Extra Project on Machine Learning - PINN for Solving DE

November 8, 2023

Name	Anis Mohammed V
Roll No:	23D0292
Subject:	CE603

1 General

In the modern era, Neural Networks (NN) are widely used for engineering applications. The application of NN was initially primarily focused on curve fitting, considering the NN's capacity as "universal function approaximator". Recently, another application of NN was ivolved in Numerical Solution of Ordinary Differential Equations (ODEs) and Partial Differential Equations(PDEs). The method was widely known as Physics-Informed Neural Networks (PINNs).

This project aims to study the PINN approach for ODE and PDE solution and solve ODE of the form $u'' - u + x = 0$ for a given boundary condition. The following section describe the general approach for PINN solving ODEs and PDEs.

2 Physics-Informed Neural Networks

Unlike traditional NN learning, the PINN approach doesn't have prior datasets that can be used for learning and validation. The approach is different from standard supervised machine learning methods. The approach is to learn the solution $f_N N$ from the physical properties of the ODEs and PDEs.

For example, consider the differential equation of the following form;

$$u'' - u + x = 0, \qquad x(0) = x(1) = 0 \tag{1}$$

The solution of the differential equation will be a function of x. And, at $x = 0$ and $x = 1$, the value of the function will be zero. Consider $f_{NN}(x)$ as the solution of the above differential equation, then the residue of the DE is;

$$\epsilon = \frac{d^2 f_{NN}}{x^2} - f_{NN} + x \tag{2}$$

The solution f_{NN} will be the output of a neural network with x as input. The advantage of neural network function approaximator is the derivatives in the ODEs can be estimated through a backward propagation algorithm (which is implemented as autograd in many NN packages like PyTorch). The best solution to the ODE will be the solution that has the least residue ϵ. In order minimize the residue, an approach similar to that of the **Collocation Method** can be adopted. Where, the average square of the residue at selected points (**Collocation Points**) is minimized. The loss function for this optimization can be termed as;

$$\mathcal{L}_{DE} = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{d^2 f_{NN}(x_j)}{x^2} - f_{NN}(x_j) + x_j\right)^2 \tag{3}$$

The above loss function is termed as internal loss function ($\mathcal{L}_{DE}$) which is associated with the differential equation alone. However, the above loss function alone is sufficient to solve the ODE. In order, to include the boundary condition, additional constraints to the loss functions are added.

$$\mathcal{L}_{BC} = (f_{NN}(x_0) - f_0)^2 \tag{4}$$

Therefore the final loss function is

$$\mathcal{L} = \mathcal{L}_{DE} + \mathcal{L}_{BC} \tag{5}$$

The parameters of the function f_{NN} are estimated through the NN learning algorithm for the above loss function.

2.1 Implementation of PINN

The PINN algorithm is implemented in an extended library of PyTorch called PyDEns. The present project uses thi s library to solve ODEs and PDEs.

3 Examples Solutions

3.1 Solution of BVP ODE using PINN

$$u'' - u + x = 0, \qquad x(0) = x(1) = 0 \tag{6}$$

[1]:
```
import sys
import numpy as np
import torch
from torch import nn
import matplotlib.pyplot as plt
sys.path.append("../../")

#from batchflow.models.torch import Block, MultiLayer

from pydens import Solver, D, V, ConvBlockModel
from pydens import NumpySampler as NS
```

```
/home/anis/.local/lib/python3.11/site-
packages/pydens/batchflow/batchflow/notifier.py:8: TqdmWarning: IProgress not
found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from tqdm.autonotebook import tqdm as tqdm_auto
```

[2]:
```
def ode(f, x):
    return D(D(f, x),x) - f + x
```

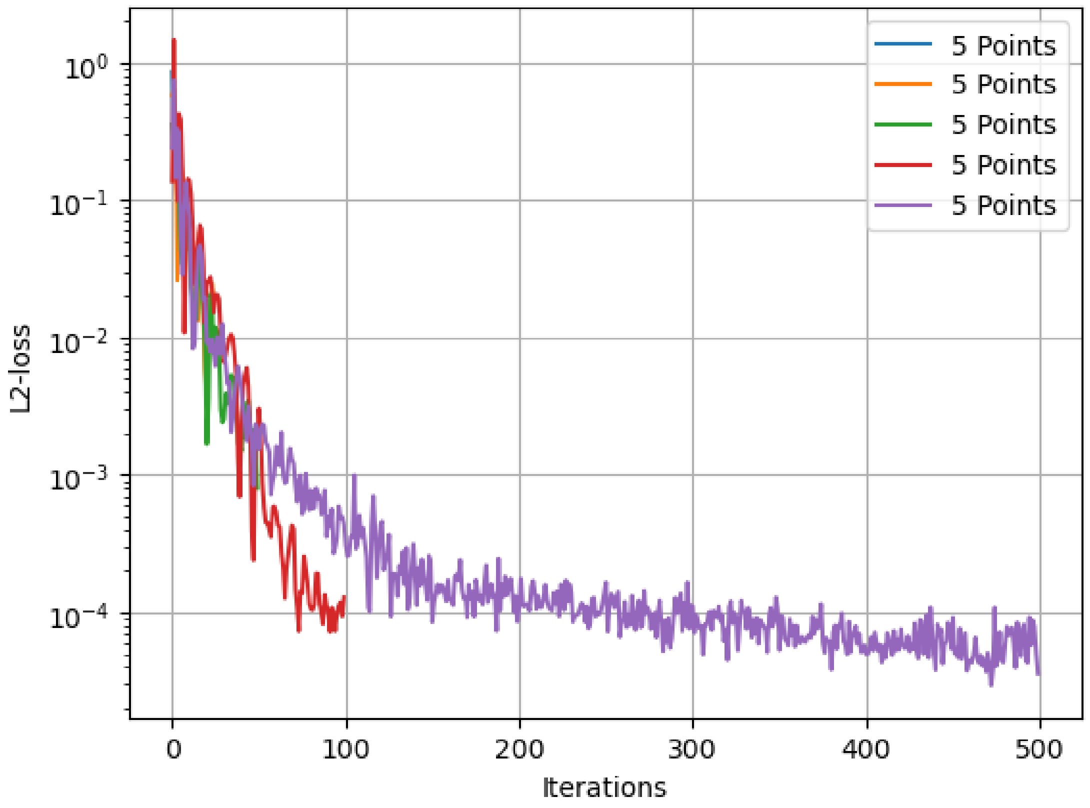
[3]:
```
solver01 = Solver(ode, ndims=1, boundary_condition=0,
                model=ConvBlockModel,
                layout='fafaf', features=[10, 10, 1], activation='Tanh')
solver02 = Solver(ode, ndims=1, boundary_condition=0,
                model=ConvBlockModel,
                layout='fafaf', features=[10, 10, 1], activation='Tanh')
solver03 = Solver(ode, ndims=1, boundary_condition=0,
                model=ConvBlockModel,
                layout='fafaf', features=[10, 10, 1], activation='Tanh')
solver04 = Solver(ode, ndims=1, boundary_condition=0,
                model=ConvBlockModel,
                layout='fafaf', features=[10, 10, 1], activation='Tanh')
solver05 = Solver(ode, ndims=1, boundary_condition=0,
                model=ConvBlockModel,
                layout='fafaf', features=[10, 10, 1], activation='Tanh')
```

[4]:
```
solver01.fit(niters=5, batch_size=40, lr=0.02)
solver02.fit(niters=25, batch_size=40, lr=0.02)
solver03.fit(niters=50, batch_size=40, lr=0.02)
solver04.fit(niters=100, batch_size=40, lr=0.02)
solver05.fit(niters=500, batch_size=40, lr=0.02)
```

```
100%|
                | 5/5 [00:00<00:00, 188.59it/s]
100%|
               | 25/25 [00:00<00:00, 554.69it/s]
100%|
               | 50/50 [00:00<00:00, 658.61it/s]
100%|
              | 100/100 [00:00<00:00, 578.33it/s]
100%|
              | 500/500 [00:00<00:00, 783.41it/s]
```

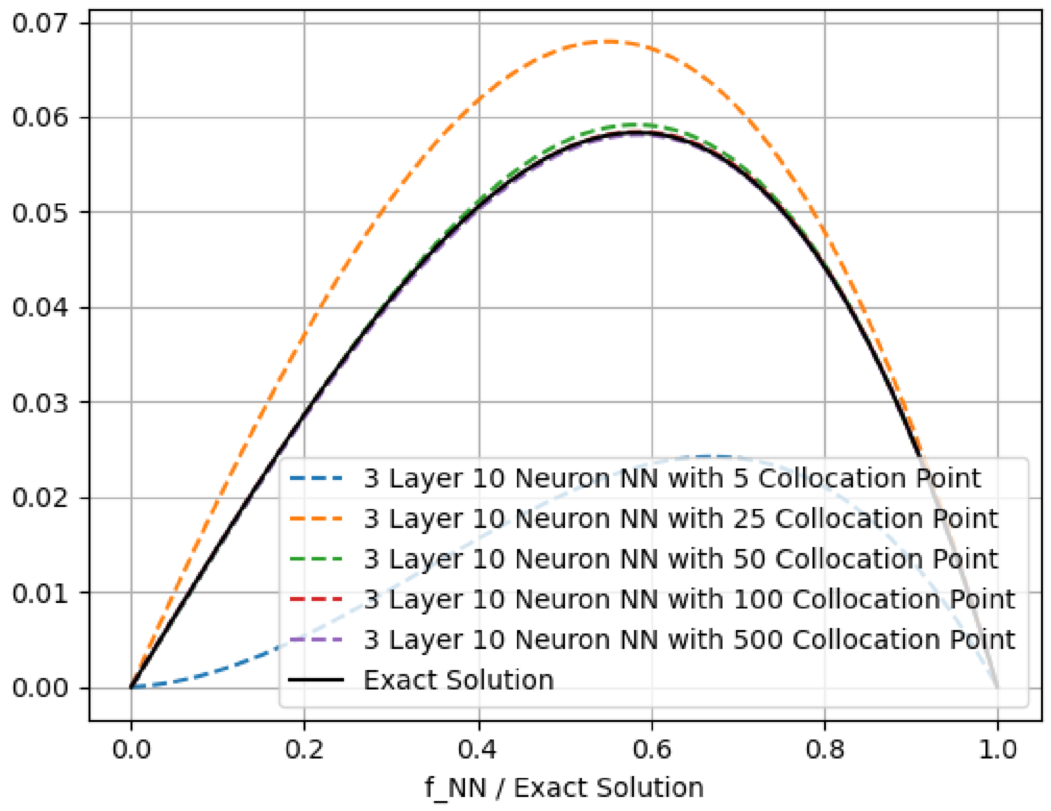
[5]:
```
plt.semilogy(solver01.losses, label='5 Points')
plt.plot(solver02.losses, label='5 Points')
plt.plot(solver03.losses, label='5 Points')
plt.plot(solver04.losses, label='5 Points')
plt.plot(solver05.losses, label='5 Points')
plt.grid()
plt.xlabel('Iterations')
plt.ylabel('L2-loss')
plt.legend()
```

```
[5]: <matplotlib.legend.Legend at 0x7f80e9adb250>
```



```
[6]: x = torch.tensor(np.linspace(0, 1, 100)).float()
f1 = solver01.predict(x)
f2 = solver02.predict(x)
f3 = solver03.predict(x)
f4 = solver04.predict(x)
f5 = solver05.predict(x)
plt.grid()
plt.plot(x, f1, label='3 Layer 10 Neuron NN with 5 Collocation Point', ls='--')
plt.plot(x, f2, label='3 Layer 10 Neuron NN with 25 Collocation Point', ls='--')
plt.plot(x, f3, label='3 Layer 10 Neuron NN with 50 Collocation Point', ls='--')
plt.plot(x, f4, label='3 Layer 10 Neuron NN with 100 Collocation Point',␣
 ↪ls='--')
plt.plot(x, f5, label='3 Layer 10 Neuron NN with 500 Collocation Point',␣
 ↪ls='--')
plt.plot(x, x-(np.sinh(x)/np.sinh(1)), label='Exact Solution', c='k')
plt.legend()
plt.xlabel('x')
plt.xlabel('f_NN / Exact Solution')
```

```
[6]: Text(0.5, 0, 'f_NN / Exact Solution')
```



```
[7]: import pandas as pd
     pd.options.display.float_format = "{:,.6f}".format
     x = torch.tensor(np.linspace(0, 1, 11)).float()
     f1 = solver01.predict(x)
     f2 = solver02.predict(x)
     f3 = solver03.predict(x)
     f4 = solver04.predict(x)
     f5 = solver05.predict(x)
     fe = x-(np.sinh(x)/np.sinh(1))
     RESULTS = pd.DataFrame.from_dict({'X':x,\
                             'NN with 5 Points':f1[:,0],\
                            'NN with 25 Points':f2[:,0],\
                            'NN with 50 Points':f3[:,0],\
                            'NN with 100 Points':f4[:,0],\
                            'NN with 500 Points':f5[:,0],\
                              'Exact':fe})
```

A comparison of results obtained from PINN and the analytical solution is shown above. It can be seen that, with a 500 number of collocation points the PINN results closely match with analytical solution. A comparison table for estimated values and exact values at 0.1 interval is shown below;

[8]:
```
RESULTS
```

[8]:
```
           X  NN with 5 Points  NN with 25 Points  NN with 50 Points  \
0   0.000000          0.000000           0.000000           0.000000
1   0.100000          0.001718           0.019615           0.014551
2   0.200000          0.005413           0.037059           0.028587
3   0.300000          0.010313           0.051381           0.041093
4   0.400000          0.015577           0.061699           0.051037
5   0.500000          0.020292           0.067209           0.057380
6   0.600000          0.023476           0.067182           0.059087
7   0.700000          0.024093           0.060969           0.055143
8   0.800000          0.021081           0.048007           0.044573
9   0.900000          0.013385           0.027814           0.026466
10  1.000000          0.000000           0.000000           0.000000

    NN with 100 Points  NN with 500 Points     Exact
0             0.000000            0.000000  0.000000
1             0.014760            0.014634  0.014766
2             0.028605            0.028407  0.028680
3             0.040766            0.040522  0.040878
4             0.050404            0.050121  0.050483
5             0.056600            0.056289  0.056591
6             0.058368            0.058050  0.058260
7             0.054665            0.054377  0.054507
8             0.044415            0.044203  0.044294
9             0.026542            0.026440  0.026518
10            0.000000            0.000000  0.000000
```

[ ]: