Solving Ordinary Differential Equations through Neural Networks

B23412 Sahil

June 24, 2025

1 OBJECTIVE

To solve a first order ordinary differential equation with the help of neural networks.

2 PURPOSE

Solving ordinary differential equations (ODEs) using neural networks serves several important purposes, particularly in scientific computing, physics, engineering, and machine learning. Here are the key motivations and benefits:

1.**Approximation of Complex Solutions**
Many ODEs, especially nonlinear or high-dimensional ones, do not have closed-form analytical solutions. Traditional numerical methods like Euler's method, Runge-Kutta, etc., can sometimes be inefficient, especially for complex boundary conditions or in high-dimensional spaces. Neural networks can serve as a function approximator to model the solution of such ODEs, offering flexibility and the ability to handle more intricate problems.

2. **Reduction of Computational Complexity**
In some cases, neural networks can provide a more efficient way to compute solutions to ODEs, particularly for problems that involve large-scale simulations, parameter sweeps, or require real-time predictions. Once trained, a neural network can often evaluate solutions faster than traditional solvers for large-scale or repeated queries.

3. **Learning from Data**
In many real-world applications, the ODE governing the system dynamics might not be fully known, or the system might involve some stochastic or uncertain elements. Neural networks can be trained on data to learn these dynamics, providing a data-driven way to solve the ODEs or at least approximate them in situations where traditional methods cannot be applied.

4. **Generalization Across Initial Conditions**
Unlike traditional solvers that compute solutions for specific initial or boundary conditions, neural networks can be trained to generalize across a wide range of conditions, enabling faster solution retrieval for varying parameters once the network has learned the system's behavior.

5. **Handling High-Dimensional Systems**
Solving high-dimensional ODEs using traditional methods becomes computationally expensive due to the curse of dimensionality. Neural networks, particularly deep learning models, are well-suited for handling high-dimensional problems. They can represent complex systems and dynamics more compactly than grid-based methods.

3 Problem Description and it's Solution

The First Order Initial Value Ordinary differential Equation is

dy/dx = -y + sin(x) , y(0)=1

where x,y are variables and dy/dx is rate of change of y with respect to x.

y(0)=1 is initial condition for this differential equation

Now, we try to solve this differential equation with the help of neural networks. We use python code to find it's solution.We use an in-built library in python which is Tensorflow with keras.Keras provides a high-level API built on top of TensorFlow, making it easier to build, train, and fine-tune neural networks without needing in-depth knowledge of the underlying mechanics. TensorFlow's automatic differentiation system (Autograd) is extremely helpful in solving ODEs using neural networks. When solving ODEs, you need to compute the derivatives of the network output with respect to the input (i.e., dy(x)/dx). TensorFlow's GradientTape allows automatic differentiation, making it efficient and convenient to calculate gradients.

Here is the code I want to include:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
import math
import matplotlib.pyplot as plt

# My First Order ODE: dy/dx = -y + sin(x)
def f(x, y):
    return -y + tf.sin(x)

# Loss function used for computing the residual of the ODE
def loss_fn(model, x_train):
    with tf.GradientTape() as tape:
        tape.watch(x_train)
        y_pred = model(x_train)             # Predict y(x) using the
            model
        dydx_pred = tape.gradient(y_pred, x_train)  # Compute dy/dx using
             autograd
    ode_residual = dydx_pred - f(x_train, y_pred)   # ODE residual
    return tf.reduce_mean(tf.square(ode_residual))  # Mean squared error


def initial_condition_loss(model, x0, y0):
    y0_pred = model(x0)
    return tf.reduce_mean(tf.square(y0_pred - y0))

# Training step that includes both ODE residual and initial condition
def train_step(model, x_train, x0, y0, optimizer):
    with tf.GradientTape() as tape:
        loss = loss_fn(model, x_train)      # ODE loss
        loss += initial_condition_loss(model, x0, y0)  # Add initial
            condition loss
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return loss

# Initial condition: y(0) = 1
x0 = tf.constant([[0.0]])
y0 = tf.constant([[1.0]])
```

```
37
38 # Create model and optimizer
39 model = models.Sequential([
40         layers.Input(shape=(1,)),    #giving input in the model
41         layers.Dense(10, activation="tanh"), #these are hidden layers
42         layers.Dense(10, activation="tanh"),
43         layers.Dense(1)     #it is the output from the model
44     ])
45 optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
46
47 x_train = tf.constant(np.linspace(0, 2, 100).reshape(-1, 1), dtype=tf.
       float32)
48
49 # Code to train the model
50 epochs = 1000
51 for epoch in range(epochs):
52     loss_value = train_step(model, x_train, x0, y0, optimizer)
53     if epoch % 100 == 0:
54         print(f"Epoch {epoch}: Loss = {loss_value.numpy()}")
55
56 x_test = np.linspace(0, 2, 100).reshape(-1, 1).astype(np.float32)
57
58 y_pred = model.predict(x_test)
59
60 for i in range(len(x_test)):
61     print(f"x = {x_test[i][0]:.4f}, y_pred = {y_pred[i][0]:.4f}")
62
63 x=[]
64 y_actual=[]
65 y=[]
66 for i  in range(len(x_test)):
67     x.append(x_test[i][0])
68     y.append(y_pred[i][0])
69     y_actual.append(((math.sin(x_test[i][0])-math.cos(x_test[i][0]))/2) +
           (1.5*math.exp(-x_test[i][0])))
70 plt.plot(x,y_actual,label="actual")
71 plt.plot(x,y,label="predicted")
72 plt.xlabel("Values of X")
73 plt.ylabel("Values of Y")
74 plt.legend()
75 plt.show()
76
77 squared_error=[]
78 for i in range(len(x_test)):
79     squared_error.append((y[i]-y_actual[i])**2)
80
81 plt.figure(figsize=(10, 6))
82 plt.plot(x,squared_error)
83 plt.xlabel("Values of X")
84 plt.ylabel("Squared of diffference b/w actual and predicted Y for given X
       ")
85 plt.show()
```

Brief Explaination of the Code/Methodology:

First of all, we create a function 'f' defining the structure of differential equation. Then, we create a function for calculating the loss during the training of neural networks. Using tf.GradientTape(), we calculate gradient of y with respect to x(that is predicted value). Model also predicts value of y which we put in differential equation to know the actual value of dy/dx at the predicted y.Now we calculate the difference between (dy/dx)predicted and dy/dx at predicted value of y.Now this function returns mean squared error of the difference between (dy/dx)predicted and actual value of dy/dx at the predicted y. Now we define a function which loss the initial value point.Firstly,it predicts y at that initial point x and then return mean

square error of real y and predicted y. Now it's time of the training function 'train-step'.Firstly, it calculates the ODE loss (loss for values of x ahead of initial x).Then it calculates the loss at initial point of differential equation and then sum up both of them.Now using tape.gradient(), we differentiate the loss with respect to model weights and biases.After that, with the help of optimizer,we assign the correct weights and biases that makes the model more accurate. After that, we define the initial conditions.So, it's time to create the model with help of keras. We define an input layer with only one neuron.The first hidden layer consists of 10 neurons and predicts the value of y according to their own weights and biases. Then we apply tanh activation function to it.tanh function introduces non-linearity to the neural network, enabling it to model complex and non-linear relationships effectively.Then we send the outputs of these 10 neurons to the 10 neurons in next hidden layer and apply same activtion function to it.From these 10 neurons, we get an output which is our predicted value of y.Here, we use Adam optimizer with a learning rate of 0.001.Now it's time to measure the accurate weights and biases for the model.We use 1000 epochs here,means 1000 times it back propagates.With each epoch, it calculates the loss value and then update the weights and biases with help of the optimizer. So we get our accurate weights and biases.Now we can use this model to predict the values of y for certain values of x. How my solution works.

4 Results And Discussion

Figure 1: actual values of Y v/s predicted values of Y

Figure 2: Square of difference between actual and predicted value of Y

Result: By the help of this graphs, we will be able to observe how good a neural network is in solving ordinary differential equations.Neural networks offer a flexible and powerful approach to solving differential equations, especially in cases where traditional methods struggle. They can approximate complex, non-linear solutions while efficiently handling high-dimensional problems. In fact, there are some errors in prediction, but still it is very useful in giving rough idea of a solution for a particular value of x.

5 References

1. Artificial Neural Networks For Engineers and Scientists:A book by S. Chakraverty and Susmita Mall
2.Youtube Video1
3.Youtube Video2