Introduction to Physics Informed Neural Networks

Agney.K.Rajeev (Roll no.: 2111010)[1, *]
[1]*School of Physical Sciences, National Institute of Science Education and Research, HBNI, Jatni-752050, India*
(Dated: November 24, 2023)

I. AIM

The project was undertaken with the following broad aims

- Understand the principles of Physics Informed Neural Networks (PINNs) and their distinctness from standard Neural Networks
- Understand the application of PINNs as Differential Equation Solvers
- From the previous aims, recognize and determine the usefulness of PINNs in solving physical phenomena and their merits and demerits compared to other methods

A. Neural Networks

A neural network can be represented as a special instance of a **Directed Graph** with weighted vertices and weighted edges.

The vertices (also known as **neurons**) are partitioned into **layers**. The first (source) layer is called the **input layer**, the last (sink) layer is called the **output layer** and the middle layers are called the **hidden layers**. Every neuron of a layer is only connected to each neuron of the next layer by a directed edge. The weight of a neuron is the **value** of the neuron and the weight of an edge is the **weight** between the neurons it connects.

(If the number of layers of a neural network is n, then the neural network is an n-partite directed graph with each hidden layer being a vertex cut.)

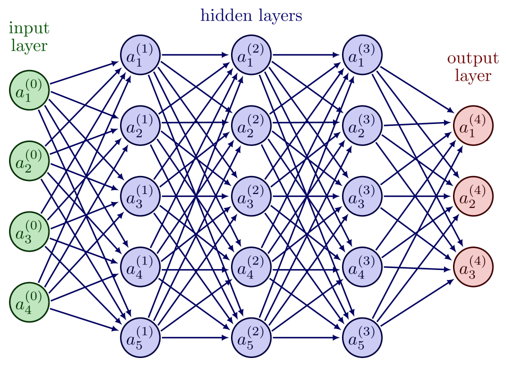


FIG. 1. Diagrammatic representation of a Neural Network

the value of a neuron in a layer (other than the input layer) is determined by the values of all the neurons in the previous layer as well as the weights of all the edges connecting them.

The value of the ith neuron in the (j+1)th layer is given by the formula

$$a_i^{(j+1)} = R\left(\left(\sum_k a_k^{(j)} w_{ki}^{(j)}\right) + b_i^{(j)}\right) \tag{1}$$

Here $a_k^{(j)}$ is the value of the kth neuron in the jth layer, $w_{ki}^{(j)}$ is the weight between $a_k^{(j)}$ and $a_i^{(j+1)}$, $b_i^{(j)}$ is a value known as the **bias** for $a_i^{(j+1)}$ and R is some non-linear function called the **activation function** (In this project, we choose the tanh function as the activation function).

If we represent the values of all neurons in a layer as a row vector (and all weights between two layers as a matrix and all biases as row vectors), we can rewrite (1) as

$$\mathbf{a^{(j+1)}} = R\left(\mathbf{a^{(j)}}.\mathbf{W^{(j)}} + \mathbf{b^{(j)}}\right) \tag{2}$$

The most important feature of a Neural Network is that it is a **Universal Function Approximator**. Which means any function can be approximated up to a tolerance within a finite range by a neural network with a finite amount of layers and neurons.

Throughout this project, we shall have our Neural Network produce an approximation to the solution function of a Differential Equation

In order to understand how well our neural network approximates a function, we employ a function which produces a value quantifying the closeness of our approximation to the true solution. This function is known as the **loss function**.

In this project, we shall use the Mean Squared Error function as our loss function which is defined as

$$L = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 \tag{3}$$

Here f is the function approximation by the neural network and the points (x_i, y_i) are called **sample points**. The sample points belong to the true solution function which we attempt to approximate.

We can see that if the prediction of the neural network $(f(x_i)$ is close to the true value (y_i), then the loss function will be minimal. Else, it will be large.

* agneyk.rajeev@niser.ac.in

The process of training the neural network is **minimizing** the loss function using **optimizers** like Gradient Descent with respect to the weights and biases.

B. Physics Informed Neural Networks

Physics-Informed Neural Networks (PINNs) are a type of Neural Networks that can embed the knowledge of any physical laws that govern a given data-set in the learning process. These physical laws are represented in the form of differential equations and are passed on to the Neural Network in the form of an automatic differentiation function.

Keeping the structure of the network intact, we shall incorporate this differential equation in the loss function.

If we write our differential equation as $Df = 0$ where D is the differential operator and f is the solution function, then we can re-imagine our loss function in the form

$$L = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 + k\frac{1}{m}\sum_{j=1}^{m}((Df)(x_j) - 0)^2 \quad (4)$$

Here, we can observe that an additional **differential loss** term has been added to the net loss. This differential loss is calculated at a new set of points (x_j, y_j) known as the collocation points. The constant k is the relative weightage given to the differential loss w.r.t the standard sample point loss.

Note that the differential loss term will have a large value if the differential equation is disobeyed at the collocation points and will be minimal if the neural network adheres to the differential equation.

C. Solving Differential Equations

We can recognize that PINNs are much more equipped to solve differential equations than standard neural networks.

While a standard neural network might be able to produce an approximate solution function with enough sample points, the solution cannot be expected to be sustained to parts of the domain where sample points are not present. Furthermore, there is little to no involvement of the differential equation in producing the output of the neural network and hence, a large number of sample points shall be required to produce a decent approximate solution. However, PINNs have information regarding the differential equation incorporated into it with the help of collocation points, which incentivizes it to maintain an approximate solution in regions without sample points. Since the ordinate value of the collocation points (y_j) is not present in the loss term, collocation points can be sampled from anywhere in the domain. This also reduces the number of sample points required to produce a good enough approximate solution. Hence, we shall attempt to use Physics Informed Neural Networks to solve some elementary differential equations (with known or understood solutions) to gauge its ability as a Differential Equation Solver.

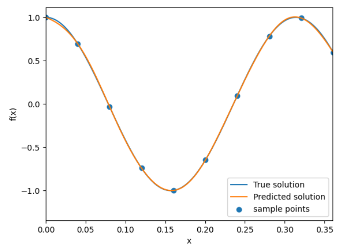


FIG. 2. Solution to the Harmonic Oscillator Equation by a standard neural network

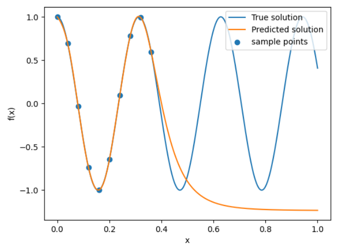


FIG. 3. Solution to the Harmonic Oscillator Equation by a standard neural network with extended domain

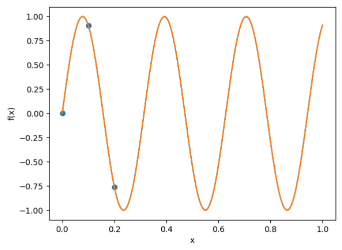


FIG. 4. Solution to the Harmonic Oscillator Equation by a physics informed neural network with extended domain

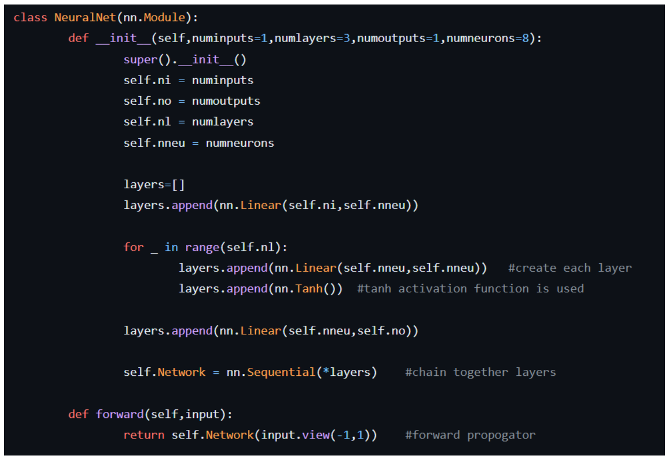
Almost any physical phenomenon can be written as a differential equation. Dynamics of a classical system can be represented using Lagrange's Equation, dynamics of a quantum system can be represented using Schrodinger's Equation, Propagation of Electromagnetic waves using Maxwell's Equations etc. Thus, solving differential equations are essential to studying any physical system.

The standard methods to solving these differential equations are using analytic simplification or numerical techniques such as Finite Element Methods (FEM), Finite Difference Methods (FDM) and Finite Volume Methods (FVM).

Although a direct comparison and bench-marking of the above mentioned methods in comparison to PINNs is beyond the scope of this project, we shall attempt to discuss some possible advantages as well as disadvantages possessed by PINNs over other methods, judging from the results obtained in solving elementary differential equations and the understanding the difference in working principle.

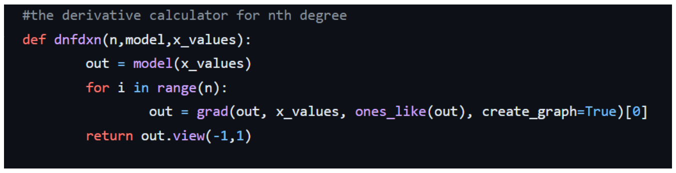
D. Code

The Model was implemented in *Python3.10*. The *Pytorch* module was as the machine learning framework. *Numpy* package was used for mathematical computations and some data structures. *Matplotlib* Library was used for all plots and diagrams. All code used in this project can be found in the following GitHub page `https://github.com/AKR211/PINN`(ref [3])



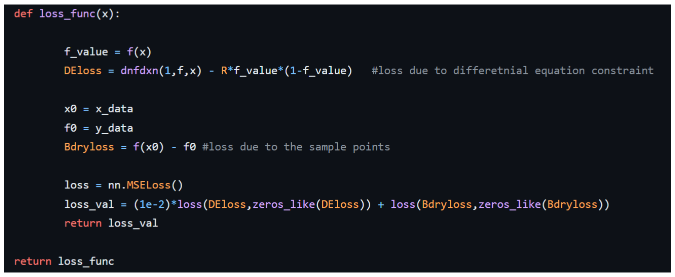
```
class NeuralNet(nn.Module):
        def __init__(self,numinputs=1,numlayers=3,numoutputs=1,numneurons=8):
                super().__init__()
                self.ni = numinputs
                self.no = numoutputs
                self.nl = numlayers
                self.nneu = numneurons

                layers=[]
                layers.append(nn.Linear(self.ni,self.nneu))

                for _ in range(self.nl):
                        layers.append(nn.Linear(self.nneu,self.nneu))   #create each layer
                        layers.append(nn.Tanh())  #tanh activation function is used

                layers.append(nn.Linear(self.nneu,self.no))

                self.Network = nn.Sequential(*layers)    #chain together layers

        def forward(self,input):
                return self.Network(input.view(-1,1))    #forward propogator
```

FIG. 5. Code for the Neural Network



```
#the derivative calculator for nth degree
def dnfdxn(n,model,x_values):
        out = model(x_values)
        for i in range(n):
                out = grad(out, x_values, ones_like(out), create_graph=True)[0]
        return out.view(-1,1)
```

FIG. 6. Code for calculating derivatives (we use automatic differentiation provided by the *autograd* module of PyTorch)

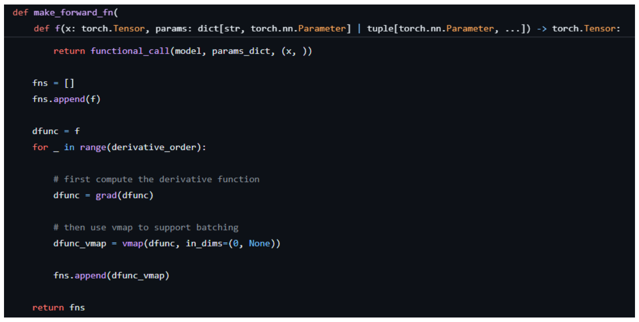


```
def loss_func(x):

        f_value = f(x)
        DEloss = dnfdxn(1,f,x) - R*f_value*(1-f_value)   #loss due to differetnial equation constraint

        x0 = x_data
        f0 = y_data
        Bdryloss = f(x0) - f0 #loss due to the sample points

        loss = nn.MSELoss()
        loss_val = (1e-2)*loss(DEloss,zeros_like(DEloss)) + loss(Bdryloss,zeros_like(Bdryloss))
        return loss_val

return loss_func
```

FIG. 7. Code for the Loss function

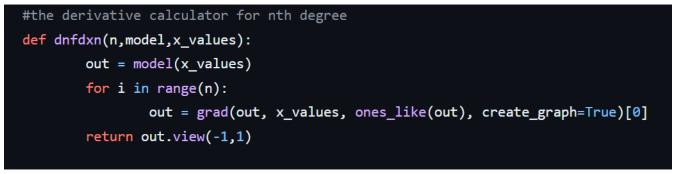
II. MODIFICATIONS

The original code snippets in ref[1] uses the functional API of PyTorch (the *functorch* module) to calculate the derivatives and the loss function by decoupling the parameters of the neural network and using *func.grad*. This involves importing additional libraries as well as being computationally slow. This approach was replaced by the in-built *requires_grad* method of PyTorch matrices which automatically computes the derivative with each function call. This speeds up the training process especially for higher order derivatives.



```
def make_forward_fn(
    def f(x: torch.Tensor, params: dict[str, torch.nn.Parameter] | tuple[torch.nn.Parameter, ...]) -> torch.Tensor:
        return functional_call(model, params_dict, (x, ))

    fns = []
    fns.append(f)

    dfunc = f
    for _ in range(derivative_order):

        # first compute the derivative function
        dfunc = grad(dfunc)

        # then use vmap to support batching
        dfunc_vmap = vmap(dfunc, in_dims=(0, None))

        fns.append(dfunc_vmap)

    return fns
```

FIG. 8. Original Code for the derivative calculation



```
#the derivative calculator for nth degree
def dnfdxn(n,model,x_values):
        out = model(x_values)
        for i in range(n):
                out = grad(out, x_values, ones_like(out), create_graph=True)[0]
        return out.view(-1,1)
```

FIG. 9. Modified Code for the derivative calculation

The previously mentioned relative weightage of the differential loss, k, was also additionally introduced. This was not implemented in the original code but was mentioned as a possible improvement (In fig. 7, k = 10^{-2}). This helps in avoiding trivial solutions for the differential equations (such as $f(x) = 0$).

Furthermore, the original article only employed the PINN to solve the Logistic Differential Equation which is a first-order ordinary differential equation in one variable. In this project, a diverse and larger pool of differential equations were attempted to be solved with a PINN. The damped harmonic oscillator equation (second order ODE), the wave equation (second order PDE in two variables) and the diffusion equation (two variable PDE which is first and second order w.r.t one variable each; extremely difficult to solve analytically) were all solved using a PINN.

III. RESULTS

Apart from obtaining a similar result to that from the original article for the logistic differential equation, the results for the other differential equations are provided

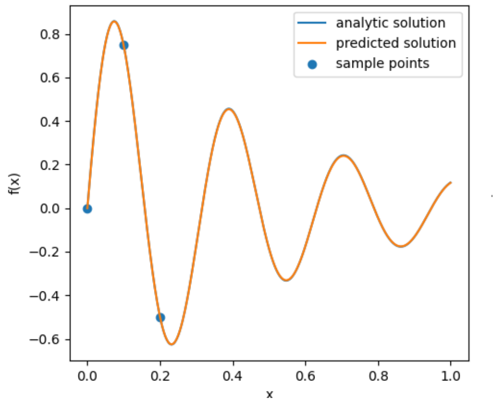


FIG. 10. Solution function for the Damped Harmonic oscillator Equation

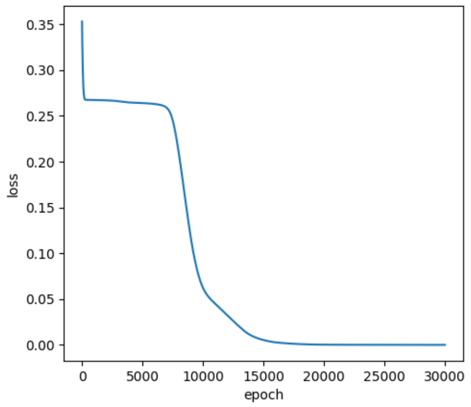


FIG. 11. Loss plot for the Equation with respect to each training step (epoch)

The damped harmonic oscillator equation is used to approximate oscillatory behaviours for small perturbations. The solution can observed to be nearly identical to the analytical solution. The final loss was calculated to be 8.44×10^{-5}.The PINN was constructed with one hidden layer of 16 neurons and was trained for 30,000 training steps with 30 collocation points. The relative weightage of the differential loss was 10^{-4}. The complexity of the neural network is the same as that of the Logistic DE but required more training to converge to the solution. This is understandable since this is a higher order ODE. The differential loss was given a relatively small weightage since we have 3 sample points (only 2 is required for an analytical solution) in the first wavelength and the differential loss simply has to propagate this to further wavelengths.

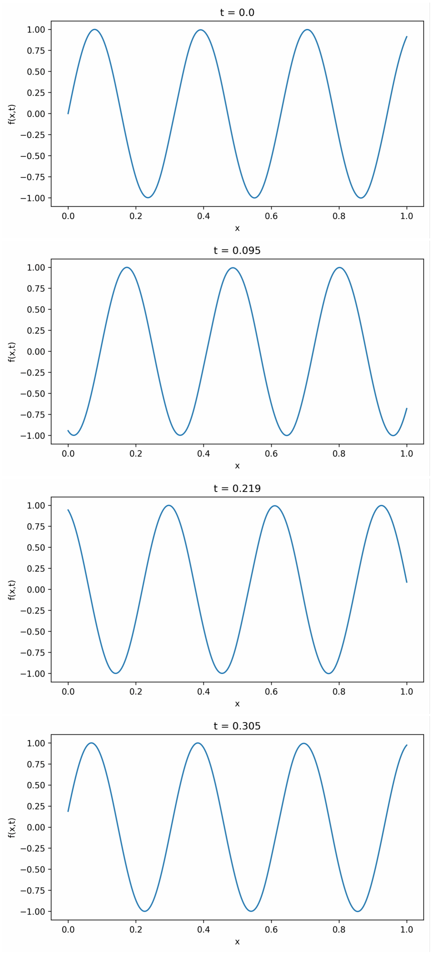


FIG. 12. Solution function for the Wave Equation, (x,f(x,t)) for different t

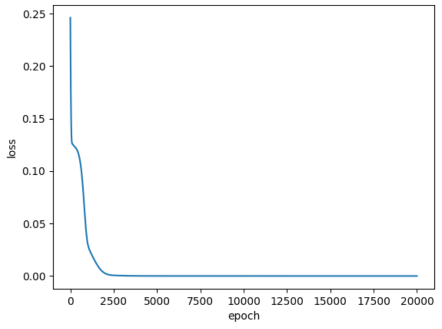


FIG. 13. Loss plot for the Equation with respect to each training step (epoch)

The wave equation describes mechanical and electromagnetic waves as well as standing wave fields. The solution was found to be sinusoidal w.r.t x and can be observed to be shifting along the positive x- direction with time. This is the expected behaviour of the solution. The final loss was calculated to be 1.2×10^{-6}.The PINN was constructed with two hidden layers of 32 neurons and was trained for 20,000 training steps with 30 collocation points. The relative weightage of the differential loss was 10^{-4}. We required an additional layer and double the neurons for each layer as compared to the previous ODEs. This is not surprising since the wave equation is a PDE with two inputs (x,t) than one.

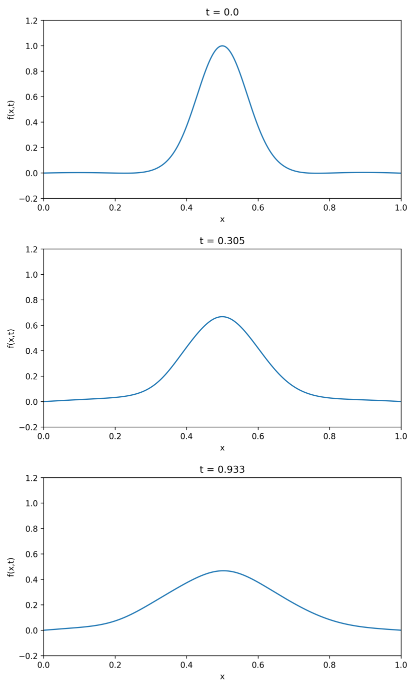


FIG. 14. Solution function for the Diffusion Equation, (x,f(x,t)) for different t

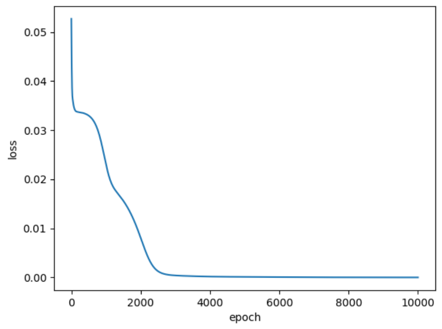


FIG. 15. Loss plot for the Equation with respect to each training step (epoch)

The diffusion equation is used to describe particle diffusion in a volume as well as thermal diffusivity in materials. The solution was found to be a Gaussian w.r.t x and can be observed to be widening with time (and the peak reducing to keep the total area constant). This is the expected behaviour of the solution. The final loss was calculated to be 5.67×10^{-6}.The PINN was constructed with two hidden layers of 32 neurons and was trained for 10,000 training steps with 30 collocation points. The relative weightage of the differential loss was 10^{-4}. The diffusion equation is quite difficult to solve analytically and can only be accomplished for certain initial conditions. The ability of the PINN to produce a close to perfect approximation is testament to its resourcefulness.

Based on the results obtained above as well as an understanding of other methods for solving differential equations, we can attempt to non-empirically predict some advantages and disadvantages of using a PINN to solve a Differential Equation.

Advantages :

- **PINNs are self-learning**
  There is no need to derive a systematic solution method. PINNs can converge to a solution by itself with appropriate supervision
- **PINNs are data-driven**
  The performance of a PINN relies majorly on the training data and not the complexity of the underlying dynamics. It can also improve upon itself with the addition of new data
- **PINNs are independent of the intermediate training steps**
  Only the final output needs to be an accurate approximation. Errors in intermediate steps do not affect the final output.
- **PINNs are not discretized**
  A PINN is a continous function (as opposed to Finite Difference Methods which only compute the solution at step sizes). It can provide the function value for any input without interpolation.

Disadvantages :

- **PINNs are self-learning**
  PINNs are a black-boxes. We do not have a solid proof as to whether the PINN is trained to be the correct function
- **PINNs are data-driven**
  PINNs can perform poorly if there is a lack of sample points. Also the computation cost for using a large number of points can be quite high
- **PINNs are independent of the intermediate training steps**
  We cannot access a partial solution. Extending the domain for approximation would require retraining the whole model
- **PINNs are not discretized**
  The computation cost or accuracy cannot be adjusted with the step size (Finite Difference Methods have the luxury of reducing the computational cost by sacrificing accuracy by increasing the step size)

Overall, we can observe that Physics Informed Neural Networks are a new paradigm of Differential Equation

Solvers. Although it is a relatively new concept with limited practical uses as of the moment, the rapid evolution of Artificial Intelligence and Machine Learning suggests a large amount of potential for PINNs in nearly every field of Physics (and other sciences). In this project, we have demonstrated that PINNs are certainly capable of solving Differential Equations of varying complexity which are relevant in physics to a sufficient accuracy as well as provide some possible merits and demerits of using a PINN for the same. Further scope of this project involves using PINNs to solve more complex Differential Equations (such as analytically unsolvable equations) as well as performing a detailed study of the performance and efficiency of PINNs versus other computational methods for solving Differential Equations.

IV. QUESTIONS

1. What is the adjacency matrix of a Neural network?

The adjacency matrix of a neural network is not usually explicitly defined as there are a large number of nodes in the network (most of which are not connected to each other.)

E.g. If a network with two hidden layers of 32 neurons each would require a 64x64 matrix out of which 75% are just zeros. It is better to just represent the network with the size of the 25% non-zero weights and specifying between which two layers they belong.

2. Try and solve the Schrodinger equation for quantum tunneling

The Time Independent Schrodinger equation can be written as

$$\frac{d^2\psi}{dx^2} + \frac{2m(E - V(x))}{\hbar^2}\psi = 0$$

For quantum tunneling to occur, we require a potential barrier such that

$$V(x) = \begin{cases} 0 & x \leq a \\ V_0 & a \leq x \leq b \\ 0 & b \leq x \end{cases}$$

where $V_0 > E$

We can observe that the approximation behaves nicely before and after crossing the barrier with the amplitude of the wave decreasing (but still being non-zero) after crossing the barrier. However, inside the barrier, an exponential decay was expected and not an oscillatory solution. This can be overcome to an extent by increasing the number of sample points in the region of the barrier but the oscillatory behaviour is still present and feasibility of obtaining so many sample points is questionable

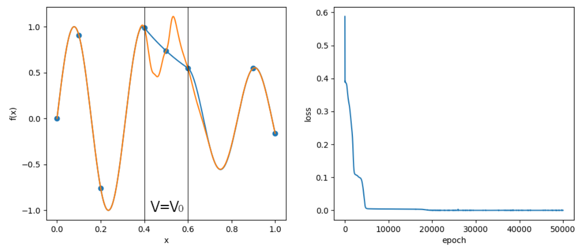


FIG. 16. Plot for quantum tunneling and the loss evolution

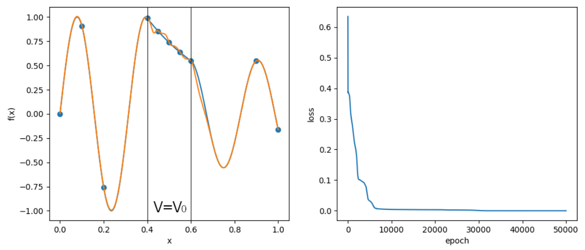
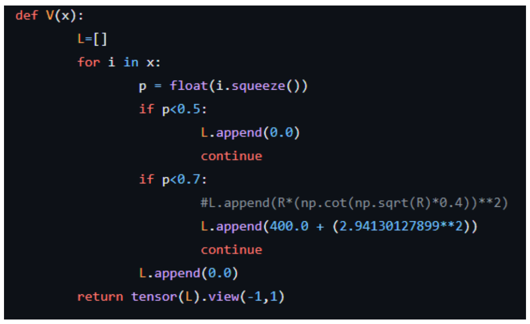


FIG. 17. Plot for quantum tunneling (more points) and the loss evolution

This likely occurs because of the piece-wise definition of the potential (and hence the differential equation and hence the solution function). the oscillatory behaviour which the Neural Network learnt from outside the barrier carries over to its behaviour inside the barrier. While a Neural Network is theoretically capable of approximating a piece-wise function, it is very difficult to train it do so as we have to train it to behave as different functions at different regions. This can be overcome by training two neural networks separately for both regions with appropriate boundary points to ensure that they are in mutual agreement. However, this would require a larger amount of computational cost and time



```
def V(x):
        L=[]
        for i in x:
                p = float(i.squeeze())
                if p<0.5:
                        L.append(0.0)
                        continue
                if p<0.7:
                        #L.append(R*(np.cot(np.sqrt(R)*0.4))**2)
                        L.append(400.0 + (2.94130127899**2))
                        continue
                L.append(0.0)
        return tensor(L).view(-1,1)
```

FIG. 18. Code for the potential function(full code can be found in the GitHub page ref[3])

3. Why are Physics-Informed Neural Networks named so?

PINNs are DE solvers. They have inherently very little to do with physics in terms of its working principle. They are named so because they combine principles from physics with neural networks to solve scientific and engineering problems. They are designed to incorporate the laws of physics as constraints into the training process of neural networks, making them informed by the underlying physical principles.

The concept of "Domain-Informed Neural Networks" can be executed to a variety of fields where you would like to ensure that neural network solutions fit with domain-specific information or limitations and not just physics. The term "Physics-Informed Neural Networks" is well-known because it was one of the first uses of this technology to earn academic interest. Similar strategies, however, can and have been applied to problems in Biology, Chemistry and other fields.

4. How are PINNs continous functions?

A discrete function is a function in which the domain and range are each a discrete set of values. A non-discrete (continuous) function is one that is continuous either on its entire domain, or on intervals within its domain. A discrete structure can produce a continuous output if the inputs are continuous and the operations executed on the input ensure that the output is continuous as well.

E.g. A matrix is a discrete structure. Consider the following function composed of a matrix multiplication

$$f(x_1, x_2) = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix}$$

if $x_1, x_2 \in R$ then $f(x_1, x_2) \in R^2$. Both domain and range are continuous and hence the function is continuous

The neural network consists only of operations described in section I, all of which (matrix multiplication, matrix addition and application of the activation function) are continuous. Hence, the Neural Network is able to produce a continuous output if provided with a continuous input.

V. REFERENCES

1. M. Dagrada, "Introduction to physics-informed neural networks",Towards Data Science, 2022

2. B. Moseley, "So, what is a physics-informed neural network?" 2021

3. My Github Repository for the Project : AKR211/PINN