Swarnadeep Seth - Physics Ph.D.

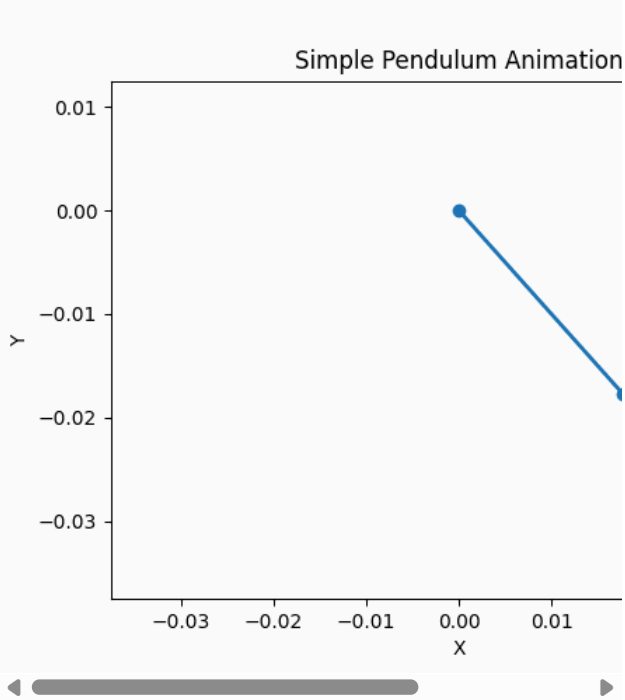
Pendulum's trajectory prediction using Physics Informed Neural Network (PINN)

Mar 23, 2023

Simple Pendulum Ordinary Differential Equation Neural Network Prediction

Simple Pendulum Problem:

A simple pendulum consists of a mass (called the pendulum bob) attached to a string or rod of fixed length. When the pendulum is displaced from its equilibrium position and released, it undergoes periodic motion.



Equations of Motion (ODE)

The motion of a simple pendulum can be described by the following second-order ordinary differential equation (ODE):

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0$$

Where:

- θ is the angular displacement of the pendulum from the vertical (radians)
- t is time (seconds)
- g is the acceleration due to gravity (m/s^2)
- L is the length of the pendulum (meters)

Exact Solution

The exact solution to the pendulum's equation of motion can be derived using the Jacobi elliptic function:

For a more detailed analysis of the non-linear pendulum and its complete solution, you can refer to the following medium article:

A Complete Solution to the Non-Linear Pendulum

Small Angle Approximation Solution

For small angular displacements ($\theta \ll 10^0$), the pendulum's motion can be approximated using the small angle approximation:

$$\sin(\theta) \approx \theta$$

Substituting this approximation into the pendulum's equation of motion yields the following second-order ordinary differential equation (ODE):

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\theta = 0$$

Which has the following exact solution:

$$\theta(t) \approx \theta_0 \cdot \cos\left(\sqrt{\frac{g}{L}} \cdot t\right)$$

Where θ_0 is the initial angular displacement.

Until and unless we solve the ODE numerically, it is very complicated to obtain the exact solution. This brings us to the use of Physics Informed Neural Networks to solve the ODE numerically which solves the following challenges:

- It is very difficult to obtain the analytical solution of every problem and need to numerically solves the equation.
- Solving complicated ODEs is computationally expensive. Specially when modeling a real experimental system where many external factors and dissipative terms are involved.
- For any change in the initial condition, we need to re-do the ODE solution.

The PINN overcomes these problems. A few good training data from experiment would be sufficient to train the neural network and then we can predict the solution for any initial condition. Also, the PINN is very fast in predicting the solution. Now, let's see how we can use PINN to solve the ODE of the simple pendulum.

Physics-Informed Neural Networks:

Physics-Informed Neural Networks (PINNs) are a class of machine learning techniques that combine neural networks with the physical laws governing a system. They incorporate domain-specific knowledge into learning, making them useful for solving partial differential equations (PDEs) and physics-based problems. PINNs use neural networks to approximate the underlying physics while fitting available data.

Some Applications of PINNs

1. Fluid Dynamics: PINNs are applied in fluid dynamics simulations to predict fluid flow patterns, turbulence, and other complex phenomena described by the Navier-Stokes equations.

2. Heat Transfer: PINNs model heat conduction, radiation, and convection in materials, aiding in designing efficient cooling systems and predicting temperature distributions.

3. Structural Mechanics: PINNs simulate stress, strain, deformation, and failure of structures under mechanical loads, contributing to safe and robust design.

4. Quantum Mechanics: In quantum chemistry, PINNs solve Schrödinger's equation to predict molecular properties, electronic structures, and chemical reactions.

5. Medical Imaging: PINNs can be used for medical image reconstruction, resolution enhancement, and denoising by incorporating physics-based constraints.

How to proceed?

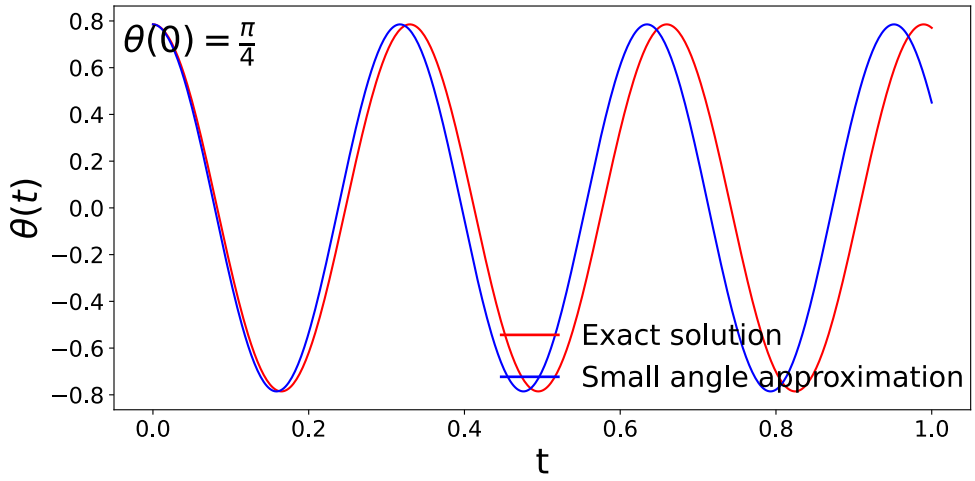
1. **Problem Formulation:** Define the physical problem and governing equations (ODEs/PDEs) with boundary and initial conditions.
2. **Data Collection:** Gather data points from experiments or simulations for neural network training.
3. **Network Architecture:** Design a neural network to approximate the solution.
4. **Loss Function:** Create a loss function that combines data fitting and physics constraints.
5. **Training:** Train the network using backpropagation and optimization techniques.
6. **Physics-Informed Training:** Include terms enforcing physics equations during training.
7. **Validation and Testing:** Evaluate the trained PINN on validation data.
8. **Deployment:** Utilize the trained PINN for predictions, simulations, and optimizations.

Physics-Informed Neural Networks bridge the gap between data-driven machine learning and domain-specific physical insights, offering versatile tools for various scientific and engineering applications.

How does the general solution look?

Let's plot the exact solution and the small angle approximation for a pendulum with the following parameters:

- $g = 9.81\, m/s^2$
- $L = 0.025\, m$
- $\theta_0 = \frac{\pi}{4}\, rad$



We can see, the small angle approximation is only valid for small angular displacements. For larger displacements, the exact solution and the small angle approximation diverge.

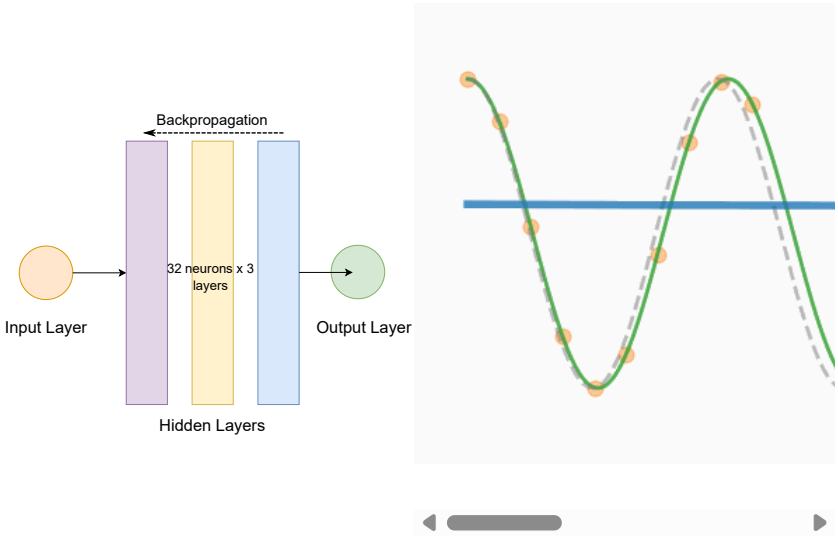
Code Snippet to plot the solutions:

```
import numpy as np
from PIL import Image
import torch
import torch.nn as nn
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Parameters
L = 0.025  # Length of pendulum
g = 9.81  # Acceleration due to gravity
w = np.sqrt(g/L)
k = w**2

end_time = 1

# ================================================
# ODE function for a simple pendulum exact solution
def simple_pendulum_eqn(state, t, L, g):
    theta, theta_dot = state
    theta_ddot = -k*np.sin(theta)
    return [theta_dot, theta_ddot]

# Initial state [theta, theta_dot]
initial_state = [np.pi/4, 0]

# Small Angle Approximation Analytical Solution
def pendulum_solution(w, x):
    return initial_state[0]*np.cos(w*x)

# ================================================
# Create a one-dimensional time array
t = np.linspace(0, end_time, 500)
x = torch.tensor(t, dtype=torch.float32).view(-1, 1)

# Numerical solution of the simple pendulum ODEs
states = odeint(simple_pendulum_eqn, initial_state,
y = torch.tensor(states[:, 0], dtype=torch.float32).
print(x.shape, y.shape)

x_data = x[0:200:20]
y_data = y[0:200:20]
print(x_data.shape, y_data.shape)

# View the analytical solution
View_Analytical_Solution = True
if View_Analytical_Solution:
    fig = plt.figure(figsize=(10, 5))
    plt.plot(x, y, label='Exact solution', color='re
    plt.plot(x, pendulum_solution(w, x), label='Smal
    plt.xlabel('t', fontsize=25)
    plt.ylabel(r'$\theta(t)$', fontsize=25)
    plt.tick_params(labelsize=16)
    plt.legend(frameon=False, fontsize=16)

    # annotate the initial condition in text
    plt.text(0.05, 0.9, r'$\theta(0) = \frac{\pi}{4}

    plt.tight_layout()
    plt.savefig('solution.svg', bbox_inches='tight',
    plt.show()
```

How does a regular neural network perform?

In artificial neural network, the loss function minimizes the difference between the predicted and the actual values. For a simple pendulum, we could use a loss function that minimizes the difference between the predicted and the actual positions of the pendulum at different time steps. The loss function would encourage the neural network to learn parameters that lead to accurate predictions.

However, without the knowledge of the underlying physics principal, ANN, tries to fit the data with a polynomial function. Below, we show the results of a regular neural network with 3 hidden layers and 32 neurons in each layer. The training data are orange dots sampled in equal intervals from the exact solution.



Physics-Informed Neural Networks:

For a physics-inspired neural network modeling a simple pendulum, we could use a loss function that minimizes the difference between the predicted behavior of the pendulum and the actual physics equations governing its motion. This might involve minimizing the mean squared error between predicted and actual positions, velocities, or energies of the pendulum at different time steps. The loss function would encourage the neural network to learn parameters that lead to physically accurate predictions.

The loss function for a PINN is a combination of a data-fitting term and a physics term. The data-fitting term is the mean squared error between the predicted and actual values of the system. The physics term is the mean squared error between the predicted and actual values of the governing physics equations.

Loss Function Description

In Physics-Informed Neural Networks (PINNs), the loss function is a crucial component that guides the network's training process. For a simple pendulum problem, the loss function typically consists of two main components:

1. **Data Fitting Term:** This term measures the discrepancy between the predicted values by the neural network and the actual observed data. It is typically calculated using a measure such as mean squared error (MSE).

$$\text{Data Loss function: } \frac{1}{N}\sum_{i=1}^{N}\left(\theta_{NN}(t_i)-\theta_{true}(t_i)\right)^2; \quad i \in [\text{training dat}$$

2. **Physics-Informed Term:** This term encodes the underlying physics of the pendulum system. It incorporates the equations of motion for the pendulum and ensures that the neural network adheres to the fundamental laws governing the pendulum's behavior. The physics-informed term is calculated by taking the second derivative of the predicted position of the pendulum with respect to time and comparing it to the second derivative of the actual position of the pendulum with respect to time. The physics-informed term is calculated at a set of points in the prediction range.

$$\text{Physics Loss function: } \frac{1}{M}\sum_{j}^{M}\left(\left[\frac{d^2}{dt^2}+\frac{g}{L}\sin\right]\theta_{NN}(t_j)\right)^2; \quad j \in [\text{pr}$$

The Physics loss term ensures that the prediction is consistent with the underlying physical process.
The PINN is trained by minimizing the loss function using gradient descent. The gradient of the loss function with respect to the parameters of the neural network is computed using automatic differentiation.

$$\text{Minimize Loss function} = \text{Initial Loss function} - \text{Learning Rate} \times \nabla \text{Lo}$$

where, Loss function = Data Loss function + Physics Loss function

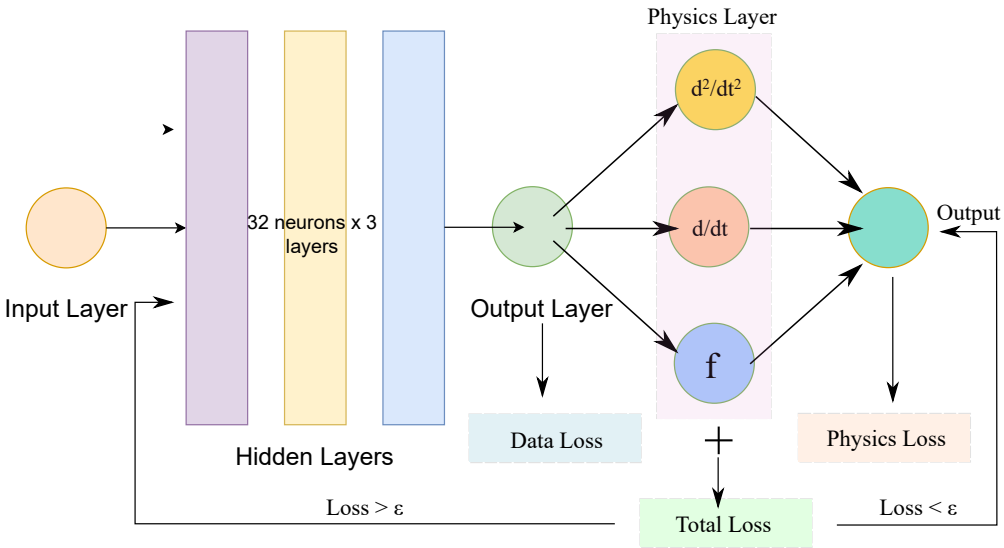
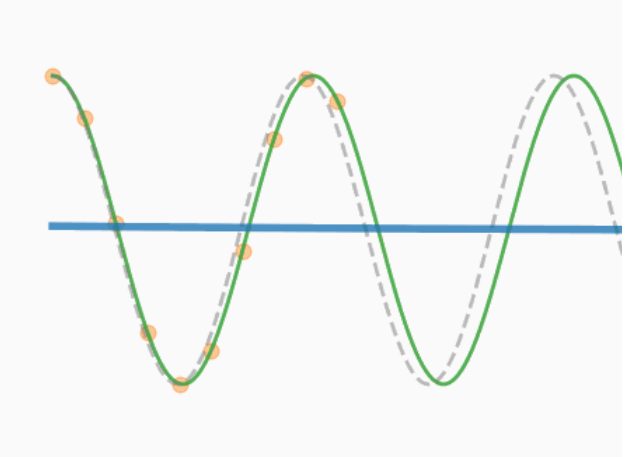


Fig: Visualization of the PINN architecture and the two types of loss functions.

PINN output:

The results of the PINN are shown below. The training data are orange dots sampled in equal intervals from the exact solution. This can represent the data collected from an experiment. The PINN is able to accurately predict the position of the pendulum at a longer time interval with very high accuracy compared to the regular neural network.



The final code Snippet to implement the NN and the PINN and the Visualization:

```
# ================================================
class FCN(nn.Module):
    "Defines a connected network"

    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN,
        super().__init__()
        activation = nn.Tanh
        self.fcs = nn.Sequential(*[
                        nn.Linear(N_INPUT, N_HIDDEN)
                        activation()])
        self.fch = nn.Sequential(*[
                        nn.Sequential(*[
                            nn.Linear(N_HIDDEN, N_HI
                            activation()]) for _ in
        self.fce = nn.Linear(N_HIDDEN, N_OUTPUT)

    def forward(self, x):
        x = self.fcs(x)
        x = self.fch(x)
        x = self.fce(x)
        return x

# ================================================
def save_gif_PIL(outfile, files, fps=5, loop=0):
    "Helper function for saving GIFs"
    imgs = [Image.open(file) for file in files]
    imgs[0].save(fp=outfile, format='GIF', append_im

def plot_result(x,y,x_data,y_data,yh,xp=None):
    "Pretty plot training results"
    plt.figure(figsize=(8,4))
    plt.plot(x,y, color="tab:green", linewidth=2, al
    plt.plot(x,pendulum_solution(w, x), color="tab:g
```

```
    plt.plot(x,yh, color="tab:blue", linewidth=4, al
    plt.scatter(x_data, y_data, s=60, color="tab:ora
    l = plt.legend(loc=(1.01,0.34), frameon=False, f
    plt.setp(l.get_texts(), color="k")
    plt.xlim(-0.05, end_time+0.05)
    plt.ylim(-1.1, 1.1)
    plt.text(1.065,0.7,"Training step: %i"%(i+1),fon
    plt.axis("off")

# ================================================
# train standard neural network to fit training data
torch.manual_seed(123)
model = FCN(1,1,32,3)
optimizer = torch.optim.Adam(model.parameters(),lr=1

files = []
for i in range(1000):
    optimizer.zero_grad()
    yh = model(x_data)
    loss = torch.mean((yh-y_data)**2)# use mean squa
    loss.backward()
    optimizer.step()

    # plot the result as training progresses
    if (i+1) % 10 == 0:

        yh = model(x).detach()

        plot_result(x, y, x_data, y_data, yh)

        file = "plots/nn_%.8i.png"%(i+1)
        plt.savefig(file, bbox_inches='tight', pad_i
        files.append(file)

        if (i+1) % 500 == 0: plt.show()
        else: plt.close("all")

save_gif_PIL("nn.gif", files, fps=20, loop=0)


# ================================================
x_physics = torch.linspace(0,end_time,30).view(-1,1)

model = FCN(1,1,32,3)
optimizer = torch.optim.Adam(model.parameters(),lr=1
files = []
for i in range(20000):
    optimizer.zero_grad()

    # compute the "data loss"
    yh = model(x_data)
    loss1 = torch.mean((yh-y_data)**2)# use mean squ

    # compute the "physics loss"
    yhp = model(x_physics)
    dx  = torch.autograd.grad(yhp, x_physics, torch.
    dx2 = torch.autograd.grad(dx,  x_physics, torch.
    physics = dx2 + k*torch.sin(yhp)
    loss2 = (1e-4)*torch.mean(physics**2)

    # backpropagate joint loss
    loss = loss1 + loss2# add two loss terms togethe
    loss.backward()
    optimizer.step()

    # plot the result as training progresses
    if (i+1) % 150 == 0:

        yh = model(x).detach()
        xp = x_physics.detach()
```

```
            plot_result(x,y,x_data,y_data,yh,xp)

            file = "plots/pinn_%.8i.png"%(i+1)
            plt.savefig(file, bbox_inches='tight', pad_i
            files.append(file)

            if (i+1) % 6000 == 0: plt.show()
            else: plt.close("all")

    save_gif_PIL("pinn.gif", files, fps=20, loop=0)
```

References for further exploration:

- Ben Moseley's blog