

PHYS 491: Generalizing Physics Informed Neural Networks for Multiple Initial Conditions of A Harmonic Oscillator

Ozan Cem Bař

Department of Physics, Bilkent University, 06800 Ankara, Turkey

December 29, 2024

Abstract

Physics-Informed neural networks, or PINNs for short, are neural networks with modified loss functions so that they can learn the underlying solution to a physical differential equation with noisy and/or low number of measurement data. They can achieve accurate results and also, they can generate solutions that are consistent with the equations describing the system. However, PINNs need to be trained for a single set of measurements of the system for a specific set of initial and boundary conditions and problem parameters. Therefore, they cannot generalize these solutions to other initial conditions, requiring the network to be trained again for a new solution. In this project, I suggested a different neural network architecture that can accept different initial conditions as input, learn different initial conditions separately and generalize the solutions for the the other initial conditions that lay in between the pre-trained initial conditions. It was found that the altered physics-informed neural network structure can generalize for several initial conditions but it is not an efficient method.

Introduction

Neural Networks

Neural networks, or artificial neural networks, are function estimators that are build up of biologically inspired neurons. The research for modeling neurons began in the 20th century. One of the first models artificial neurons was given by Warren McCulloch and Walter Pitts in 1943, where they introduced the first mathematical model of a neuron (1). McCulloch-Pitts neuron simplified the neuron to a binary activation model with a threshold and the connection weights that determined how much the information from the other neurons is scaled, as seen in figure 1. The work of McCulloch and Pitts paved the road for neural networks.

In 1958, Frank Rosenblatt extended these ideas with the invention of the perceptron, a single-layer neural network capable of performing simple classifications by adjusting its weights through supervised learning (3). Rule for each neuron activation in the perceptron model is calculated as,

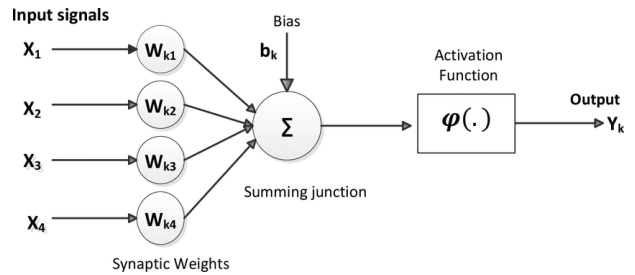


Figure 1: An illustration of the McCulloch-Pitts neuron model (2).

$$y = \varphi \left(\sum_{j=0}^m w_j x_j \right) \quad (1)$$

where w_j are the connection weights and φ is a nonlinear activation function. With perceptron, it was possible to adjust the weights so that basic recognition tasks could be made. However, expanding this single layer structure was problematic, for there was no efficient method of fitting the parameters of the network. This solution came with the

introduction of the backpropagation.

Backpropagation, that was developed by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams in 1986, was one of the most significant milestones for artificial neural networks to gain the importance they have today (4). Backpropagation provided an algorithm that provided a systematic method for iteratively optimizing the parameters of several layers of neurons, allowing deeper architectures to emerge.

The information flow in modern feed-forward artificial neural networks progresses in two phases: forward propagation and backpropagation. Forward propagation is passing the data through the neural network to obtain the predictions of the network. For a neural network with L fully connected layers, the forward propagation calculations is as follows,

$$\begin{aligned} Z^{(k)} &= W^{(k)}A^{(k-1)} + b^{(k)} \\ A^{(k)} &= \sigma_k(Z^{(k)}) \end{aligned} \quad (2)$$

where $A^{(k-1)}$ is the activation values of the previous layer’s neurons, $W^{(k)}$ are the connection weights matrix that connect the $k - 1$ -th layer to the k -th layer, $b^{(k)}$ are bias terms, and σ_k is a nonlinear activation function, such as $\tanh(x)$. Using the activation values of the previous layer, the next layer of activations $A^{(k)}$ is calculated. Here, $A^{(0)}$ is the input data and final layer activation, $A^{(L)}$, is the neural network’s output. In the training phase of the neural network, the error in the networks prediction is quantized using a cost function \mathcal{L} . The goal of the training procedure is the minimize this cost function throughout the dataset. The cost function is chosen depending on the nature of the problem. A common cost function that is used for regression tasks is Euclidean distance,

$$\mathcal{L}(y, A^{(L)}) = \sqrt{\sum_{i=1}^N (y_i - A_i^{(L)})^2} \quad (3)$$

while for categorization tasks, cross-entropy loss is more common,

$$\mathcal{L}(y, A^{(L)}) = - \sum_{i=1}^N y_i \ln A_i^{(L)} \quad (4)$$

where the y is the vector of true values. Finally, to optimize the weights in order to minimize the cost function for all training samples, the gradient with

respect to the network’s weights and biases is calculated, and a gradient descent is performed in the opposite direction of the gradient vector:

$$W(n) = W(n-1) - \eta \frac{\partial \mathcal{L}}{\partial W} \quad (5)$$

where W is an arbitrary network parameter and n is the number of the training step. Each pass over the training data set is called an ‘epoch’.

With the advancements in parallel processing units, such as GPUs, development in the neural networks has accelerated. Several other configurations of layers and neurons emerged for better performance in different fields such as convolutional neural networks (CNN) image recognition tasks, long-short term memory (LSTM) for processing time series data such as speech and text, and transformer structures that can perform parallel processing on sequential data with its attention mechanism (5).

Physics-Informed Neural Networks

Physics-informed neural networks (PINN) are an emerging method in scientific computation that makes use of neural networks equipped with a specialized cost function to solve systems whose progression is given by a set of deterministic equations.

The need for PINNs emerged as a response to the growing challenges in scientific and engineering computations, particularly for solving complex systems of partial differential equations (PDEs) that describe physical phenomena. Traditional numerical methods such as finite element, and finite difference, while effective, require significant computational resources, especially in high-dimensional problems or when dealing with irregular geometries. These methods also depend heavily on finely meshed grids, making them less adaptable to dynamic or data-sparse scenarios. Furthermore, in inverse problems, where unknown system parameters must be deduced from sparse or noisy data, conventional approaches often struggle due sensitivity to measurement errors (6).

At the same time, the rise of machine learning demonstrated the potential of neural networks as powerful approximators capable of capturing complex patterns (7). However, purely data-driven approaches require large datasets and lack generalizability beyond their training domain. This high-

lighted the need for a method that could integrate physical laws with sparse data to improve accuracy and generalizability. PINNs were developed in order to fill this gap, leveraging neural networks not only as function approximators but also as a means to encode and enforce physical constraints within the solution process.

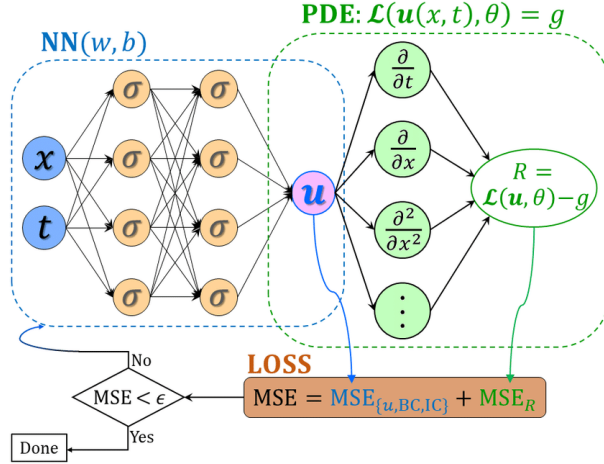


Figure 2: The general structure of PINN. The cost function has a residual term that is a measure of how much u has deviated from the target differential equation (8).

Functionally, PINN is a neural network whose inputs are the free variables of the system (i.e., t , x , y , z) and outputs are the value, or values, of the solution of the system (i.e. $\Psi(r, t)$). Similar to traditional neural network training, PINNs are trained using measurement data obtained from the system. However, unlike conventional approaches, the cost function in PINNs is specifically designed to incorporate the residuals of the governing equations, accounting for the error in the derivatives of the outputs with respect to the inputs, as well as the boundary conditions, as seen in figure 2. This formulation enables PINNs to ensure physical consistency across arbitrary points in the domain, allowing the model to satisfy both the provided data and the underlying physics of the system, even with relatively limited datasets. (9).

Despite the promising capabilities that PINNs offer, they are constrained on how much they can learn. PINNs are initially designed to be fitted over a set of measurements that are sampled over a single set of initial conditions and/or boundary conditions. This

means that learns only for one initial conditions of the system at a time, and cannot provide accurate results for any other initial condition. In this project, the input layer of a PINN was arranged so that it can accept the initial conditions of a harmonic oscillator system as inputs. Then, the network was trained for several initial conditions of this system simultaneously. Finally, PINN's predictions for the values that lay between the initial conditions, that the network was trained on, were evaluated and PINN's accuracy and ability to generalize to these points was evaluated.

Methodology

Structure of PINNs

All of the implementations of this project were performed in Python version 3.11.9, and Pytorch library was used for performing the operations associated with the neural networks. For the structure of PINN, a neural network with three hidden layers with 128 neurons, among with an input layer with three neurons, and an output layer with a single neuron is used to train the network for the harmonic oscillator solutions. The three input neurons accept the values time t , initial position x_0 , and the initial velocity v_0 . The single output neuron return the value $u(t, x_0, v_0)$ that is the network's prediction of the value of $x(t)$, given that $x(0) = x_0$ and $\dot{x}(t) = v_0$. To obtain the baseline PINN results without the proposed changes, a neural network with same dimentionis is implemented. However, instead of having a three neuron input layer, an inpult layer with one neuron is implemented, that neuron's input being time t . So that PINN's output $u(t)$ is an estimation of $x(t)$.

Training of PINNS

Initially, the baseline PINN is trained for a given initial condition, $x_0 = 0.$, $v_0 = 20.$ of a harmonic oscillator,

$$m\ddot{x} + \mu\dot{x} + kx = 0 \quad (6)$$

where the constant that determine the behaviour of the system are taken as, $m = 1kg$ $\mu = 4\frac{kg}{s}$, and

$k = 400 \frac{\text{kg}}{\text{s}^2}$. So that, the oscillator is under-damped and has the general solution,

$$x(t) = Ae^{-\delta t} \cdot \cos(\omega t + \phi) \quad (7)$$

where $\delta = \frac{\mu}{2m}$, $\omega_0 = \sqrt{k/m}$, $\omega = \sqrt{\delta^2 - \omega_0^2}$. The values A and ϕ are determined from the initial conditions as,

$$A = -\frac{\sqrt{2\delta x_0 v_0 + v_0^2 + (\delta x_0)^2 + (x_0 \omega)^2}}{\omega} \quad (8)$$

$$\phi = 2 \tan^{-1} \left(\frac{-v_0 - \delta x_0}{x_0 \omega - A \omega} \right) \quad (9)$$

where x_0 and v_0 constitute the initial position and velocity. With the information of the differential equation, the cost function is defined as follows,

$$\begin{aligned} \mathcal{L}(t) = & \beta_1 \cdot |u(0) - x_0| + \beta_2 \cdot \left| \frac{du(0)}{dt} - v_0 \right| \\ & + \beta_3 \cdot \left(\frac{d^2 u(t)}{dt^2} - \mu \frac{du(t)}{dt} - k \cdot u(t) \right) \end{aligned} \quad (10)$$

where β_1 is mean-squared error term for the initial position, β_2 term is mean-squared error term for the initial velocity, and β_3 term, which is called the physics loss, accounts for the deviations from a true solution of the differential equation. Here, the β terms are selected as,

$$\begin{aligned} \beta_1 &= 10 \\ \beta_2 &= 1 \\ \beta_3 &= 0.01 \end{aligned} \quad (11)$$

t specified in the β_3 term is the time points on which the physics loss is calculated on. After the training, the results of the trained PINN are displayed and explained.

Then, the proposed PINN structure is trained for a set of values that encapsulates the domain of interest. The initial condition points that are used for training and testing the network are seen in figure 3. Then, the same predictions are made over the same set of initial conditions.

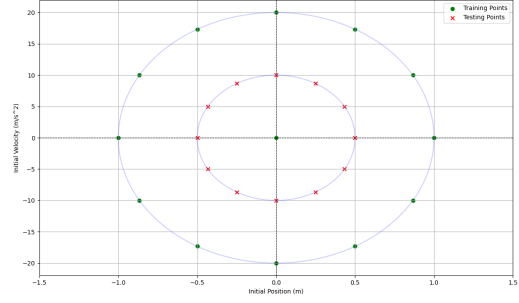


Figure 3: The training points and testing point that are used for the experiments.

Results

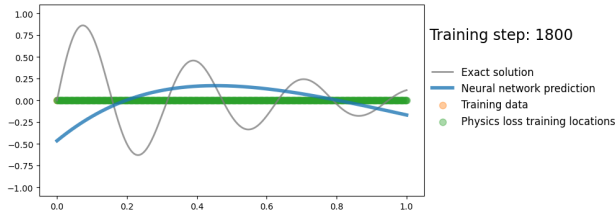
After fitting the unaltered PINN to the initial conditions $x_0 = 0.$, $v_0 = 20$. for the harmonic oscillator with parameters $k = 400$, $\mu = 4$, it was observed that PINN converged to the target solution curve quite accurately. Some of the selected training steps can be seen in figure 4.

At the earlier stages of the training, PINN cannot satisfy the initial conditions. But as it learns the initial conditions, the network learns the rest of the solution more easily by incrementally adapting the for the provided differential equation. It can be seen that network can learn the curvature of the solution and how it should progress as time moves forward. After two minutes of training, PINN was trained for 12600 training steps, and achieved an average of 0.03 difference per plotting point.

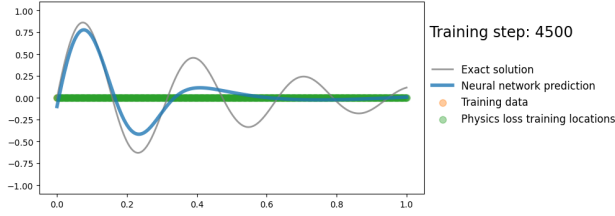
After modifying the PINN to be able to adapt for several initial conditions, network learning slowed down drastically, as all of the initial conditions that the network needed to learn had to be shown the network repeatedly. After the the decrease in network's prediction costs stagnated, the training is stopped. In figure 5, the PINN's predictions for some of the initial training data can be seen.

It can be seen that when the number of points that PINN needed to learn increased, its ability to fit each individual solution decreased. The fitting results from the testing points is shared in figure 6.

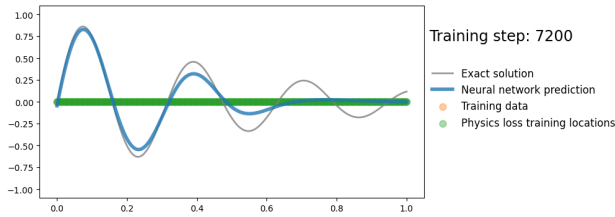
It is seen that PINN can, on some points predict accurately. For earlier time values ($t < 0.5$), it is observed that model has learned the shape of the solution up to a certain extend. However, the predictions



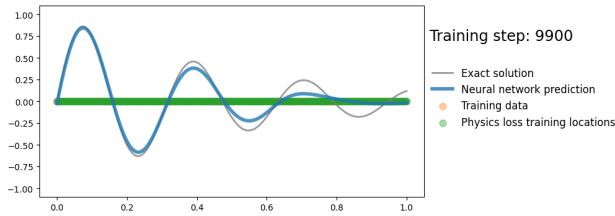
(a) Training step: 1800



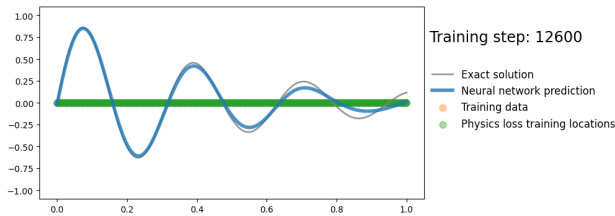
(b) Training step: 4500



(c) Training step: 7200

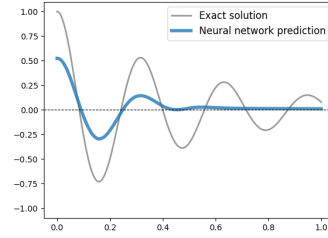


(d) Training step: 9900

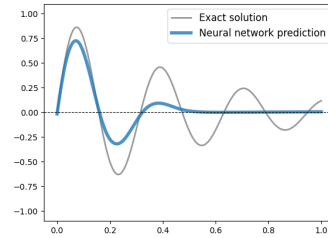


(e) Training step: 12600

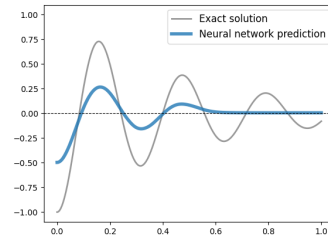
Figure 4: Progression of the unaltered PINN's predictions at different training steps.



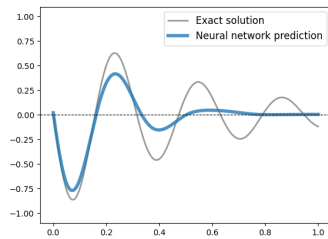
(a) $x_0 = 1., v_0 = 0.$



(b) $x_0 = 0., v_0 = 20.$



(c) $x_0 = -1., v_0 = 0.$



(d) $x_0 = 0., v_0 = -20.$

Figure 5: The predictions of the altered PINN structure for the initial points that are used for the training after the training is completed.

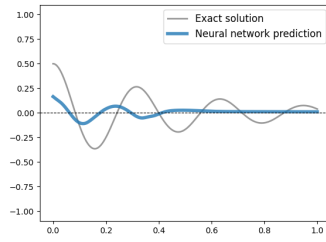
are mostly flawed and results cannot be used as a representative of the actual solution.

Conclusion

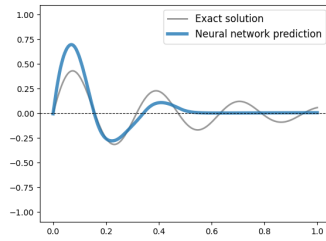
In this project, physics-informed neural networks, which is a neural network implementation for scientific computation, is explained. Their working principle is studied and suggested a possible augmentation to their structure, with the intend to enable the network to learn different sets of initial conditions of the system, is implemented and experimented upon. As observed in the previous section, the results indicated that generalizing PINN for multiple sets of measurement data was not an efficient process. Model could not fit any of the solutions fully. Model could, although imprecisely, generalize the solutions for the testing points that lay in between the training points. However, the model's predictions grew more inaccurate as the prediction time increased, both for the training and testing sets. Here are some of the problems associated with the the training of the multiple initial condition generalized PINN:

- Training was multiple times slower, due to model having problem fitting all of the training data simultaneously.
- In order to learn the information associated with all of the training points, the model's layer sizes need to be increased. Increased layer sizes, additionally slowed down the training further.
- Convergence is not guaranteed for all of the training points.
- In order of obtain high accuracy results in the testing points, the testing points need to be selected in between the training points and close to them.
- Increasing the size of the domain in which the model can make meaningful predictions, the training points need to be increased, making the training more difficult and unstable.

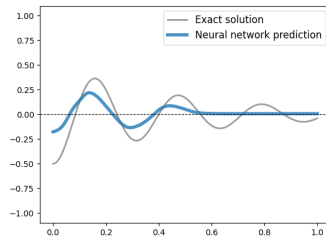
In conclusion, it can be said that augmenting the input layer of PINNs in order to train for several ini-



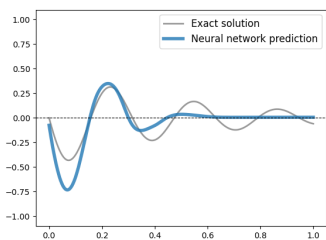
(a) $x_0 = 0.5, v_0 = 0.$



(b) $x_0 = 0., v_0 = 10.$



(c) $x_0 = -0.5, v_0 = 0.$



(d) $x_0 = 0., v_0 = -10.$

Figure 6: The predictions of the altered PINN structure for the testing points.

tial conditions was not an efficient method for generating solutions to a physical system.

References

- [1] McCulloch WS, Pitts W. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*. 1943 12;5(4):115-33. Available from: <https://doi.org/10.1007/bf02478259>.
- [2] Riadi A, Botutihe M. Visitor satisfaction prediction of the 'Pantai Pohon Cinta' beach tourism using the backpropagation algorithm with particle swarm optimization feature selection. *ILKOM Jurnal Ilmiah*. 2021 08;13:117-24.
- [3] Rosenblatt F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*. 1958 1;65(6):386-408. Available from: <https://doi.org/10.1037/h0042519>.
- [4] Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. *Nature*. 1986 10;323(6088):533-6. Available from: <https://doi.org/10.1038/323533a0>.
- [5] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is all you need. *arXiv (Cornell University)*. 2017 1:359-366. Available from: <https://arxiv.org/abs/1706.03762>.
- [6] Schönheinz H. G. Strang / G. J. Fix, *An Analysis of the Finite Element Method*. (Series in Automatic Computation. XIV + 306 S. m. Fig. Englewood Cliffs, N. J. 1973. Prentice-Hall, Inc. ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik. 1975 1;55(11):696-7. Available from: <https://doi.org/10.1002/zamm.19750551121>.
- [7] Hornik K, Stinchcombe M, White H. Multilayer feedforward networks are universal approximators. *Neural Networks*. 1989 1;2(5):359-66. Available from: [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8).

[8] Meng X, Li Z, Zhang D, Karniadakis G. PPINN: Parareal Physics-Informed Neural Network for time-dependent PDEs; 2019.

[9] Raissi M, Perdikaris P, Karniadakis GE. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*; 2017. Available from: <https://arxiv.org/abs/1711.10561>.

Appendix: Code

Listing 1: Python Code from File

```
1 from PIL import Image
2
3 import numpy as np
4 import torch
5 import torch.nn as nn
6 import matplotlib.pyplot as plt
7
8 device = ('cuda' if torch.cuda.
9           is_available()
10          else 'cpu')
11
12 device = 'cpu'
13 torch.set_default_device(device)
14 print(f'Default device is : {device}')
15
16 def plot_result(x,y,x_data,y_data,yh
17               ,xp=None):
18     "Pretty plot training results"
19     plt.figure(figsize=(8,4))
20     plt.plot(x,y, color="grey",
21             linewidth=2, alpha=0.8, label
22             ="Exact solution")
23     plt.plot(x,yh, color="tab:blue",
24             linewidth=4, alpha=0.8,
25             label="Neural network
26             prediction")
27     plt.scatter(x_data, y_data, s
28               =60, color="tab:orange",
29               alpha=0.4, label='Training
30               data')
31     if xp is not None:
32         plt.scatter(xp, -0*torch.
33                   ones_like(xp), s=60,
34                   color="tab:green", alpha
35                   =0.4,
```

```

24         label='Physics'
25         loss_training
26         locations')
27     l = plt.legend(loc=(1.01,0.34),
28                   frameon=False, fontsize="
29                   large")
30     plt.setp(l.get_texts(), color="k"
31             ")
32     plt.xlim(-0.05, 1.05)
33     plt.ylim(-1.1, 1.1)
34     plt.text(1.065,0.7,"Training
35             step: %i"%(i+1),fontsize="xx-
36             large",color="k")
37
38 def save_gif_PIL(outfile, files, fps
39 =5, loop=0):
40     "Helper function for saving GIFs
41     "
42     imgs = [Image.open(file) for
43             file in files]
44     imgs[0].save(fp=outfile, format=
45                 'GIF', append_images=imgs
46                 [1:], save_all=True, duration
47                 =int(1000/fps), loop=loop)
48
49 def oscillator_init_cond(d, w0, x,
50 x0, v0):
51     assert d < w0
52     w = np.sqrt(w0**2 - d**2)
53     A = -np.sqrt(2*d*x0*v0 + v0**2 +
54               (d*x0)**2 + (x0*w)**2) / w
55     phi = 2 * np.arctan((-v0 -d*x0)
56                       / (x0*w - np.sqrt(2*d*x0*v0 +
57               v0**2 + (d*x0)**2 + (x0*w)
58               **2)))
59     y = A * np.exp(-d*x) * np.cos(w*
60     x + phi)
61     return y
62
63 class FCN(nn.Module):
64     "Defines a connected network"
65
66     def __init__(self, N_INPUT,
67                 N_OUTPUT, N_HIDDEN, N_LAYERS)
68     :
69         super().__init__()
70         activation = nn.Tanh
71         self.fcs = nn.Sequential(*[
72             nn.Linear(
73                 N_INPUT,
74                 N_HIDDEN)
75             ,
76             activation()
77             ])
78         self.fch = nn.Sequential(*[
79             nn.
80             Sequential
81             (*[
82             nn.
83             Linear
84             (
85                 N_HIDDEN
86                 ,
87                 N_HIDDEN
88                 ),
89             activation
90             (])
91         for _
92         in
93         range
94         (
95             N_LAYERS
96             -1)])
97         self.fce = nn.Linear(
98             N_HIDDEN, N_OUTPUT)
99
100     def forward(self, x):
101         x = self.fcs(x)
102         x = self.fch(x)
103         x = self.fce(x)
104         return x
105
106     ### For regular PINN
107     d, w0 = 2, 20
108     mu, k = 2*d, w0**2
109     data_points = 256
110     x0, v0 = 0., 20.
111     beta = [10, 1e-2, 1]
112     x = torch.linspace(0,1,500).view
113     (-1,1)
114     y = oscillator_init_cond(d, w0, x,
115     x0, v0)
116     x_data = torch.tensor([0, x0, v0]).
117     view(1,3)
118     y_data = y[0]
119     # print(x_data, x_data.shape)
120     x_physics = torch.linspace(0,1,
121     data_points).requires_grad_(True)
122     # sample locations over the
123     problem domain

```

```

85
86 input_physics = torch.stack([
    x_physics, torch.ones_like(
    x_physics) * x0, torch.ones_like(
    x_physics) * v0], dim=1).view
    (-1,3)
87 x_in = torch.stack([x, torch.
    ones_like(x) * x0, torch.
    ones_like(x) * v0], dim=1).view
    (-1,3)
88
89
90 torch.manual_seed(123)
91 model_regular = FCN(3,1,32,3)
92 optimizer = torch.optim.Adam(
    model_regular.parameters(), lr=1e
    -4)
93 files = []
94
95 # for i in range(12001):
96 stop_loss_lim = 0.5
97 loss = 1000
98 i = -1
99 while loss > stop_loss_lim:
100     i += 1
101
102     optimizer.zero_grad()
103
104     # compute the "data loss"
105     yh = model_regular(x_data)
106     loss1 = torch.mean((yh - x0)**2)
107     # use mean squared error
108
109     # compute the "physics loss"
110     yhp = model_regular(
        input_physics)
111     dx = torch.autograd.grad(yhp,
        input_physics, torch.
        ones_like(yhp), create_graph=
        True)[0][:, 0:1] # computes
        dy/dx
112     dx2 = torch.autograd.grad(dx,
        input_physics, torch.
        ones_like(dx), create_graph=
        True)[0][:, 0:1] # computes d
        ^2y/dx^2
113     physics = dx2 + mu*dx + k*yhp #
        computes the residual of the
        1D harmonic oscillator
        differential equation
114     loss2 = torch.mean(physics**2)
115
116     # print(dx.shape, dx2.shape, yhp
        .shape)
117
118     # print(dx[0], v0)
119
120     loss3 = torch.mean((dx[0] - v0)
        **2)
121
122     # backpropagate joint loss
123     loss = beta[0] * loss1 + beta[1]
        * loss2 + beta[2] * loss3 #
        add two loss terms together
124     loss.backward()
125     optimizer.step()
126
127     # plot the result as training
128     progresses
129     if (i+1) % 150 == 0:
130
131         yh = model_regular(x_in).
            detach()
132         xp = x_in[:,0].view(-1,1).
            detach()
133         print(f"{i+1}th iteration ->
            x0 loss: {loss1:.4f} |
            v0 loss: {loss3:.4f} |
            physics loss: {loss2:.4f
            }| Scaled Total loss: {
            loss:.4f}")
134
135         plot_result(x,y,x_data[:,0],
            y_data,yh,xp)
136
137         file = "plots_regular/pinn_
            %.8i.png"%(i+1)
138         plt.savefig(file,
            bbox_inches='tight',
            pad_inches=0.1, dpi=100,
            facecolor="white")
139         files.append(file)
140
141         print(torch.sum(torch.abs(y-
            yh)))
142
143         if (i+1) % 6000 == 0: plt.
            show()
144         else: plt.close("all")
145
146         save_gif_PIL("pinn.gif", files, fps
            =20, loop=0)
147
148     ### For plotting the training and
149     target points
150     import numpy as np
151     import matplotlib.pyplot as plt

```

```

150 # Set axis limits
151 # Parameters for the ellipse
152 v_max = 20 # Semi-major axis
153 x_max = 1 # Semi-minor axis
154
155 # Generate points on the ellipse
156 theta = np.linspace(0, 2 * np.pi,
157 500) # Angle parameter
158 x = x_max * np.cos(theta)
159 y = v_max * np.sin(theta)
160
161 # Select 8 equally spaced points on
162 the ellipse
163 theta_training = np.linspace(0, 2 *
164 np.pi + np.pi/6, 14)[: -1] #
165 Exclude the last point to avoid
166 overlap
167 x_training = x_max * np.cos(
168 theta_training)
169 y_training = v_max * np.sin(
170 theta_training)
171 x_training[-1] = 1e-5
172 y_training[-1] = 1e-5
173
174 theta_testing = np.linspace(0, 2*np
175 pi, 13)[: -1]
176 x_test = x_max * np.cos(
177 theta_testing) / 2
178 y_test = v_max * np.sin(
179 theta_testing) / 2
180
181 # Plot the ellipse
182 plt.figure(figsize=(20, 6))
183 plt.plot(x, y, color="blue", alpha
184 =0.2) # Make the ellipse line
185 dimmer
186 plt.plot(x/2, y/2, color="blue",
187 alpha=0.2) # Make the ellipse
188 line dimmer
189 # plt.scatter([0], [0], color="green
190 ") # Marking the center
191 plt.scatter(x_training, y_training,
192 color="green", label="Training
193 Points", marker='o') # 8 points
194 plt.scatter(x_test, y_test, color="
195 red", label="Testing Points",
196 marker='x')
197
198 # Mark and scale the axes
199 plt.axhline(0, color='black',
200 linewidth=0.8, linestyle='--')
201 plt.axvline(0, color='black',
202 linewidth=0.8, linestyle='--')
203
204 # Set axis limits
205 plt.ylim(-22, 22)
206 plt.xlim(-1.5, 1.5) # Make the y-
207 axis longer
208
209 # Label the axes
210 plt.xlabel("Initial Position (m)")
211 plt.ylabel("Initial Velocity (m/s^2)
212 ")
213 plt.legend()
214 plt.grid(True)
215
216 ### Generating the training and
217 testing points
218 data_points = 256
219 d, w0 = 2, 20
220 mu, k = 2*d, w0**2
221
222 init_list = np.stack((x_training,
223 y_training), axis=1, dtype=np.
224 float32)
225
226 x = torch.linspace(0,1,500).view
227 (-1,1)
228 x_physics = torch.linspace(0,1,
229 data_points).requires_grad_(True)
230 # sample locations over the
231 problem domain
232
233 y_list = []
234 x_data_list = []
235 y_data_list = []
236 input_physics_list = []
237 x_in_list = []
238
239 for x0, v0 in init_list:
240 y = oscillator_init_cond(d, w0,
241 x, x0, v0)
242
243 x_data = torch.tensor([0, x0, v0
244 ]).view(1,3)
245 y_data = y[0]
246
247 input_physics = torch.stack([
248 x_physics, torch.ones_like(
249 x_physics) * x0, torch.
250 ones_like(x_physics) * v0],
251 dim=1).view(-1,3)
252 x_in = torch.stack([x, torch.
253 ones_like(x) * x0, torch.
254 ones_like(x) * v0], dim=1).
255 view(-1,3)

```

```

219     y_list.append(y)
220     x_data_list.append(x_data)
221     y_data_list.append(y_data)
222     input_physics_list.append(
223         input_physics)
224     x_in_list.append(x_in)
225
226
227
228     ##### Training of the augmented PINN
229     beta = [10, 1e-2, 1]
230
231
232     num_init_states = len(init_list)
233     print(f'There are {num_init_states}
234           initial States:')
235
236     torch.manual_seed(123)
237     model = FCN(3,1,128,3)
238     optimizer = torch.optim.Adam(model.
239         parameters(),lr=1e-3)
240
241     # for i in range(12001):
242     stop_loss_lim = 1.
243     max_loss = 1000.
244     loss_list = torch.ones((
245         num_init_states,)) * max_loss
246     i = -1
247
248     while loss_list.max().item() >
249         stop_loss_lim:
250         i += 1
251
252         which_r0 = i % num_init_states
253
254         x0, v0 = init_list[which_r0]
255         y = y_list[which_r0]
256         x_data = x_data_list[which_r0]
257         y_data = y_data_list[which_r0]
258         input_physics =
259             input_physics_list[which_r0]
260         x_in = x_in_list[which_r0]
261
262         optimizer.zero_grad()
263
264         # compute the "data loss"
265         yhp = model(x_data)
266         dx = torch.autograd.grad(yhp,
267             input_physics, torch.
268             ones_like(yhp), create_graph=
269             True)[0][:, 0:1] # computes
270             dy/dx
271         dx2 = torch.autograd.grad(dx,
272             input_physics, torch.
273             ones_like(dx), create_graph=
274             True)[0][:, 0:1] # computes d
275             ^2y/dx^2
276         physics = dx2 + mu*dx + k*yhp #
277             computes the residual of the
278             1D harmonic oscillator
279             differential equation
280         loss2 = torch.mean(physics**2)
281
282         # print(dx.shape, dx2.shape, yhp
283             .shape)
284         # print(dx[0], v0)
285
286         loss3 = torch.mean((dx[0] - v0)
287             **2)
288
289         # backpropagate joint loss
290         loss = beta[0] * loss1 + beta[1]
291             * loss2 + beta[2] * loss3 #
292             add two loss terms together
293         loss_list[which_r0] = loss
294
295         loss.backward()
296         optimizer.step()
297
298         # plot the result as training
299         progresses
300         if (i+1) % 298 == 0:
301
302             yh = model(x_in).detach()
303             xp = x_in[:,0].view(-1,1).
304                 detach()
305             print(f"{i+1}th iteration ->
306                 x0 loss: initial state
307                 no.: {which_r0} | x0
308                 loss: {loss1:.4f} | v0
309                 loss: {loss3:.4f} |
310                 physics loss: {loss2:.4f
311                 }| Scaled Total loss: {
312                 loss:.4f}")
313
314             plot_result(x,y,x_data[:,0],
315                 y_data,yh,xp)

```

```

294     file = "plots_augmented/
295         pinn_%.8i.png"%(i+1)
296     plt.savefig(file,
297                 bbox_inches='tight',
298                 pad_inches=0.1, dpi=100,
299                 facecolor="white")
300     plt.close("all")
301
302     ##### Testing the augmented for
303     testing and training data points
304
305     beta = [10, 1e-2, 1]
306
307     t = torch.arange(0, 2*torch.pi, 0.1)
308     A = 1
309
310     my_plot_names = []
311
312     for x0, v0 in zip(x_test, y_test):
313
314         x = torch.linspace(0,1,500).view
315         (-1,1)
316         x_in = torch.stack([x, torch.
317         ones_like(x) * x0, torch.
318         ones_like(x) * v0], dim=1).
319         view(-1,3)
320
321         y = oscillator_init_cond(d, w0,
322         x, x0, v0)
323         y_out = model(x_in).detach()
324
325         figure = plt.figure()
326         plt.plot(x, y, color="grey",
327                 linewidth=2, alpha=0.8, label
328                 ="Exact solution")
329         plt.plot(x_in[:,0], y_out, color
330                 ="tab:blue", linewidth=4,
331                 alpha=0.8, label="Neural
332                 network prediction")
333         plt.axhline(0, color='black',
334                 linewidth=0.8, linestyle='--',
335                 )
336         l = plt.legend(frameon=True,
337                 fontsize="large")
338         plt.setp(l.get_texts(), color="k
339                 ")
340         plt.xlim(-0.05, 1.05)
341         plt.ylim(-1.1, 1.1)
342
343         plt_name = f"my_plots_aug/{x0:.2
344         f}_{v0:.2f}_train.png"
345         my_plot_names.append(plt_name)
346
347         plt.savefig(plt_name)
348         plt.close(fig=figure)
349
350     plt.show()
351
352     my_plot_names = []
353
354     for x0, v0 in zip(x_training,
355         y_training):
356
357         x = torch.linspace(0,1,500).view
358         (-1,1)
359         x_in = torch.stack([x, torch.
360         ones_like(x) * x0, torch.
361         ones_like(x) * v0], dim=1).
362         view(-1,3)
363
364         y = oscillator_init_cond(d, w0,
365         x, x0, v0)
366         y_out = model(x_in).detach()
367
368         figure = plt.figure()
369         plt.plot(x, y, color="grey",
370                 linewidth=2, alpha=0.8, label
371                 ="Exact solution")
372         plt.plot(x_in[:,0], y_out, color
373                 ="tab:blue", linewidth=4,
374                 alpha=0.8, label="Neural
375                 network prediction")
376         plt.axhline(0, color='black',
377                 linewidth=0.8, linestyle='--',
378                 )
379         l = plt.legend(frameon=True,
380                 fontsize="large")
381         plt.setp(l.get_texts(), color="k
382                 ")
383         plt.xlim(-0.05, 1.05)
384         plt.ylim(-1.1, 1.1)
385
386         plt_name = f"my_plots_aug/{x0:.2
387         f}_{v0:.2f}_test.png"
388         my_plot_names.append(plt_name)
389
390         plt.savefig(plt_name)
391         plt.close(fig=figure)
392
393     plt.show()

```