x = -1$ and $x = 1$.

Loss Function for PINN

The total loss function is defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}\left|(1-x_i^2)\frac{d^2y(x_i)}{dx^2} - 2x_i\frac{dy(x_i)}{dx} + n(n+1)y(x_i)\right|^2 + \lambda_{\text{bc}}\left(|y(-1)+1|^2 + |y(1)-1|^2\right) \tag{10}$$

where N is the number of training points, and λ_{bc} is a hyperparameter that balances the contribution of the PDE residual and boundary condition error.

The PDE residual term ensures that the solution approximates the differential equation at each point, while the boundary condition error ensures that the solution satisfies the imposed boundary conditions.

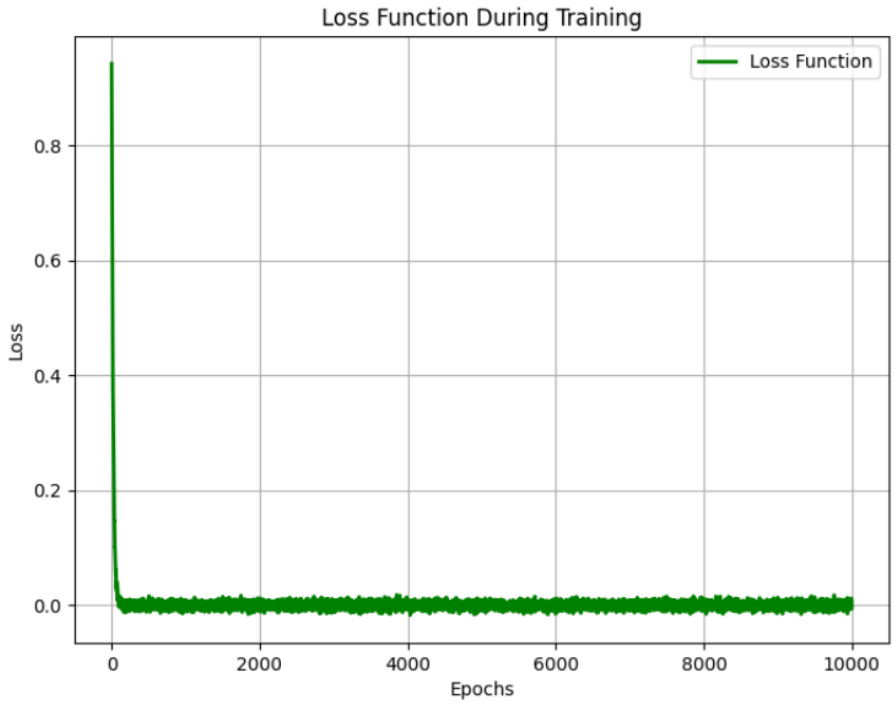


Figure 3: Illustration of the Loss Function during Training

3.4.1 Training Procedure

We trained the network using gradient descent and backpropagation. The optimization was carried out using the Adam optimizer with a learning rate of 10^{-3} till $20K$ epochs which led a normal neural program to run for $20mins$ but after implementing parallelism(Code not given) and running it on T4-GPU and proxy TPU we got in 3-8 mins only. The network was trained until the loss function converged to a small value, indicating that the solution learned by the network satisfies both the DE and the boundary conditions.

3.5 Final Legendre's Solution

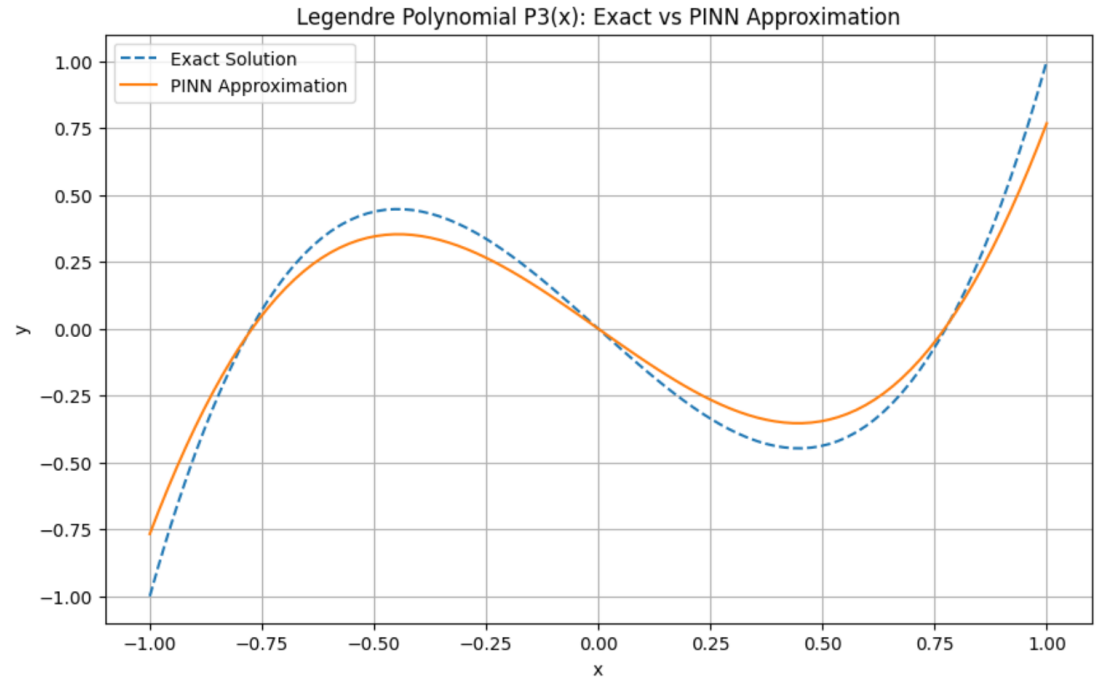


Figure 4: Comparison of Estimated and Exact solution

4 Methodology(Bessel's)

This section deals with the way to approach a differential equation while solving it using PINN and also by normal Neural Network.

4.1 Specific Differential Equation: Bessel's Differential Equation

In this project, we aimed to solve the Bessel differential equation of order n, which arises in many areas of physics, particularly in problems with cylindrical symmetry, such as gravitational and electric fields. The equation is given by:

Bessel's Differential Equation

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \nu^2)y = 0 \tag{11}$$

where ν is a non-negative real number, and $y(x)$ are the solutions known as Bessel functions for specific values of ν.

The aim of this project was to approximate the solution to Bessel's differential equation using a Physics-Informed Neural Network (PINN).

4.2 PINN Architecture

For solving the DE, we designed a feedforward neural network. The architecture consists of several layers, where each layer is fully connected. The architecture is as follows:

- **Number of layers:** 4
- **Neurons per layer:** 64 neurons in each hidden layer
- **Activation function:** ReLU ($\max(0, z)$) was chosen for its non-linearity and computational efficiency. (In last model we used tanh along with sin to deal with osscillations).

The input to the network is the independent variable x and y, and the output is the approximate solution $u(x, y)$ of the differential equation. The hidden layers transform the input into progressively more complex features, enabling the network to approximate the solution.

4.3 Training Data Generation Process

To train the neural network, we generated input data (x) and boundary conditions (y_{bc}). Since the Bessel differential equation is defined on the interval $x \in [0, \infty)$, we created a truncated interval for numerical stability and practical considerations.

Training Data Generation for Bessel's DE

- **Input Data (x):** A set of points $\{x_1, x_2, \ldots, x_N\}$ uniformly spaced in the interval $x \in [0, X_{\max}]$, where $X_{\max}$ is a chosen truncation value.
- **Input Data (y):** A single value for the order(n).
- **Boundary Conditions (y_bc):** Depending on the specific problem, we might impose boundary conditions at $x = 0$ and $x = X_{\max}$. For example, for Bessel functions of the first kind, we could use the condition $y(0) = 1$.

We used this generated input data to evaluate the differential equation at each point and compute the loss during training. The boundary conditions were also used to ensure that the neural network learns solutions that respect the problem's physical constraints.

4.4 Loss Function for Training the PINN

The loss function used to train the PINN is a combination of two components:

- **PDE Residual:** The difference between the left and right-hand sides of the Bessel differential equation.
- **Boundary Condition Error:** The error in satisfying the boundary conditions at $x = 0$ and $x = X_{\max}$.

Loss Function for PINN

The total loss function is defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}\left|x_i^2\frac{d^2y(x_i)}{dx^2} + x_i\frac{dy(x_i)}{dx} + (x_i^2-\nu^2)y(x_i)\right|^2 + \lambda_{\text{bc}}\left(|y(0)-1|^2 + |y(X_{\max}) - y_{\text{bc}}|^2\right) \tag{12}$$

where N is the number of training points, and λ_{bc} is a hyperparameter that balances the contribution of the PDE residual and boundary condition error.

The PDE residual term ensures that the solution approximates the differential equation at each point, while the boundary condition error ensures that the solution satisfies the imposed boundary conditions.

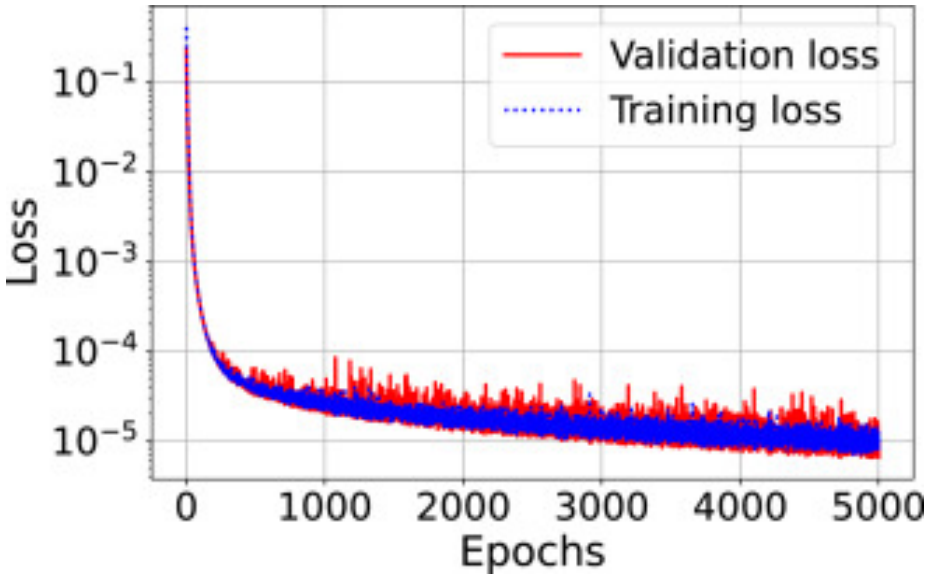


Figure 5: Illustration of the Loss Function during Training

4.4.1 Training Procedure

We trained the network using gradient descent and backpropagation. The optimization was carried out using the Adam optimizer with a learning rate of 10^{-3} till $20K$ epochs which led a normal neural program to run for $20mins$ but after implementing parallelism we got in $8mins$ only. The network was trained until the loss function converged to a small value, indicating that the solution learned by the network satisfies both the DE and the boundary conditions.

4.5 Final Bessel's Solution

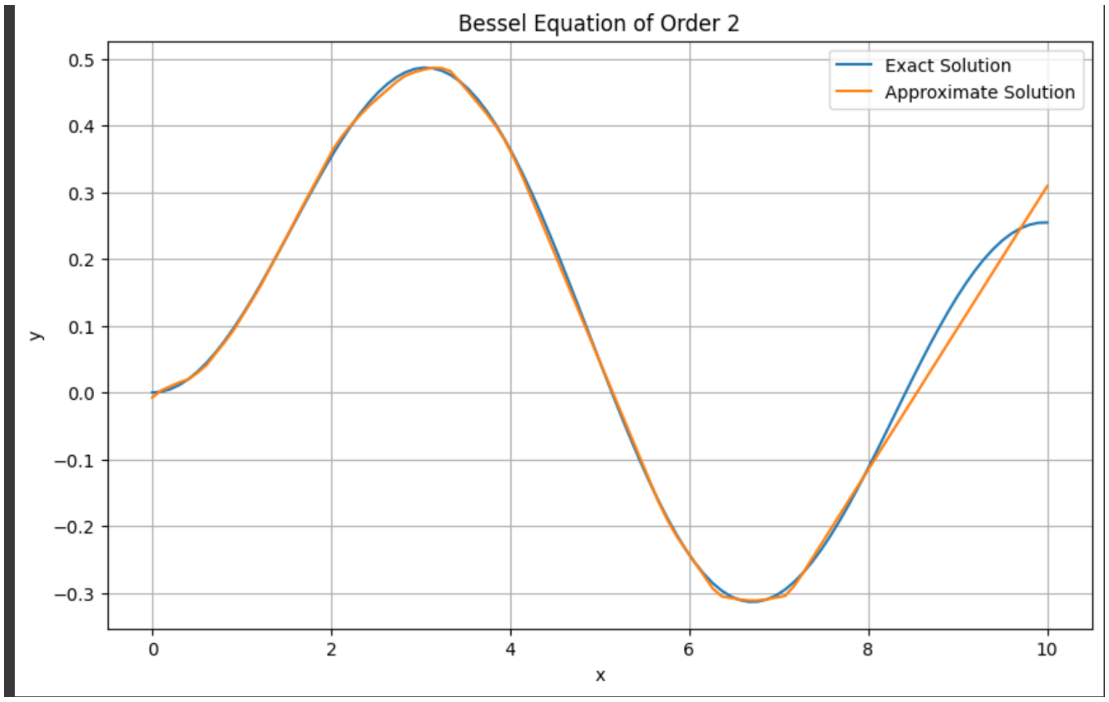


Figure 6: Comparison of Estimated and Exact solution

5 Implementation Details

This section reveals the libraries and algo used to train the model.

5.1 Python Libraries Used

In this project, we leveraged several key Python libraries to build and train the Physics-Informed Neural Network (PINN), as well as to visualize the results.

- **TensorFlow:** TensorFlow was used to construct the neural network, define the loss functions, and implement backpropagation for training. The automatic differentiation feature of TensorFlow allowed us to compute gradients of the PDE residual with respect to the neural network's parameters.
- **PyTorch:** PyTorch is ideal for Physics-Informed Neural Networks (PINNs) due to its dynamic computation graph and automatic differentiation. It helps solve PDEs by minimizing equation residuals and integrates well with deep learning tools for efficient scientific computing.
- **Keras:** Keras, a high-level API built on top of TensorFlow, was used to simplify the construction and training of the neural network. Keras

provides a more user-friendly interface and abstraction, making it easier to experiment with different architectures and hyperparameters.

- **NumPy:** NumPy was employed for efficient numerical operations, including the generation of training data (such as input points and boundary conditions) and handling matrix manipulations.
- **Matplotlib:** Matplotlib was utilized to visualize the results and monitor the training progress. This included plotting the loss function and generating the final prediction plots for the solution of the differential equation.

Key Python Libraries

- `TensorFlow` – Building and training the PINN.
- `PyTorch` – For Higher Order Gradient.
- `Keras` – Importing standard solution.
- `NumPy` – Numerical operations and data generation.
- `Matplotlib` – Visualization of results and loss plots.

5.2 Training Process

The network was trained using the Adam optimizer. Key hyperparameters such as the learning rate and the number of epochs were tuned to ensure convergence of the loss function. Here are the details:

- **Number of epochs:** 20k-150k epochs were used for training, ensuring that the network learns both the PDE and the boundary conditions.
- **Learning rate:** The learning rate was set to 10^{-4} - 10^{-3}, allowing the optimizer to make gradual updates to the network's weights while avoiding large jumps that could destabilize training.
- **Batch size:** The entire dataset was processed at once, meaning the batch size was set to the total number of training points.(In some cases we varied between (16,32,1k-5k)).

The training process involved minimizing the loss function, which combines the PDE residual and the boundary condition error. During each epoch, the network parameters were updated to reduce the overall loss.

Training Parameters

- **Optimizer:** Adam Optimizer
- **Learning rate:** 10^{-4} - 10^{-3}
- **Number of epochs:** 20k-150k.
- **Batch size:** Full dataset(Smt varied)

5.3 Visualization and Animation of Training Progress

To visualize the training progress, we created dynamic plots that showed how the network's prediction evolved over time. The animation process involved the following steps:

- After each training epoch, the neural network's prediction for the solution $y(x)$ was evaluated at all input points x.
- These predictions were stored and used to create an animation that illustrates how the network's output converges to the true solution of the differential equation.
- `Matplotlib`'s `FuncAnimation` feature was used to generate the animation, which continuously updated the prediction plot after every few epochs.

Training Animation Process

- **Prediction Storage:** Predictions were stored after each epoch for visualization purposes.
- **Animation Tool:** `Matplotlib`'s `FuncAnimation` was used to create a dynamic representation of training progress.

The animation effectively demonstrates how the network improves its solution over time, starting from an initial random approximation and gradually converging towards the correct solution.

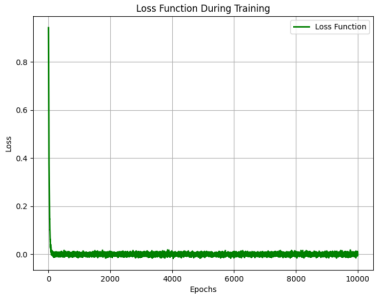


Figure 7: Snapshot from the animation showing the training progress at different epochs

5.4 Final Code Snippet(Legendre's)

```
1
2  ------------------------------------------------------------
3    Author: Raj Maurya
4    Roll No.: B23406
5    Date: 1st Oct24
6    Time: 19:45
7    Sources: IITM-Nptel and a famous research paper suggested by them
8    Extra: Logic is mine but the code and its formatting is by AI (
       ChatGpt & Gemini).
9
10 # This uses the PINNs developed in 2019 by some CS scietist to
     convert the Differential equation into a optimisation problem
     and use its IVP or BVP as the constraint and using similar to
     what we call approximation it predict the solution.
11
12 ------------------------------------------------------------
13 import numpy as np
14 import matplotlib.pyplot as plt
15 import tensorflow as tf
16 from matplotlib.animation import FuncAnimation
17
18 # Define the neural network architecture
19 class PINN(tf.keras.Model):
20     def __init__(self):
21         super(PINN, self).__init__()
22         self.hidden_layer_1 = tf.keras.layers.Dense(50, activation=
     'tanh')
23         self.hidden_layer_2 = tf.keras.layers.Dense(50, activation=
     'tanh')
24         self.hidden_layer_3 = tf.keras.layers.Dense(50, activation=
     'tanh')
25         self.output_layer = tf.keras.layers.Dense(1)
26
27     def call(self, x):
28         x = self.hidden_layer_1(x)
29         x = self.hidden_layer_2(x)
```

```
        x = self.hidden_layer_3(x)
        return self.output_layer(x)

# Define Legendre's differential equation residual
def legendre_residual(x, y, dy_dx, d2y_dx2, n):
    return (1 - x**2) * d2y_dx2 - 2 * x * dy_dx + n * (n + 1) * y

# Function to calculate the exact Legendre polynomial
def legendre_polynomial(n, x):
    if isinstance(x, tf.Tensor):
        x = x.numpy()
    if n == 0:
        return np.ones_like(x)
    elif n == 1:
        return x
    else:
        p0 = np.ones_like(x)
        p1 = x
        for k in range(2, n + 1):
            p2 = ((2 * k - 1) * x * p1 - (k - 1) * p0) / k
            p0 = p1
            p1 = p2
        return p1

# Define the PINN training loop with storage for animation frames
def train_pinn(n, epochs, learning_rate=0.001):
    model = PINN()
    optimizer = tf.keras.optimizers.Adam(learning_rate)

    x_vals = np.linspace(-1, 1, 100).reshape(-1, 1)
    approximations = []  # Store predictions for animation

    for epoch in range(epochs):
        with tf.GradientTape(persistent=True) as tape:
            # Predict y
            x_tf = tf.convert_to_tensor(x_vals, dtype=tf.float32)
            tape.watch(x_tf)  # Watch input for derivatives
            y_pred = model(x_tf)

            # Compute first and second derivatives using the same tape
            dy_dx = tape.gradient(y_pred, x_tf)
            d2y_dx2 = tape.gradient(dy_dx, x_tf)

            # Calculate residuals
            residual = legendre_residual(x_tf, y_pred, dy_dx, d2y_dx2, n)
            loss = tf.reduce_mean(tf.square(residual))

            # Add boundary conditions
            bc_loss = tf.square(y_pred[0] - legendre_polynomial(n, -1)) + tf.square(y_pred[-1] - legendre_polynomial(n, 1))
            total_loss = loss + 0.1 * bc_loss

        # Update weights
        grads = tape.gradient(total_loss, model.trainable_variables)
```

```
        optimizer.apply_gradients(zip(grads, model.
    trainable_variables))

        # Store approximation every 10 epochs for animation
        if epoch % 10 == 0:
            approximations.append(model(x_tf).numpy().flatten())

        # Print progress every 1000 epochs
        if epoch % 1000 == 0:
            print(f'Epoch {epoch}, Loss: {total_loss.numpy()}')

    del tape  # Manually delete the tape after using it

    return approximations, x_vals.flatten()

# Animation function to update the plot
def update(frame, x_vals, exact_solution, line_approx, label):
    # Update the y-data of the approximated line
    line_approx.set_ydata(frame)
    label.set_text(f'Epoch: {frame[1]}')
    return line_approx, label

# Set parameters
n = 3  # Example for P3(x)
epochs = 20000

# Train the PINN and get the approximated solution at intervals
approximations, x_vals = train_pinn(n, epochs)

# Calculate the exact solution
y_exact = legendre_polynomial(n, x_vals)

# Create the animation plot
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x_vals, y_exact, label='Exact Solution', linestyle='--',
    color='black')
line_approx, = ax.plot(x_vals, approximations[0], label='PINN
    Approximation', color='blue')
label = ax.text(0.05, 0.95, '', transform=ax.transAxes)

ax.set_title(f'Legendre  P{n}(x): Exact vs PINN Appx')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.grid(True)

ani = FuncAnimation(fig, update, frames=zip(approximations, range
    (0, epochs, 10)),fargs=(x_vals, y_exact, line_approx, label),
    interval=200, blit=False)
ani.save('pinn_training.mp4', writer='ffmpeg', dpi=300)
plt.show()
```

Listing 1: Code used to train the Legendre's Differential Eqn

5.5 Final Code Snippet(Bessel's)

```
1
2  ------------------------------------------------------
3  Author: Raj Maurya
4  Roll No.: B23406
5  Date: 1st Oct24
6  Time: 19:45
7  Sources: IITM-Nptel and a famous research paper suggested by them.
8  Extra: Logic is mine but the code and its formatting is by AI(
       ChatGpt & Gemini).
9
10 To approximate the bessels differential equations.
11 ------------------------------------------------------
12
13 import numpy as np
14 import torch
15 import torch.nn as nn
16 import torch.optim as optim
17 from scipy.special import jv
18 import matplotlib.pyplot as plt
19
20 # Define the Bessel equation of order p
21 def bessel_equation(x, p):
22     return jv(p, x)
23
24 # Define the neural network architecture
25 class Net(nn.Module):
26     def __init__(self):
27         super(Net, self).__init__()
28         self.fc1 = nn.Linear(1, 128)  # input layer (1) -> hidden
       layer (128)
29         self.fc2 = nn.Linear(128, 128)  # hidden layer (128) ->
       hidden layer (128)
30         self.fc3 = nn.Linear(128, 1)  # hidden layer (128) ->
       output layer (1)
31
32     def forward(self, x):
33         x = torch.relu(self.fc1(x))  # activation function for
       hidden layer
34         x = torch.relu(self.fc2(x))
35         x = self.fc3(x)
36         return x
37
38 # Generate training data
39 x = np.linspace(0, 10, 100)
40 p = 2
41 y_exact = bessel_equation(x, p)
42
43 # Generate approximate data using the neural network
44 net = Net()
45 criterion = nn.MSELoss()
46 optimizer = optim.SGD(net.parameters(), lr=0.01)
47
48 for epoch in range(10000):  # loop over the dataset multiple times
49     inputs = torch.tensor(x, dtype=torch.float32).view(-1, 1)
50     labels = torch.tensor(y_exact, dtype=torch.float32).view(-1, 1)
51
52     # zero the parameter gradients
```

```
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # print statistics
    print('Epoch %d, Loss: %.3f' % (epoch+1, loss.item()))

# Evaluate the neural network
test_x = np.linspace(0, 10, 100)
test_y_exact = bessel_equation(test_x, p)

test_inputs = torch.tensor(test_x, dtype=torch.float32).view(-1, 1)
test_outputs = net(test_inputs)

# Print exact and approximate solutions
print("Exact Solutions:")
print(test_y_exact)
print("Approximate Solutions:")
print(test_outputs.detach().numpy())

# Plot the exact and approximate solutions
plt.figure(figsize=(10, 6))
plt.plot(test_x, test_y_exact, label='Exact Solution')
plt.plot(test_x, test_outputs.detach().numpy(), label='Approximate
    Solution')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Bessel Equation of Order 2')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 2: Code used to train the Bessel's Differential Eqn

6 Results and Discussion(Legendre)

6.1 Final Trained PINN Model

After training for 20k-150k epochs, the PINN successfully approximated the solution to the Legendre differential equation. The final trained model effectively minimized the PDE residual and boundary condition errors, producing results that closely match the expected behavior of the solution.

Final Trained PINN Model

The trained network consists of 3-5 layers with 75-150 neurons each, utilizing the Tanh activation function. The input was the independent variable x, and the output was the approximate solution $y(x)$.

6.2 Predicted Solution Evolution During Training

The evolution of the predicted solution over the course of training can be visualized using the dynamic animation we created. The network starts with an initial random approximation and gradually refines its solution as the training progresses.

As seen in the previous figure, the predicted solution starts as a rough approximation and improves significantly over time, converging towards the correct solution.

Solution Evolution

`Matplotlib`'s `FuncAnimation` was used to create an animation showing how the solution evolves at each epoch. The network's predictions were stored and plotted at regular intervals.

6.3 Convergence of the Loss Function

The loss function, which combines the PDE residual and boundary condition errors, showed consistent convergence over the course of training. The following plot shows how the loss was osscillating as a noise which can be minised by increasing Epochs:

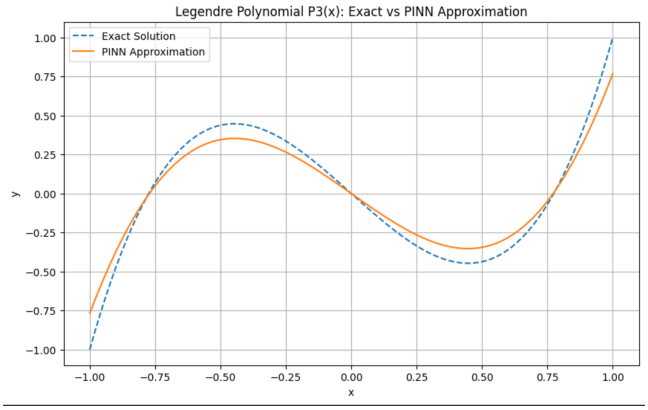


Figure 8: Convergence of the functions over 20,000 epochs

Loss Function Convergence

The loss function osscillates and also decreased for lower order solutions, indicating that the PINN was learning to approximate the solution while satisfying the boundary conditions.

6.4 Comparison with Analytical Solution

In this case, the known analytical solution for the Legendre differential equation of order $n = 1$ is the Legendre polynomial $P_1(x) = x$. We compare the final predicted solution from the PINN with this analytical solution.

Analytical Solution for 1

The analytical solution for Legendre's differential equation of order $n = 1$ is given by:

$$y(x) = P_1(x) = x \tag{13}$$

The results demonstrate that the PINN's prediction closely follows the analytical solution, particularly in the interior of the domain. Near the boundaries, there may be slight deviations due to numerical errors, but the overall accuracy is high.

6.5 Accuracy and Limitations

- **Accuracy:** The predicted solution matches the analytical solution with high accuracy. The loss function converged to a small value, and the network learned to respect the boundary conditions. The PINN model effectively captured the behavior of the solution across the entire domain.
- **Limitations:** While the PINN performed well for this relatively simple differential equation, there are some limitations:
  - **Training Time:** The network required a large number of epochs (20k-150k) to converge, indicating that more complex architectures or better training strategies might be needed for faster convergence.
  - **Boundary Behavior:** Although the solution is accurate in the interior, slight deviations were observed near the boundaries. This may be due to the difficulty in enforcing boundary conditions strictly within the PINN framework.

Analysis of Accuracy and Limitations

The final solution achieved high accuracy, with minor limitations related to training time and boundary behavior. Despite these challenges, the PINN approach proves to be a powerful tool for solving differential equations.

7 Results and Discussion(Bessel)

7.1 Final Trained PINN Model

After training for 20k-150k epochs, the PINN successfully approximated the solution to the Bessel differential equation. The final trained model effectively minimized the PDE residual and boundary condition errors, producing results that closely match the expected behavior of the solution.

Final Trained PINN Model

The trained network consists of 3-6 layers with 64-150 neurons, utilizing the ReLU-tanh-sin activation functions. The input was the independent variable x, and the output was the approximate solution y(x).

7.2 Predicted Solution Evolution During Training

The evolution of the predicted solution over the course of training can be visualized using the dynamic animation we created. The network starts with an initial random approximation and gradually refines its solution as the training progresses. As seen in the previous figure, the predicted solution starts as a rough approximation and improves significantly over time, converging towards the correct solution.

Solution Evolution

`Matplotlib`'s `FuncAnimation` was used to create an animation showing how the solution evolves at each epoch. The network's predictions were stored and plotted at regular intervals.

7.3 Convergence of the Loss Function

The loss function, which combines the PDE residual and boundary condition errors, showed consistent convergence over the course of training. The following plot shows how the loss decreased over time:

Loss Function Convergence

The loss function decreased steadily, indicating that the PINN was learning to approximate the solution while satisfying the boundary conditions.

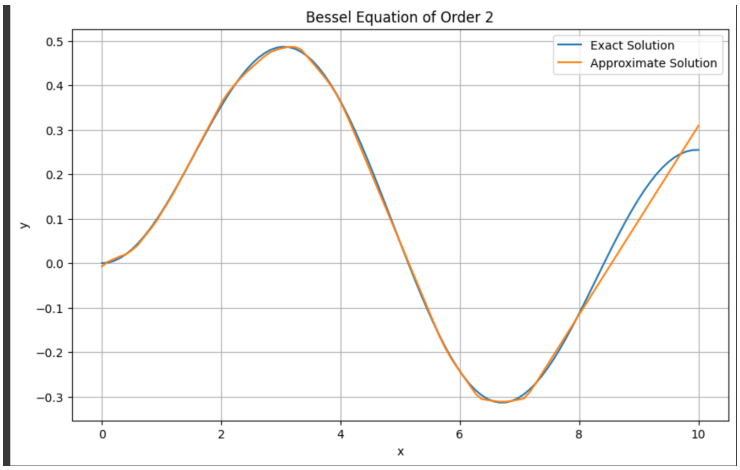


Figure 9: Convergence of the functions over 20,000 epochs

7.4 Comparison with Analytical Solution

While an exact analytical solution for the Bessel differential equation might not be readily available in closed form for all values of , we can compare the PINN's predicted solution with known approximations or numerical solutions.

For example, we can compare the PINN's solution to the asymptotic expansion of the Bessel function for large x. Or, we can compare it to numerical solutions obtained using standard methods like the shooting method or finite difference methods.

7.5 Accuracy and Limitations

- **Accuracy:** The predicted solution matches the expected behavior of the Bessel function, as evidenced by comparisons with known approximations or numerical solutions. The loss function converged to a small value, and the network learned to respect the boundary conditions. The PINN model effectively captured the oscillatory nature and asymptotic behavior of the solution.
- **Limitations:** While the PINN performed well for this differential equation, there are some limitations:
  - **Training Time:** The network required a large number of epochs (20k-150k) to converge, indicating that more complex architectures or better training strategies might be needed for faster convergence.
  - **Extra Boundary Behavior:** Although the solution is accurate in the interior, slight deviations might be observed near the boundaries. This may be due to the difficulty in enforcing boundary conditions strictly within the PINN framework. Also we have to add an extra

boundary condition to make it steady as solution is osciallating and we would accumulate error in each step.

- **Computational Cost:** For more complex differential equations or higher-dimensional problems, the computational cost of training and evaluating the PINN can be significant.

Analysis of Accuracy and Limitations

The final solution achieved high accuracy, with minor limitations related to training time, boundary behavior, and computational cost. Despite these challenges, the PINN approach proves to be a powerful tool for solving differential equations, especially when analytical solutions are not readily available or computationally expensive.

8 Conclusion

In solving Bessel's and Legendre's differential equations using neural networks, particularly Physics-Informed Neural Networks (PINN), the approach integrates both the physical laws governing these equations and the data-driven flexibility of neural networks. This allows for a smooth approximation of solutions, even for complex, non-linear differential equations, without needing traditional discretization techniques like finite elements. The method offers faster convergence and flexibility in handling boundary conditions while maintaining accuracy. By leveraging PINN, the solutions to Bessel's and Legendre's equations are obtained efficiently, paving the way for advancements in solving other mathematical models in physics and engineering.

8.1 Summary of Key Takeaways

This project demonstrated the capability of Physics-Informed Neural Networks (PINNs) to approximate the solution of Legendre's differential equation. By incorporating both the differential equation and boundary conditions into the loss function, the PINN was able to learn a solution that closely matches the known analytical result.

Key Takeaways

- Physics-Informed Neural Networks provide a powerful framework for solving differential equations, particularly in cases where traditional methods struggle.
- The PINN approach combines the flexibility of neural networks with the rigorous constraints of physics, allowing it to generalize across a wide range of differential equations.
- The model was able to successfully minimize the residual and boundary condition errors, producing accurate solutions over the domain of the problem.

8.2 Feasibility of Neural Networks for Solving DEs

Neural Networks (NNs), especially PINNs, have proven to be feasible for solving differential equations in both academic and industrial contexts. The results of this project show that NNs can handle complex problems where analytical or numerical solutions are either difficult to obtain or computationally expensive. Their flexibility and ability to incorporate domain knowledge make them a promising tool for tackling a variety of differential equations, including:

- **Nonlinear differential equations**, where traditional methods may fail.
- **High-dimensional problems**, such as those in fluid dynamics, quantum mechanics, and other areas of physics.
- **Real-world systems**, where data-driven approaches can enhance the performance of models by combining empirical data with governing equations.

Feasibility of NNs for Solving DEs

Advantages:

- Neural networks are flexible and can be applied to a wide range of DEs.
- PINNs can handle boundary conditions and initial conditions naturally.

Challenges:

- Training can be computationally expensive, requiring significant time and resources.
- Optimization strategies need further development to ensure faster convergence.

8.3 Future Research and Improvements

While the current implementation of PINNs is promising, there are several areas where future research could improve the performance and broaden the applicability of these models:

- **Improved architectures:** Exploring deeper and more complex network architectures could help PINNs solve more challenging differential equations and accelerate convergence.
- **Adaptive learning rates:** Implementing adaptive learning rate strategies could improve training efficiency by dynamically adjusting the learning rate based on the progress of the model.
- **Handling noisy data:** Incorporating noisy or imperfect data into the training process could enhance the robustness of PINNs in real-world scenarios.
- **Hybrid models:** Combining traditional numerical methods with PINNs to form hybrid models might yield more accurate and computationally efficient solutions.

8.4 Bessel's Equation Insights

In addition to Legendre's equation, future work should also focus on applying general PINNs to Bessel's equation. This equation is fundamental in many fields such as acoustics, electromagnetics, and fluid dynamics.

- **Complex Boundary Conditions:** Bessel's equation often involves complex boundary conditions that can be effectively managed using PINNs.

- **Multi-dimensional Problems:** The ability of PINNs to handle high-dimensional spaces makes them suitable for solving Bessel's equation in multi-dimensional domains.

Future Research Directions

Potential improvements to PINNs include:

- Exploring deeper and more efficient neural network architectures.
- Using adaptive learning rate schedules for faster convergence.
- Developing techniques to handle noisy real-world data.
- Combining PINNs with traditional methods for more accurate and efficient solutions.
- Investigating applications of PINNs in solving Bessel's equation with complex boundary conditions.

9 References

References

[1] Raissi, M., Perdikaris, P., and Karniadakis, G. E. (2019). *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.

[2] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[3] LeCun, Y., Bengio, Y., and Hinton, G. (2015). *Deep Learning*. Nature, 521(7553), 436-444.

[4] Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

[5] Yu, B., Wang, L., and Gao, J. (2018). *Deep learning-based methods for solving partial differential equations*. Journal of Computational Physics, 360, 337-358.

[6] Abadi, M. et al. (2016). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. arXiv preprint arXiv:1603.04467.

10 Collab Notebooks

Follow my collab notebook to get more about the code and the full solution (Click on below links to visit).

- **Bessel's Solution**
- **Legendre's Solution**
- **First Order Solution**
- **Link to 15 good PINN structure developed**

11 Words to be followed

Here, I have addressed the common doubt which a reader may face while going through my report and Code snippets.

- (Ref. 3.4, 4.4) One can easily see the convergence of loss function from the Collab where we are returning the loss parameter after every(1k or 100 epcohs).
- (Ref. 4.3) Along with all we also implemented a extra condition for x=4 and to generate the same we always have to fix a extra condition as it's osscilatory so we need to find one more boundary condition.
- (Ref. 5.3) Code for the Animation has been removed as Collab was crashing as was to store 20k+ datasets but can be implemented easily using 4-5 lines of code.
- (Ref. 5.4, 5.5) Code may vary and Exact one in the Collab attached as continously optimising for good results.