Solving Green's function using neural networks to model electrostatic correlations

Nikhil Agrawal *CHEM247 Spring 2022: Final Project proposal*

The study of electrical double layer resulting from a charged surface is at the heart of colloidal and interfacial sciences. However, the standard mean-field PB theory fails to describe systems with large surface-charge density, high counter-ion valency and high ion concentration because it ignores the electrostatic correlation. To model ion-ion correlations phenomena one has to go beyond mean-field level and include fluctuations in the theory. For this a Gaussian Renormalized Fluctuation theory was given by Zhen-Gang Wang at Caltech. One of the key equations in this theory for symmetric planar systems is

$$\mathcal{L}G(z,z) = -\frac{\partial^2 G(s,z,z')}{\partial z^2} + s^2(z)G(s,z,z') = \frac{\delta(z,z')}{\epsilon} \tag{1}$$

where ϵ is the dielectric constant of the system, s is a smooth function and G is the Green's function we want to solve for. The aim of this project will be to use artificial neural networks (ANN) to approximate the function G in [0,L]. The following boundary conditions (BC) for regions $z < 0, z' \geq 0$ and $z > L, z' \leq L$ respectively will be used:

$$\frac{\partial G(s,z,z')}{\partial z'} = s(z=0)G(s,z,z') \tag{2}$$

$$\frac{\partial G(s,z,z')}{\partial z'} = -s(z=L)G(s,z,z') \tag{3}$$

Existing finite difference methods to solve for the above problem become very complex and inefficient for the three dimensional analog of the Equation 1. Solving these three-dimensional PDE is very important if we want to solve for ion-ion correlations in asymmetric systems. Through this project I aim to see the capabilities of the machine learning framework to solve for such two point correlation functions.

I have decided to divide the project into two parts. The first half will involve solving Equation 1 for the case when $s(z) = \text{constant}$. In this case there is an analytical solution to the PDE given by

$$G(s,z,z') = \frac{e^{-s|z-z'|}}{2s\epsilon} \tag{4}$$

Using this analytical solution as my labelled data I will use a supervised learning approach on a fully connected neural network to finalize the architecture of the network. I will use a suitable random number generator to create a big enough sample data set for (z, z'). Let's call the neural network for this case to be $NN_1(z,z)$ which satisfies $NN_1(z,z) = G(z, z\text{''}, s(z) = constant)$.

In the second half of the project using the architecture of NN_1, I will try to train a new neural network, NN_2, which will replicate the function $G(z, z, s(z))$ when $s(z)$ is not a constant. I am planning to use a simple linear or tanh type function for $s(z)$. To train the network I we will use Equations 1,2 and 3 to write a mean-square error type loss function. The derivataives of G or NN_2 with respect to z can be calculated using automatic differentiation tools of Pytorch. The singularity associated with the dirac delta function can be approximated using a continuous Gaussian density function with variance tending to zero. Fortunately, there has been some work on a similar problem before from which I plan to borrow some ideas (Teng et al. 2021, arXiv:2105.11045v1). Another piece of literature on which I will rely upon is the pioneering work on Physics Inspired Neural Networks by M. Raissi et al. 2019.

Step 1: Supervised Learning to Optimize the architecture

In this notebook I have successfully finished the step 1 of the proposed project. For the cases of constant s=1 and s=2 I have been able to optimize a single artificial neural network architecture which can predict the solution to greens function upto an accuracy of atleast 98%. The labelled data I used for supervised learning was the analytical solution given in Equation 4.

Neural Network Trainer Code

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from torch.optim import SGD, Adam
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
from sklearn.model_selection import train_test_split
from functools import wraps
from time import time
```

In [2]:
```
def timing(f):
    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        print('func:%r  took: %2.4f sec' % (f.__name__,  te-ts))
        return result
    return wrap

def create_chunks(complete_list, chunk_size=None, num_chunks=None):

    chunks = []
    if num_chunks is None:
        num_chunks = math.ceil(len(complete_list) / chunk_size)
    elif chunk_size is None:
        chunk_size = math.ceil(len(complete_list) / num_chunks)
    for i in range(num_chunks):
        chunks.append(complete_list[i * chunk_size: (i + 1) * chunk_s
ize])
    return chunks

def error_analy(x,y):
    error = np.sum(np.true_divide(abs(x-y),y))*100/(len(y))
    return error
```

In [3]:
```
class Trainer_analy():
    def __init__(self, model, error_fn,loss_fn, learning_rate, epoch,
batch_size):

        self.model = model
        self.optimizer = Adam(model.parameters(),learning_rate,weight
_decay=1e-5)
        self.epoch = epoch
        self.batch_size = batch_size

    @timing
    def train(self, error_fn,loss_fn,inputs, outputs, val_inputs, val
_outputs,early_stop,l2,silent=False):
        ### convert data to tensor of correct shape and type here ###
        inputs = torch.FloatTensor(inputs)
        val_inputs = torch.FloatTensor(val_inputs)
        losses = []
        errors = []
        val_losses = []
        val_errors = []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        l2_lambda=1e-5
        for n_epoch in tqdm(range(self.epoch), leave=False):
            self.model.train()
            batch_indices = list(range(inputs.shape[0]))
            random.shuffle(batch_indices)
            batch_indices = create_chunks(batch_indices, chunk_size=s
elf.batch_size)
            epoch_loss = 0
            epoch_error = 0
            for batch in batch_indices:
                batch_importance = len(batch) / len(outputs)
                batch_input = inputs[batch]
                batch_output = outputs[batch]
                ### make prediction and compute loss with loss functi
on of your choice on this batch ###
                batch_predictions = self.model.forward(batch_input).s
queeze(-1)
                #print(batch_predictions.shape)
                loss = loss_fn(batch_predictions,torch.FloatTensor(ba
tch_output))
                if l2:
                    ### Compute the loss with L2 regularization ###
                    l2_norm = sum([p.pow(2.0).sum() for p in self.mod
el.parameters()])
                    loss = loss + l2_lambda * l2_norm
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                ### Compute epoch_loss and epoch_error
                epoch_loss += loss.detach().item()*batch_importance
                error = error_fn(batch_predictions.detach().numpy(),b
atch_output)
                epoch_error += error*batch_importance
```

```
            val_loss, val_error = self.evaluate(val_inputs, val_outpu
ts, error_fn,print_error=False)
            if n_epoch % 25 ==0 and not silent:
                print("Epoch %d/%d - Loss: %.3f - error: %.3f" % (n_e
poch + 1, self.epoch, epoch_loss, epoch_error))
                print("              Val_loss: %.3f - Val_error: %.3f
" % (val_loss, val_error))
            losses.append(epoch_loss)
            errors.append(epoch_error)
            val_losses.append(val_loss)
            val_errors.append(val_error)
            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()
        if early_stop:
            self.model.load_state_dict(weights)

        return self.model,{"losses": losses, "errors": errors, "val_l
osses": val_losses, "val_errors": val_errors}

    def evaluate(self, inputs, outputs,error_fn,print_error=True):
        loss_fn = nn.MSELoss()
        inputs = torch.FloatTensor(inputs)
        predictions = self.model.forward(inputs).squeeze(-1)
        losses = loss_fn(predictions,torch.FloatTensor(outputs)).item
()
        error = np.sum(error_fn(predictions.detach().numpy(),outputs
))
        if print_error:
            print("Error: %.3f" % error)
        return losses, error
```

Training and validation Code

In [4]:
```
from sklearn.model_selection import train_test_split,KFold

def train_and_val(model,error_fn,loss_fn,train_X,train_y,val_X,val_y,
epochs,batch_size,lr,early_stop,l2,pde,draw_curve=True):

    if pde==False:
        ann_trainer = Trainer_analy(model,error_fn,loss_fn,lr, epochs
, batch_size)
    else:
        ann_trainer = Trainer_pde(model,error_fn,loss_fn,lr, epochs,
batch_size)

    model,ledger  = ann_trainer.train(error_fn,loss_fn,train_X,train_
y,val_X,val_y,early_stop,l2)
    val_array = ledger['val_losses']
    train_array = ledger['losses']
    val_error= ledger['val_errors']
    train_error = ledger['errors']
    train_error_all=[]
    if draw_curve:
        plt.figure()
        plt.plot(np.arange(len(val_array))+1,val_array,label='Validat
ion loss')
        plt.plot(np.arange(len(train_array))+1,train_array,label='Tra
ining loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

    if draw_curve:
        plt.figure()
        plt.plot(np.arange(len(val_error))+1,val_error,label='Validat
ion Error')
        plt.plot(np.arange(len(train_error))+1,train_error,label='Tra
ining Error')
        plt.xlabel('Epochs')
        plt.ylabel('Error')
        plt.legend()

    if early_stop:
            report_idx= np.argmin(ledger["val_losses"])
    else:
            report_idx=-1
    ### Recover the model weight ###
    weights = model.parameters()

    return model,weights
```

Code for evaluating accuracy/error in test data set

```
In [5]: def evaluate_ind(model, inputs, outputs,error_fn,print_error=True):
                loss_fn = nn.MSELoss()
                inputs = torch.FloatTensor(inputs)
                predictions = model.forward(inputs).squeeze(-1)
                error = error_fn(predictions.detach().numpy(),outputs)
                if print_error:
                    print("Testing set Precentage Error: %.3f" % error)
                return error
```

Cross-fold validation code

```
In [6]: def Kfold(k,model_func,error_fn, loss_fn,Xs,ys,test_X, test_y,epochs,
batch_size,lr,early_stop,l2,pde):
    # The total number of examples for training the network

    total_num = len(Xs)
    # Built in K-fold function in Sci-Kit Learn
    kf=KFold(n_splits=k,shuffle=True)
    # record error for each model
    train_error_all=[]
    test_error_all=[]

    for train_selector,test_selector in kf.split(range(total_num)):
        ### Decide training examples and testing examples for this fo
ld ###
        train_Xs= Xs[train_selector]
        test_Xs= Xs[test_selector]
        train_ys= ys[train_selector]
        test_ys= ys[test_selector]
        model = model_func()
        print(f" parameters:", sum([len(item.flatten()) for item in m
odel.parameters()]))
        model, weights =train_and_val(model,error_fn,loss_fn,train_Xs
,train_ys,test_Xs,test_ys,epochs,batch_size,lr,early_stop,l2,pde,draw
_curve = True)

        #errors = evaluate_ind(model,test_X, test_y, error_fn,print_e
rror=True)

    return model,weights
```

Optimized architecture of the neural network.

In [7]:
```
from torch import nn
import torch


class ann(nn.Module):
    def __init__(self):
        super(ann, self).__init__()
        a = 100
        inp = 3
        self.Linear = nn.ModuleList([nn.Linear(inp,a),nn.Linear(a,a)
                                     ,nn.Linear(a,a),nn.Linear(a,a)
                                     ,nn.Linear(a,100),nn.Linear(100,
1)])
        self.activation = nn.ModuleList([nn.Tanh()])
    def forward(self, x):
        for i in range(len(self.Linear)-1):
            x = self.activation[0](self.Linear[i](x))
        x = self.Linear[-1](x)
        return x

print(ann())
```

```
ann(
  (Linear): ModuleList(
    (0): Linear(in_features=3, out_features=100, bias=True)
    (1): Linear(in_features=100, out_features=100, bias=True)
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): Linear(in_features=100, out_features=100, bias=True)
    (4): Linear(in_features=100, out_features=100, bias=True)
    (5): Linear(in_features=100, out_features=1, bias=True)
  )
  (activation): ModuleList(
    (0): Tanh()
  )
)
```

Labelled data generation function from analytical solution

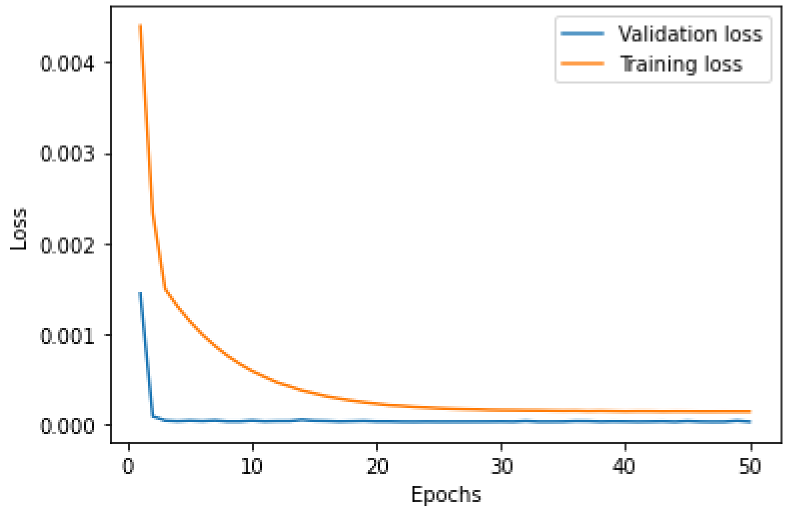
In [8]:
```
def data_generator(N,s):
    train_X = np.zeros((N,3))
    train_y = np.zeros((N))


    for i in range(N):
        train_X[i,0] = s#0*np.random.rand() + 0.01 #s
        train_X[i,1] = 2*np.random.rand() - 1.0 #x
        train_X[i,2] = 2*np.random.rand() - 1.0 #x'
        train_y[i] = np.exp(-train_X[i,0]*abs(train_X[i,1]-train_X[i,
2]))/(2*train_X[i,0])

    return train_X,train_y
```

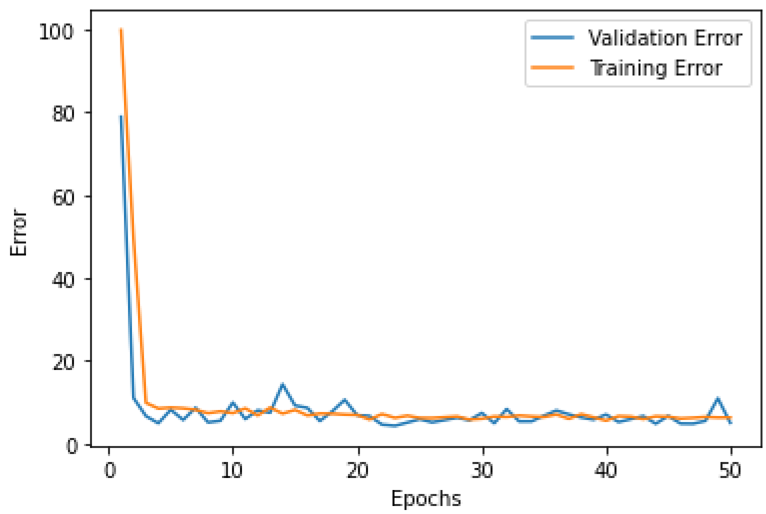
Neural Network training and testing

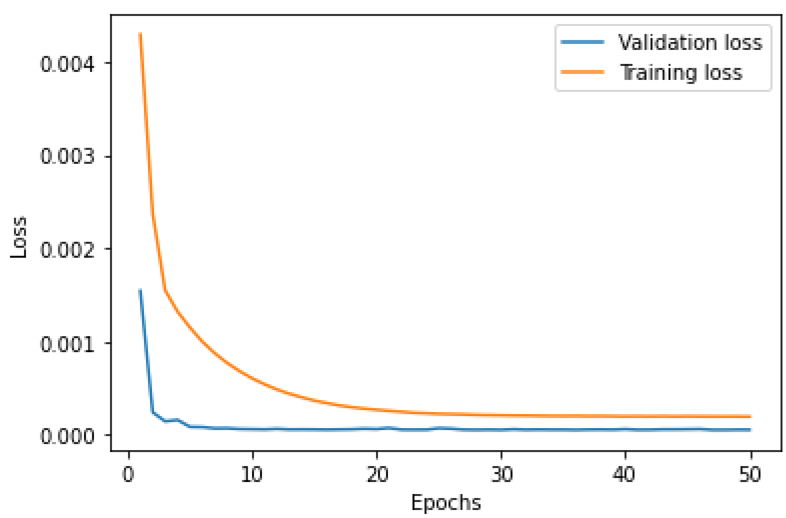
In [9]:
```
train_X,train_y = data_generator(100000,2)
test_X,test_y = data_generator(1000,2)
model, weights = Kfold(5,ann,error_analy,nn.MSELoss(),train_X,train_y
,test_X,test_y,50,500,lr=1e-3,early_stop=True,l2=True,pde=False)
```

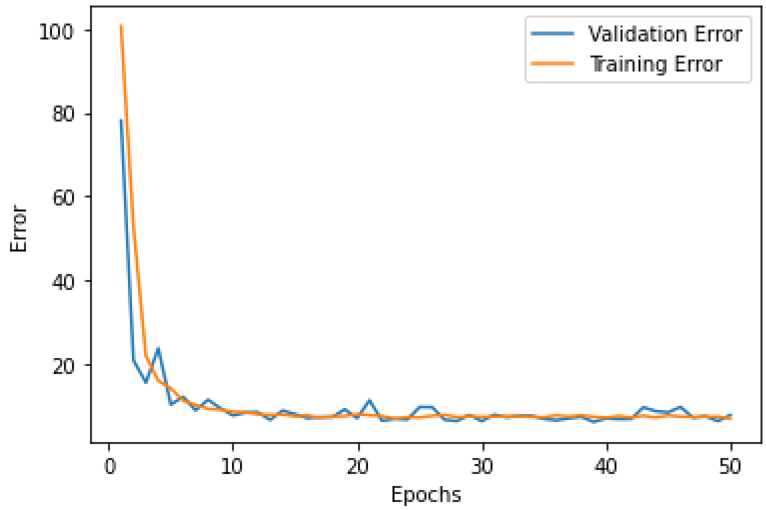
```
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.92it/s]

Epoch 1/50 - Loss: 0.004 - error: 99.834
              Val_loss: 0.001 - Val_error: 78.878

 52%|█████▏    | 26/50 [00:13<00:12,  1.99it/s]

Epoch 26/50 - Loss: 0.000 - error: 6.222
              Val_loss: 0.000 - Val_error: 5.189


func:'train'  took: 24.9730 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.94it/s]

Epoch 1/50 - Loss: 0.004 - error: 100.720
              Val_loss: 0.002 - Val_error: 78.134

 52%|█████▏    | 26/50 [00:13<00:12,  1.99it/s]

Epoch 26/50 - Loss: 0.000 - error: 7.706
              Val_loss: 0.000 - Val_error: 9.775


func:'train'  took: 26.5428 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:24,  1.97it/s]

Epoch 1/50 - Loss: 0.004 - error: 101.821
              Val_loss: 0.001 - Val_error: 71.541

 52%|█████▏    | 26/50 [00:20<00:18,  1.32it/s]

Epoch 26/50 - Loss: 0.000 - error: 7.720
              Val_loss: 0.000 - Val_error: 9.246


func:'train'  took: 34.1506 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.95it/s]

Epoch 1/50 - Loss: 0.005 - error: 109.574
              Val_loss: 0.001 - Val_error: 72.238

 52%|█████▏    | 26/50 [00:13<00:12,  1.96it/s]

Epoch 26/50 - Loss: 0.000 - error: 6.920
              Val_loss: 0.000 - Val_error: 9.168


func:'train'  took: 25.1049 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.94it/s]

Epoch 1/50 - Loss: 0.004 - error: 97.840
              Val_loss: 0.001 - Val_error: 70.276
```

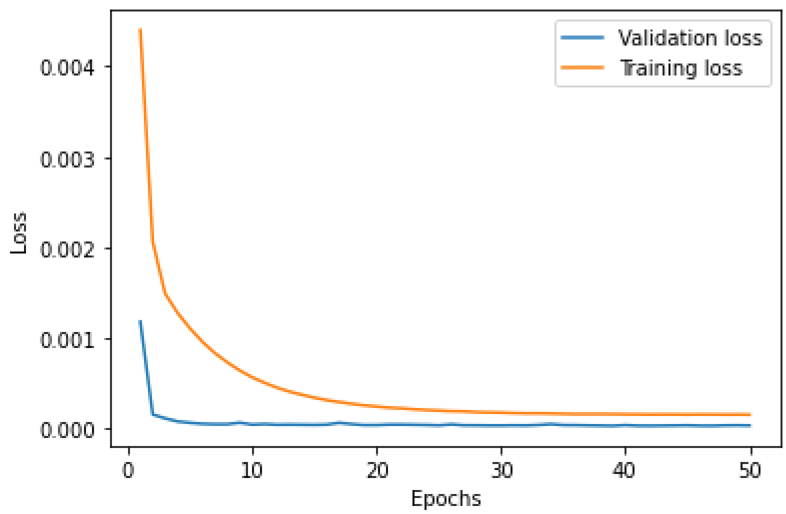
```
 52%|█████     | 26/50 [00:13<00:12,  2.00it/s]

Epoch 26/50 - Loss: 0.000 - error: 8.250
              Val_loss: 0.000 - Val_error: 5.310


func:'train'  took: 25.3280 sec
```

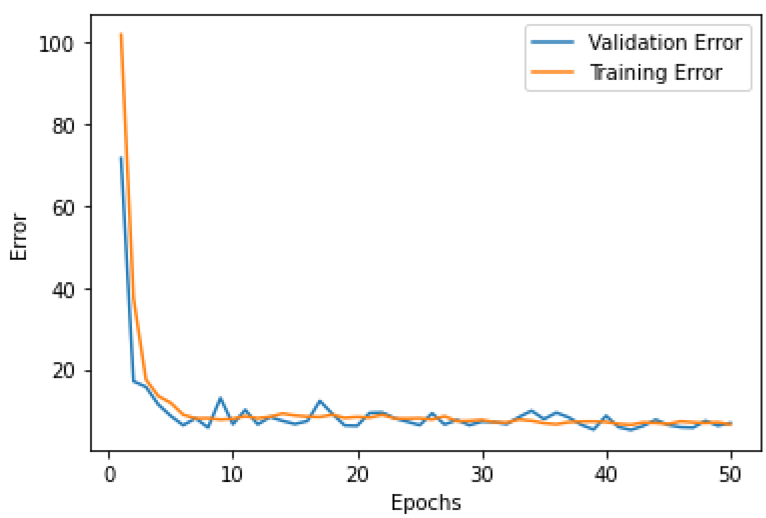


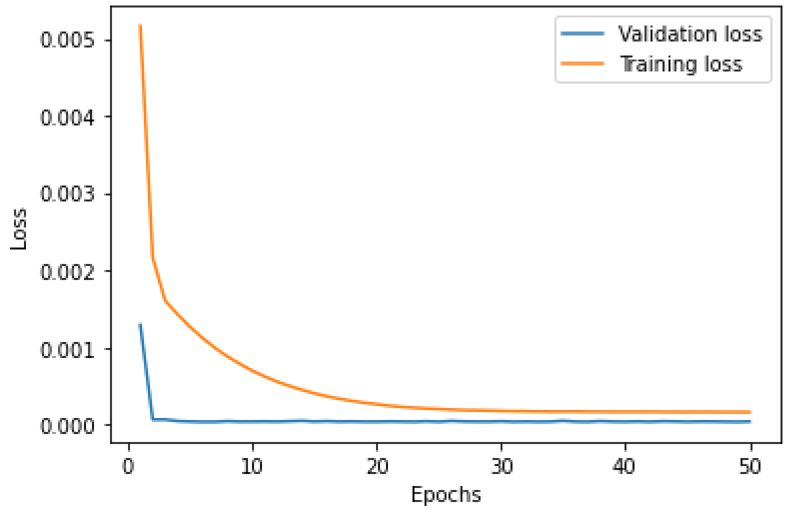


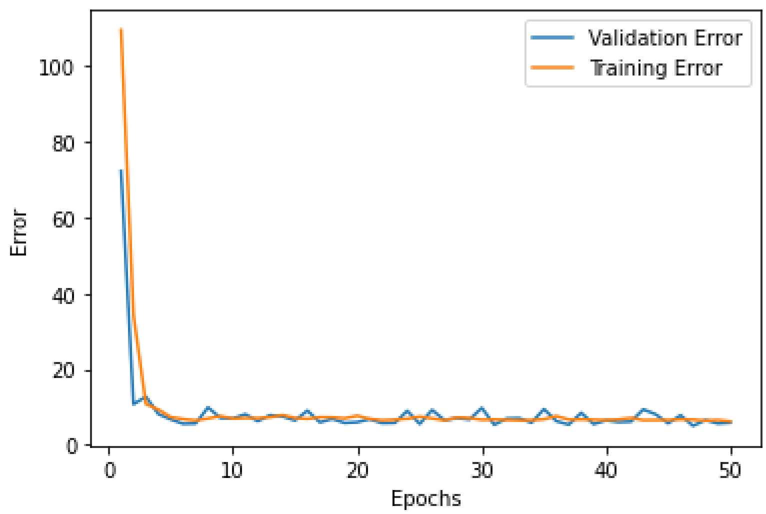


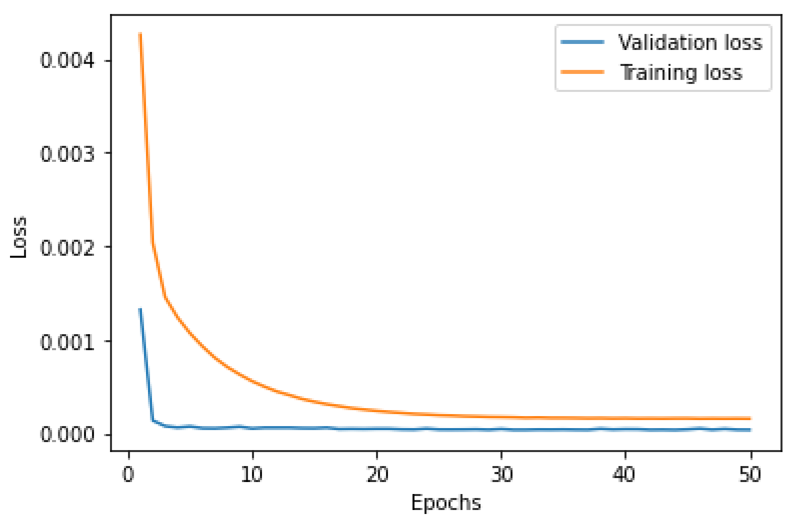

Validation Error
Training Error
Error
Epochs

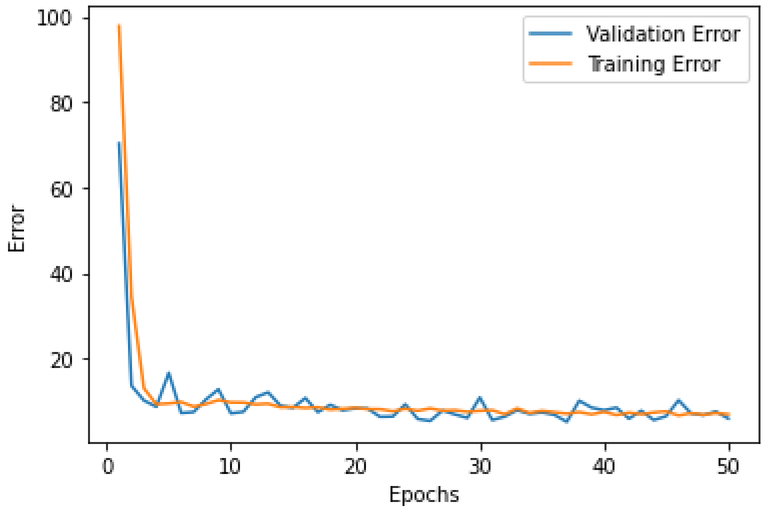

Validation loss
Training loss
Loss
Epochs


Validation Error
Training Error
Error
Epochs


Validation loss
Training loss
Loss
Epochs
0.000
0.001
0.002
0.003
0.004
0.005
0
10
20
30
40
50


Validation Error
Training Error
Error
Epochs
0
20
40
60
80
100
0
10
20
30
40
50


Validation loss
Training loss
Loss
Epochs
0.000
0.001
0.002
0.003
0.004
0
10
20
30
40
50




```
In [10]: print('Percentage error in test set for s=2')

         errors = evaluate_ind(model,test_X, test_y, error_analy,print_error=T
         rue)
```

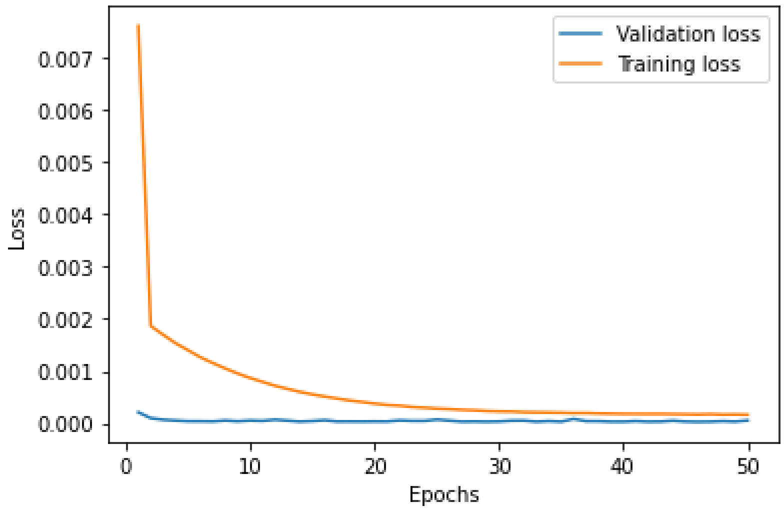
```
Percentage error in test set for s=2
Testing set Precentage Error: 5.725
```

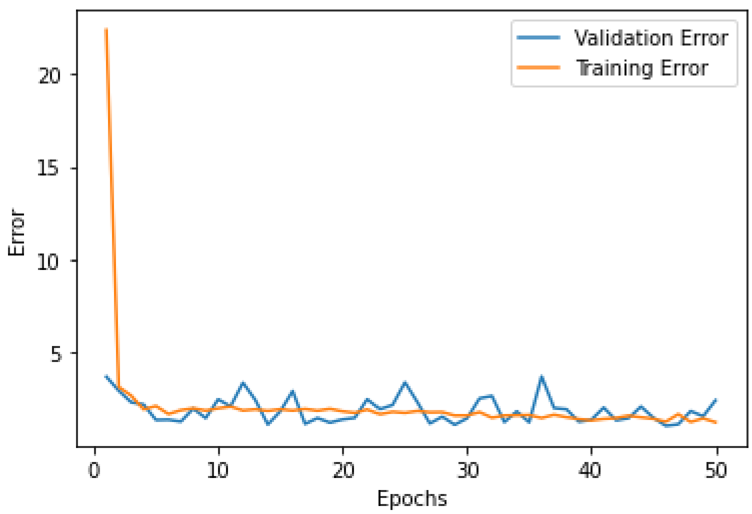
In [12]:

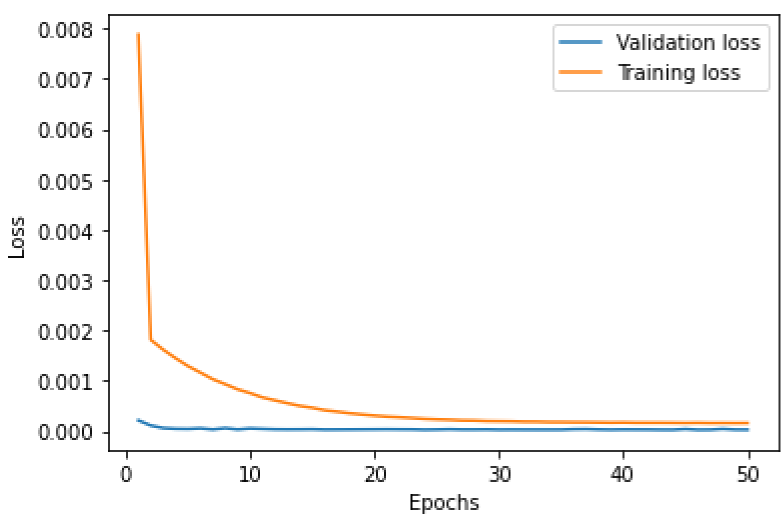
```
train_X,train_y = data_generator(100000,1)
test_X,test_y = data_generator(1000,1)
model, weights = Kfold(5,ann,error_analy,nn.MSELoss(),train_X,train_y
,test_X,test_y,50,500,lr=1e-3,early_stop=True,l2=True,pde=False)
```

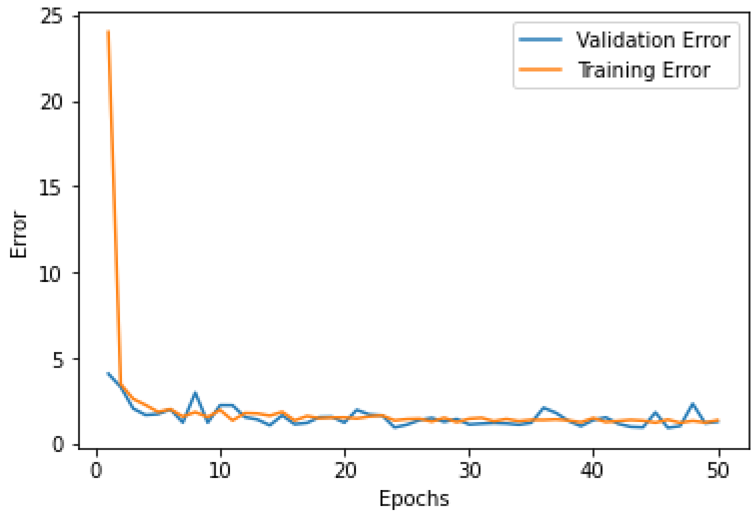
```
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.91it/s]

Epoch 1/50 - Loss: 0.008 - error: 22.362
              Val_loss: 0.000 - Val_error: 3.716

 52%|█████▏    | 26/50 [00:13<00:12,  1.94it/s]

Epoch 26/50 - Loss: 0.000 - error: 1.869
              Val_loss: 0.000 - Val_error: 2.342


func:'train'  took: 26.3063 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.95it/s]

Epoch 1/50 - Loss: 0.008 - error: 24.013
              Val_loss: 0.000 - Val_error: 4.106

 52%|█████▏    | 26/50 [00:13<00:12,  1.97it/s]

Epoch 26/50 - Loss: 0.000 - error: 1.482
              Val_loss: 0.000 - Val_error: 1.403


func:'train'  took: 25.4020 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:25,  1.93it/s]

Epoch 1/50 - Loss: 0.006 - error: 20.571
              Val_loss: 0.000 - Val_error: 4.509

 52%|█████▏    | 26/50 [00:13<00:12,  2.00it/s]

Epoch 26/50 - Loss: 0.000 - error: 1.625
              Val_loss: 0.000 - Val_error: 1.399


func:'train'  took: 28.6110 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:24,  1.97it/s]

Epoch 1/50 - Loss: 0.007 - error: 21.195
              Val_loss: 0.000 - Val_error: 3.761

 52%|█████▏    | 26/50 [00:12<00:12,  1.98it/s]

Epoch 26/50 - Loss: 0.000 - error: 1.536
              Val_loss: 0.000 - Val_error: 1.466


func:'train'  took: 25.7126 sec
 parameters: 40901

  2%|▏         | 1/50 [00:00<00:24,  1.96it/s]

Epoch 1/50 - Loss: 0.007 - error: 20.279
              Val_loss: 0.000 - Val_error: 3.555
```

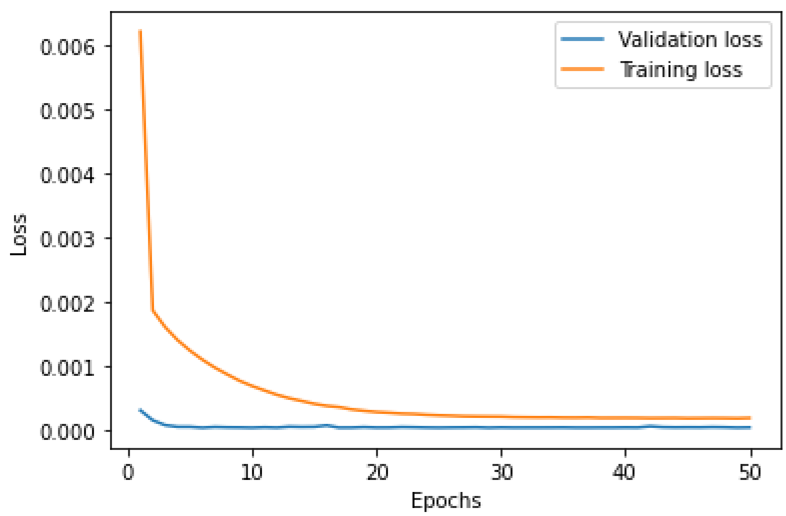
```
 52%|█████     | 26/50 [00:14<00:14,  1.60it/s]

Epoch 26/50 - Loss: 0.000 - error: 1.624
              Val_loss: 0.000 - Val_error: 2.568


func:'train'  took: 29.0541 sec
```

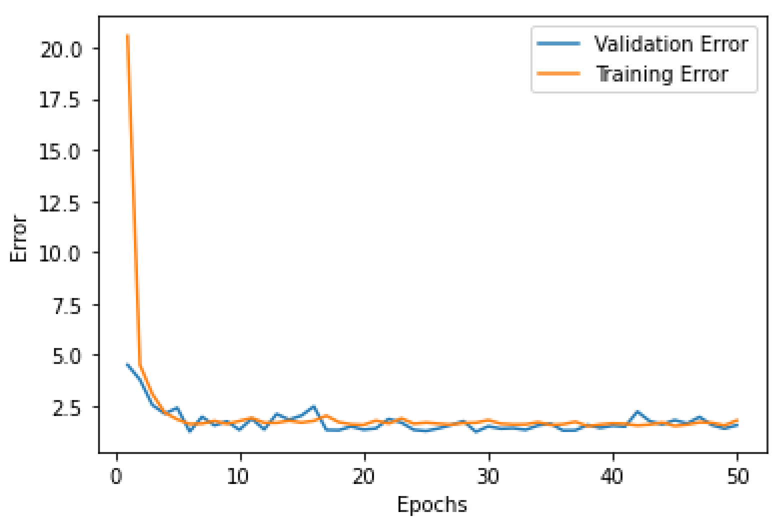


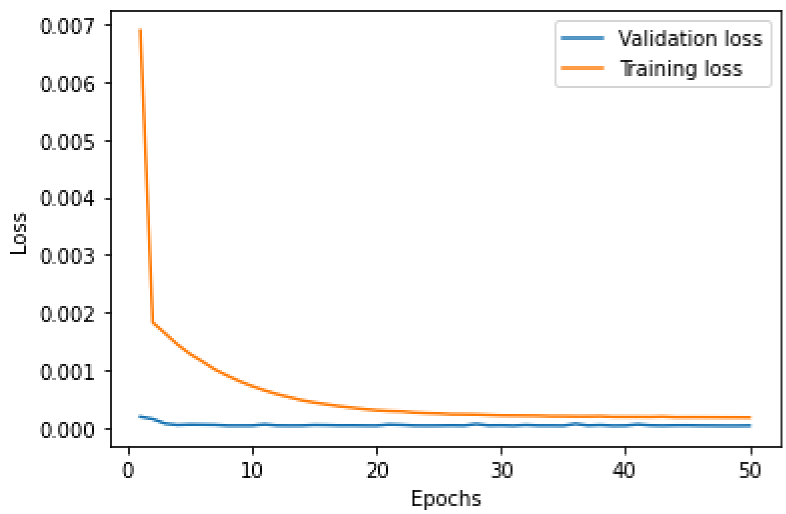


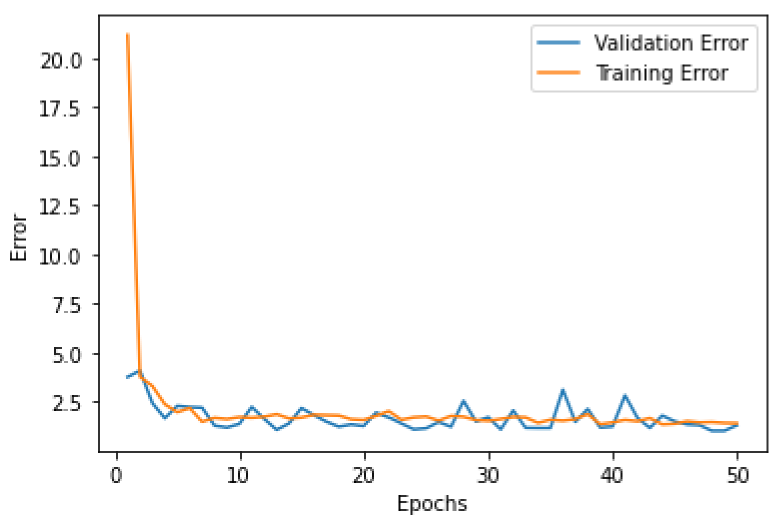


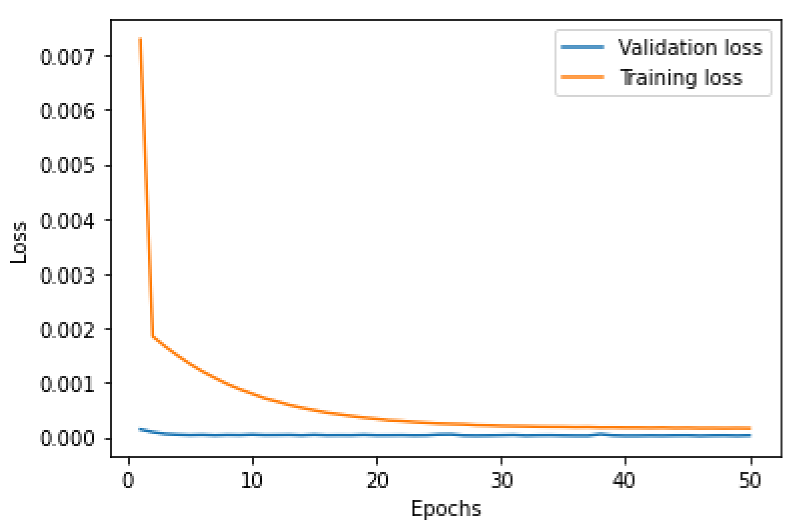

Validation Error
Training Error
Error
Epochs
0
5
10
15
20
25
30
40
50

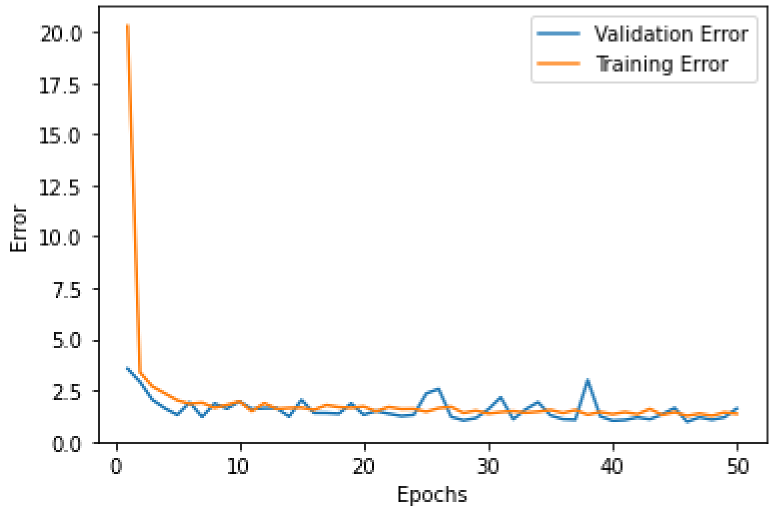

Validation loss
Training loss
Loss
Epochs
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0
10
20
30
40
50


Validation Error
Training Error
Error
Epochs
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
10
20
30
40
50


Validation loss
Training loss
Loss
Epochs


Validation Error
Training Error
Error
Epochs


Validation loss
Training loss
Loss
Epochs



In [13]:
```
print('Percentage error in test set for s=1')

errors = evaluate_ind(model,test_X, test_y, error_analy,print_error=True)
```

```
Percentage error in test set for s=1
Testing set Precentage Error: 1.605
```

In [ ]:

Step 2: Solving the PDE with Neural Network

In this step, I have attempted to solve the PDE when s is not a constant and there is no analytical solution. I tried to solve the PDE for the case when s(x) goes from 1 to 2 in a tanh like fashion when x goes from -1 to 1. I have used a semi-supervised learning type approach where instead of labelled data I incorporate the original PDE (Equation 1) and Boundary conditions (Equation 2 and 3) in the loss function.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from torch.optim import SGD, Adam
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
from sklearn.model_selection import train_test_split
from functools import wraps
from time import time
from math import*

def timing(f):
    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        print('func:%r  took: %2.4f sec' % (f.__name__,  te-ts))
        return result
    return wrap

def create_chunks(complete_list, chunk_size=None, num_chunks=None):

    chunks = []
    if num_chunks is None:
        num_chunks = math.ceil(len(complete_list) / chunk_size)
    elif chunk_size is None:
        chunk_size = math.ceil(len(complete_list) / num_chunks)
    for i in range(num_chunks):
        chunks.append(complete_list[i * chunk_size: (i + 1) * chunk_s
ize])
    return chunks
```

Training data generators

In [2]:
```
def data_generator_pde(N):
    train_X = np.zeros((N,1))
    #train_X = np.zeros(N)
    y = np.linspace(-1,1,N)
    for i in range(N):
        train_X[i,0]=y[i]
    return train_X,train_X
```

In [3]:
```
def data_generator(N,s):
    train_X = np.zeros((N,3))
    train_y = np.zeros((N))
    y = np.linspace(-1,1,N)

    for i in range(N):
        train_X[i,0] = s#0*np.random.rand() + 0.01 #s
        train_X[i,1] = y[i]
        train_X[i,2] = 0*2*np.random.rand() - 0*1.0 #x'
        train_y[i] = np.exp(-train_X[i,0]*abs(train_X[i,1]-train_X[i,
2]))/(2*train_X[i,0])

    return train_y
```

Optimized architecture of the neural network.

In [4]:
```python
from torch import nn
import torch


class ann(nn.Module):
    def __init__(self):
        super(ann, self).__init__()
        a = 200
        inp = 1
        self.Linear = nn.ModuleList([nn.Linear(inp,a),nn.Linear(a,a)
                                     ,nn.Linear(a,a),nn.Linear(a,a)
                                     ,nn.Linear(a,100),nn.Linear(100,
1)])
        self.activation = nn.ModuleList([nn.Tanh()])
    def forward(self, x):
        for i in range(len(self.Linear)-1):
                x = self.activation[0](self.Linear[i](x))
        x = self.Linear[-1](x)
        return x

print(ann())
```

```
ann(
  (Linear): ModuleList(
    (0): Linear(in_features=1, out_features=200, bias=True)
    (1): Linear(in_features=200, out_features=200, bias=True)
    (2): Linear(in_features=200, out_features=200, bias=True)
    (3): Linear(in_features=200, out_features=200, bias=True)
    (4): Linear(in_features=200, out_features=100, bias=True)
    (5): Linear(in_features=100, out_features=1, bias=True)
  )
  (activation): ModuleList(
    (0): Tanh()
  )
)
```

Neural Network Trianer

In [5]:

```
class Trainer_pde():
    def __init__(self, model, error_fn,loss_fn, learning_rate, epoch,
batch_size):

        self.model = model()
        self.optimizer = Adam(self.model.parameters(),learning_rate,w
eight_decay=1e-5)
        self.epoch = epoch
        self.batch_size = batch_size

    @timing
    def train(self,error_fn,loss_fn,inputs,val_inputs,early_stop,l2,s
ilent=False):
        ### convert data to tensor of correct shape and type here ###
        inputs = torch.tensor(inputs,dtype=torch.float32,requires_gra
d = True)
        val_inputs = torch.tensor(val_inputs,dtype=torch.float32,requ
ires_grad = True)
        losses = []
        errors = []
        val_losses = []
        val_errors = []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        l2_lambda=1e-5
        for n_epoch in tqdm(range(self.epoch), leave=False):
            self.model.train()
            batch_indices = list(range(inputs.shape[0]))
            batch_indices = create_chunks(batch_indices, chunk_size=s
elf.batch_size)
            #print(batch_indices)
            epoch_loss = 0
            epoch_error = 0
            for batch in batch_indices:
                batch_importance = len(batch) / len(inputs)
                batch_input = inputs[batch]
                ### make prediction and compute loss with loss functi
on of your choice on this batch ###
                batch_predictions = self.model.forward(batch_input).s
queeze(-1)
                deriv = [torch.autograd.grad(outputs=out, inputs=batc
h_input, allow_unused=True,  retain_graph=True, create_graph=True)[0]
[i] for i, out in enumerate(batch_predictions)]
                first_deriv = torch.empty((len(deriv)),requires_grad=
False)
                for i in range(len(deriv)):
                    first_deriv[i] = (deriv[i])
                sec_deriv =  torch.empty((len(deriv)),requires_grad=F
alse)
                deriv2 = [torch.autograd.grad(outputs=out, inputs=bat
ch_input,allow_unused=True, retain_graph=True, create_graph=True)[0][
i] for i, out in enumerate(first_deriv)]
                for i in range(len(deriv)):
                    sec_deriv[i] = (deriv2[i])
                answer = torch.mul(torch.tensor(-1,dtype=torch.float6
4),sec_deriv) + torch.mul(torch.pow(s_tanh(batch_input),2),batch_pred
```

```
ictions) - delta(batch_input,1e-5)
                answer1 = first_deriv[0] - torch.mul(s_tanh(batch_inp
ut[0]),batch_predictions[0])
                answer2 = first_deriv[-1] + torch.mul(s_tanh(batch_in
put[-1]),batch_predictions[-1])
                loss = nn.MSELoss()(answer,torch.zeros(len(deriv))) +
nn.MSELoss()(answer1,torch.zeros(1)) +  nn.MSELoss()(answer2,torch.ze
ros(1))
                if l2:
                    ### Compute the loss with L2 regularization ###
                    l2_norm = sum([p.pow(2.0).sum() for p in self.mod
el.parameters()])
                    loss = loss + l2_lambda * l2_norm
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                ### Compute epoch_loss and epoch_error
                epoch_loss += loss.detach().item()*batch_importance
                error = np.sqrt(epoch_loss)
                epoch_error += error*batch_importance

            predictions =self.model.forward(val_inputs).squeeze(-1)
            deriv = [torch.autograd.grad(outputs=out, inputs=val_inpu
ts, allow_unused=True,  retain_graph=True, create_graph=True)[0][i] f
or i, out in enumerate(predictions)]
            first_deriv = torch.empty((len(deriv)),requires_grad=Fals
e)
            for i in range(len(deriv)):
                    first_deriv[i] = deriv[i]
            sec_deriv =  torch.empty((len(deriv)),requires_grad=False
)
            deriv2 = [torch.autograd.grad(outputs=out, inputs=val_inp
uts,allow_unused=True, retain_graph=True, create_graph=True)[0][i] fo
r i, out in enumerate(first_deriv)]
            for i in range(len(deriv)):
                    sec_deriv[i] = deriv2[i]
            answer = torch.mul(torch.tensor(-1,dtype=torch.float64),s
ec_deriv) + torch.mul(torch.pow(s_tanh(val_inputs),2),predictions) -
delta(val_inputs,1e-5)
            answer1 = first_deriv[0] - torch.mul(s_tanh(val_inputs[0
]),predictions[0])
            answer2 = first_deriv[-1] + torch.mul(s_tanh(val_inputs[-
1]),predictions[-1])
            loss = nn.MSELoss()(answer,torch.zeros(len(deriv))) +  nn
.MSELoss()(answer1,torch.zeros(1)) +  nn.MSELoss()(answer2,torch.zero
s(1))
            val_loss = loss.detach().numpy()
            val_error = np.sqrt(val_loss)
            #val_loss, val_error = self.evaluate(val_inputs, val_outp
uts, error_fn,print_error=False)
            if n_epoch % 25 ==0 and not silent:
                print("Epoch %d/%d - Loss: %.3f - error: %.3f" % (n_e
poch + 1, self.epoch, epoch_loss, epoch_error))
                print("              Val_loss: %.3f - Val_error: %.3f
" % (val_loss, val_error))
            losses.append(epoch_loss)
            errors.append(epoch_error)
```

```
            val_losses.append(val_loss)
            val_errors.append(val_error)
            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()
        if early_stop:
            self.model.load_state_dict(weights)

        return self.model,{"losses": losses, "errors": errors, "val_l
osses": val_losses, "val_errors": val_errors}
```

In [6]:
```
from sklearn.model_selection import train_test_split,KFold

def train_and_val(model,error_fn,loss_fn,train_X,val_X,epochs,batch_s
ize,lr,early_stop,l2,draw_curve=True):

    ann_trainer = Trainer_pde(model,error_fn,loss_fn,lr, epochs, batc
h_size)

    model,ledger  = ann_trainer.train(error_fn,loss_fn,train_X,val_X,
early_stop,l2)
    val_array = ledger['val_losses']
    train_array = ledger['losses']
    val_error= ledger['val_errors']
    train_error = ledger['errors']
    train_error_all=[]
    if draw_curve:
        plt.figure()
        plt.plot(np.arange(len(val_array))+1,val_array,label='Validat
ion loss')
        plt.plot(np.arange(len(train_array))+1,train_array,label='Tra
ining loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

    if draw_curve:
        plt.figure()
        plt.plot(np.arange(len(val_error))+1,val_error,label='Validat
ion Error')
        plt.plot(np.arange(len(train_error))+1,train_error,label='Tra
ining Error')
        plt.xlabel('Epochs')
        plt.ylabel('Error')
        plt.legend()

    if early_stop:
            report_idx= np.argmin(ledger["val_losses"])
    else:
            report_idx=-1
    ### Recover the model weight ###
    weights = model.parameters()

    return model,weights
```

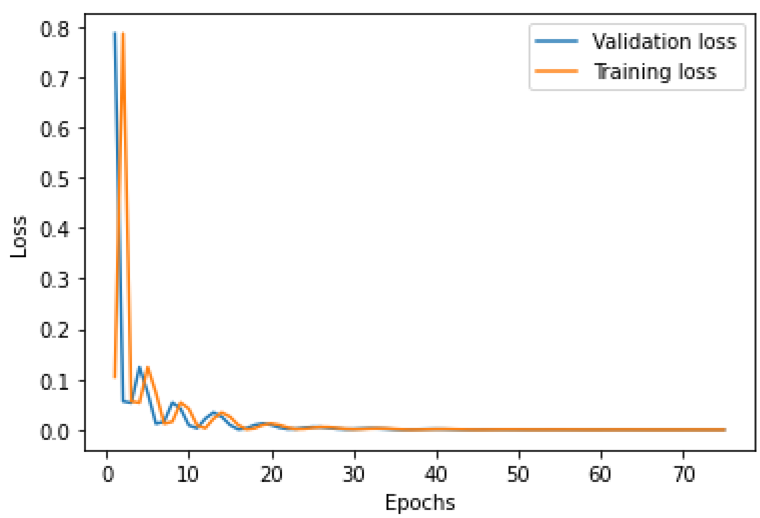
Delta function approximator for PDE

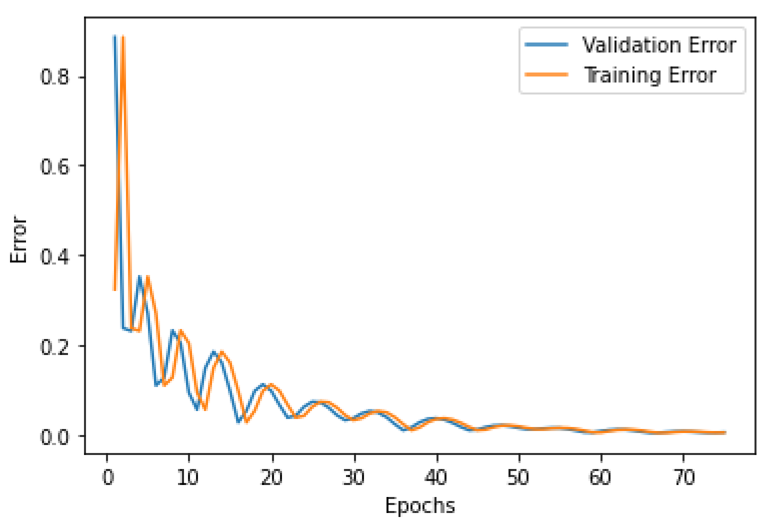
In [7]:
```
def delta(x,eps):
    ans = torch.zeros(x.size(dim=0))
    torch.pi = torch.acos(torch.zeros(1)).item() * 2
    for i in range(x.size(dim=0)):
        #ans[i] = torch.tensor(1.0/(eps*sqrt(2.*torch.pi))*torch.exp
(-(x[i])*x[i]/(2.*eps**2)),dtype=torch.float32,requires_grad=True)
        ans[i] = 1.0/(eps*sqrt(2.*torch.pi))*torch.exp(-(x[i])*x[i]/(
2.*eps**2)).clone().detach().requires_grad_(True)
    return ans
    #return torch.tensor(1.0/(eps*sqrt(2.*np.pi))*torch.exp(-(x-y)**
2/(2.*eps**2)),dtype=torch.float32,requires_grad=True)

def s_tanh(x):
    ans = torch.zeros(x.size(dim=0))
    for i in range(x.size(dim=0)):
        ans[i] = 1.5*torch.tanh(torch.mul(x[i],10)) + 0.5
    return ans
    #return 1.5*np.tanh(x*10) + 0.5
```

Loss and Error Function

In [11]:
```
N=20
train_X,train_y = data_generator_pde(N)
model, weights = train_and_val(ann,nn.MSELoss,nn.MSELoss,train_X,trai
n_X,75,N,lr=1e-3,early_stop=True,l2=False)
```

```
  1%||          | 1/75 [00:00<00:44,  1.66it/s]

Epoch 1/75 - Loss: 0.105 - error: 0.324
              Val_loss: 0.786 - Val_error: 0.886

 35%|███        | 26/75 [00:15<00:29,  1.68it/s]

Epoch 26/75 - Loss: 0.005 - error: 0.073
              Val_loss: 0.005 - Val_error: 0.072

 68%|███████    | 51/75 [00:30<00:14,  1.67it/s]

Epoch 51/75 - Loss: 0.000 - error: 0.015
              Val_loss: 0.000 - Val_error: 0.012


func:'train'  took: 44.4100 sec
```





In [9]:
```
def error_analy(x,y):
    error = np.sum(np.true_divide(abs(x-y),y))*100/(len(y))
    return error

def evaluate_ind(model, inputs, outputs,error_fn,print_error=True):
        loss_fn = nn.MSELoss()
        inputs = torch.FloatTensor(inputs)
        predictions = model.forward(inputs).squeeze(-1)
        print(predictions)
        print(outputs)
        error = error_fn(predictions.detach().numpy(),outputs)
        if print_error:
            print("Testing set Precentage Error: %.3f" % error)
        return error
```

In [10]:
```
test_y = data_generator(N,1)

evaluate_ind(model,train_X,test_y,error_analy)
```

```
tensor([ 7.9071e-03,  7.2247e-03,  6.5728e-03,  5.9541e-03,  5.3697e-
03,
         4.8199e-03,  4.3034e-03,  3.8182e-03,  3.3616e-03,  2.9308e-
03,
         2.5229e-03,  2.1353e-03,  1.7661e-03,  1.4143e-03,  1.0800e-
03,
         7.6430e-04,  4.6987e-04,  2.0020e-04, -4.0427e-05, -2.4769e-
04],
       grad_fn=<SqueezeBackward1>)
[0.18393972 0.20435757 0.22704186 0.25224418 0.28024402 0.31135193
 0.34591291 0.38431026 0.42696983 0.47436474 0.47436474 0.42696983
 0.38431026 0.34591291 0.31135193 0.28024402 0.25224418 0.22704186
 0.20435757 0.18393972]
Testing set Precentage Error: 98.858
```

Out[10]:
```
98.85818344289994
```