URM

January 30, 2024

1 PINN for Uniform Rectilinear Motion

1.1 Importing Necessary Packages

[ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import grad
import numpy as np
from matplotlib import pyplot as plt
```

1.2 Determining Equation of Motion

Below a Python function is developed to facilitate the development of our mathematical function, which afterwards will be used to define the network's loss function.

Given we are dealing with a Uniform Rectilinear Motion (URM), the differential equation is given by

$$\frac{dx}{dt} - v = 0, \tag{1}$$

with boundary condition given by

$$x(t=0) = 10$$

where x refers to a temporal function of position, t to time and v to a constant velocity. Basically, this equation describes the movement of a body with constant velocity.

For the purpose of this study, the domains considered will be

$$t \in [0, 1]$$

and

$$v = 2.$$

1.3 Network's Architecture

In the following cell, we define the archictecture's hyperparameters, note that the activation function applied here is a hyperbolic tangent (Tanh) for it being non-linear and ranging from -1 to 1. It is also interesting to pay attention to the fact that we are building the linear neural network from scratch due the lack of a PyTorch module specific for it, i.e., we are building a custom module.

[ ]:
```
class LinearNN(nn.Module):
    def __init__(
            self,
            num_inputs: int=1,
            num_layers: int=5,
            num_neurons: int=11,
            act: nn.Module = nn.Tanh()
    ) -> None:
        super().__init__()
        self.num_inputs = num_inputs
        self.num_neurons = num_neurons
        self.num_layers = num_layers

        layers = []

        # Input layer
        layers.append(nn.Linear(self.num_inputs, num_neurons))

        # Hidden layers
        for _ in range(num_layers):
            layers.extend([nn.Linear(num_neurons, num_neurons), act])

        # Output layers
        layers.append(nn.Linear(num_neurons, 1))

        # Building the network as sequential
        self.network = nn.Sequential(*layers)

    # Setting up the output
    def forward(self, x:torch.Tensor) -> torch.Tensor:
        return self.network(x.reshape(-1,1)).squeeze()
```

1.4 Loss Function

Here the loss function for our network will be defined, we begin by defining the initial conditions $x0 = x(t = 0) = 10$ and $v = 2$, afterwards a tensor of 100 elements is defined to receive the possible temporal coordinates that the model may assume. Considering the initial conditions established and the temporal tensor, it is now possible to define a function that returns the position x at a given time.

```
[ ]: # Random list of 100 temporal cordinates in 1D
     t = torch.rand(100, 1)

     # Position coordinate of a border
     x0 = 10
     # Velocity
     v = 2

     def x_analytical(t):
        return x0 + v*t
```

Finally, the model and training data are created, hence providing enough resources to define the loss function itself, which is defined by applying the model to the temporal training data (generating thereafter the $x_p red$ variable), the loss will also consider the temporal derivative of the position predicted thus it is also computed.

In conclusion, the loss function will return the mean squared error between the predicted and actual values through two different methods. The first method considers equation (1) whilst substituing the first term by the one predicted by the model. As for the second method, it considers de position provided by the model against the analytical result.

```
[ ]: # Running the structure created
     model = LinearNN()

     # Parameters for training
     t_train = torch.rand(100,1, requires_grad =True)

     x_analytic = x_analytical(t_train)

     # Loss function
     def loss_fn(model, t_train):

         x_pred = model(t_train)
         xt_train_dot = grad(x_pred, t_train, grad_outputs=torch.ones_like(x_pred),␣
      ↪create_graph=True)[0][:,0]

         return torch.mean(torch.square(xt_train_dot - v))  + torch.mean(torch.
      ↪square(x_pred - x_analytic))
```

1.5 Model Training

In this section, the model is being trained along 1000 cycles (epochs) through the Adam optimizer at a learning rate of 0.001. One may note that a gradient descent is applied and the graph is being retained for later analysis. It is also shown the evolution of the loss at each cycle.

```
[ ]: epochs = 1000

     optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```python
for epoch in range(epochs):

    optimizer.zero_grad()

    loss = loss_fn(model, t_train)
    loss.backward(retain_graph=True)
    optimizer.step()

    print(f"Epoch {epoch} with loss {float(loss)}")
```

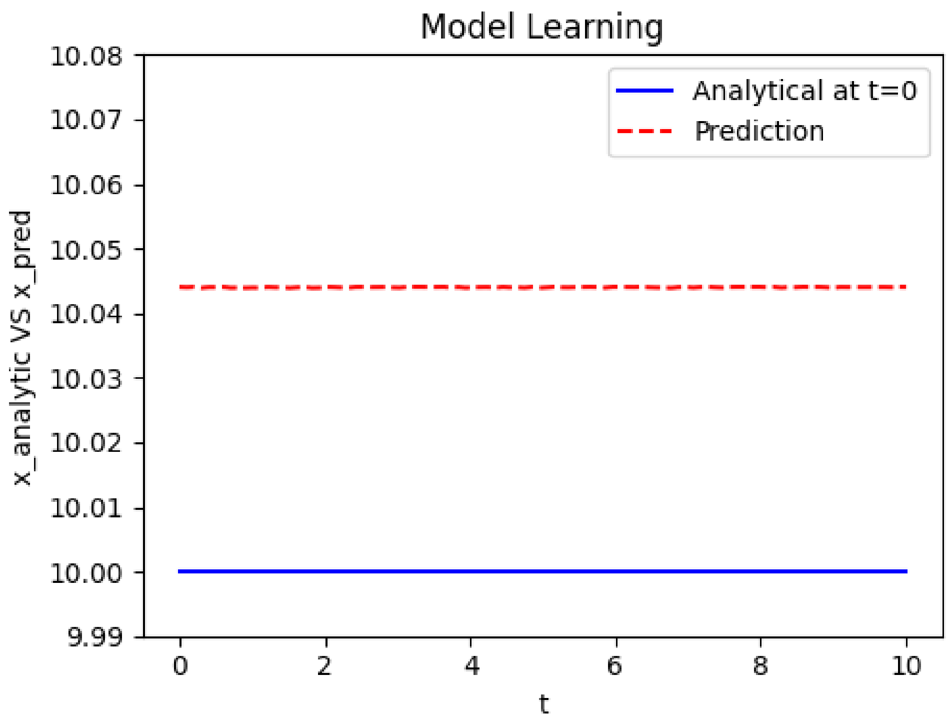
```
Epoch 0 with loss 125.00753784179688
Epoch 1 with loss 124.69764709472656
Epoch 2 with loss 124.3872299194336
Epoch 3 with loss 124.07585144042969
Epoch 4 with loss 123.76353454589844
Epoch 5 with loss 123.45028686523438
Epoch 6 with loss 123.13609313964844
Epoch 7 with loss 122.82093048095703
Epoch 8 with loss 122.50476837158203
Epoch 9 with loss 122.18753051757812
Epoch 10 with loss 121.86917114257812
Epoch 11 with loss 121.54960632324219
Epoch 12 with loss 121.22879028320312
Epoch 13 with loss 120.90658569335938
Epoch 14 with loss 120.58293151855469
Epoch 15 with loss 120.25769805908203
Epoch 16 with loss 119.93074035644531
Epoch 17 with loss 119.60188293457031
Epoch 18 with loss 119.27091217041016
Epoch 19 with loss 118.93758392333984
Epoch 20 with loss 118.60157012939453
Epoch 21 with loss 118.26255798339844
Epoch 22 with loss 117.92015075683594
Epoch 23 with loss 117.57402801513672
Epoch 24 with loss 117.22380828857422
Epoch 25 with loss 116.8691177368164
Epoch 26 with loss 116.50960540771484
Epoch 27 with loss 116.1449966430664
Epoch 28 with loss 115.77487182617188
Epoch 29 with loss 115.39897918701172
Epoch 30 with loss 115.01700592041016
Epoch 31 with loss 114.62866973876953
Epoch 32 with loss 114.23367309570312
Epoch 33 with loss 113.83175659179688
Epoch 34 with loss 113.42269134521484
Epoch 35 with loss 113.00621032714844
Epoch 36 with loss 112.58209991455078
Epoch 997 with loss 5.253895282745361
Epoch 998 with loss 5.2478742599487305
Epoch 999 with loss 5.241888046264648
```

1.6 Analysis of Perfomance

Finally, an analysis of the model performance is developed through visualization of its learning evolution in time. In the following graph, it is seen that the maintains a constant prediction for all coordinate points what is accordance to the movement described. Besides such behaviour, the graph also shows that the difference between the analytical and predicted values is smaller than 0.25, implying the model performed well.

[ ]:
```
x_plot = np.linspace(0, 10, 100)
t_plot = np.zeros_like(x_plot)
x_pred = model(t).cpu().detach().numpy()
mean_p = np.mean(x_pred)

fig, ax = plt.subplots()
ax.set_ylim(9.99, 10.08)
plt.subplots_adjust(left=0.25, bottom=0.25)
l, = plt.plot(x_plot, x_analytical(t_plot), 'b', label='Analytical at t=0')
p, = plt.plot(x_plot, x_pred, '--r', label='Prediction')
# ax.hlines(mean_p, xmin=0, xmax=10, color='g', linestyle='-', label='Mean of␣
 ↪Prediction')
plt.title('Model Learning')
plt.xlabel('t')
plt.ylabel('x_analytic VS x_pred')
plt.legend(loc='upper right');
```


Model Learning
Analytical at t=0
Prediction
x_analytic VS x_pred
t
10.08
10.07
10.06
10.05
10.04
10.03
10.02
10.01
10.00
9.99
0
2
4
6
8
10

```
[ ]:
```