24-787: Machine Learning and Artificial Intelligence for Engineers
Ryan Wu
ID: weihuanw
Homework 7
Due: Mar 16, 2024

Concept Questions:

Problem 1

$x = [1, -6, 10, 8, -2, 3]$; $w = [7, -2, -4, 5, 2, -6]$

$a = (1 \cdot 7) + (-6 \cdot -2) + (10 \cdot -4) + (8 \cdot 5) + (-2 \cdot 2) + (3 \cdot -6)$

$a = 7 + 12 - 40 + 40 - 4 - 18 \rightarrow a = -3$

$y = f(a) = \frac{1}{1+e^{-a}} = \frac{1}{1+e^{3}} \cong 0.0474$ #

Problem 2

$$W_3 = \begin{bmatrix} 10 & -3 \\ 16 & 9 \\ 12 & -7 \end{bmatrix}$$

Problem 3
The softmax activation function is used in the output layer for multi-class classification problems to produce a probability distribution over multiple classes.

Problem 4
All of the above.

Problem 1

Problem Description

In this problem you will create your own neural network to fit a function with two input features x_0 and x_1, and predict the output, y. The structure of your neural network is up to you, but you must describe the structure of your network, training parameters, and report an MSE for your fitted model on the provided data.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- Visualization of provided data
- Visualization of trained model with provided data
- Trained model MSE
- Discussion of model structure and training parameters

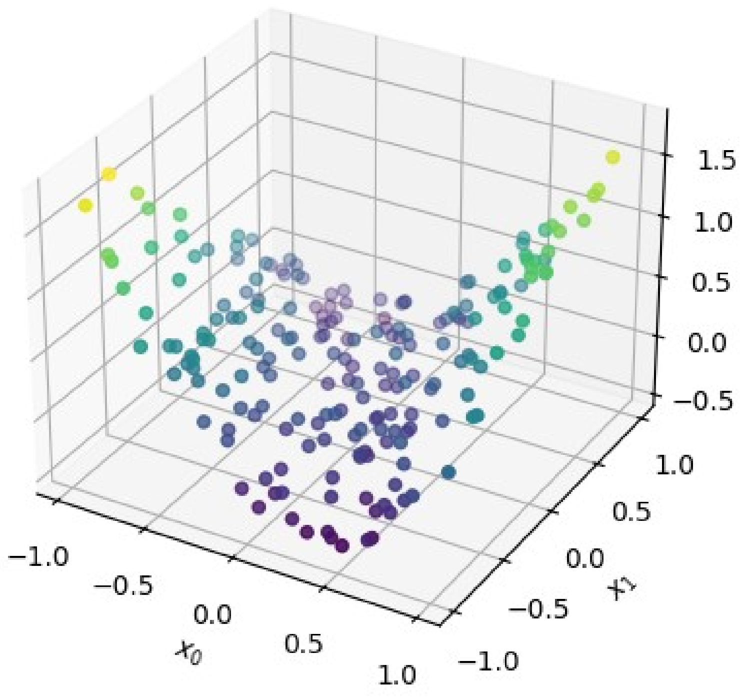
Imports and Utility Functions:

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

def dataGen():
    # Set random seed so generated random numbers are always the same
    gen = np.random.RandomState(0)
    # Generate x0 and x1
    x = 2*(gen.rand(200,2)-0.5)
    # Generate y with x0^2 - 0.2*x1^4 + x0*x1 + noise
    y = x[:,0]**2 - 0.2*x[:,1]**4 + x[:,0]*x[:,1] +
0.4*(gen.rand(len(x))-0.5)

    return x, y

def visualizeModel(model):
    # Get data
    x, y = dataGen()
    # Number of data points in meshgrid
    n = 25
    # Set up evaluation grid
    x0 = torch.linspace(min(x[:,0]),max(x[:,0]),n)
    x1 = torch.linspace(min(x[:,1]),max(x[:,1]),n)
    X0, X1 = torch.meshgrid(x0, x1, indexing = 'ij')
    Xgrid = torch.vstack((X0.flatten(),X1.flatten())).T
    Ypred = model(Xgrid).reshape(n,n)
    # 3D plot
    fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
    # Plot data
    ax.scatter(x[:,0],x[:,1],y, c = y, cmap = 'viridis')
    # Plot model

ax.plot_surface(X0.detach().numpy(),X1.detach().numpy(),Ypred.detach()
.numpy(), color = 'gray', alpha = 0.25)

ax.plot_wireframe(X0.detach().numpy(),X1.detach().numpy(),Ypred.detach
().numpy(),color = 'black', alpha = 0.25)
    ax.set_xlabel('$x_0$')
    ax.set_ylabel('$x_1$')
    ax.set_zlabel('$y$')
    plt.show()
```

Generate and visualize the data

Use the `dataGen()` function to generate the x and y data, then visualize with a 3D scatter plot.

```
# YOUR CODE GOES HERE
# generate data
x, y = dataGen()

# plot 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[:, 0], x[:, 1], y, c=y, cmap='viridis')
ax.set_xlabel('$x_0$')
ax.set_ylabel('$x_1$')
ax.set_zlabel('$y$')
plt.show()
```



Create and train a neural network using PyTorch

Choice of structure and training parameters are entirely up to you, however you will need to provide reasoning for your choices. An MSE smaller than 0.02 is reasonable.

```
# YOUR CODE GOES HERE
# import utility fucntions
from torch import optim, nn
import torch.nn.functional as F

# covert data to tensor
x = torch.Tensor(x)
y = torch.Tensor(y).reshape(-1,1)

# define model
class Net_3_layer(nn.Module):
    def __init__(self, N_hidden=5, N_in=2, N_out=1, activation =
F.relu):
        super().__init__()
        self.lin1 = nn.Linear(N_in, N_hidden)
        self.lin2 = nn.Linear(N_hidden, N_hidden)
        self.lin3 = nn.Linear(N_hidden, N_hidden)
        self.lin4 = nn.Linear(N_hidden, N_out)
        self.act = activation

        # other activation functions
        if activation == F.relu:
            self.act = nn.ReLU()
        elif activation == F.sigmoid:
            self.act = nn.Sigmoid()
        elif activation == F.tanh:
            self.act = nn.Tanh()
        elif activation == F.leaky_relu:
            self.act = nn.LeakyReLU()
        elif activation == F.gelu:
            self.act = nn.GELU()
        else:
            self.act = nn.Identity()

    def forward(self,x):
        x = self.lin1(x)
        x = self.act(x)  # Activation of first hidden layer
        x = self.lin2(x)
        x = self.act(x)  # Activation at second hidden layer
        x = self.lin3(x)
        x = self.act(x)  # Activation at third hidden layer
        x = self.lin4(x) # (No activation at last layer)

        return x

# instantiate model
# ReLU
# model = Net_3_layer(N_hidden=30, N_in=2, N_out=1, activation=F.relu)
# Sigmoid
# model = Net_3_layer(N_hidden=30, N_in=2, N_out=1,
activation=F.sigmoid)
# Tanh
# model = Net_3_layer(N_hidden=30, N_in=2, N_out=1, activation=F.tanh)
# Leaky ReLU
model = Net_3_layer(N_hidden=5, N_in=2, N_out=1,
activation=F.leaky_relu)
# GELU
# model = Net_3_layer(N_hidden=30, N_in=2, N_out=1, activation=F.gelu)

# lose curve list
loss_curve = []

# train model
# lr = 0.0005     # learning rate
# lr = 0.00005
lr = 0.005
epochs = 1500   # number of epochs
loss_fcn = F.mse_loss

# Set up optimizer to optimize the model's parameters using Adam with
the selected learning rate
```

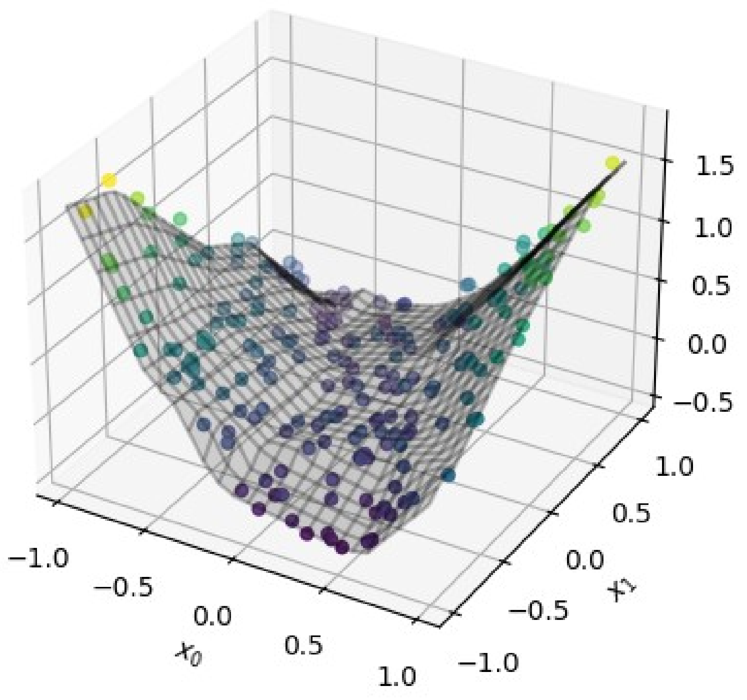
```
opt = optim.Adam(params = model.parameters(), lr=lr)

# Training loop
for epoch in range(epochs):
    out = model(x) # Evaluate the model
    loss = loss_fcn(out,y) # Calculate the loss -- error between
network prediction and y

    loss_curve.append(loss.item())

    # Print loss progress info 25 times during training
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch} of {epochs}... \tAverage loss:
{loss.item()}")

    # Move the model parameters 1 step closer to their optima:
    opt.zero_grad()
    loss.backward()
    opt.step()
```

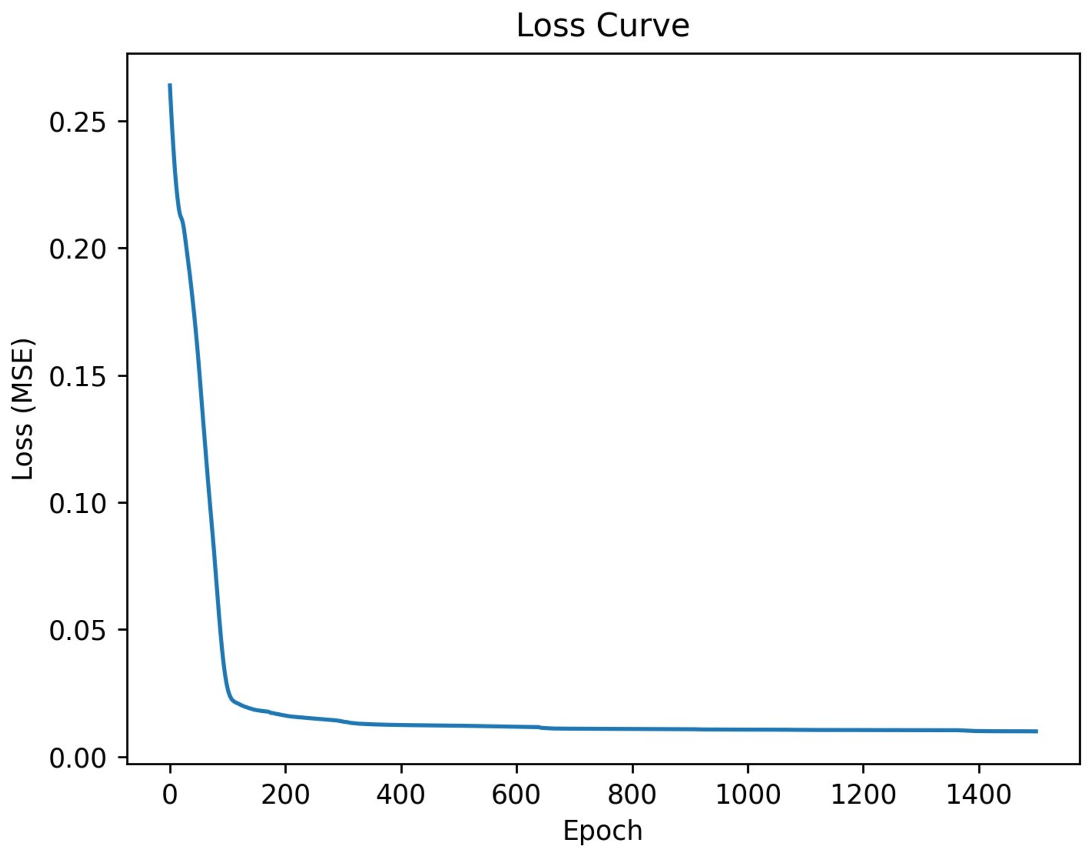
```
Epoch 0 of 1500...    Average loss: 0.26397496461868286
Epoch 60 of 1500...   Average loss: 0.12510962784290314
Epoch 120 of 1500...  Average loss: 0.020606068894267082
Epoch 180 of 1500...  Average loss: 0.017011607065796852
Epoch 240 of 1500...  Average loss: 0.015141118317842484
Epoch 300 of 1500...  Average loss: 0.013714760541915894
Epoch 360 of 1500...  Average loss: 0.012585285119712353
Epoch 420 of 1500...  Average loss: 0.012331795878708363
Epoch 480 of 1500...  Average loss: 0.012156221084296703
Epoch 540 of 1500...  Average loss: 0.01196255162358284
Epoch 600 of 1500...  Average loss: 0.01173623837530613
Epoch 660 of 1500...  Average loss: 0.011054372414946556
Epoch 720 of 1500...  Average loss: 0.010915595106780529
Epoch 780 of 1500...  Average loss: 0.010857533663511276
Epoch 840 of 1500...  Average loss: 0.010802366770803928
Epoch 900 of 1500...  Average loss: 0.010755450464785099
Epoch 960 of 1500...  Average loss: 0.010601027868688107
Epoch 1020 of 1500...      Average loss: 0.010563092306256294
Epoch 1080 of 1500...      Average loss: 0.010495845228433609
Epoch 1140 of 1500...      Average loss: 0.010422760620713234
Epoch 1200 of 1500...      Average loss: 0.01038841251283884
Epoch 1260 of 1500...      Average loss: 0.010382584296166897
Epoch 1320 of 1500...      Average loss: 0.010339329019188881
Epoch 1380 of 1500...      Average loss: 0.010162290185689926
Epoch 1440 of 1500...      Average loss: 0.009946865029633045
```

Visualize your trained model

Use the provided `visualizeModel()` function by passing in your trained model to see your models predicted function compared to the provided data

```
# YOUR CODE GOES HERE
# visualize model
visualizeModel(model)

# plot loss curve
plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```





Discussion

Report the MSE of your trained model on the generated data. Discuss the structure of your network, including the number and size of hidden layers, choice of activation function, loss function, optimizer, learning rate, number of training epochs.

YOUR ANSWER GOES HERE

Model structure and parameters:

```
3 hidden layers with 5 neurons per hidden layer, Leaky Relu as
activation function, Mean Squared Error (MSE) as loss function,
optimized using the Adam optimizer with a learning rate of 0.005 for
1500 training epochs.
```

Model performance:

```
The MSE of my trained model on the generated data is around 0.0099.
```

Model design reasoning and parameter experiments:

```
The model's activation function and learning rate were determined by
trial and error. Several activation functions like ReLu, Sigmoid,
Tanh, GELU, and Leaky ReLu were explored, and the Leaky ReLu results
in the lowest MSE. I also experimented with varying learning rates to
determine the model outcome. The results showed that a smaller
learning rate converged faster with a lower MSE in our use case. I
also test the number of neurons in each hidden layer. The results
appeared to be overfitting the data with a larger number of neurons.
Other parameters were referred from M7-L2-P2.
```

Problem 2

Problem Description

In this problem you will train a neural network to classify points with features x_0 and x_1 belonging to one of three classes, indicated by the label y. The structure of your neural network is up to you, but you must describe the structure of your network, training parameters, and report an accuracy for your fitted model on the provided data.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- Visualization of provided data
- Visualization of trained model with provided data
- Trained model accuracy
- Discussion of model structure and training parameters

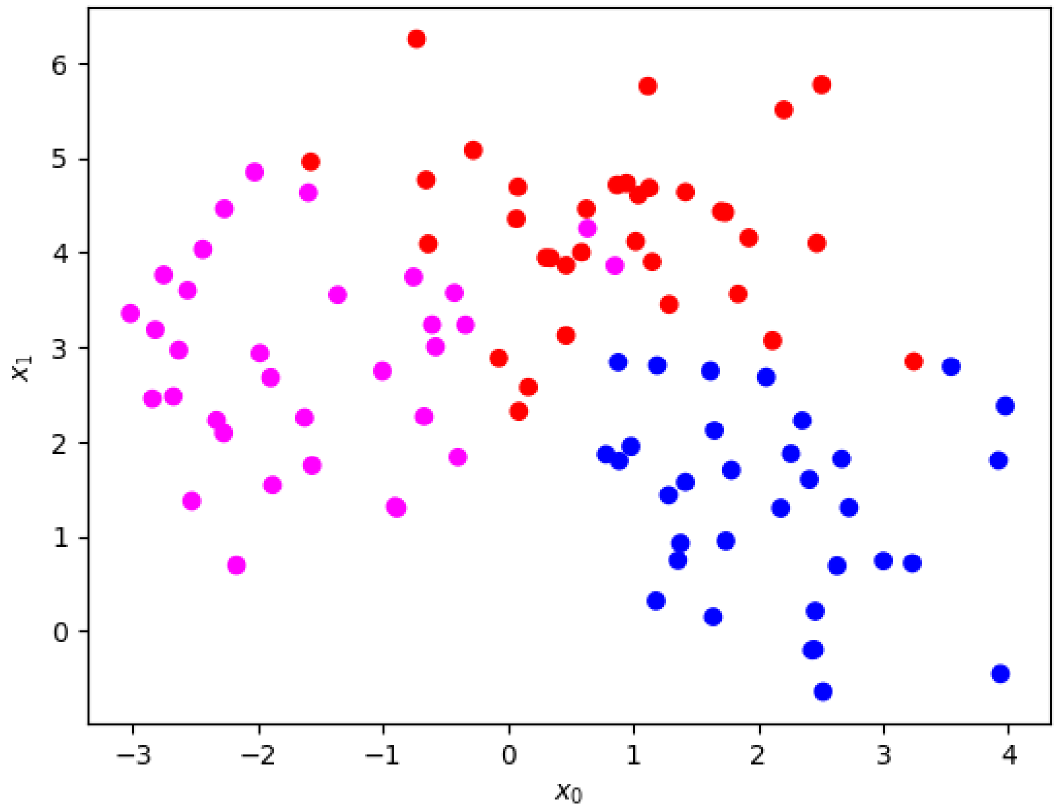
Imports and Utility Functions:

```python
import torch
import torch.nn as nn
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def dataGen():
    # random_state = 0 set so generated samples are identical
    x, y = datasets.make_blobs(n_samples = 100, n_features = 2,
centers = 3, random_state = 0)
    return x, y

def visualizeModel(model):
    # Get data
    x, y = dataGen()
    # Number of data points in meshgrid
    n = 100
    # Set up evaluation grid
    x0 = torch.linspace(min(x[:,0]), max(x[:,0]),n)
    x1 = torch.linspace(min(x[:,1]), max(x[:,1]),n)
    X0, X1 = torch.meshgrid(x0, x1, indexing = 'ij')
    Xgrid = torch.vstack((X0.flatten(),X1.flatten())).T
    Ypred = torch.argmax(model(Xgrid), dim = 1)
    # Plot data
    plt.scatter(x[:,0], x[:,1], c = y, cmap =
ListedColormap(['red','blue','magenta']))
    # Plot model
    plt.contourf(Xgrid[:,0].reshape(n,n), Xgrid[:,1].reshape(n,n), 
Ypred.reshape(n,n), cmap = ListedColormap(['red', 'blue', 'magenta']), 
alpha = 0.15)
    plt.xlabel('$x_0$')
    plt.ylabel('$x_1$')
    plt.show()
```

Generate and visualize the data

Use the `dataGen()` function to generate the x and y data, then visualize with a 2D scatter plot, coloring points according to their labels.

```
# YOUR CODE GOES HERE
# generate data
x, y = dataGen()

# plot 2D scatter plot
plt.scatter(x[:,0], x[:,1], c = y, cmap = 
ListedColormap(['red','blue','magenta']))
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.show()
```



Create and train a neural network using PyTorch

Choice of structure and training parameters are entirely up to you, however you will need to provide reasoning for your choices. An accuracy of 0.9 or more is reasonable.

Hint: think about the number out nodes in your output layer and choice of output layer activation function for this multi-class classification problem.

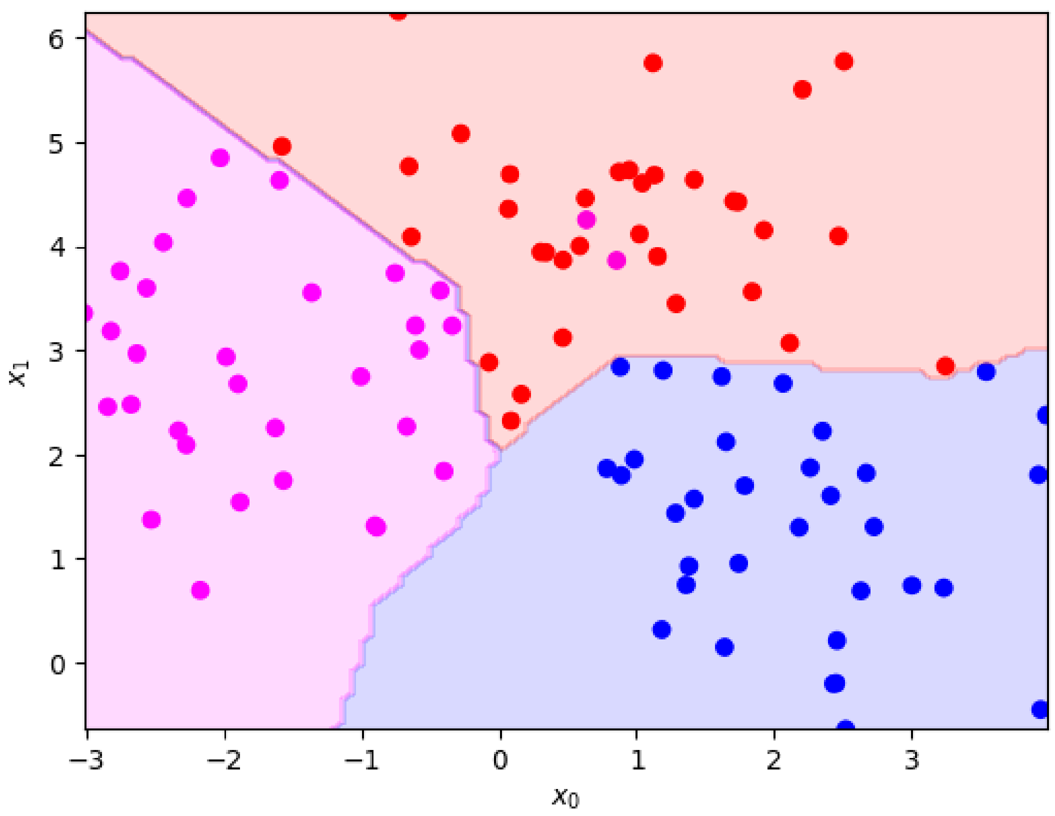
```
# YOUR CODE GOES HERE
# import utility functions
from torch import optim, nn
import torch.nn.functional as F

# convert data to tensor
x = torch.Tensor(x)
y = torch.Tensor(y).long()

# define model
class Net_3_layer(nn.Module):
    def __init__(self, N_hidden=5, N_in=2, N_out=3, activation =
F.relu):
        super().__init__()
        self.lin1 = nn.Linear(N_in, N_hidden)
        self.lin2 = nn.Linear(N_hidden, N_hidden)
        self.lin3 = nn.Linear(N_hidden, N_hidden)
        self.lin4 = nn.Linear(N_hidden, N_out)
        self.act = activation

        # other activation functions
        if activation == F.relu:
            self.act = nn.ReLU()
        elif activation == F.sigmoid:
            self.act = nn.Sigmoid()
        elif activation == F.tanh:
            self.act = nn.Tanh()
        elif activation == F.leaky_relu:
            self.act = nn.LeakyReLU()
        elif activation == F.gelu:
            self.act = nn.GELU()
        else:
            self.act = nn.Identity()

    def forward(self,x):
        x = self.lin1(x)
        x = self.act(x)  # Activation of first hidden layer
        x = self.lin2(x)
        x = self.act(x)  # Activation at second hidden layer
        x = self.lin3(x)
        x = F.softmax(x, dim=1)  # Activation at third hidden layer
(softmax for classification problem)
        # x = self.act(x)  # Activation at third hidden layer
        x = self.lin4(x) # (No activation at last layer)

        return x

# instantiate model
# ReLU
# model = Net_3_layer(N_hidden=5, N_in=2, N_out=3, activation=F.relu)
# Sigmoid
# model = Net_3_layer(N_hidden=5, N_in=2, N_out=3,
activation=F.sigmoid)
# Tanh
# model = Net_3_layer(N_hidden=5, N_in=2, N_out=3, activation=F.tanh)
# Leaky ReLU
model = Net_3_layer(N_hidden=5, N_in=2, N_out=3,
activation=F.leaky_relu)
# GELU
# model = Net_3_layer(N_hidden=5, N_in=2, N_out=3, activation=F.gelu)

# lose curve list
loss_curve = []

# train model
# lr = 0.0005     # learning rate
# lr = 0.00005
lr = 0.005
epochs = 1500   # number of epochs
loss_fcn = nn.CrossEntropyLoss()  # loss function

# Set up optimizer to optimize the model's parameters using Adam with
the selected learning rate
opt = optim.Adam(params = model.parameters(), lr=lr)

# Training loop
for epoch in range(epochs):
    out = model(x) # Evaluate the model
    loss = loss_fcn(out,y) # Calculate the loss -- error between
network prediction and y

    loss_curve.append(loss.item())

    # Print loss progress info 25 times during training
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch} of {epochs}... \tAverage loss:
{loss.item()}")

    # Move the model parameters 1 step closer to their optima:
    opt.zero_grad()
    loss.backward()
    opt.step()
```

```
Epoch 0 of 1500...    Average loss: 1.1398591995239258
Epoch 60 of 1500...   Average loss: 0.8802828788757324
Epoch 120 of 1500...  Average loss: 0.6326925158500671
Epoch 180 of 1500...  Average loss: 0.4106394052505493
Epoch 240 of 1500...  Average loss: 0.30089858174324036
Epoch 300 of 1500...  Average loss: 0.23634019494056702
Epoch 360 of 1500...  Average loss: 0.20372045040130615
Epoch 420 of 1500...  Average loss: 0.18274754285812378
Epoch 480 of 1500...  Average loss: 0.16840240359306335
Epoch 540 of 1500...  Average loss: 0.15831558406352997
Epoch 600 of 1500...  Average loss: 0.15109066665172577
Epoch 660 of 1500...  Average loss: 0.11400409787893295
Epoch 720 of 1500...  Average loss: 0.10398337990045547
Epoch 780 of 1500...  Average loss: 0.09954670071601868
Epoch 840 of 1500...  Average loss: 0.09643104672431946
Epoch 900 of 1500...  Average loss: 0.09402863681316376
Epoch 960 of 1500...  Average loss: 0.09212164580821991
Epoch 1020 of 1500...      Average loss: 0.09054950624704361
Epoch 1080 of 1500...      Average loss: 0.08923567086458206
Epoch 1140 of 1500...      Average loss: 0.0881190076470375
Epoch 1200 of 1500...      Average loss: 0.0871620625257492
Epoch 1260 of 1500...      Average loss: 0.08632531762123108
Epoch 1320 of 1500...      Average loss: 0.08559451252222061
Epoch 1380 of 1500...       Average loss: 0.08495081216096878
Epoch 1440 of 1500...       Average loss: 0.08438059687614441
```

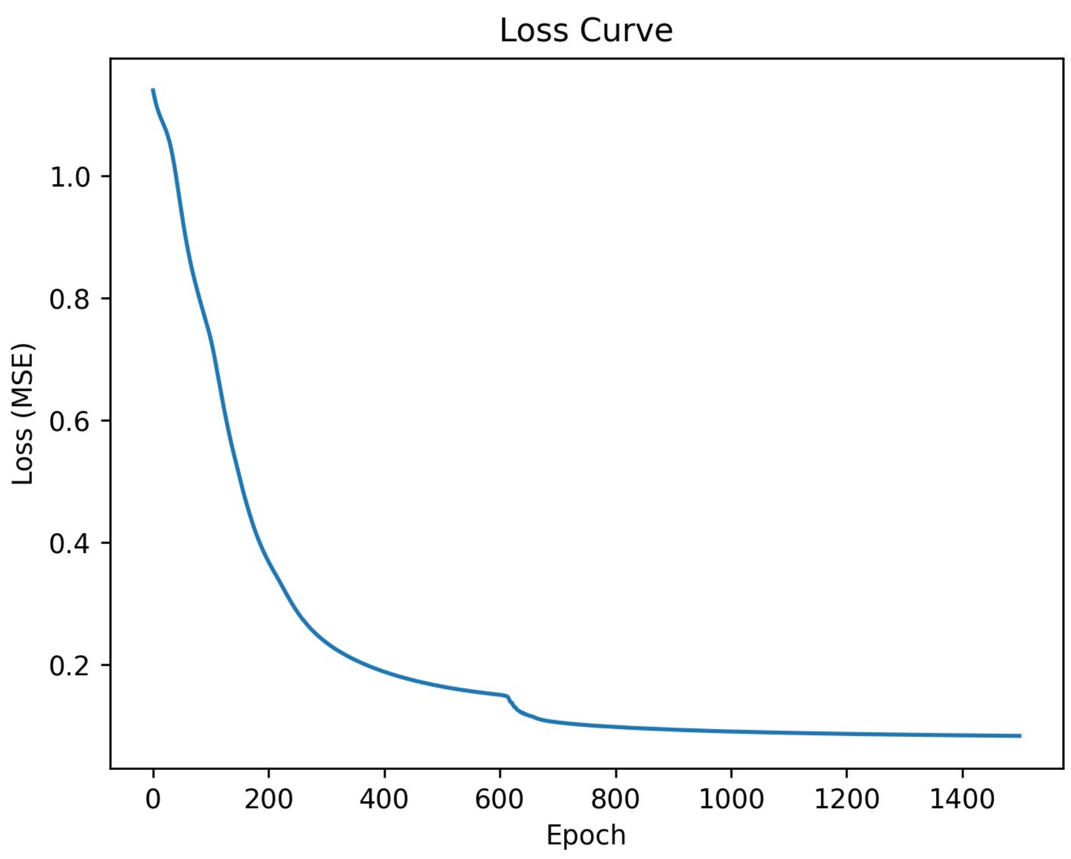
Visualize your trained model

Use the provided `visualizeModel()` function by passing in your trained model to see your models predicted function compared to the provided data

```
# YOUR CODE GOES HERE
# visualize model
visualizeModel(model)

# model accuracy
#model's predictions
y_pred = torch.argmax(model(x), dim=1)
# calculate the accuracy of the model
accuracy = (y_pred == y).float().mean()
accuracy = accuracy * 100
print (f"Model accuracy: {accuracy.item():.2f}%")

# plot loss curve
plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```



```
Model accuracy: 98.00%
```



Discussion

Report the accuracy of your trained model on the generated data. Discuss the structure of your network, including the number and size of hidden layers, choice of activation function, loss function, optimizer, learning rate, number of training epochs.

Model structure and parameters:

```
3 hidden layers with 5 neurons per hidden layer and an output layer
with 3 neurons and a Softmax activation function (suitable for multi-
class problem), Leaky Relu as activation function, Cross-entropy loss
as the loss function for multi-class problems, optimized using the
Adam optimizer with a learning rate of 0.005 for 1500 training epochs.
```

Model performance:

```
The Cross-entropy loss of my trained model on the generated data is
around 0.0886. The model prediction accuracy is around 98%.
```

Model design reasoning and parameter experiments:

The model's activation function and learning rate were determined by trial and error. Several activation functions like ReLu, Sigmoid, Tanh, GELU, and Leaky ReLu were explored, and the Leaky ReLu results in the lowest MSE. I also experimented with varying learning rates to determine the model outcome. The results showed that a smaller learning rate converged faster in our use case and overfitting may occur if not done properly. I also test the number of neurons in each hidden layer. The results appeared to be overfitting the data with a larger number of neurons. Other parameters were referred from M7-L2-P2 and M7-HW1.

M7-L1 Problem 1

In this problem, you will implement a perceptron function that can take in multiple inputs at once as a matrix and output the result of multiplying by a weight matrix and adding a bias vector. Then you will use this function in a loop to implement a multilayer perceptron.

```
import numpy as np
np.set_printoptions(precision=3)
```

Function: `perceptron_layer()`

Complete the function definition for `perceptron_layer(x, weight, bias)`. Inputs:

- `x`: An $N \times n$ matrix of N inputs, each with n features.
- `weight`: An $m \times n$ weight matrix, to be multiplied by the input `x`
- `bias`: A 1-D array of m biases, to be added to the m outputs

Return:

- $N \times m$ output a

a can be obtained by multiplying the weight matrix by the inputs, then adding bias. You must figure out how to make the dimensions work out (e.g. by transposing as necessary) to give the correct size result.

A nonlinear activation would be applied after this function in the context of an MLP, so don't include it in the function. A test case is included for you to check for correctness.

```
def perceptron_layer(x, weight, bias):
    a = np.dot(x, weight.T) + bias
    return a

# Example: N = 3, n = 2, m = 4
x = np.array([[1,2],[3,4],[5,6]])
weight = np.array([[-1.5, -3], [0.5, 1], [1, 1.5], [2, -2]])
bias = np.array([3, -2, .5, -1])
a = perceptron_layer(x, weight, bias)
result = np.array(np.array([[ -4.5,   0.5,   4.5,  -3. ],[-13.5,
3.5,   9.5,  -3. ],[-22.5,   6.5,  14.5,  -3. ]]))

print("Your result", a, sep="\n")
print("Correct result:", result, sep="\n")
```

```
Your result
[[ -4.5   0.5   4.5  -3. ]
 [-13.5   3.5   9.5  -3. ]
 [-22.5   6.5  14.5  -3. ]]
Correct result:
[[ -4.5   0.5   4.5  -3. ]
```

```
 [-13.5   3.5   9.5  -3. ]
 [-22.5   6.5  14.5  -3. ]]
```

Function: `MLP()`

Now by looping through several perceptron layers, you can create a multilayer perceptron (AKA a Neural Network!). Complete the function below to do this. Inputs:

- `x`: An $N \times n$ matrix of N inputs, each with n features.
- `weights`: A list of weight matrices
- `biases`: A list of bias vectors

Return:

- Result of applying each perceptron layer with activation, to the input one-by-one

Apply sigmoid activation (a sigmoid function is given) on all layers EXCEPT the final layer.

A test case is provided for you to check your function.

```
def sigmoid(x):
    return 1./(1.+np.exp(-x))

def MLP(x, weights, biases):
    # YOUR CODE GOES HERE
    for i in range(len(weights)-1):
        x = sigmoid(perceptron_layer(x, weights[i], biases[i]))
    # not applying sigmoid to the last layer
    x = perceptron_layer(x, weights[-1], biases[-1])
    return x

# Example
np.random.seed(0)
dims = [2,6,8,3,1]
weights = []
biases = []
for i,_ in enumerate(dims[:-1]):
    weights.append(np.random.standard_normal([dims[i+1],dims[i]]))
    biases.append(np.random.rand(dims[i+1]))
x = np.random.uniform(-10,10,size=[10,2])

result = np.array([[0.029],[0.267],[0.314],[0.027],[0.319],[0.297],
[0.331],[0.343],[0.187],[0.335]])
y = MLP(x, weights, biases)

print("   Your result: ", y.T, ".T",sep="")
print("Correct result: ", result.T, ".T", sep="")
```

```
   Your result: [[0.029 0.267 0.314 0.027 0.319 0.297 0.331 0.343
0.187 0.335]].T
```

```
Correct result: [[0.029 0.267 0.314 0.027 0.319 0.297 0.331 0.343
0.187 0.335]].T
```

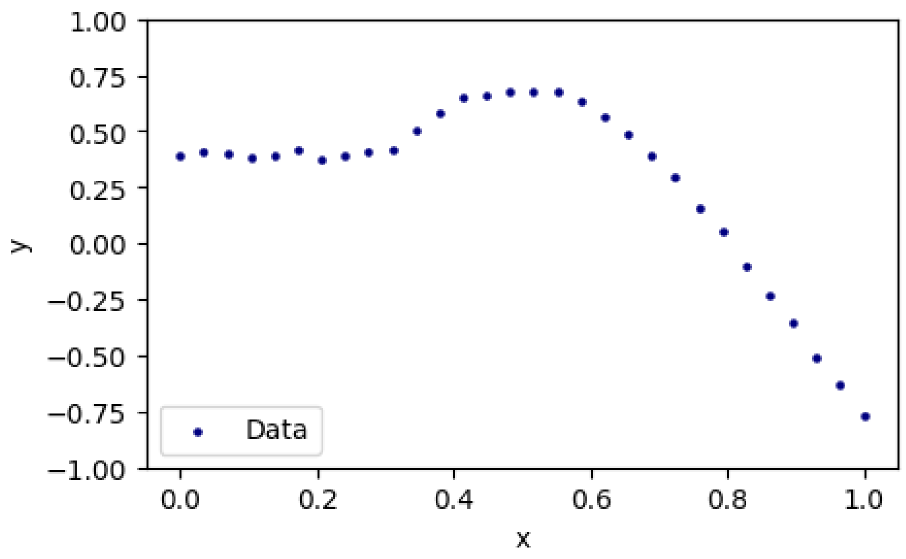
M7-L1 Problem 2

In this problem, you will explore what happens when you change the weights/biases of a neural network.

Neural networks act as functions that attempt to map from input data to output data. In training a neural network, the goal is to find the values of weights and biases that minimize the loss between their output and the desired output. This is typically done with a technique called backpropagation; however, here you will simply note the effect of changing specific weights in the network which has been pre-trained.

First, load the data and initial weights/biases below:

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

x = np.array([0.        , 0.03448276, 0.06896552, 0.10344828,
0.13793103,0.17241379, 0.20689655, 0.24137931, 0.27586207,
0.31034483,0.34482759, 0.37931034, 0.4137931 , 0.44827586,
0.48275862,0.51724138, 0.55172414, 0.5862069 , 0.62068966,
0.65517241,0.68965517, 0.72413793, 0.75862069, 0.79310345,
0.82758621,0.86206897, 0.89655172, 0.93103448, 0.96551724,
1.        ]).reshape(-1,1)
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,
0.394214  ,0.41651437,  0.37573321,  0.39571087,  0.41265936,
0.41953955,0.50596807,  0.58059196,  0.6481607 ,  0.66050901,
0.67741369,0.67348567,  0.67696078,  0.63537378,  0.56446933,
0.48265412,0.39540671,  0.29878237,  0.15893846,  0.05525194, -
0.10070259,-0.23055219, -0.35288448, -0.51317604, -0.63377544, -
0.76849408]).reshape(-1,1)

weights = [np.array([[-5.90378086,  0,  0 ]]).T,
           np.array([[ 0.8996511 ,  4.75805319, -0.95266992],[-
0.99667812, -0.89303165,  3.19020423],[-1.65213421, -2.93268438,
2.61332701]]).T,
           np.array([[ 1.71988943, -1.56198034, -3.31173131]])]

biases = [np.array([ 2.02112296, -3.47589349, -1.11586831]),
np.array([ 1.35350721, -0.11181542, -4.0283719 ]),
np.array([0.51626399])]

plt.figure(figsize=(5,3))
plt.scatter(x,y,s=5,c="navy",label="Data")
plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



MLP Function

Copy in your MLP function (and all necessary helper functions) below. Make sure it is called `MLP()`. In this case, you can plug in `x`, `weights`, and `biases` to try and predict `y`. Make sure you use the sigmoid activation function after each layer (except the final layer).

```python
# YOUR CODE GOES HERE
# perceptron layer function
def perceptron_layer(x, weight, bias):
    a = np.dot(x, weight.T) + bias
    return a

# sigmoid function
def sigmoid(x):
    return 1./(1.+np.exp(-x))

# MLP function
def MLP(x, weights, biases):
    # YOUR CODE GOES HERE
    for i in range(len(weights)-1):
        x = sigmoid(perceptron_layer(x, weights[i], biases[i]))
    # not applying sigmoid to the last layer
    x = perceptron_layer(x, weights[-1], biases[-1])
    return x
```

Varying weights

The provided network has 2 hidden layers, each with 3 neurons. The weights and biases are shown below. Note the weights w_a and w_b -- these are left for you to investigate:

$$ \underline{x; (N\times 1)} \rightarrow \sigma\left(w = \begin{bmatrix} -5.9\ \ boldsymbol{w_a}\ \boldsymbol{w_b}\ \end{bmatrix}

;b = \begin{bmatrix} 2.02\ -3.48\ -1.12\ \end{bmatrix}'

\right)\rightarrow

\underline{(N\times 3)} \rightarrow \sigma\left(w = \begin{bmatrix} 0.9 & -1. & -1.65\ 4.76 & -0.89 & -2.93\ -0.95 & 3.19 & 2.61\ \end{bmatrix}

;b = \begin{bmatrix} 1.35\ -0.11\ -4.03\ \end{bmatrix}'

\right)\rightarrow

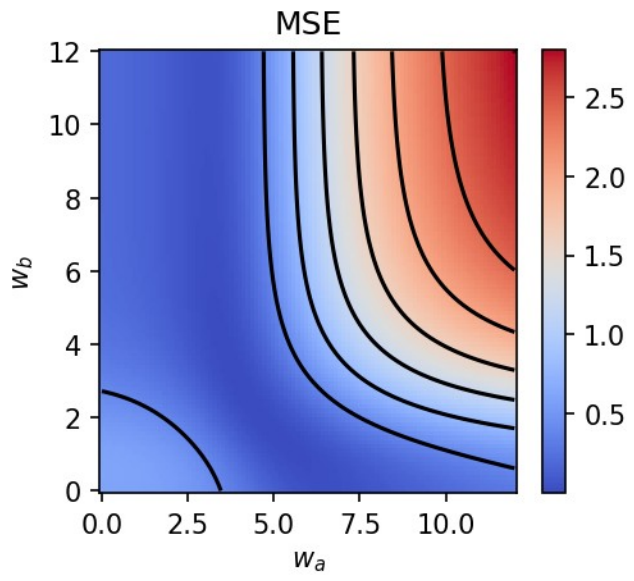
\underline{(N\times 3)} \rightarrow \sigma\left(w = \begin{bmatrix} 1.72\ -1.56\ -3.31\ \ end{bmatrix}'

;b = \begin{bmatrix} 0.52\ \end{bmatrix}'

\right)\rightarrow \underline{\hat{y}; (N\times 1)} $$

We can compute the MSE for each combination of $\left(w_a, w_b\right)$ to see where MSE is minimized.

```python
def MSE(y, pred):
    return np.mean((y.flatten()-pred.flatten())**2)

vals = np.linspace(0,12,100)
was, wbs = np.meshgrid(vals,vals)
mses = np.zeros_like(was.flatten())

for i in range(len(was.flatten())):
    ws, bs = weights.copy(), biases.copy()
    ws[0][1,0] = was.flatten()[i]
    ws[0][2,0] = wbs.flatten()[i]
    mses[i] = MSE(y, MLP(x, ws, bs))
mses = mses.reshape(was.shape)

plt.figure(figsize = (3.5,3),dpi=150)
plt.title("MSE")
plt.contour(was,wbs,mses,colors="black")
plt.pcolormesh(was,wbs,mses,shading="nearest",cmap="coolwarm")
plt.xlabel("$w_a$")
plt.ylabel("$w_b$")
plt.colorbar()
plt.show()
```



```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual,
Layout, FloatSlider, Dropdown


def plot(wa, wb):
    ws, bs = weights.copy(), biases.copy()
    ws[0][1,0] = wa
    ws[0][2,0] = wb

    xs = np.linspace(0,1)
    ys = MLP(xs.reshape(-1,1), ws, bs)


    plt.figure(figsize=(10,4),dpi=120)

    plt.subplot(1,2,1)
    plt.contour(was,wbs,mses,colors="black")
    plt.pcolormesh(was,wbs,mses,shading="nearest",cmap="coolwarm")
    plt.title(f"$w_a = {wa:.1f}$;  $w_b = {wb:.1f}$")
    plt.xlabel("$w_a$")
    plt.ylabel("$w_b$")
    plt.scatter(wa,wb,marker="*",color="black")
    plt.colorbar()

    plt.subplot(1,2,2)
    plt.scatter(x,y,s=5,c="navy",label="Data")
    plt.plot(xs,ys,"r-",linewidth=1,label="MLP")
    plt.title(f"MSE = {MSE(y, MLP(x, ws, bs)):.3f}")
    plt.legend(loc="lower left")
    plt.ylim(-1,1)
    plt.xlabel("x")
    plt.ylabel("y")

    plt.show()


slider1 = FloatSlider(
    value=0,
    min=0,
    max=12,
    step=.5,
    description='wa',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)

slider2 = FloatSlider(
    value=0,
    min=0,
    max=12,
    step=.5,
    description='wb',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)



interactive_plot = interactive(
    plot,
    wa = slider1,
    wb = slider2
    )
output = interactive_plot.children[-1]
output.layout.height = '500px'

interactive_plot
```

{"model_id":"d67e866732c2403286652e39ffea65e9","version_major":2,"version_minor":0}

Questions

1. For $w_a = 4.0$, what walue of w_b gives the lowest MSE (to the nearest 0.5)?

- *ANSWER:* For $w_a = 4.0$, w_b of 3.0 gives the lowest MSE.

1. For the large values of w_a and w_b, describe the MLP's predictions.

- *ANSWER:* For the large values of w_a and w_b, the MLP's predictions before less and less accurate with larger MSE values.

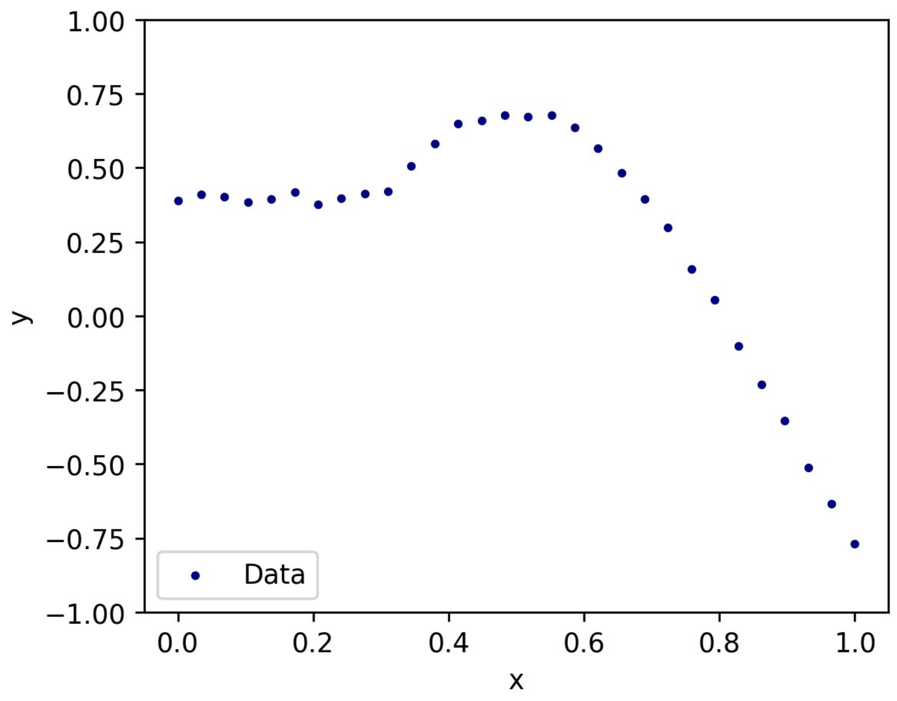
M7-L2 Problem 1

In this function you will:

- Learn to use SciKit-Learn's `MLPRegressor()` model
- Look at the loss curve of an sklearn neural network
- Try out multiple activation functions

First, load the data in the following cell. This is the same data from M7-L1-P2

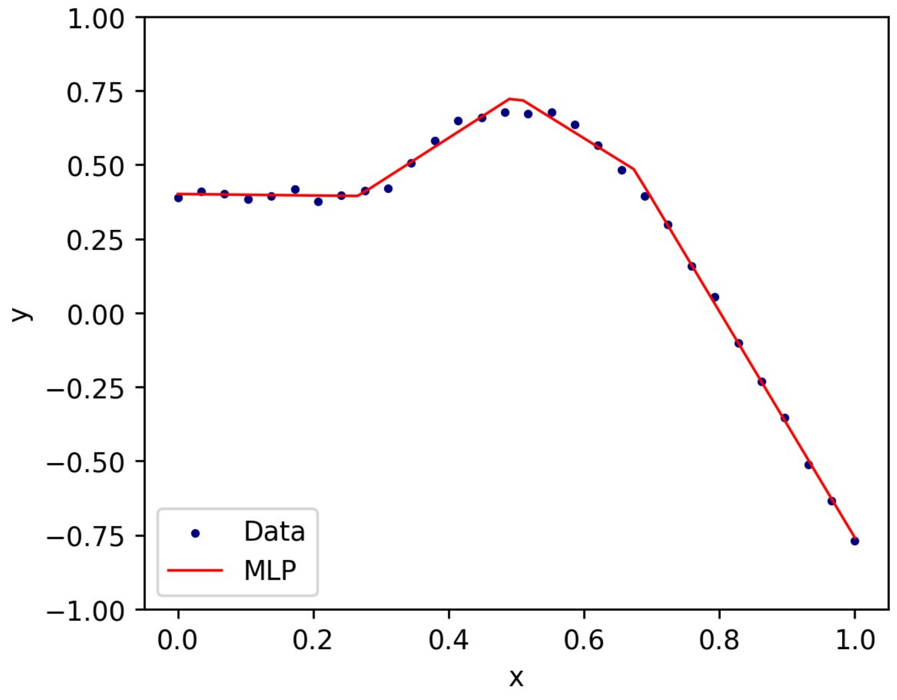
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor

x = np.array([0.        , 0.03448276, 0.06896552, 0.10344828,
0.13793103,0.17241379, 0.20689655, 0.24137931, 0.27586207,
0.31034483,0.34482759, 0.37931034, 0.4137931 , 0.44827586,
0.48275862,0.51724138, 0.55172414, 0.5862069 , 0.62068966,
0.65517241,0.68965517, 0.72413793, 0.75862069, 0.79310345,
0.82758621,0.86206897, 0.89655172, 0.93103448, 0.96551724,
1.        ]).reshape(-1,1)
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,
0.394214  ,0.41651437,  0.37573321,  0.39571087,  0.41265936,
0.41953955,0.50596807,  0.58059196,  0.6481607 ,  0.66050901,
0.67741369,0.67348567,  0.67696078,  0.63537378,  0.56446933,
0.48265412,0.39540671,  0.29878237,  0.15893846,  0.05525194, -
0.10070259,-0.23055219, -0.35288448, -0.51317604, -0.63377544, -
0.76849408])


plt.figure(figsize=(5,4),dpi=250)
plt.scatter(x,y,s=5,c="navy",label="Data")
plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



```
MLPRegressor()
```

Here, we create a simple MLP Regressor in sklearn and plot the results. The model is created and fitted in the same way as any other sklearn model. We choose hidden layer sizes 10,10. Note that our input and output are both 1-D, but we don't need to specify this at initialization.

```
mlp = MLPRegressor(hidden_layer_sizes=[10,10], max_iter=10000, tol=1e-
10) # Tune here
mlp.fit(x, y)

xs = np.linspace(0,1)
ys = mlp.predict(xs.reshape(-1,1))

plt.figure(figsize=(5,4),dpi=250)
plt.scatter(x,y,s=5,c="navy",label="Data")
plt.plot(xs,ys,"r-",linewidth=1,label="MLP")
plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Tuning training hyperparameters

Chances are, the model above did a poor job fitting the data. Try changing the following parameters when initializing the `MLPRegressor` in the cell above:

- `max_iter` (this will need to be very large)
- `tol` (this will need to be very small)

You can read about what these do at https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

Question

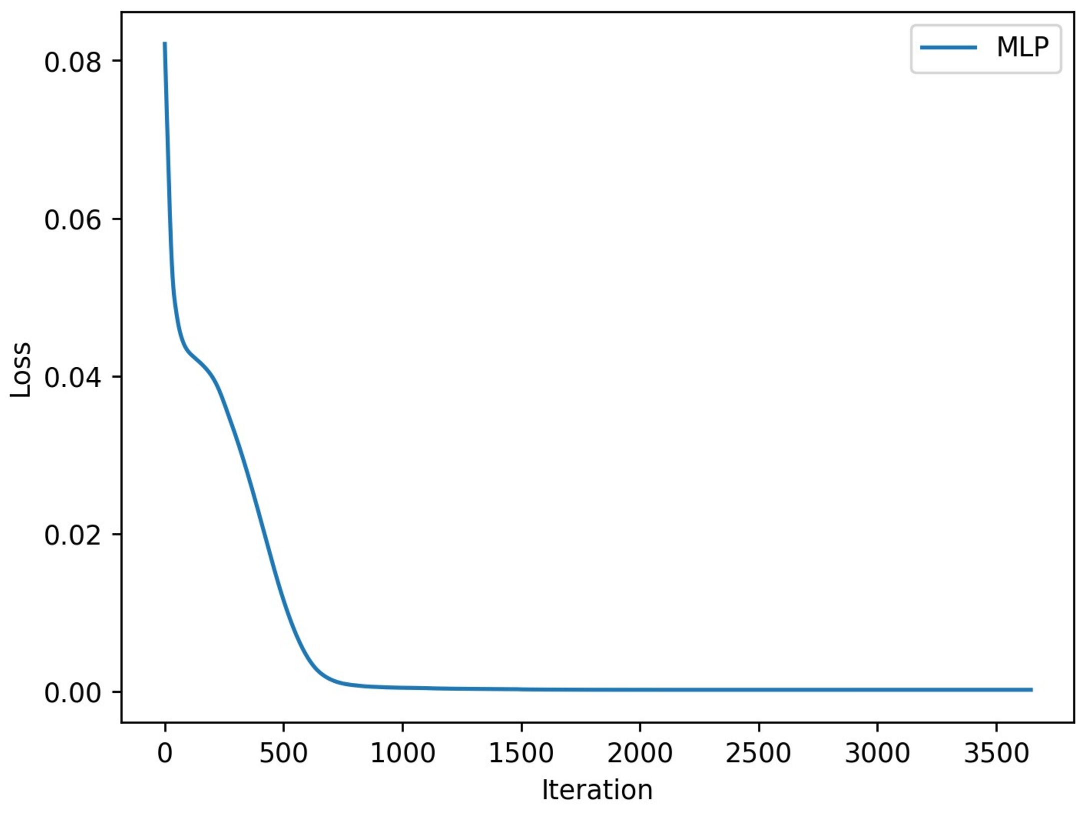
1. What values of `max_iter` and `tol` gave you a reasonable fit?

   `max_iter` = 10000 and `tol` = 1e-10 gave a reasonable fit.

Loss Curve

We can look at the loss curve by accessing `mlp.loss_curve_`. Let's plot this below:

```
loss = mlp.loss_curve_
plt.figure(dpi=250)
plt.plot(loss,label="MLP")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Activation Functions

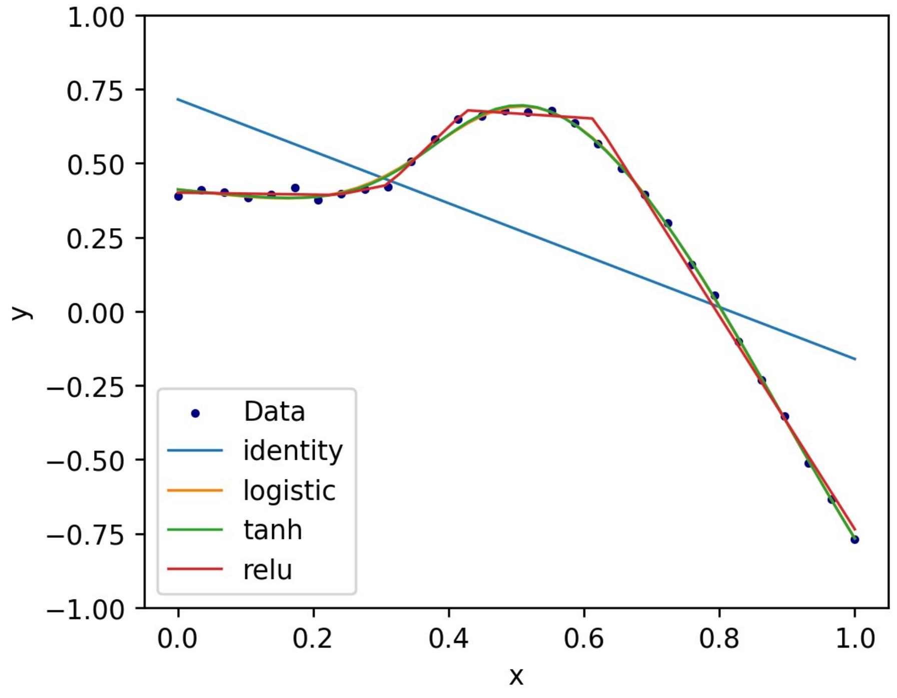
Sklearn provides the following activation functions:

- `"identity"` (This is a linear function, it should not give good results)
- `"logistic"` (We call this 'sigmoid', although both this and tanh are sigmoid functions)
- `"tanh"`
- `"relu"`

Run the following cell to train a model on each. They can be accessed via, for example: `models["relu"]` for the relu activation model

```
activations = ["identity","logistic","tanh","relu"]
models = dict()

for act in activations:
    model = MLPRegressor([10,10],random_state=50,
activation=act,max_iter=100000,tol=1e-11)
    model.fit(x,y)
    models[act] = model

xs = np.linspace(0,1)
plt.figure(figsize=(5,4),dpi=250)
plt.scatter(x,y,s=5,c="navy",label="Data")
```

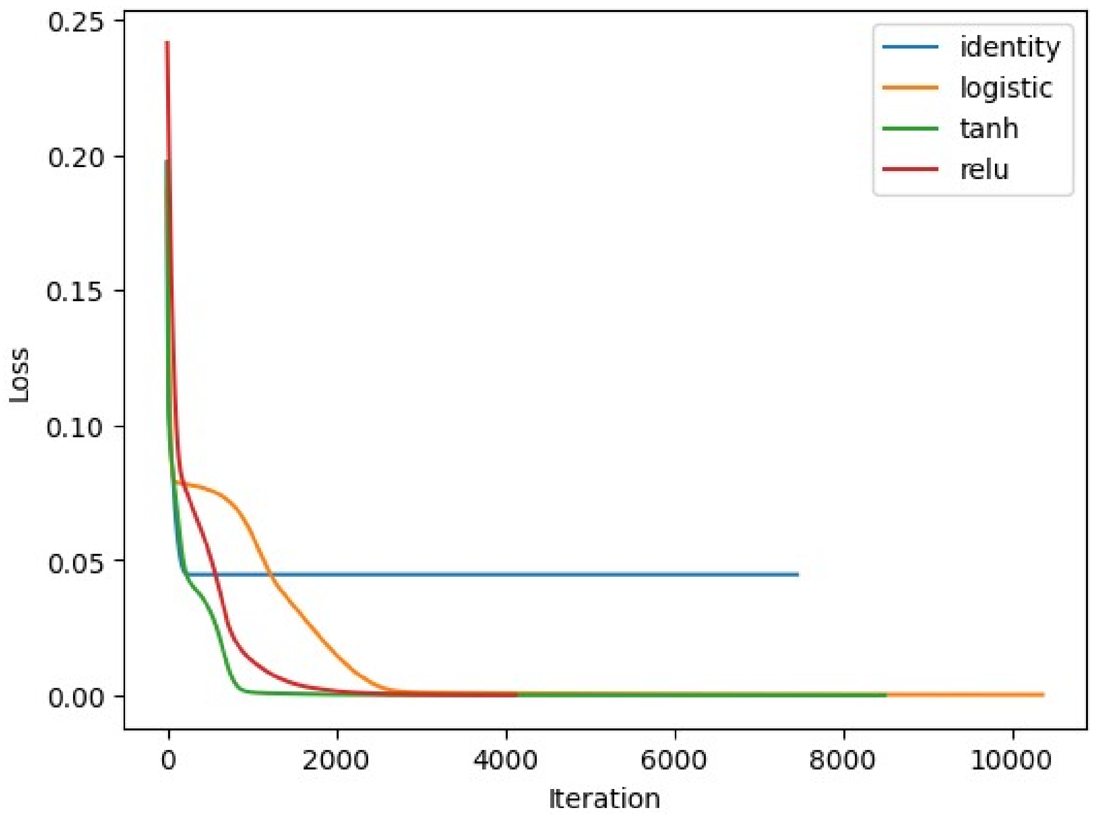
```
for act in activations:
    model = models[act]
    ys = model.predict(xs.reshape(-1,1))
    plt.plot(xs,ys,linewidth=1,label=act)

plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Loss curves

Now, create another loss curve plot, but this time, include all four MLP models with a legend indicating which activation function corresponds to each curve.

```
# YOUR CODE GOES HERE
for act in activations:
    model = models[act]
    loss = model.loss_curve_
    plt.plot(loss,label=act)

plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Questions

1. Which activation functions produced a good fit?

   The logistic, tanh, and relu functions produced a good fit.

2. Which activation function's model converged the "slowest"?

   The logistic activation function model converged the slowest.

3. Of the networks that fit well, which activation function's model converged the "fastest"?

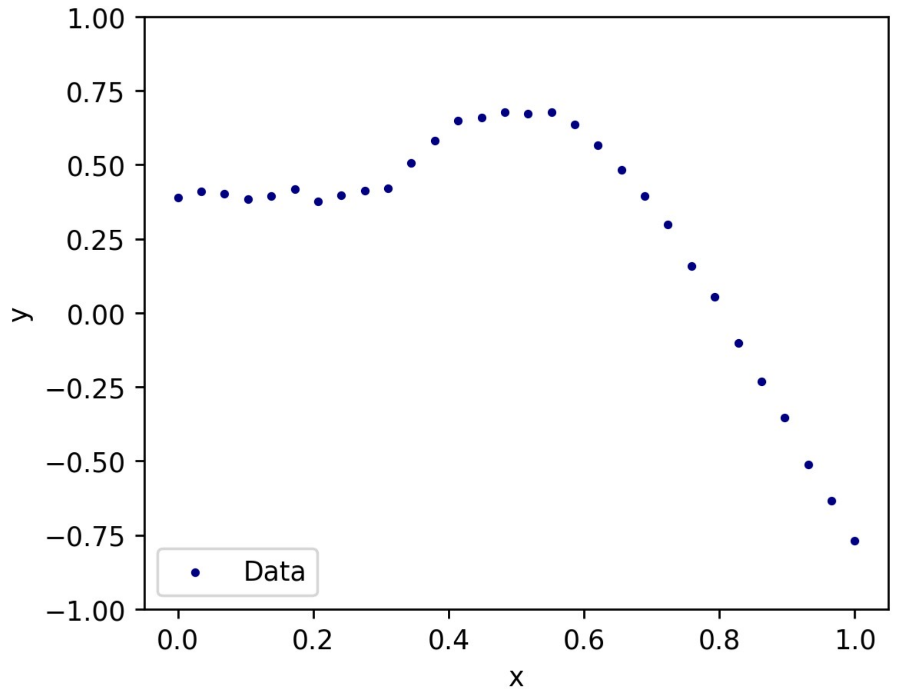
   Of the networks that fit well, the tanh activation funvtion model converged the fastest.

M7-L2 Problem 2

Here you will create a simple neural network for regression in PyTorch. PyTorch will give you a lot more control and flexibility for neural networks than SciKit-Learn, but there are some extra steps to learn.

Run the following cell to load our 1-D dataset:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import optim, nn
import torch.nn.functional as F

x = np.array([0.        , 0.03448276, 0.06896552, 0.10344828,
0.13793103,0.17241379, 0.20689655, 0.24137931, 0.27586207,
0.31034483,0.34482759, 0.37931034, 0.4137931 , 0.44827586,
0.48275862,0.51724138, 0.55172414, 0.5862069 , 0.62068966,
0.65517241,0.68965517, 0.72413793, 0.75862069, 0.79310345,
0.82758621,0.86206897, 0.89655172, 0.93103448, 0.96551724,
1.        ]).reshape(-1,1)
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,
0.394214  ,0.41651437,  0.37573321,  0.39571087,  0.41265936,
0.41953955,0.50596807,  0.58059196,  0.6481607 ,  0.66050901,
0.67741369,0.67348567,  0.67696078,  0.63537378,  0.56446933,
0.48265412,0.39540671,  0.29878237,  0.15893846,  0.05525194, -
0.10070259,-0.23055219, -0.35288448, -0.51317604, -0.63377544, -
0.76849408]).reshape(-1,1)


plt.figure(figsize=(5,4),dpi=250)
plt.scatter(x,y,s=5,c="navy",label="Data")
plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



PyTorch Tensors

PyTorch models only work with PyTorch Tensors, so we need to convert our dataset into a tensors.

To convert these back to numpy arrays we can use:

- `x.detach().numpy()`
- `y.detach().numpy()`

```
x = torch.Tensor(x)
y = torch.Tensor(y)
```

PyTorch Module

We create a subclass whose superclass is `nn.Module`, a basic predictive model, and we must define 2 methods.

`nn.Module` subclass:

- `__init__()`
  - runs when creating a new model instance
  - includes the line `super().__init__()` to inherit parent methods from `nn.Module`
  - sets up all necessary model components/parameters
- `forward()`
  - runs when calling a model instance

- performs a forward pass through the network given an input tensor.

This class `Net_2_layer` is an MLP for regression with 2 layers. At initialization, the user inputs the number of hidden neurons per layer, the number of inputs and outputs, and the activation function.

```
class Net_2_layer(nn.Module):
    def __init__(self, N_hidden=6, N_in=1, N_out=1, activation =
F.relu):
        super().__init__()
        # Linear transformations -- these have weights and biases as
trainable parameters,
        # so we must create them here.
        self.lin1 = nn.Linear(N_in, N_hidden)
        self.lin2 = nn.Linear(N_hidden, N_hidden)
        self.lin3 = nn.Linear(N_hidden, N_out)
        self.act = activation

    def forward(self,x):
        x = self.lin1(x)
        x = self.act(x)  # Activation of first hidden layer
        x = self.lin2(x)
        x = self.act(x)  # Activation at second hidden layer
        x = self.lin3(x) # (No activation at last layer)

        return x
```

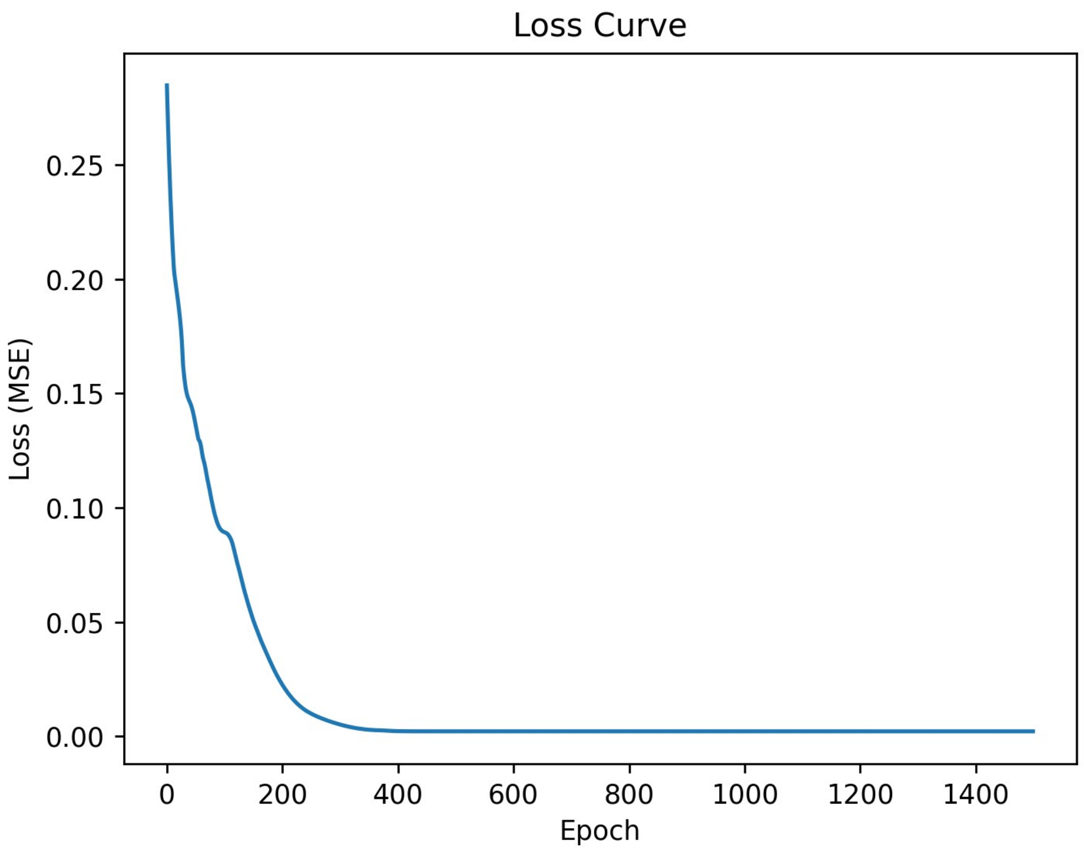
Instantiate a model

This model has 6 neurons at each hidden layer, and it uses ReLU activation.

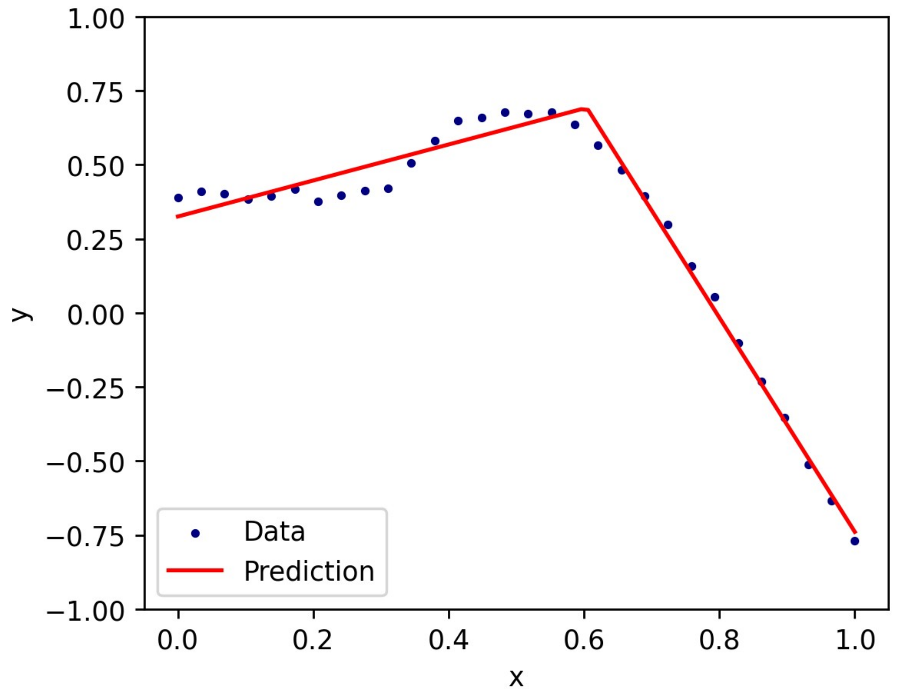
```
model = Net_2_layer(N_hidden = 6, activation = F.relu)
loss_curve = []
```

Training a model

```
# Training parameters: Learning rate, number of epochs, loss function
# (These can be tuned)
lr = 0.005
epochs = 1500
loss_fcn = F.mse_loss

# Set up optimizer to optimize the model's parameters using Adam with
the selected learning rate
opt = optim.Adam(params = model.parameters(), lr=lr)

# Training loop
for epoch in range(epochs):
    out = model(x) # Evaluate the model
    loss = loss_fcn(out,y) # Calculate the loss -- error between 
network prediction and y

    loss_curve.append(loss.item())

    # Print loss progress info 25 times during training
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch} of {epochs}... \tAverage loss: 
{loss.item()}")

    # Move the model parameters 1 step closer to their optima:
    opt.zero_grad()
    loss.backward()
    opt.step()

Epoch 0 of 1500...    Average loss: 0.2846282422542572
Epoch 60 of 1500...   Average loss: 0.12524329125881195
Epoch 120 of 1500...  Average loss: 0.07749786227941513
Epoch 180 of 1500...  Average loss: 0.03207949176430702
Epoch 240 of 1500...  Average loss: 0.011363743804395199
Epoch 300 of 1500...  Average loss: 0.0051547870971262455
Epoch 360 of 1500...  Average loss: 0.0027225306257605553
Epoch 420 of 1500...  Average loss: 0.0022241221740841866
Epoch 480 of 1500...  Average loss: 0.0022069227416068316
Epoch 540 of 1500...  Average loss: 0.0022069131955504417
Epoch 600 of 1500...  Average loss: 0.002206912962719798
Epoch 660 of 1500...  Average loss: 0.002206910867244005
Epoch 720 of 1500...  Average loss: 0.0022069106344133615
Epoch 780 of 1500...  Average loss: 0.0022069106344133615
Epoch 840 of 1500...  Average loss: 0.0022069106344133615
Epoch 900 of 1500...  Average loss: 0.0022069106344133615
Epoch 960 of 1500...  Average loss: 0.0022069106344133615
Epoch 1020 of 1500...      Average loss: 0.002206911100074649
Epoch 1080 of 1500...      Average loss: 0.002206911100074649
Epoch 1140 of 1500...      Average loss: 0.002206911100074649
Epoch 1200 of 1500...      Average loss: 0.002206911565735936
Epoch 1260 of 1500...      Average loss: 0.002206911565735936
Epoch 1320 of 1500...      Average loss: 0.002206911565735936
Epoch 1380 of 1500...      Average loss: 0.002206911565735936
Epoch 1440 of 1500...      Average loss: 0.002206911565735936

plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```



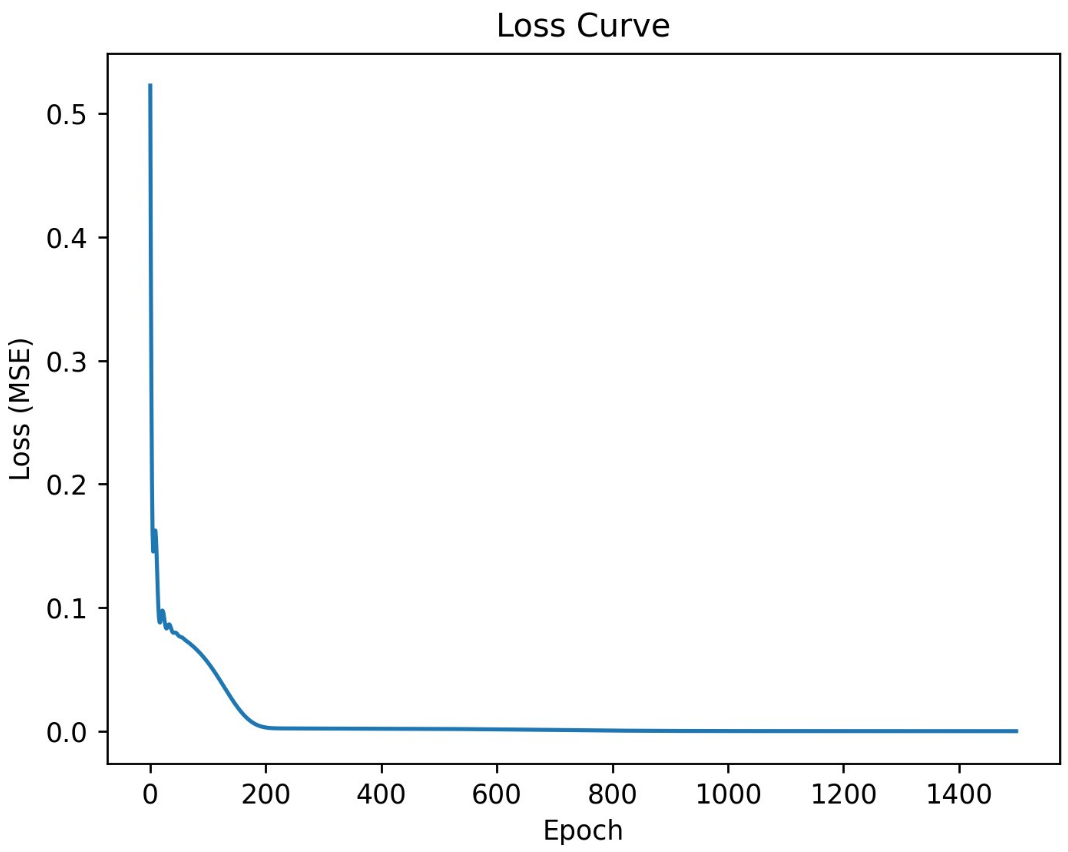
```
xs = torch.linspace(0,1,100).reshape(-1,1)
ys = model(xs)

plt.figure(figsize=(5,4),dpi=250)
plt.scatter(x,y,s=5,c="navy",label="Data")
plt.plot(xs.detach().numpy(),
ys.detach().numpy(),"r-",label="Prediction")
plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



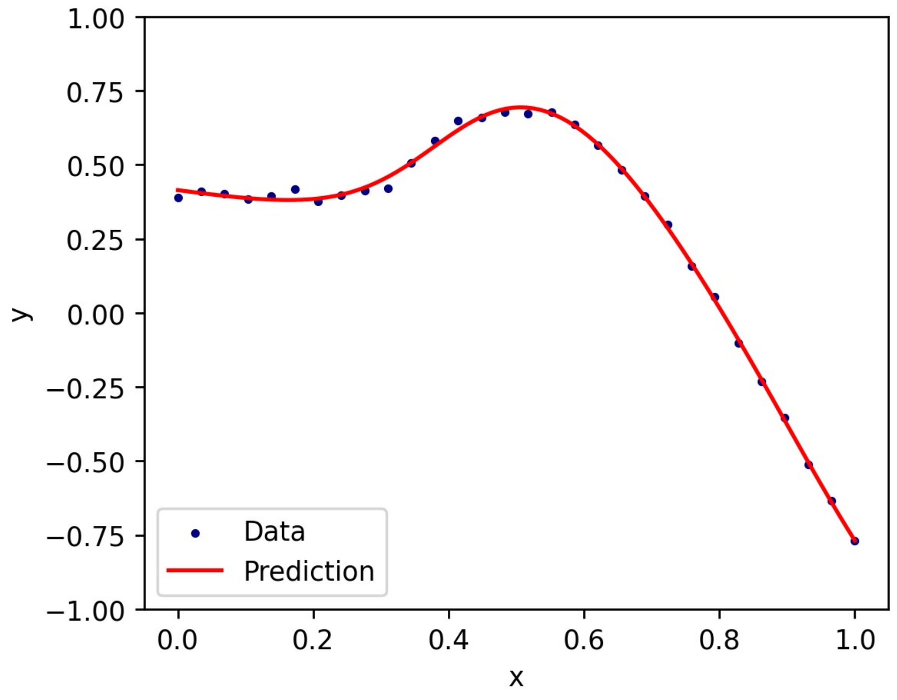
Your Turn

In the cells below, create a new instance of `Net_2_layer`. This time, use 20 neurons per hidden layer, and an activation of `F.tanh`. Plot the loss curve and a visualization of the prediction with the data.

```
# YOUR CODE GOES HERE
# 20 neruons per hidden layer, and an activation function of F.tanh
model1 = Net_2_layer(N_hidden = 20, activation = F.tanh)
loss_curve = []

# Training parameters: Learning rate, number of epochs, loss function
lr = 0.005
epochs = 1500
loss_fcn = F.mse_loss

# Set up optimizer to optimize the model's parameters using Adam with
the selected learning rate
opt = optim.Adam(params = model1.parameters(), lr=lr)

# Training loop
for epoch in range(epochs):
    out = model1(x) # Evaluate the model
    loss = loss_fcn(out,y) # Calculate the loss -- error between
network prediction and y

    loss_curve.append(loss.item())
    # Print loss progress info 25 times during training
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch} of {epochs}... \tAverage loss:
{loss.item()}")

    # Move the model parameters 1 step closer to their optima:
    opt.zero_grad()
    loss.backward()
    opt.step()

Epoch 0 of 1500...     Average loss: 0.5223419666290283
Epoch 60 of 1500...    Average loss: 0.07408086210489273
Epoch 120 of 1500...   Average loss: 0.04191172122955322
Epoch 180 of 1500...   Average loss: 0.006381927989423275
Epoch 240 of 1500...   Average loss: 0.002428209176287055
Epoch 300 of 1500...   Average loss: 0.002365131163969636
Epoch 360 of 1500...   Average loss: 0.0022913666907697916
Epoch 420 of 1500...   Average loss: 0.002200573915615678
Epoch 480 of 1500...   Average loss: 0.0020795234013348818
Epoch 540 of 1500...   Average loss: 0.0019061658531427383
Epoch 600 of 1500...   Average loss: 0.0016590988961979747
Epoch 660 of 1500...   Average loss: 0.001347280340269208
Epoch 720 of 1500...   Average loss: 0.0010233799694105983
Epoch 780 of 1500...   Average loss: 0.0007491824799217284
Epoch 840 of 1500...   Average loss: 0.0005512729403562844
Epoch 900 of 1500...   Average loss: 0.0004194226348772645
Epoch 960 of 1500...   Average loss: 0.0003373590006958693
Epoch 1020 of 1500...       Average loss: 0.00029409301350824535
Epoch 1080 of 1500...       Average loss: 0.00027269343263469636
Epoch 1140 of 1500...       Average loss: 0.00026076758513227105
Epoch 1200 of 1500...       Average loss: 0.00025270588230341673
Epoch 1260 of 1500...       Average loss: 0.0002462092961650342
Epoch 1320 of 1500...       Average loss: 0.0002403251564828679
Epoch 1380 of 1500...       Average loss: 0.0002346175751881674
Epoch 1440 of 1500...       Average loss: 0.0002288554678671062

plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```



```
xs = torch.linspace(0,1,100).reshape(-1,1)
ys = model1(xs)

plt.figure(figsize=(5,4),dpi=250)
plt.scatter(x,y,s=5,c="navy",label="Data")
plt.plot(xs.detach().numpy(),
ys.detach().numpy(),"r-",label="Prediction")
plt.legend(loc="lower left")
plt.ylim(-1,1)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0.0
0.2
0.4
0.6
0.8
1.0
x
Data
Prediction