THE ROLLING CODES

MUSTAFA TOSUN

090200142

ATINÇ BAŞ

090200144

ELİFNUR MUTLU

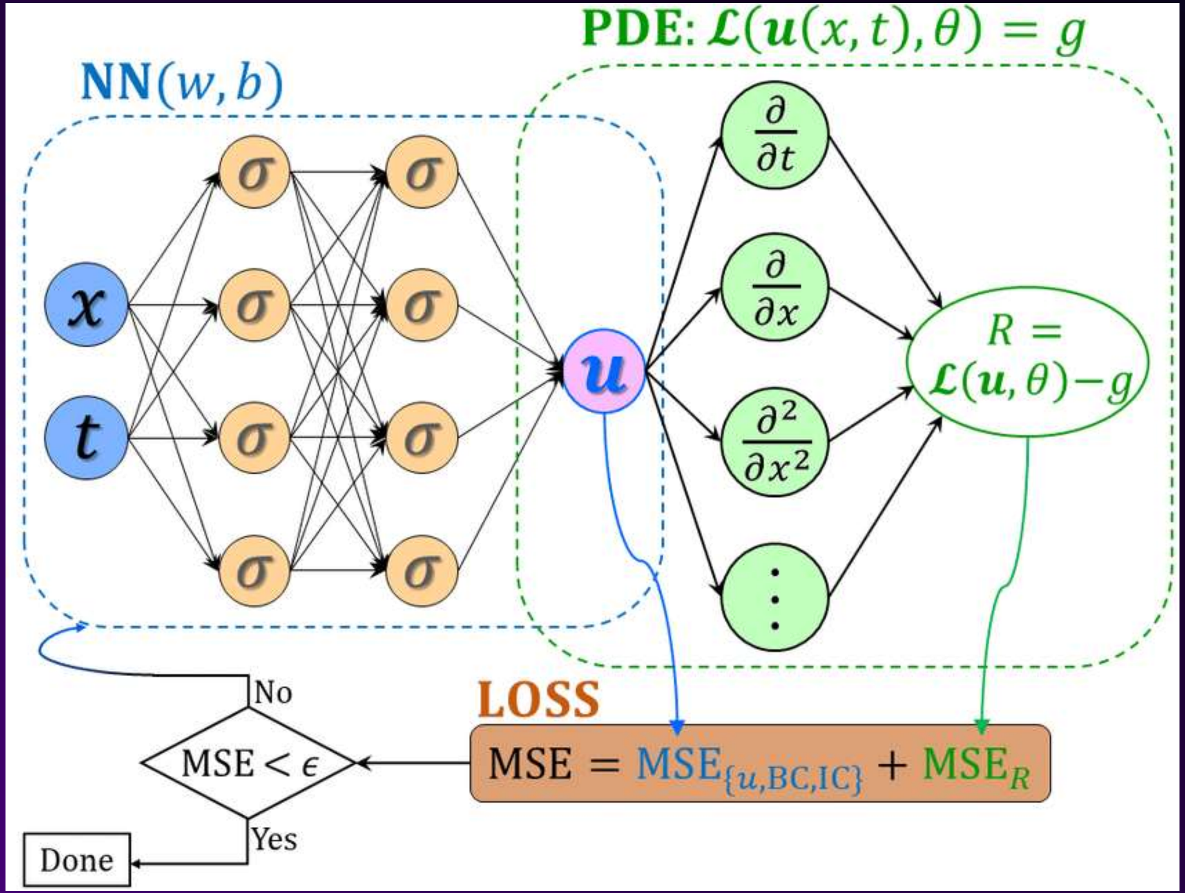
090200151

TUNAHAN AKGÜL

090200107

WHAT IS PINN?

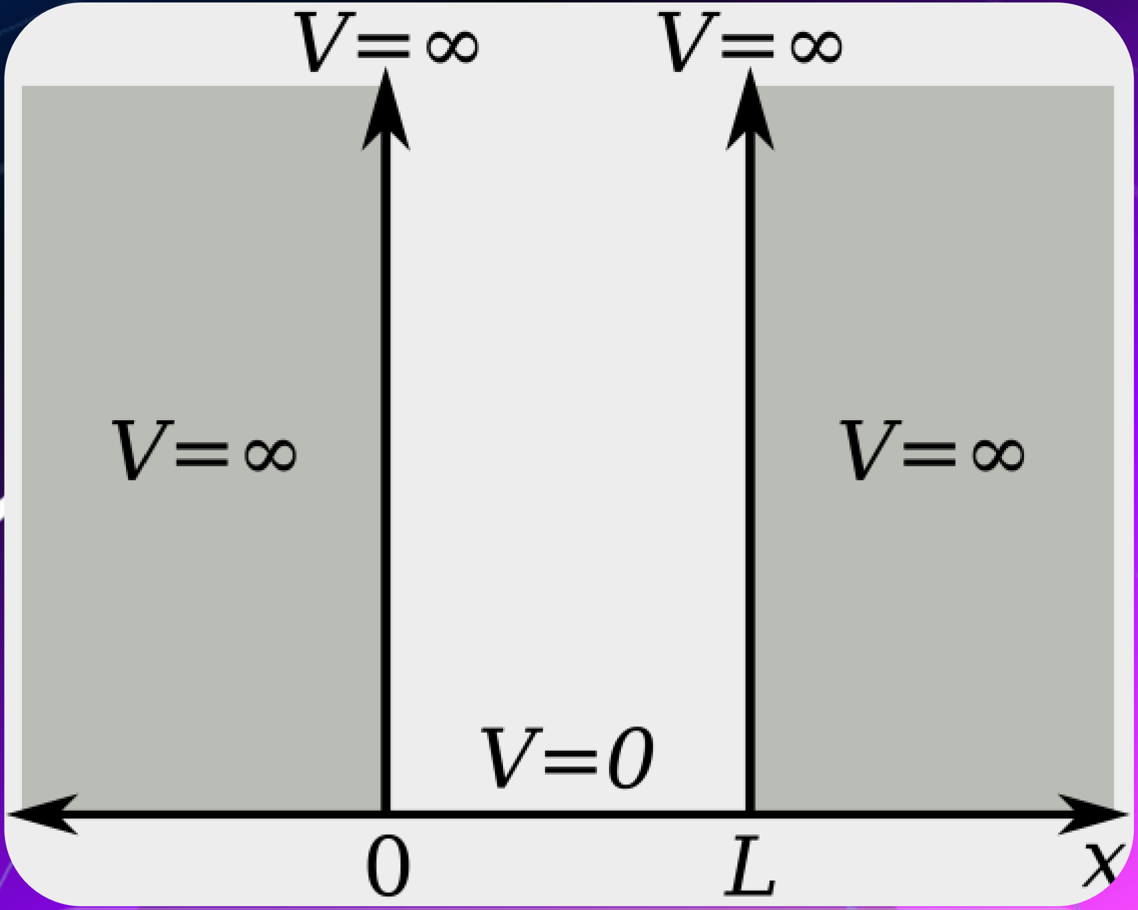
Physics-Informed Neural Networks (PINN) is an approach that combines artificial neural networks with physical laws to solve differential equations. By incorporating physical laws into the loss function of the model, it reduces the need for extensive data.



PROJECT OBJECTIVE

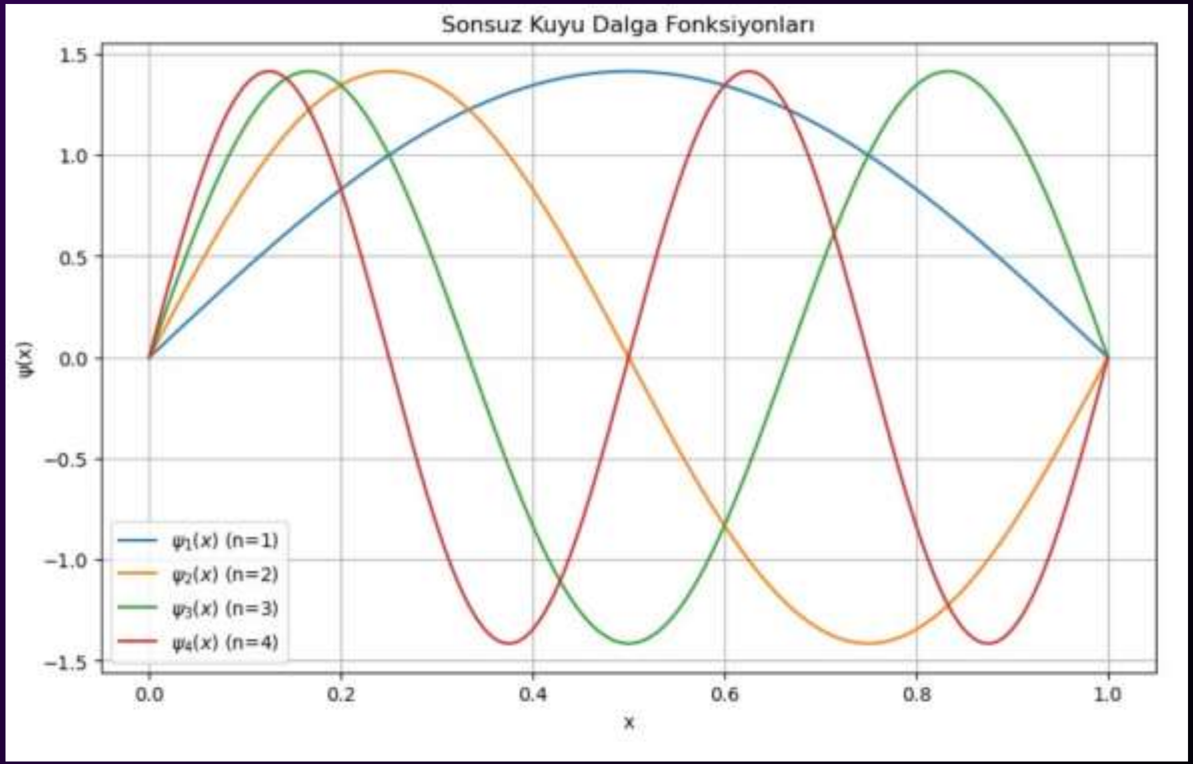
- To solve the Schrödinger equation in an infinite potential well using PINN.
- To estimate the energy levels (E) of a quantum system using the PINN model.
- To compare PINN predictions with analytical solutions (wave function and energy levels).

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V(\boldsymbol{x})\psi = E\psi$$



Analytical Wave Functions

- To use analytical wave functions and energy levels as reference data
- To evaluate the accuracy of predictions obtained with the PINN model.



$$\psi(x) = \sqrt{\frac{2}{a}} \sin\left(\frac{n\pi}{a} x\right)$$

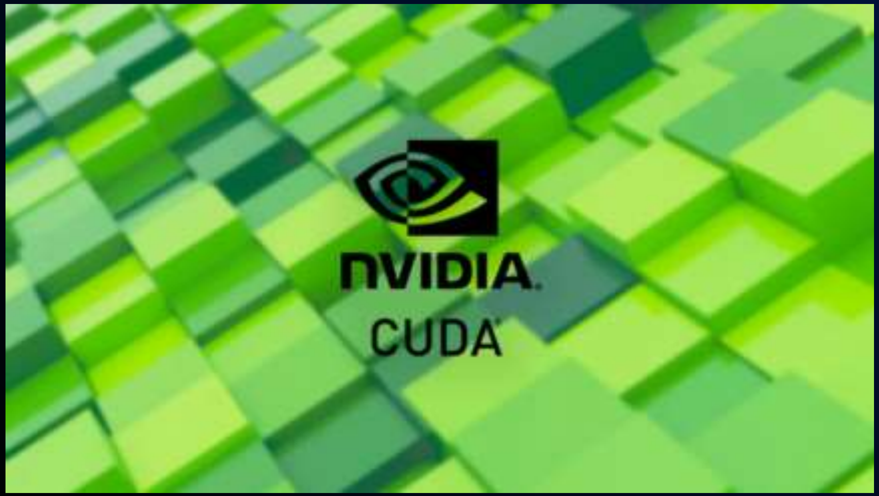
$$E_n = \frac{n^2 \pi^2 \hbar^2}{2ma^2}$$

WHAT WE LEARNED

```
In [11]: import torch
         import torch.nn as nn
         import numpy as np
         import matplotlib.pyplot as plt
         import torch.nn.init as init
         #device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
         torch.set_default_device('cuda')  # Varsayılan cihazı GPU olarak ayarla
         torch.set_default_dtype(torch.float32)  # Varsayılan veri tipini float32 olarak ayarla
         from matplotlib.animation import FuncAnimation
         import copy
         import warnings
         warnings.filterwarnings("ignore")
```

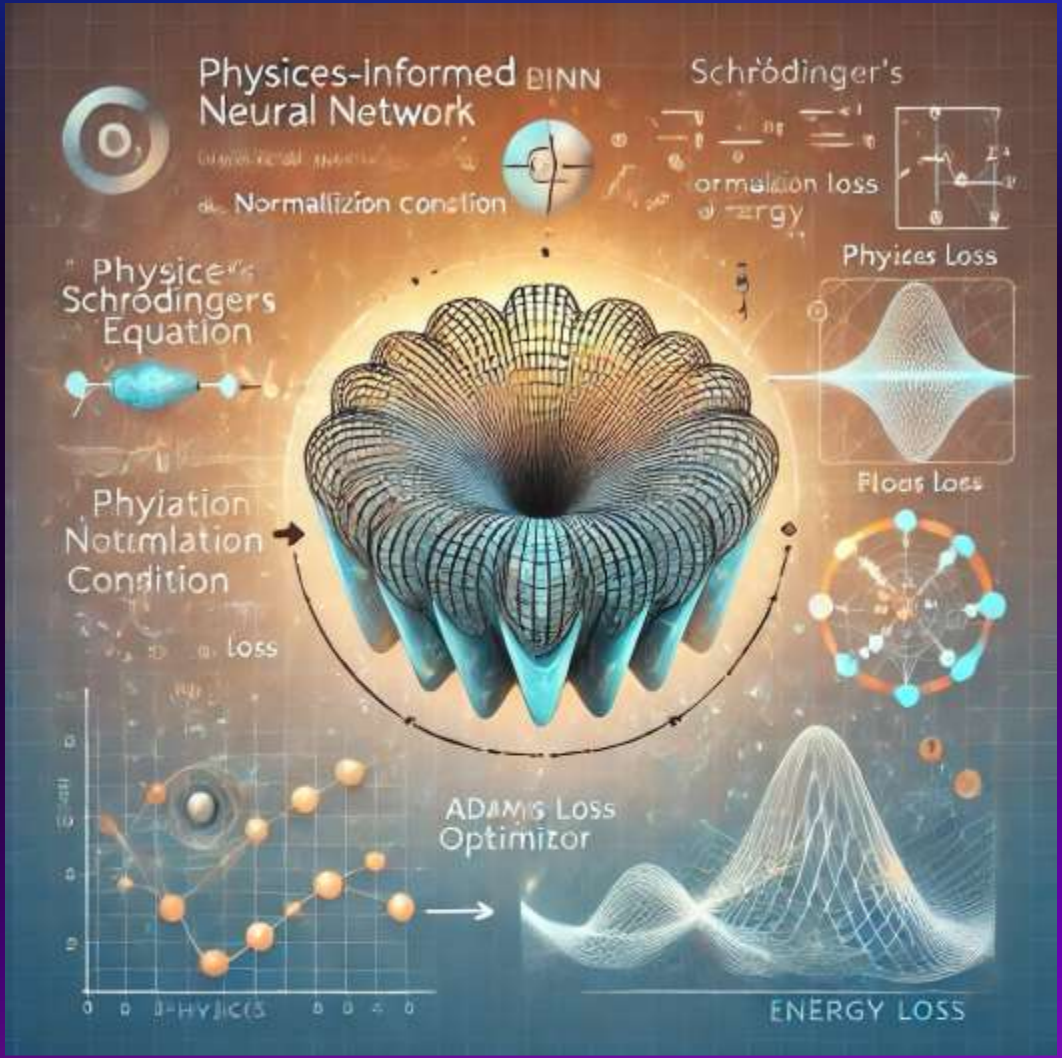
```
In [3]: class mySin(torch.nn.Module):
            @staticmethod
            def forward(input):
                return torch.sin(input)
```

```
In [4]: class NNs(nn.Module):

            def __init__(self):

                super(NNs,self).__init__()

                #Her katman çıktısına doğrusal olmayan bir dönüşüm uyguluyor.
                self.activation = mySin()
                #başka activation functionlar da denenebilir

                #Enerji seviyeleri için 1-1 doğrusal katman oluşturuyor
                self.input_En = nn.Linear(1,1) #w.i + b

                #2 nöron - 50
                self.hidden11 = nn.Linear(2, 64) #gizli
                self.hidden12 = nn.Linear(64, 64) #gizli
                self.output_layer = nn.Linear(64, 1) #sonuç


            def forward(self, X):

                #yukarıda en için tanımladığım layera gönderdim ve çıktı aldım
                #En_out = self.input_En(torch.ones_like(X))

                En_out = self.input_En(torch.ones_like(X))
                h1 = self.hidden11(torch.cat((X, En_out),1))
                hidden11_outputs = self.activation(h1)
                hidden12_outputs = self.activation(self.hidden12(hidden11_outputs))
                output = self.output_layer(hidden12_outputs)

                return output, En_out
```



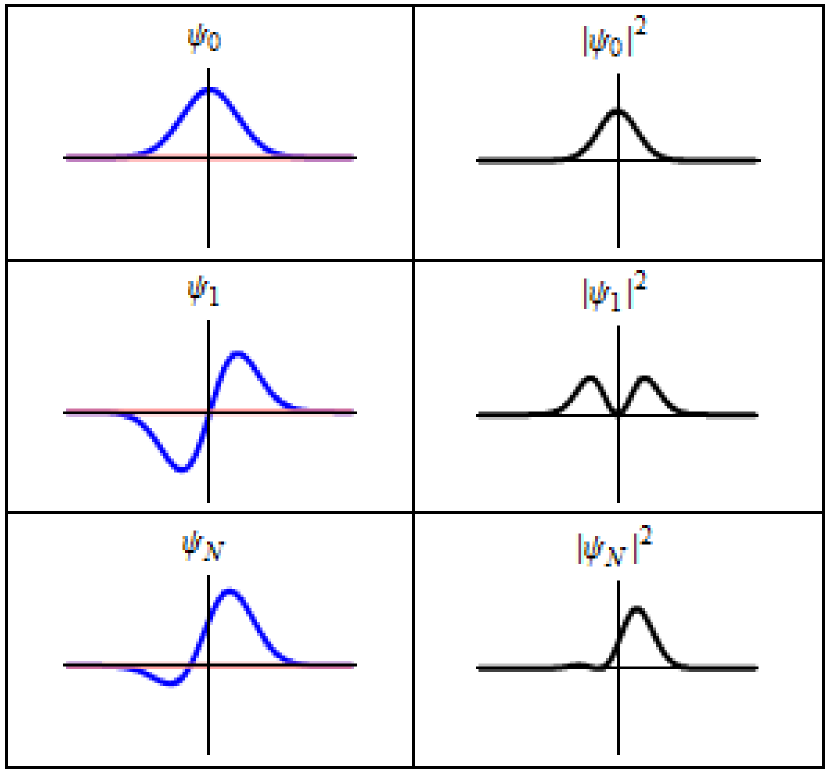
GENERAL STRUCTURE

- The activation function was selected.
- The neural network section was created.
- The required derivative functions were defined.
- Layers were constructed.
- The physics-informed section was implemented, and loss functions were defined.
- The output was generated.
- Model solutions were compared with analytical solutions.

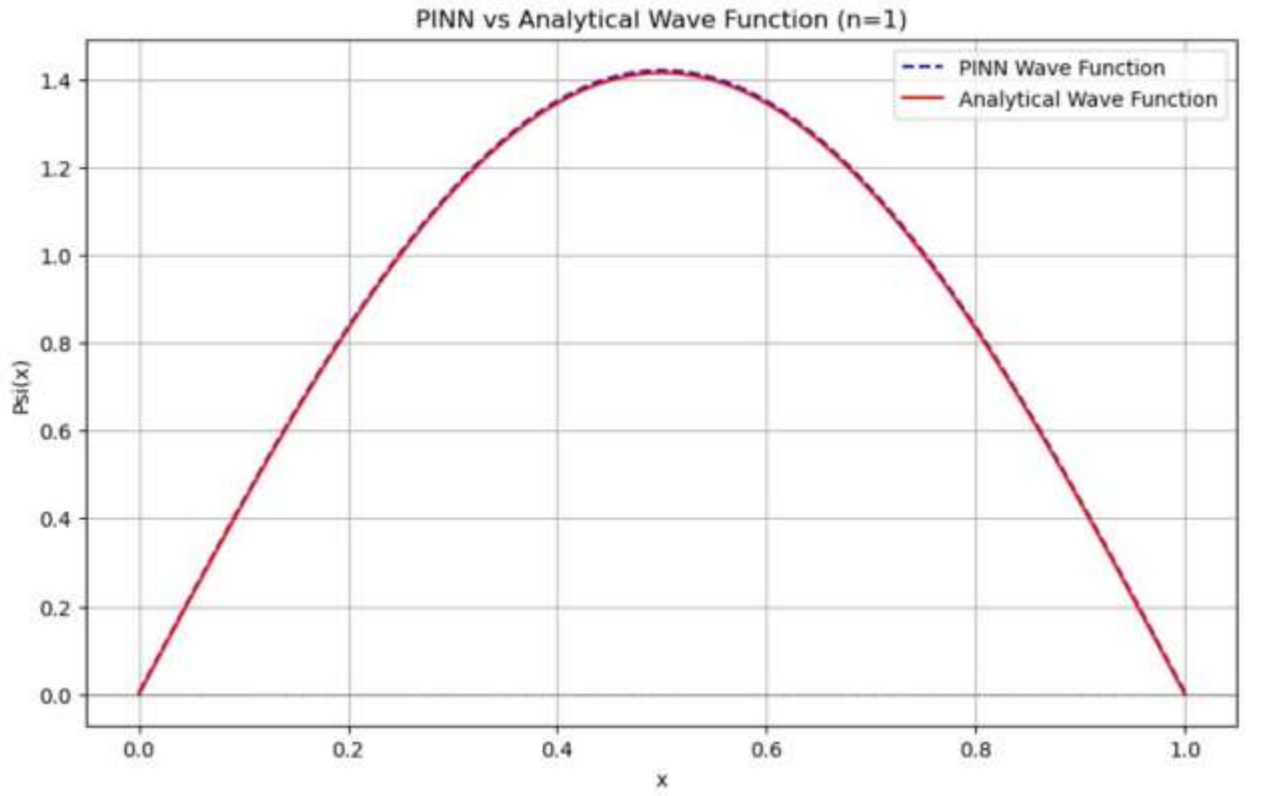


DEVELOPMENT OF THE SOLUTION

```python
# Loss function
def pinn_loss(model, x, n):
    psi = model(x)
    psi_x = torch.autograd.grad(psi, x, grad_outputs=torch.ones_like(psi), create_graph=True)[0]
    psi_xx = torch.autograd.grad(psi_x, x, grad_outputs=torch.ones_like(psi_x), create_graph=True)[0]

    # Physics loss (Schrodinger equation)
    hamiltonian = -0.5 * psi_xx
    physics_loss = torch.mean((hamiltonian - analytical_energy(n) * psi)**2)

    # Boundary loss
    boundary_loss = torch.mean(model(torch.tensor([[0.0], [1.0]], device=device)) ** 2)

    # Normalization loss
    norm_loss = torch.abs(torch.mean(psi ** 2) - 1)

    return physics_loss + 10 * boundary_loss + norm_loss
```



CODE OUTPUT



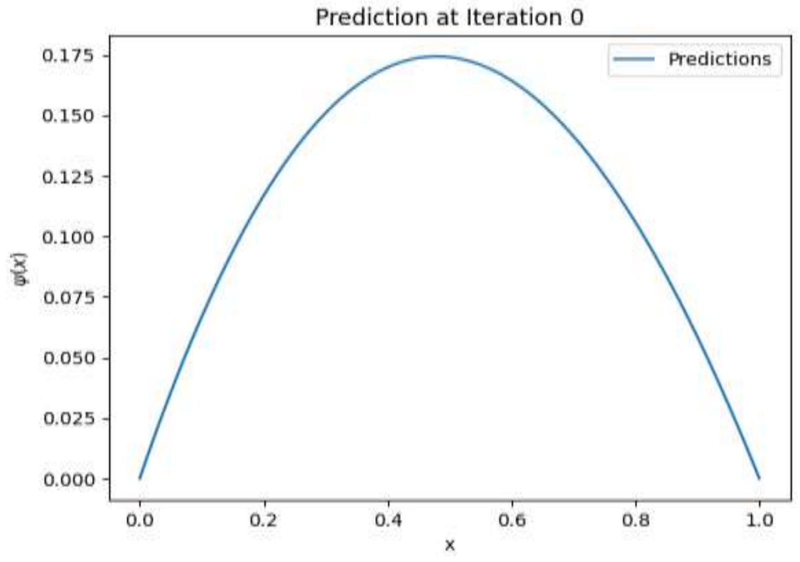

```
PINN Predicted Energy: 4.886312
Analytical Energy (n=1): 4.934802
```

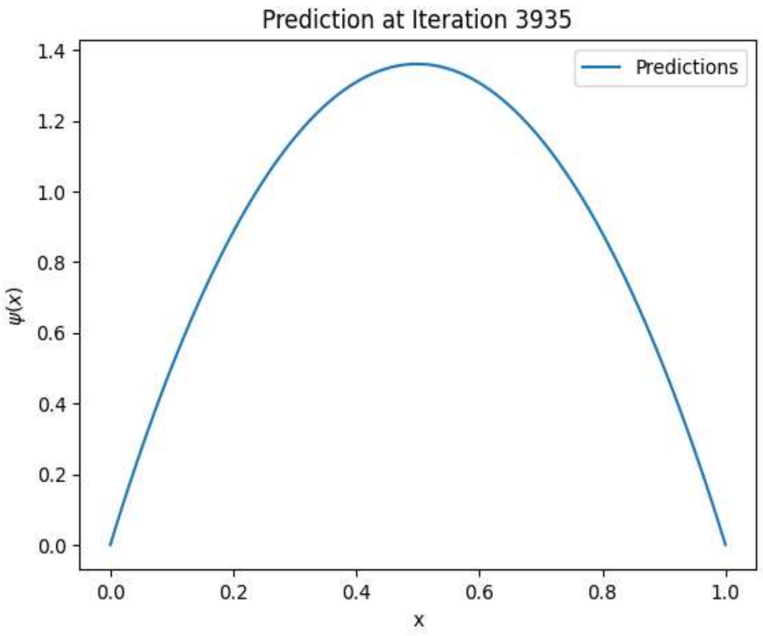
```
Epoch [1000/20000], Loss: 1.00000978
Epoch [2000/20000], Loss: 0.99999261
Epoch [3000/20000], Loss: 0.99998105
Epoch [4000/20000], Loss: 0.99977702
Epoch [5000/20000], Loss: 0.01353911
Epoch [6000/20000], Loss: 0.00839493
Epoch [7000/20000], Loss: 0.00518855
Epoch [8000/20000], Loss: 0.00655821
Epoch [9000/20000], Loss: 0.01330785
Epoch [10000/20000], Loss: 0.00719083
Epoch [11000/20000], Loss: 0.00204477
Epoch [12000/20000], Loss: 0.00465074
Epoch [13000/20000], Loss: 0.00775713
Epoch [14000/20000], Loss: 0.00378387
Epoch [15000/20000], Loss: 0.01048750
Epoch [16000/20000], Loss: 0.00608433
Epoch [17000/20000], Loss: 0.00571223
Epoch [18000/20000], Loss: 0.01009253
Epoch [19000/20000], Loss: 0.00836833
Epoch [20000/20000], Loss: 0.00627762
```

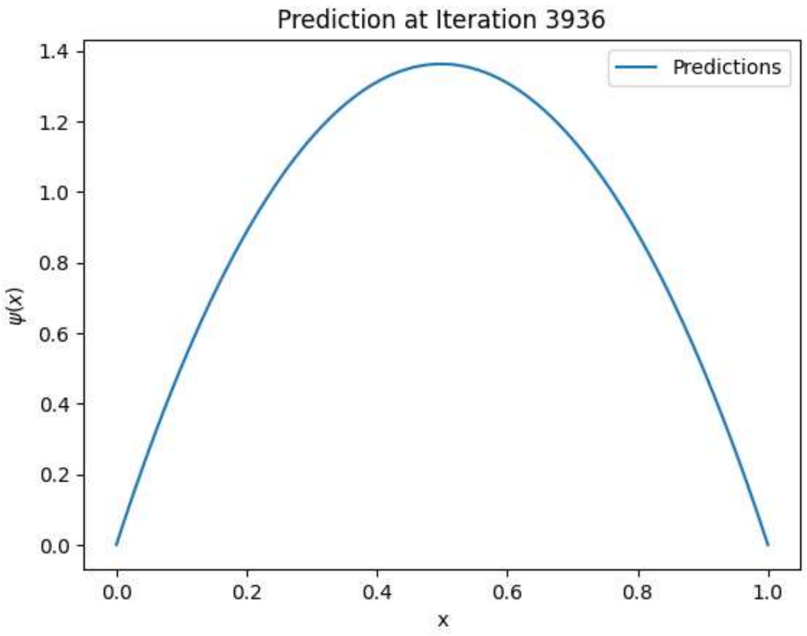
DEVELOPMENT OF THE SOLUTION: PART TWO

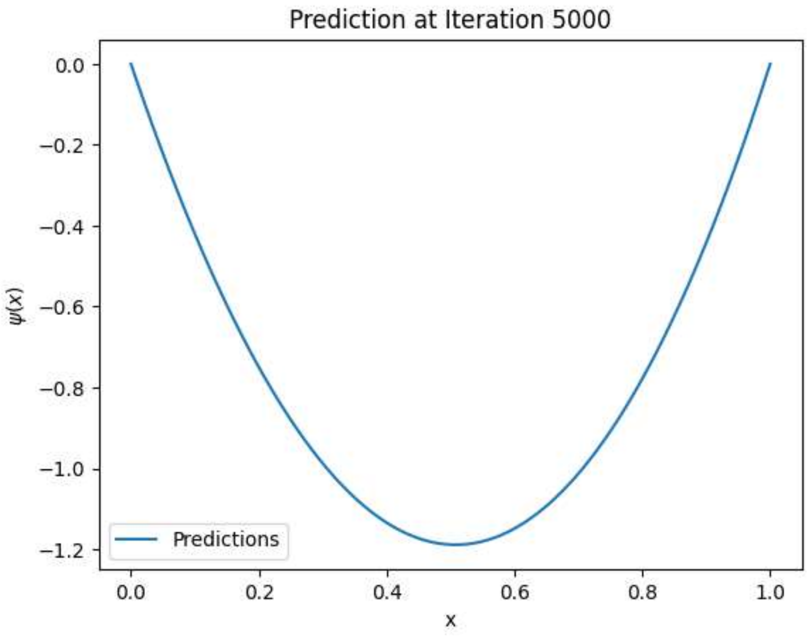
- Energy minimization loss $\Longrightarrow e^{a(E_{PINN}-E_{init})}=0$, with a = 0.8; $E_{init}=\pi^2/2$ when n = 1

```
def energy_minimization_loss(self, E_PINN, E_init=None, a=0.8):
    if E_init is None:
        E_init = torch.zeros_like(E_PINN).cuda().requires_grad_(True)
    energy_loss = (torch.exp(a * (E_PINN - E_init))).mean()
    return energy_loss
```









FINAL CODE

```python
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

```python
# Analitik çözümler
def analytical_wavefunction(x, n):
    return np.sqrt(2) * np.sin(n * np.pi * x)

def analytical_energy(n):
    return (np.pi**2 * n**2) / 2
```

```python
# Özel aktivasyon fonksiyonu
class MySin(torch.nn.Module):
    @staticmethod
    def forward(input):
        return torch.sin(input)
```

```python
# Neural Network sınıfı
class NNs(nn.Module):
    def __init__(self):
        super(NNs, self).__init__()
        self.activation = MySin()
        self.input_En = nn.Linear(1, 1)
        self.hl1 = nn.Linear(2, 50)
        self.hl2 = nn.Linear(50, 50)
        self.output_layer = nn.Linear(50, 1)

    def forward(self, X):
        En_out = self.input_En(torch.ones_like(X))
        hl1_outputs = self.activation(self.hl1(torch.cat((X, En_out), dim=1)))
        hl2_outputs = self.activation(self.hl2(hl1_outputs))
        output = self.output_layer(hl2_outputs)
        return output, En_out
```

PARAMETRIC SOLUTION AND DERIVATIVE FUNCTIONS

```python
def wavefunction(self, x):
    x = torch.tensor(x, requires_grad=True).float().view(-1, 1)
    psi, En = self.forward(x)
    wavefunction = (
        (1 - torch.exp(-1.0 * (x - 0))) *
        (1 - torch.exp(-1.0 * (1 - x))) *
        psi[:, 0:1]
    )
    return wavefunction
```

```python
# Türev fonksiyonları
def dfx(x, f):
    gopts = torch.ones(x.shape, dtype=torch.float)
    return torch.autograd.grad([f], [x], grad_outputs=gopts, create_graph=True)[0]

def d2fx(x, f):
    gopts = torch.ones(x.shape, dtype=torch.float)
    return torch.autograd.grad(dfx(x, f), [x], grad_outputs=gopts, create_graph=True)[0]
```

PYHSICS INFORMED NN

```python
# Kayıp fonksiyonları

def wavefunction_loss(x, wavefunction, E_predicted):
    psi_dx = dfx(x, wavefunction)
    psi_d2x = d2fx(x, wavefunction)
    residue = (psi_d2x / 2 + E_predicted * wavefunction)
    return (residue.pow(2)).mean()


def normalization_loss(wavefunction):
        norm_loss =((-(torch.dot(wavefunction[:,0],wavefunction[:,0]))+200).pow(2))
        return norm_loss
```

```python
class PINNs:
    def __init__(self, X, n=1):
        self.model = NNs()

        # X python listesi veyahut np.array ise float32 tensore dönüştürüyor
        self.X = torch.FloatTensor(X).view(-1, 1).requires_grad_() # !sütun

        self.test_x = np.linspace(0, 1, 200)

        ### Hiperparametreler
        lr=1e-3 # küçükken yavaş - hassas, büyükken hızlı - kararsız
        betas = [0.999, 0.9999] # büyüdükçe yavaş - istikrarlı
        ###

        #
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr, betas=(0.999, 0.9999))

        self.Loss_history = []
        self.Phys_loss_history = []
        self.Norm_loss_history = []
        self.henryjin_parameter_history = []

        self.n = n
        self.E_true = analytical_energy(n)
```

TRANSITION TO OTHER ENERGY LEVELS

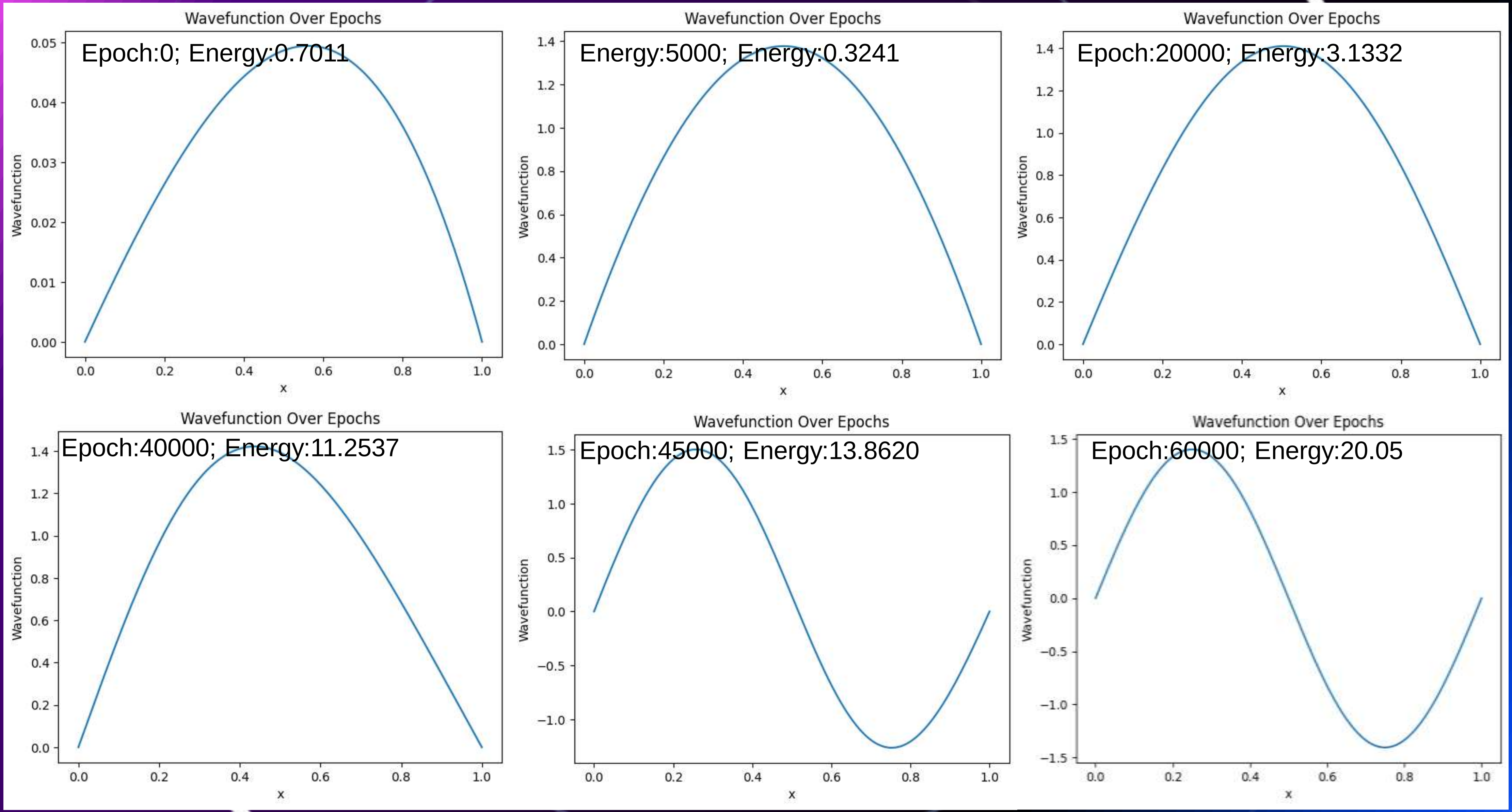
```
    self.henryjin_parameter = -4
def loss_henryjin(self, E_predicted):
    return torch.exp((-E_predicted + self.henryjin_parameter)).pow(2)
    """
    Trick: parametreyi sürekli artırarak tahmin edilen enerjiyi artmaya zorluyoruz
    (exp(10) çok büyük bir sayı, hızlı bir şekilde düşürmeye çalışıyor)
    ki farklı n değerleri için dalga fonksiyonumuzu bulalım
    """


def wavefunction_transform(self,x, wf):
    wf_out = (
        (1 - torch.exp(-1.0 * (x - 0))) *
        (1 - torch.exp(-1.0 * (1 - x))) *
        wf[:, 0:1]
    )
    return wf_out

def last_wf_En(self, x):
    psi,En = self.model.forward(x)
    psi = self.wavefunction_transform(x,psi)
    return psi,En
```

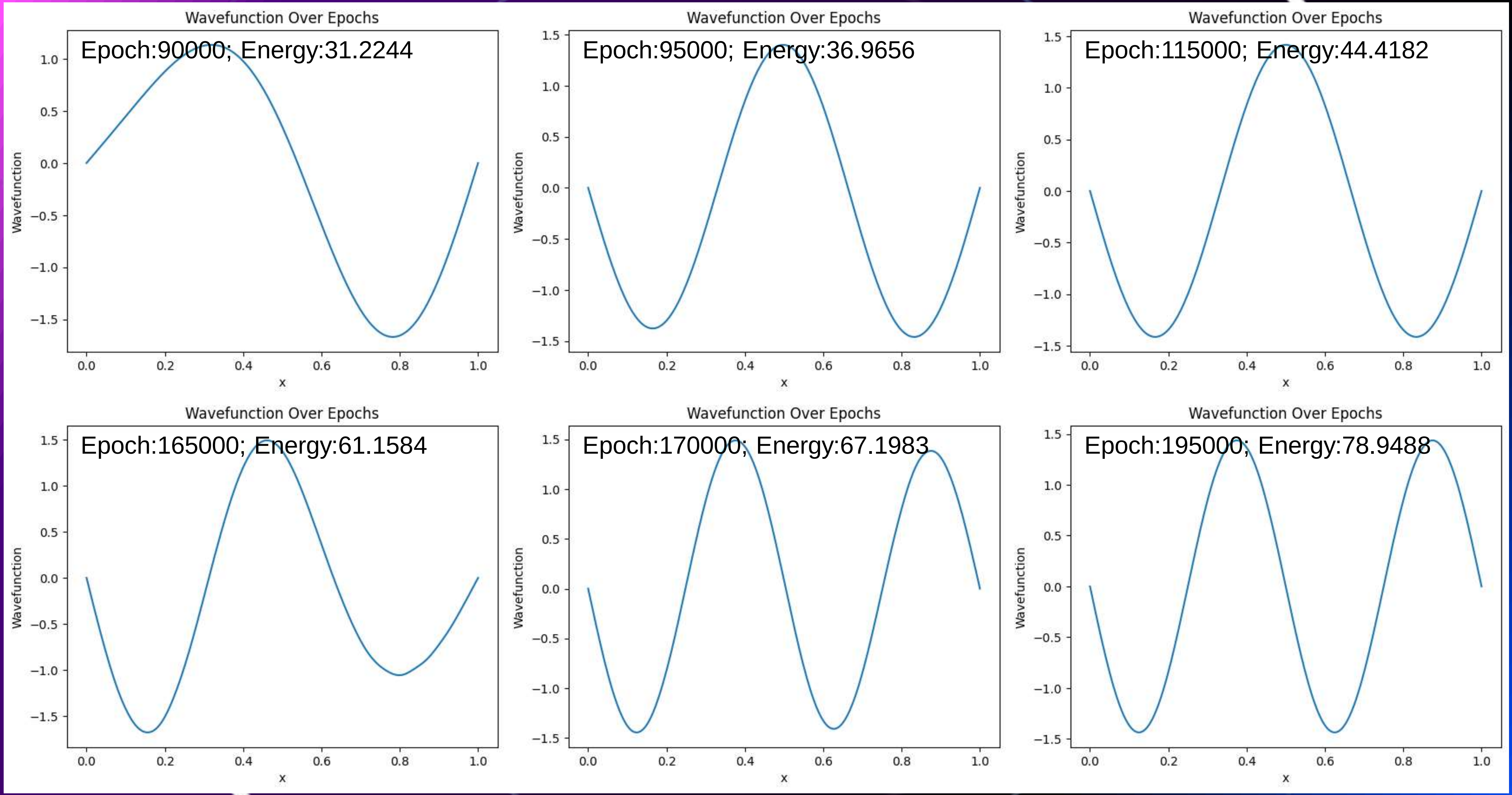
TRAIN AND EVALUATE

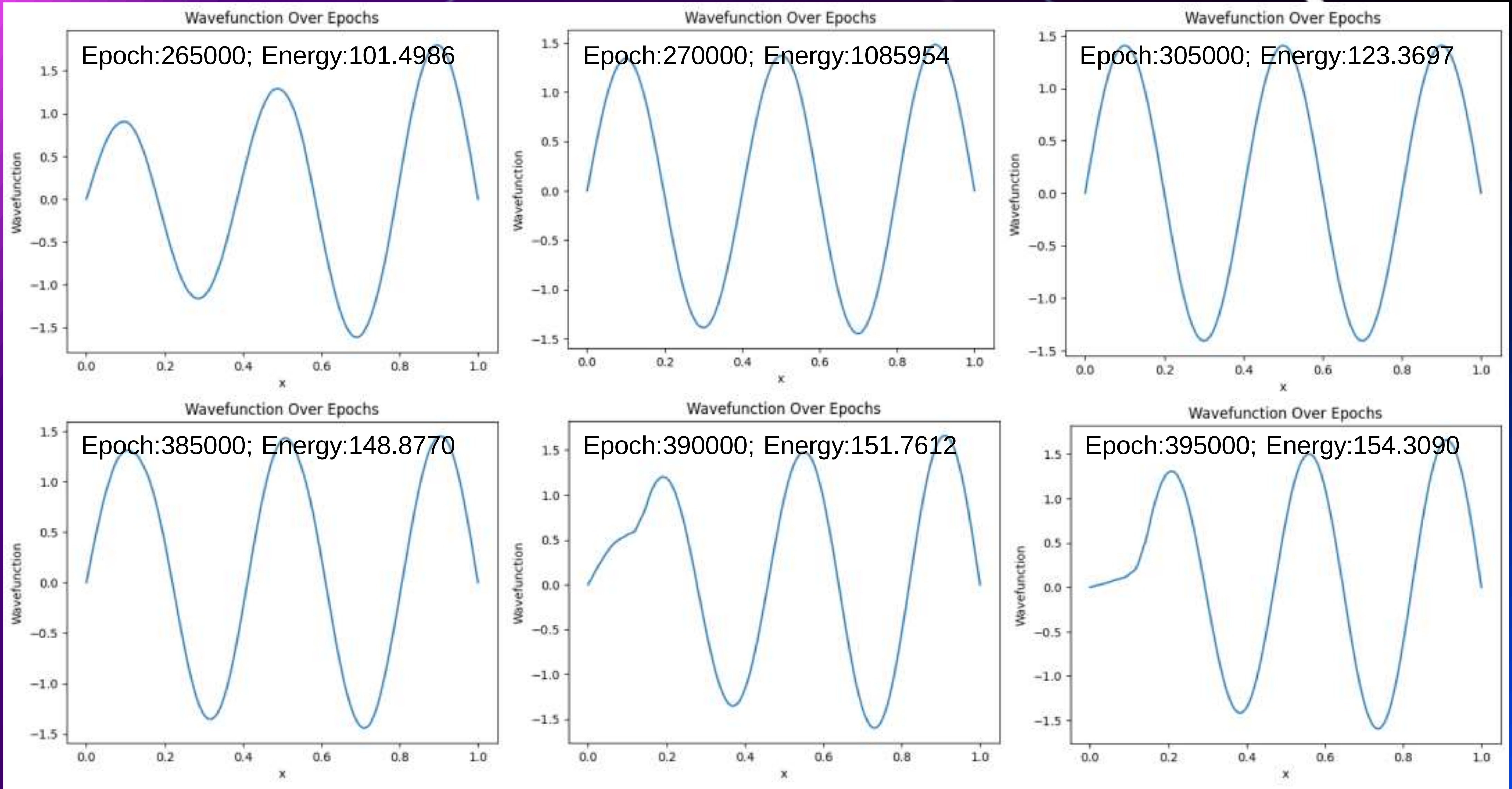
```python
def train(self, epochs=45000):


    for epoch in range(epochs):
        wavefunction, En = self.last_wf_En(self.X)
        phys_loss = wavefunction_loss(self.X, wavefunction, En.mean())
        norm_loss = normalization_loss(wavefunction)


        henryjin_loss = self.loss_henryjin(En.mean())
        total_loss = phys_loss + norm_loss + henryjin_loss


        self.optimizer.zero_grad() #gradyanlarımızı sıfırlıyoruz
        total_loss.backward() # zincir kuralı kullanarak loss fonksiyonlarının gradyanını hesaplıyor
        self.optimizer.step() # parametreleri güncelleniyor..

        # loss çetelesi tutmak için
        self.Loss_history.append(total_loss.item())
        self.Phys_loss_history.append(phys_loss.item())
        self.Norm_loss_history.append(norm_loss.item())

        self.henryjin_parameter_history.append(henryjin_loss.item())

        if epoch % 2500 == 0:
          self.henryjin_parameter += 1
        if epoch % 5000 == 0:
            print(f"Epoch {epoch}, Total Loss: {total_loss.item()}, Schrodinger Loss: {phys_loss.item()}, Norm Loss: {norm_loss.item()}, Energy: {En[0]}, Henryji
            plt.plot(self.test_x, self.evaluate(self.test_x)[0] ) #wavefunction.detach().numpy()
            plt.xlabel("x")
            plt.ylabel("Wavefunction")
            plt.title("Wavefunction Over Epochs")
            plt.show()
```

```python
def evaluate(self,x_test_tensor):

    #liste veya arrayse diye
    x_test = torch.FloatTensor(x_test_tensor).view(-1, 1) #!sütun

    # değerlendirme aşamasında parametse güncellemeye gerek yok
    with torch.no_grad(): # bellek kullanımı azaltılıyor, hızlı çalışıyor
       psi,En = self.last_wf_En(x_test)

    return psi,En
```

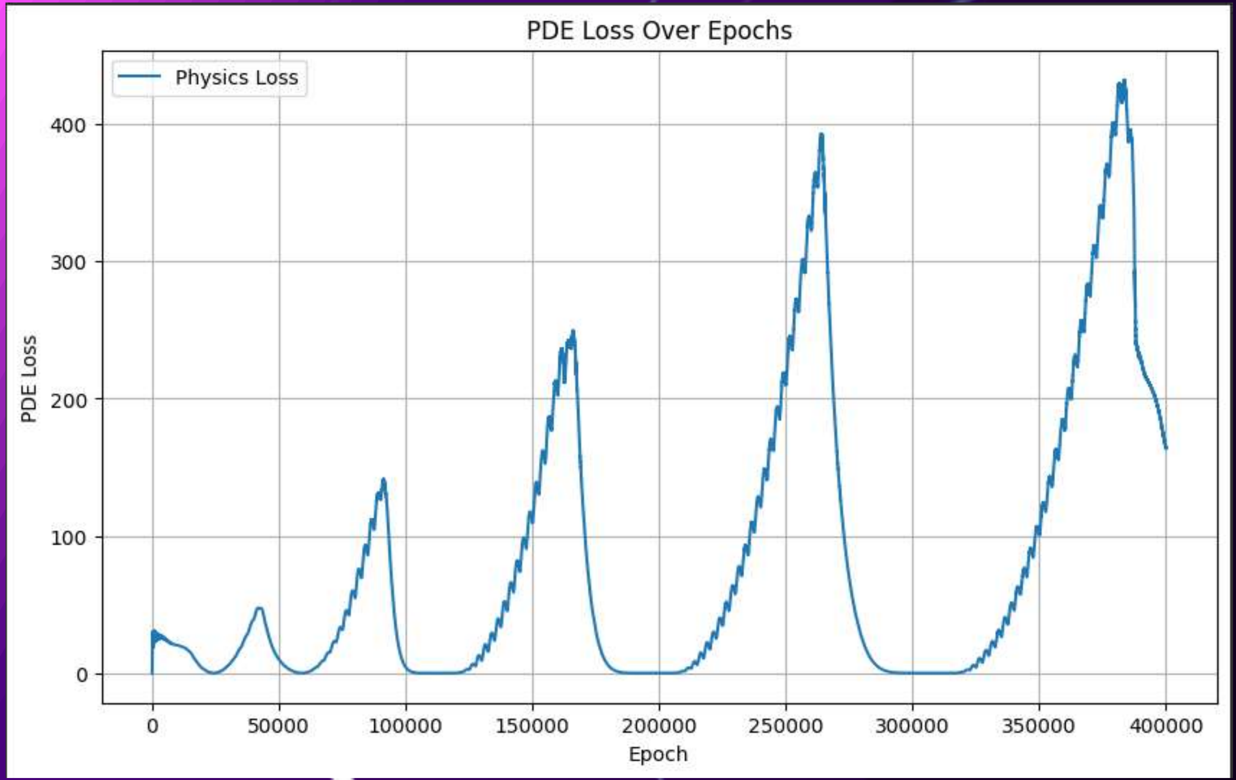
PSI – ENERGY VALUES

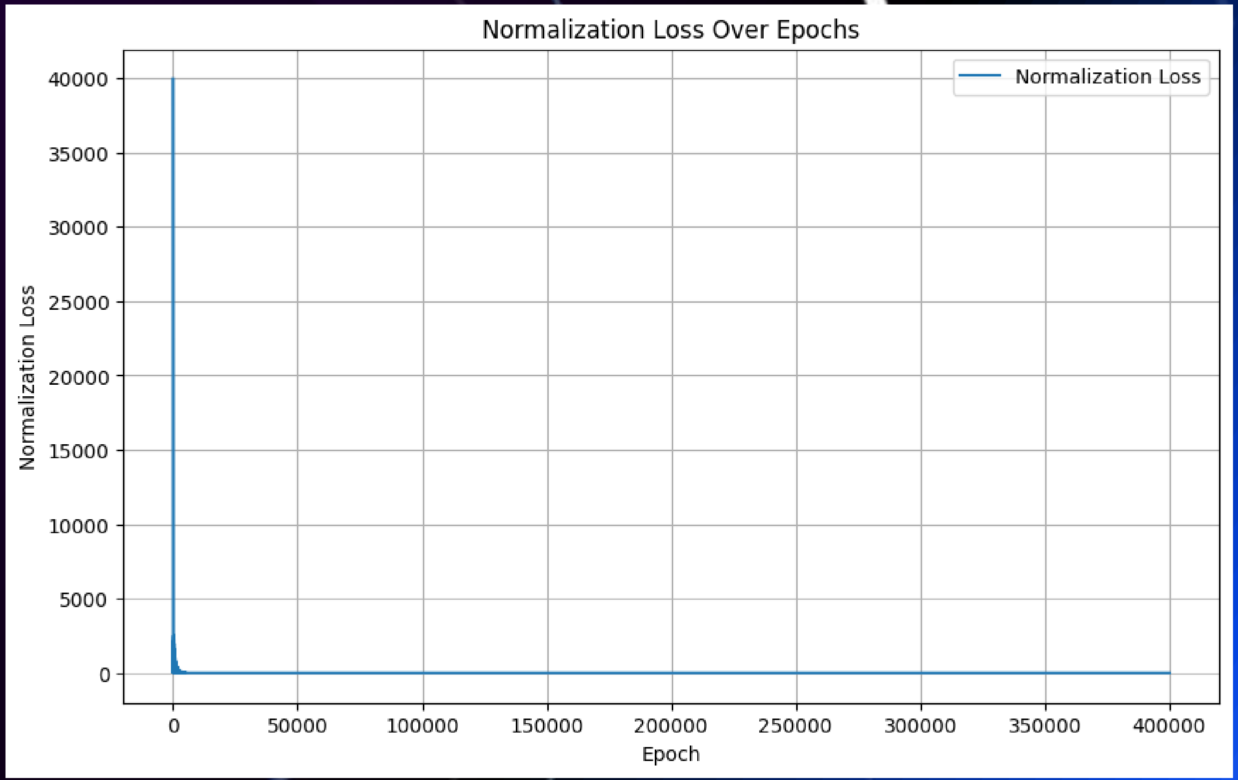


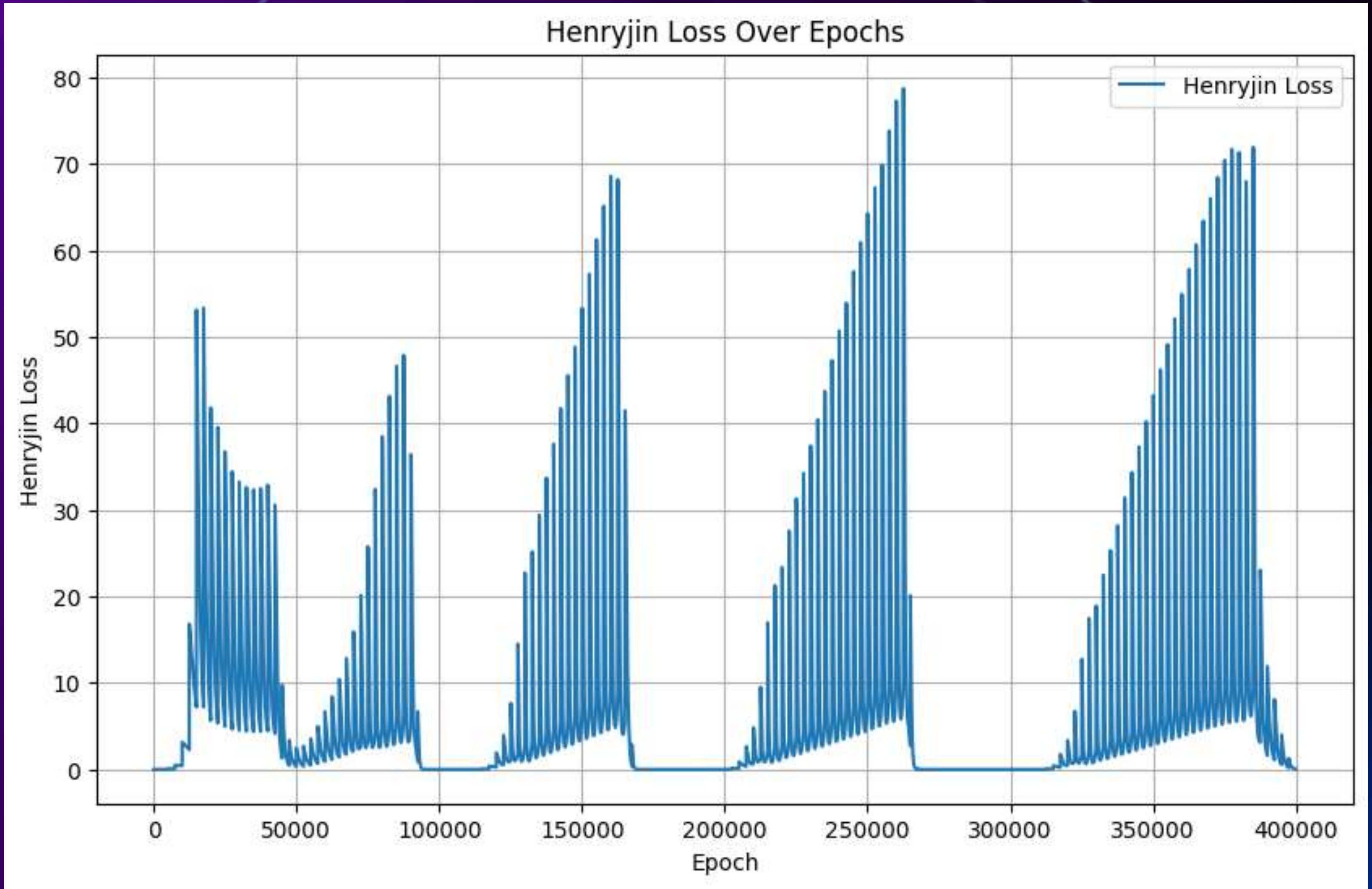

Wavefunction Over Epochs
Epoch:90000; Energy:31.2244
Wavefunction Over Epochs
Epoch:95000; Energy:36.9656
Wavefunction Over Epochs
Epoch:115000; Energy:44.4182
Wavefunction Over Epochs
Epoch:165000; Energy:61.1584
Wavefunction Over Epochs
Epoch:170000; Energy:67.1983
Wavefunction Over Epochs
Epoch:195000; Energy:78.9488
Wavefunction
x


Wavefunction Over Epochs
Epoch:265000; Energy:101.4986
Wavefunction Over Epochs
Epoch:270000; Energy:1085954
Wavefunction Over Epochs
Epoch:305000; Energy:123.3697
Wavefunction Over Epochs
Epoch:385000; Energy:148.8770
Wavefunction Over Epochs
Epoch:390000; Energy:151.7612
Wavefunction Over Epochs
Epoch:395000; Energy:154.3090
Wavefunction
x

LOSS PLOTS


PDE Loss Over Epochs
Physics Loss
PDE Loss
0
100
200
300
400
0
50000
100000
150000
200000
250000
300000
350000
400000
Epoch


Normalization Loss Over Epochs
Normalization Loss
Normalization Loss
0
5000
10000
15000
20000
25000
30000
35000
40000
0
50000
100000
150000
200000
250000
300000
350000
400000
Epoch


Henryjin Loss Over Epochs
Henryjin Loss
80
70
60
50
40
30
20
10
0
0
50000
100000
150000
200000
250000
300000
350000
400000
Epoch
Henryjin Loss

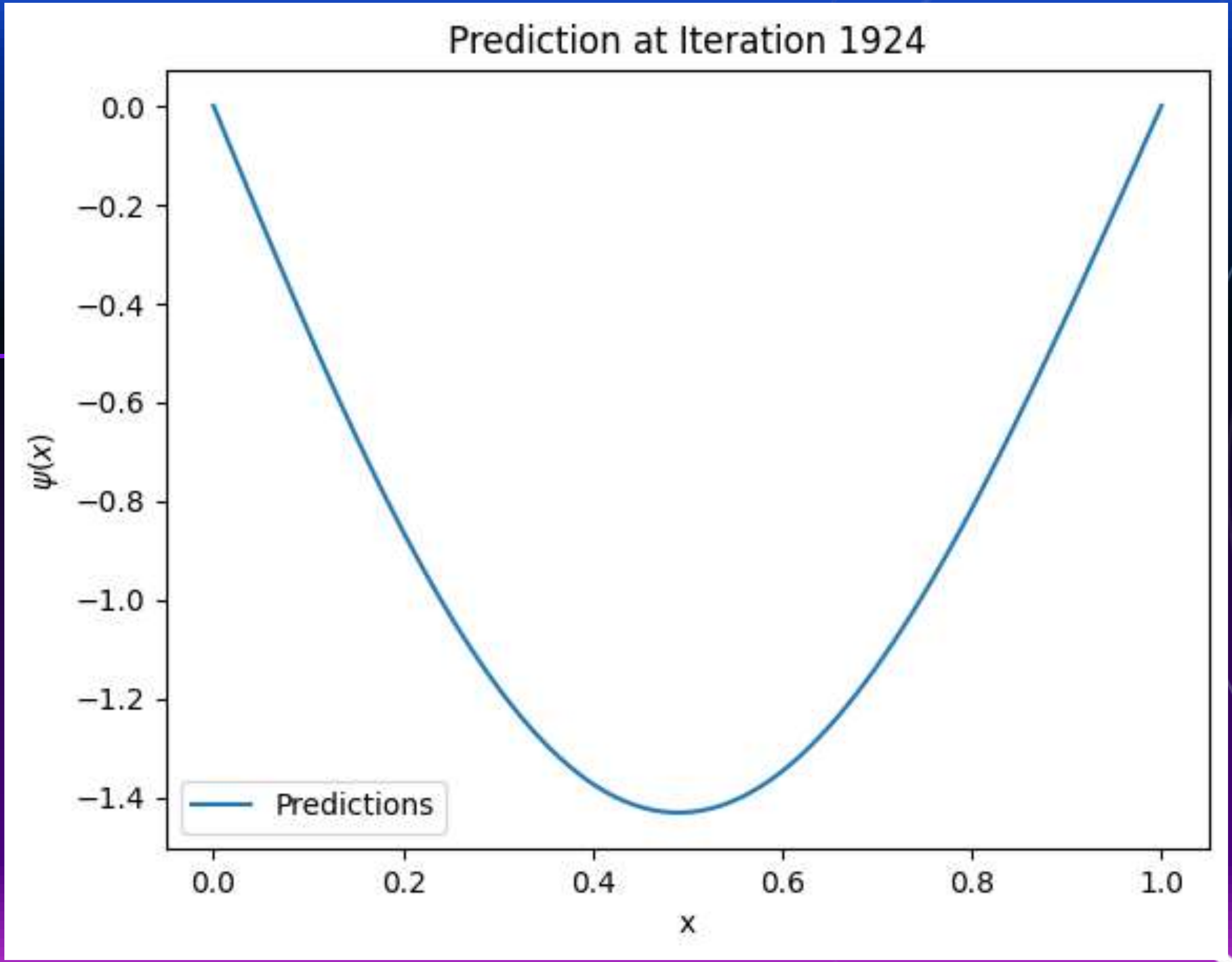
DIFFERENT LOSS FUNCTIONS AND ACCELERATION

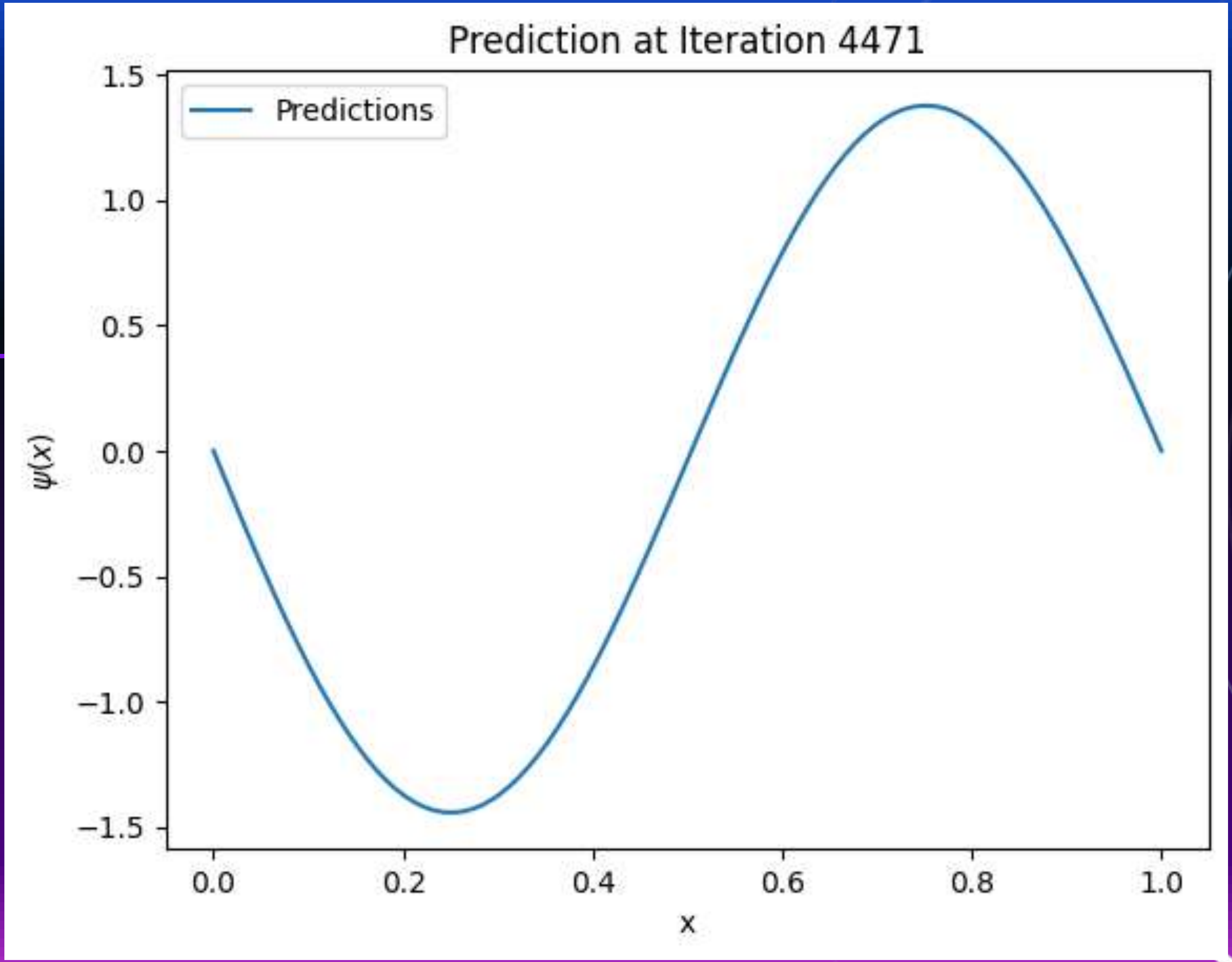
Loss Functions

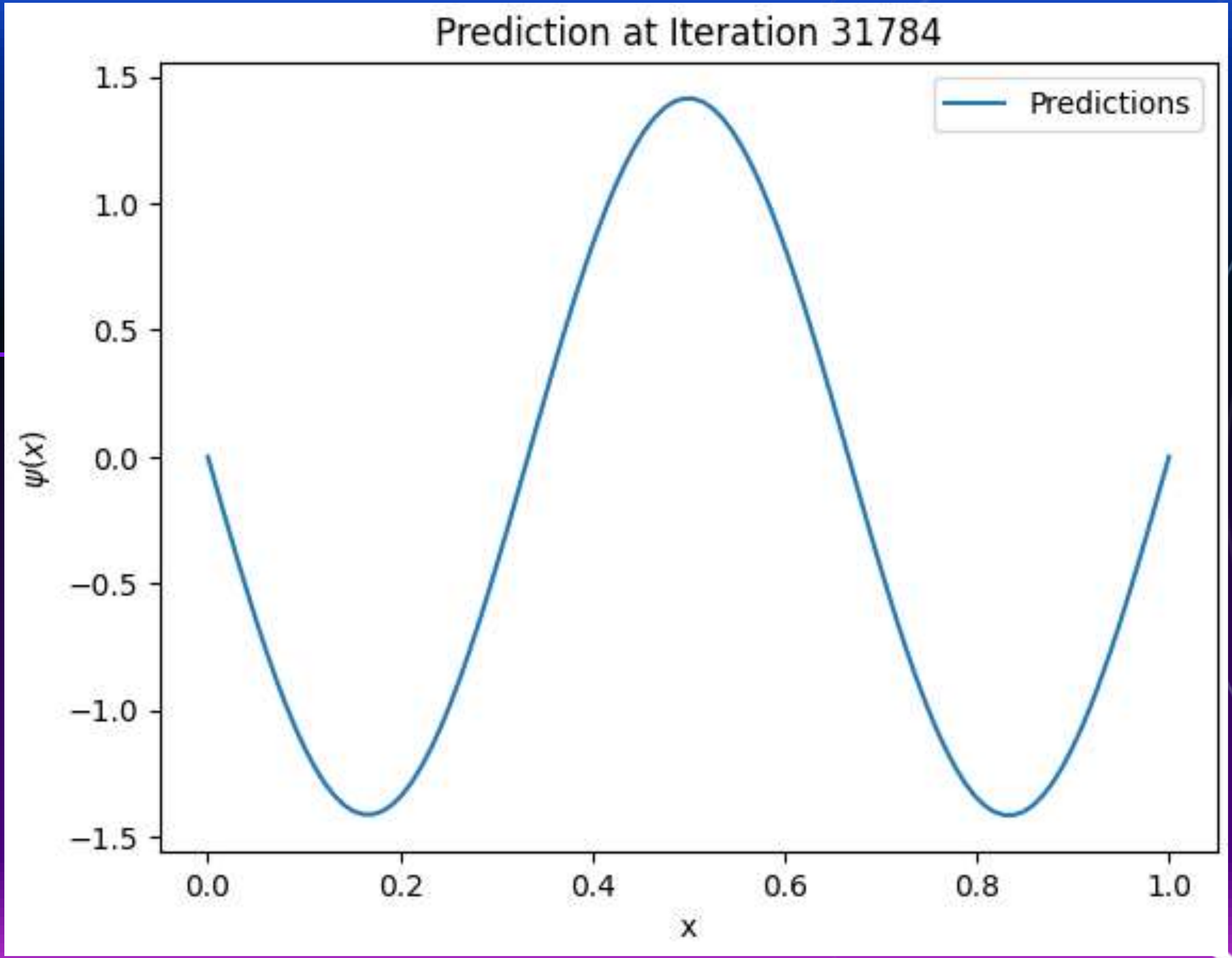
- Integral loss $\Longrightarrow \frac{\partial v(x')}{\partial x'}|_{x'=x} = |\psi(x)|^2$
- Orthogonality loss $\Longrightarrow \sum_i \langle \psi(x)|\psi_i(x)\rangle = 0$

```
if 0.99 <= integral <= 1.1 and 4.92 < En[0].item() < 4.94:

    normalization_weight = 0.001
    eigen_weight = 1000
    self.walle += 2
    print(f"Iteration {i}, Losses: {loss_components}, New_Normal_Weight: {normalization_weight}, New_Eigen_Weight: {eigen_weight}")
    print(f"|psi|^2: {integral}")
    print(f"Iteration {i}, Loss: {total_loss}, Energy: {En[0].item()}, Normal_Weight: {normalization_weight}, Eigen_Weight: {eigen_weight}")
    plt.plot(self.x_test.cpu().detach().numpy(), self.predict(self.x_test)[0], label='Predictions')
    plt.xlabel("x")
    plt.ylabel("$\psi(x)$")
    plt.title(f"Prediction at Iteration {i}")
    plt.legend()
    plt.show()

elif integral <= 0.005:

    normalization_weight = 100
    eigen_weight = 1
```

```
def integral_loss(self, x, psi):

    psi_squared = torch.abs(psi) ** 2

    dx = float(x[1] - x[0])

    v_x = torch.cumsum((psi_squared[:-1] + psi_squared[1:]) / 2, dim=0) * dx

    v_x = torch.cat((torch.zeros_like(v_x[:1]), v_x))

    derivative_v_x = dfx(x, v_x)

    integral_loss = ((derivative_v_x - psi_squared) ** 2).mean()

    return integral_loss

def orthogonality_loss(self, x, psi, previous_psi=0):


    epsilon = 1e-8
    previous_psi = self.nn.forward(x)[0]

    # Min-Max normalizasyonu
    psi_min, psi_max = psi.min(), psi.max()
    previous_psi_min, previous_psi_max = previous_psi.min(), previous_psi.max()

    psi_normalized = (psi - psi_min) / (psi_max - psi_min + epsilon)
    previous_psi_normalized = (previous_psi - previous_psi_min) / (previous_psi_max - previous_psi_min + epsilon)

    # Dot product and orthogonality loss
    dot_product = torch.sum(psi_normalized * previous_psi_normalized, dim=0)
    orthagonal_loss = torch.abs(dot_product).pow(2).mean()

    return orthagonal_loss
```


Prediction at Iteration 1924
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
ψ(x)
0.0
0.2
0.4
0.6
0.8
1.0
x
Predictions

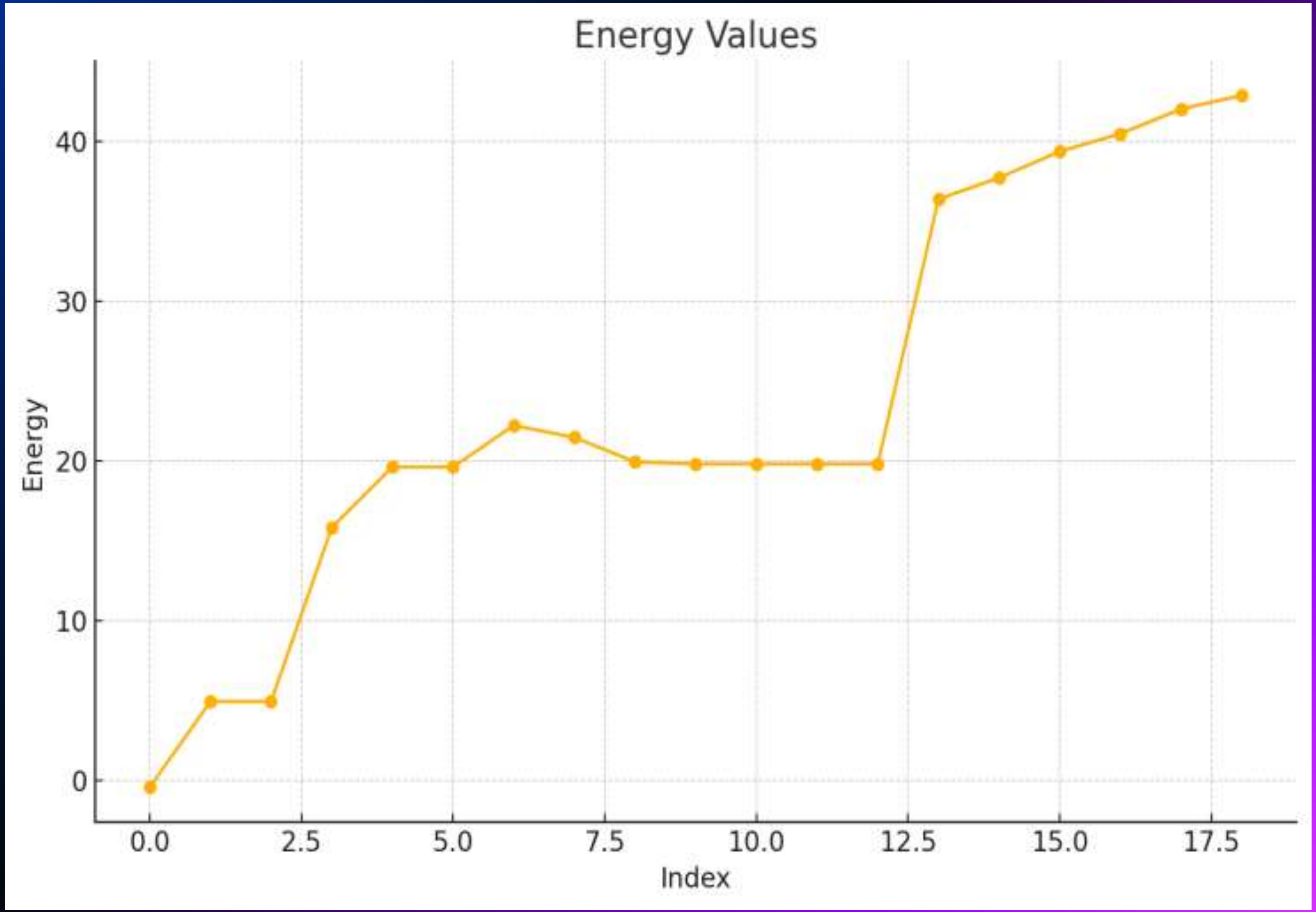

Prediction at Iteration 4471
Predictions
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
ψ(x)
0.0
0.2
0.4
0.6
0.8
1.0
x



Prediction at Iteration 31784
Predictions
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
$\psi(x)$
0.0
0.2
0.4
0.6
0.8
1.0
x

RESULT

The energy values corresponding to the n=1, n=2, and n=3 energy levels were calculated using approximately 32,000 iterations.



BIBLIOGRAPHY

Jin, Henry, Marios Mattheakis, and Pavlos Protopapas. "Physics-Informed Neural Networks for Quantum Eigenvalue Problems." arXiv.org, February 24, 2022. https://arxiv.org/abs/2203.00451.

Jin, Henry, Marios Mattheakis, and Pavlos Protopapas. "Unsupervised Neural Networks for Quantum Eigenvalue Problems." arXiv.org, October 10, 2020. https://arxiv.org/abs/2010.05075.

Brevi, Lorenzo, Antonio Mandarino, and Enrico Prati. "A Tutorial on the Use of Physics-Informed Neural Networks to Compute the Spectrum of Quantum Systems." arXiv.org, September 11, 2024. http://arxiv.org/abs/2407.20669v2.

Moseley, Ben. "Harmonic-Oscillator-Pinn." GitHub, 2021. https://github.com/benmoseley/harmonic-oscillator-pinn.

Sushmit, Mushrafi Munim. "Physics-Informed Neural Networks (PINNs) for Solving Physical Systems." GitHub, February 17, 2024. https://github.com/mushrafi88/Physics-Informed-Neural-Networks-for-Quantum-Dynamics.

THANK YOU

github.com/tosunm20/PINNs_Quantum_Potential_We