Solving Differential Equations Using Neural Networks

Aarsh Sanghavi (2301CS01)
IIT Patna

April 4, 2025

Abstract

Differential equations are widely used in mathematical modeling to describe real-world phenomena in physics, engineering, and other scientific disciplines. Traditional numerical methods for solving differential equations can be computationally expensive and may not generalize well to complex or high-dimensional problems. This paper presents an alternative approach using neural networks to approximate solutions to differential equations. By leveraging automatic differentiation and deep learning techniques, we train a neural network to minimize a loss function based on both the differential equation and initial conditions. We implement this method using PyTorch and demonstrate its effectiveness in solving a sample equation.

1 Introduction

Differential equations describe the relationship between a function and its derivatives. They appear in many scientific and engineering disciplines, such as physics (motion equations), chemistry (reaction rates), and finance (Black-Scholes equation). The general form of a first-order ordinary differential equation (ODE) is:

$$\frac{dy}{dx} = f(x, y), \tag{1}$$

where $f(x, y)$ is a given function that defines the relationship between x and y. Given an initial condition:

$$y(x_0) = y_0, \tag{2}$$

our goal is to determine $y(x)$ that satisfies this equation over a given domain.

2 Algorithm for Neural Network-Based Solver

The process of solving a differential equation using a neural network consists of the following steps:

2.1 Step 1: Define the Problem

We start by specifying the differential equation we want to solve. Consider the equation:

$$\frac{dy}{dx} = e^{x^2}, \tag{3}$$

with the initial condition:

$$y(0) = 0. \tag{4}$$

2.2 Step 2: Represent the Solution as a Neural Network

Instead of solving for $y(x)$ explicitly, we define a neural network function:

$$\hat{y}(x; \theta), \tag{5}$$

where θ represents the parameters (weights and biases) of the network. The input is x, and the output is $y(x)$. The neural network serves as a universal function approximator capable of learning the solution.

2.3 Step 3: Compute the Derivative

Since the differential equation involves dy/dx, we need to compute this derivative. Using PyTorch's **automatic differentiation**, we compute:

$$D = \frac{d\hat{y}}{dx}. \tag{6}$$

This allows us to evaluate how well the network satisfies the equation.

2.4 Step 4: Define the Loss Function

The neural network is trained by minimizing a **loss function** that ensures:

1. The differential equation is satisfied:

$$\left| \frac{d\hat{y}}{dx} - e^{x^2} \right|^2$$

2. The initial condition holds:

$$|\hat{y}(x_0) - y_0|^2$$

The total loss is:

$$L = L_{\text{eq}} + L_{\text{init}}, \tag{7}$$

where:

- L_{eq} enforces the equation constraints.
- L_{init} ensures the network satisfies the initial condition.

2.5 Step 5: Train the Neural Network

We train the neural network using **gradient-based optimization**. The loss function is minimized using the Adam optimizer. Over multiple epochs, the network parameters are adjusted to approximate the true solution.

3 Implementation in Python Using PyTorch

The following Python code implements the above approach:

Listing 1: Solving a Differential Equation Using PyTorch

```
import torch
import torch.nn as nn

# Define x values and initial conditions
x = torch.arange(-2, 2, 0.05, requires_grad=True).unsqueeze(1)
x0 = torch.tensor([[0.0]], requires_grad=True)  # Initial point
y0 = torch.tensor([[0.0]], requires_grad=True)  # Known value at x0

# Define the differential equation
def differential_equation(x, y, D):
    lhs = D
    rhs = torch.exp(x**2)
    return lhs, rhs

# Define the neural network
class NeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(1, 50),
            nn.Tanh(),
            nn.Linear(50, 50),
            nn.Tanh(),
            nn.Linear(50, 50),
            nn.Tanh(),
            nn.Linear(50, 1)
        )

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

# Training the neural network
torch.manual_seed(42)
model = NeuralNet()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
loss_fn = nn.MSELoss()

epochs = 10000
for epoch in range(epochs):
    model.train()

    # Compute prediction and derivative
    Y = model(x)
    D = torch.autograd.grad(Y, x, grad_outputs=torch.ones_like(Y), create_graph=
    # Compute loss
    l, r = differential_equation(x, Y, D)
    loss_eq = loss_fn(l, r)

    # Enforce initial condition
    loss_initial = loss_fn(model(x0), y0)

    # Total loss
    loss = loss_eq + loss_initial

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 10 == 0:
        print(f"Epoch␣{epoch},␣Loss:␣{loss.item()}")

# Evaluate the trained model
with torch.inference_mode():
    final_y = model(x).detach().numpy()
```

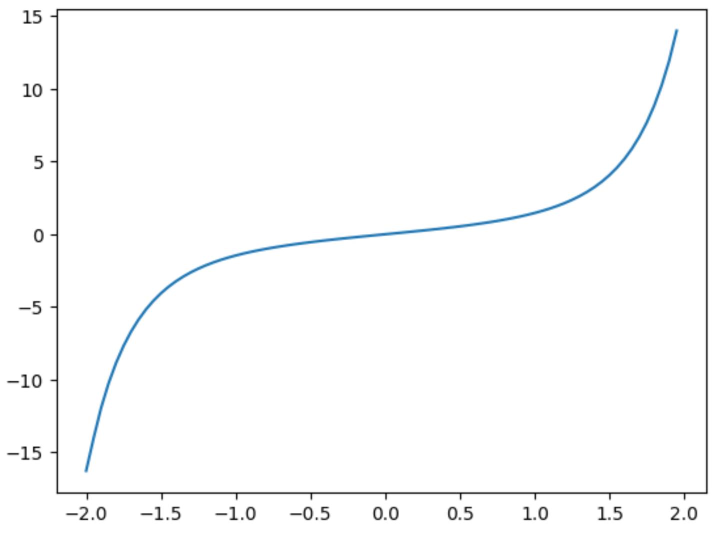

Figure 1: Solution of the differential equation $\frac{dy}{dx} = e^{x^2}$ using a neural network.

4 Results and Discussion

After training, the neural network successfully approximates $y(x)$. The loss function ensures both the differential equation and initial conditions are satisfied. The approach is particularly useful for cases where traditional methods struggle.

5 Conclusion

This paper demonstrated how neural networks can solve differential equations by approximating the solution function. The flexibility of this approach allows for solving complex equations where analytical solutions are difficult to obtain. Future research can extend this method to partial differential equations and higher-order differential equations.

6 Code Repository

The complete Jupyter Notebook is available at: Notebook Link