Solving a System of First Order ODEs with a PINN

A good way to see how to implement a Physics-Informed Neural Network (PINN) to solve a physics-based problem that involves a system of equations, is to do so by example.

PINN

A PINN is a special type of neural network that incorporates the physics within the training process for the neural network. Traditional neural networks learn to model the data by minimizing a data-fit loss function, but a PINN includes the residuals from the underlying physics differential equations within the loss function. This allows the PINN model to obey the physics when fitting the data. The PINN really shines under limited data scenarios because the physics of the equations (ODEs, PDEs) shape the solution.

Example: A 2-D System of ODEs

Suppose we have these two ODEs and initial conditions (ICs) that describe two coupled harmonic oscillators or electrical circuits for a time domain $t \in [0, 5]$:

$$\frac{dx}{dt} = -2x - y$$

$$\frac{dy}{dt} = x - 2y$$

$$x(0) = 1$$

$$y(0) = 0$$

with analytical solutions,

$$x(t) =^{1}_{2} \left(e^{-t} + e^{-3t}\right)$$

$$x(t) =^{1}_{2} \left(-e^{-t} + e^{-3t}\right)$$

PINN Key Concept

The idea is to represent the unknown functions $x(t)$ and $y(t)$ as an output of a neural network. The input of the network is the time t and the network will learn to approximate the functions $x(t)$ and $y(t)$ by minimizing a loss function that includes both of these:

1. The residuals of the ODEs

2. The intial conditions. By training the neural network to minimize this combined loss, we ensure that the network learns a solution that satisfies both the governing equatinos and the given initial conditions.

Neural Network Architecture

The NN will have the following structure:

- Input layer with 1 neuron for the input t
- Hidden layers with non-linear activation functions (e.g. $Tanh$ or $ReLU$) to learn the underlying structure of the problem
- Output layer to provide the 2 variables we want, $x(t)$ and $y(t)$

The NN will be trained to minimize the residuals of the ODEs and the error in the ICs.

Loss Function

The most critical part of a NN and especially for the PINN is the way we set up the loss function to satisfy the physics (residuals of the ODEs) and the ICs.

1. ODE Residual Loss
2. IC Loss The **total loss** is the sum of the two contributions:

$$TotalLoss = ODEResidualLoss + ICLoss$$

Variations of the solutions can also include data loss terms and options on how one "adds" the losses, e.g. weighted loss.

ODE Residual Loss

Rearranging the ODEs so that they sum to zero will put the ODEs into a *residual loss* format as shown here:

$$f_x(t) = \frac{dx}{dt} + 2x + y$$

$$f_y(t) = \frac{dy}{dt} - x + 2y$$

These are embedded into the PINN as residuals to minimize. The NN output should minimize the residuals to satisfy the ODEs at each point in the domain t. To measure the residual loss between ODEs and data points, we will use the *mean squared sum error*:

$$MSE(f_x(t)) = \frac{1}{N}\sum_{i=1}^{N}(f_x(t_i))^2$$

$$MSE(f_y(t)) = \frac{1}{N}\sum_{i=1}^{N}(f_y(t_i))^2$$

- The MSE penalizes any deviation of the network's output from satisfying the ODEs.
- The NN is trained to minimize the residuals.

IC Loss

To satisfy the ICs, we rearrange them into loss terms:

$$IC_x = (x(0) - 1)^2 \text{and} IC_y = (y(0) - 0)^2 = (y(0))^2$$

- The IC losses ensure that the NN solution satisfies the ICs.
- The NN is penalized heavily for not satisfying the ICs.

Total Loss Function

The *total loss function* is simply the sum of all the losses. Alternatively, the architect can unequally weight these losses.

$$TLF = MSE(f_x(t)) + MSE(f_y(t)) + IC_x + IC_y$$

- MSE_x satisfies $f_x(t)$
- MSE_y satisfies $f_y(t)$
- IC_x satisfies $x(0) = 1$
- IC_y satisfies $y(0) = 0$

Role of Collection Points

Collection points are specific points (t_i) where we enforce ODEs and ICs (and/or BCs).

- These points are siilar to data points in traditional machine learning because we generate the data at these points.
- Ensures the system of ODEs is satisfied throughout the domain.
- Often chosen to be evenly spaced throughout the domain to provide a uniform representation of the problem space.

Universal Approximation Theorem

The UAT simply states that a simple NN with enough neurons in the hidden layers and suitable activation functions can approximate any continuous function on a closed and bounded interval to any desired level of accuracy.

Training the NN

The NN is trained by minimizing the total loss function (TLF) during each epoch of training.

1. NN predicts x and y at various t
2. $\dot{x}$ and $\dot{y}$ are computed with automatic differentiation
3. The ODE residuals are calculated
4. The TLF is evaluated; gradients are computed to update the NN weights that minimize the TLF (e.g. gradient descent method)
5. Over time (epochs), the NN learns the satisfy the ODEs and the ICs.

Implementation

1. Define NN architecture
2. Define loss function
3. Train the NN on the data
4. Evaluate the NN outputs
5. Visualize the results

Python Code

Imports (system setup)

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
```

PINN Class

In [2]:
```
class PINN(nn.Module):
    def __init__(self):
        super(PINN, self).__init__()
        self.dense1 = nn.Sequential(
            nn.Linear(1, 64),
            nn.Tanh()
        )
        self.dense2 = nn.Sequential(
            nn.Linear(64, 64),
            nn.Tanh()
        )
        self.dense3 = nn.Linear(64, 2)  # Outputs x(t), y(t)

    def forward(self, t):
        t = self.dense1(t)
```

```
        t = self.dense2(t)
        return self.dense3(t)
```

Loss

Key steps:

1. Compute the gradient for NN's estimate of $\frac{dx}{dt}$ and $\frac{dy}{dt}$
2. Compute the ODE residuals

In [3]:
```
def loss_fn(model, t):
    """ Total Loss Function """
    # Enable gradient tracking
    t.requires_grad_(True)

    # Forward pass
    output = model(t)
    x = output[:, 0:1]
    y = output[:, 1:2]

    # Compute gradients dx/dt and dy/dt
    dxdt = torch.autograd.grad(outputs=x, inputs=t,
                               grad_outputs=torch.ones_like(x),
                               create_graph=True)[0]
    dydt = torch.autograd.grad(outputs=y, inputs=t,
                               grad_outputs=torch.ones_like(y),
                               create_graph=True)[0]

    # Residual ODEs
    res_x = dxdt + 2 * x + y
    res_y = dydt + x + 2 * y

    # Initial Conditions (IC)
    ICx = (x[0] - 1) ** 2
    ICy = (y[0] - 0) ** 2

    # Total loss
    loss = (torch.mean(res_x ** 2) +
            torch.mean(res_y ** 2) +
            ICx + ICy)

    return loss
```

Training function

The key steps are:

1. Epoch loop - update the model weights in each epoch
2. Calculate the loss with a goal to exit loop when loss is $< 10^{-3}$ or $< 10^{-4}$
3. Calculate the gradient of the loss with respect to the NN weights and NN biases

4. Update the optimizer to update the model parameters so that the loss is further reduced
5. Log the progress

In [4]:
```python
def train(model, t, epochs, optimizer):
    for epoch in range(epochs):
        # Zero the gradients
        optimizer.zero_grad()

        # Compute the loss
        loss = loss_fn(model, t)

        # Backpropagation
        loss.backward()

        # Update the model parameters
        optimizer.step()

        # Print loss periodically
        if epoch % 500 == 0:
            print(f"Epoch {epoch} Loss: {loss.item()}")
```

Set up the steps and Execute

1. Define our model with the PINN class
2. Define the gradient optimizer, e.g. **Adam**, to control the learning rate for the model's weights during training.
3. Loop the model and optimizer to adjust the model parameters to minimize the loss and ensure the NN satisfies both the physics and the ICs.

In [5]:
```python
# Run our model
model = PINN()

# Define the optimizer
# optimizer = tf.keras.optimizer.Adam(learning_rate=0.001) # control rate
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Control rate

# Define domain (t ∈ [0, 5]) -- "Collection Points"
# t = tf.convert_to_tensor(np.linspace(0,5,100)[:, None], dtype=tf.float32)
t = torch.tensor(np.linspace(0, 5, 100).reshape(-1, 1), dtype=torch.float32)

# Train the model
train(model, t, epochs=4000, optimizer=optimizer)

# Test the model
t_test = torch.tensor(np.linspace(0, 5, 300).reshape(-1, 1), dtype=torch.flo
with torch.no_grad():
    predictions = model(t_test)
    x_pred, y_pred = predictions[:, 0].numpy(), predictions[:, 1].numpy()

# Test the model
```

```
# t_test = tf.convert_to_tensor(np.linspace(0,5,300)[:, None], dtype=tf.floa
# x_pred, y_pred = model(t_test).numpy().T
t_test = torch.tensor(np.linspace(0, 5, 300).reshape(-1, 1), dtype=torch.flo
with torch.no_grad():
    predictions = model(t_test)
    x_pred, y_pred = predictions[:, 0].numpy(), predictions[:, 1].numpy()
```

```
Epoch 0 Loss: 2.5325562953948975
Epoch 500 Loss: 0.0005296792951412499
Epoch 1000 Loss: 9.405978926224634e-05
Epoch 1500 Loss: 4.4266285840421915e-05
Epoch 2000 Loss: 1.9517354303388856e-05
Epoch 2500 Loss: 5.513037467608228e-05
Epoch 3000 Loss: 4.875488684774609e-06
Epoch 3500 Loss: 3.5780437883659033e-06
```

Define analytical solution for comparison

In [8]:
```
# Truth solution
def solution(t):
    t = np.asarray(t)  # Ensure it's a standard ndarray
    x_t = 0.5 * (np.exp(-t) + np.exp(-3*t))
    y_t = 0.5 * (-1*np.exp(-t) + np.exp(-3*t))
    return x_t, y_t

# Compare results:
x_true, y_true = solution(t_test)
```

Visualize

In [9]:
```
plt.close('all')
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'Times New Roman'
plt.rcParams['font.size'] = 12
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(t_test, x_true, label=r'Analytical $x(t)$', color='red')
plt.plot(t_test, x_pred, label=r'PINN $x(t)$', color='blue')
plt.title(r'PINNs vs Analytical Solution $x(t)$', fontsize=14)
plt.xlabel(r'Time $t$')
plt.ylabel(r'$x(t)$')
plt.grid(True)
plt.legend(fontsize=12, loc="upper right")

plt.subplot(1,2,2)
plt.plot(t_test, y_true, label=r'Analytical $y(t)$', color='red')
plt.plot(t_test, y_pred, label=r'PINN $y(t)$', color='blue')
plt.title(r'PINNs vs Analytical Solution $y(t)$', fontsize=14)
plt.xlabel(r'Time $t$')
plt.ylabel(r'$y(t)$')
plt.grid(True)
plt.legend(fontsize=12, loc="upper right")
```

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