Import Required Libraries

```
import torch
from torch import Tensor
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import itertools
from collections import OrderedDict


# Device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

print(device)

if device == 'cuda':
    print(torch.cuda.get_device_name())
```

Skip the part if you are running your code in local system

```
from google.colab import drive
drive.mount('/content/drive')
```

Load the training data from numpy

```
d = np.load("/content/drive/My Drive/antiderivative_aligned_train.npz", allow_pickle=True)
y_train=d["X"][1].astype(np.float32) #output locations (100,1)
u_train = d["X"][0].astype(np.float32) # input functions (150,100)
s_train = d["y"].astype(np.float32) # output functions (150,100)
```

Define Network Archietecture

```
# the deep neural network
class DNN(torch.nn.Module):
    def __init__(self, layers):
        #define your sequential model here with activations

    def forward(self, x): #forward pass
        out = self.layers(x)
        return out
```

The DeepONet Archietecture

```
class PI_DeepONet():
    def __init__(self, branch_layers, trunk_layers,u_train, y_train, s_train):

        self.u_train =        #convert to torch tensor
        self.y_train =        #convert to torch tensor
        self.s_train =        #convert to torch tensor


        self.branch_net = DNN(branch_layers).to(device)  # The branch Network

        self.trunk_net = DNN(trunk_layers).to(device)    # The trunk Network

        branch_params =                #extract the network Parameters in list format
        trunk_params =                 #extract the network Parameters
        params = branch_params+trunk_params

        self.optimizer =torch.optim.LBFGS(params, #########
                              #######)

        self.optimizer_Adam = torch.optim.SGD(params, lr=0.01)

        self.iter = 0    #initiate iteration
```

```
    def operator_net(self,  u, y):    # Define DeepONet architecture
        B = self.branch_net(u) #output from branch Network
        T = self.trunk_net(y)  #output from branch Network

        output = torch.matmul(B, torch.transpose(T,0,1))


        return output

    # Define operator loss
    def loss_operator(# #):

        pred =          # Compute forward pass
        loss =          # Compute loss
        return loss

    def loss_func(self):   #Define loss function for optimization step
        loss = self.loss_operator(##)
        self.optimizer.zero_grad()
        loss.backward()
        return loss

    def train(self, nIter):
        model_loss=np.array([])
        for epoch in range(nIter):
            loss= self.loss_operator(self.u_train,self.y_train)

             # Backward and optimize

            self.optimizer_Adam.zero_grad()
            loss.backward()
            self.optimizer_Adam.step()

            model_loss=np.append(model_loss,###)   #get the loss value for each iteration

            if epoch % 10 == 0:
               # print loss and iteration

        # Backward and optimize
        self.optimizer.step(self.loss_func)

        return model_loss

    # Evaluates predictions at test points
    def predict_s(self, u_star,y_star):
        #####
        ####
        s = self.operator_net(u_star, y_star) #predict
        s = s.detach().cpu().numpy()
        return s
```

Train the Model

```
# Initialize model

branch_layers = [100, 50, 50, 50, 50, 50]
trunk_layers =  [1, 50, 50, 50, 50, 50]

model = PI_DeepONet(branch_layers, trunk_layers,u_train, y_train, s_train)


'Neural Network Summary'
print(model)
nIter=1000
loss=model.train(nIter)
```

Load Test Data

```
d = np.load("/content/drive/My Drive/antiderivative_aligned_test.npz", allow_pickle=True)
B=d["X"][0].astype(np.float32)
u_test = d["X"][0].astype(np.float32); y_test=d["X"][1].astype(np.float32)
s_test = d["y"].astype(np.float32)
```

```
#Model Predictions
s_pred = model.predict_s(u_test, y_test)


# Plot Visualization
```


✓ 4s completed at 16:23