Problem 1:

$$a = 3 * (-6) + (-2) * 2 + 8 * 5 + 10 * (-4) + (-6) * (-2) + 1 * 7$$
$$a = -3$$
$$y = f(a) = \frac{1}{1 + e^{a}} = 0.95257$$
$$\boldsymbol{y = 0.95257}$$

Problem 2:

1. W_3

Problem 3:

1. 2

Problem 4:

1. 4

M7-HW1

October 30, 2023

1 Problem 1

1.1 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:

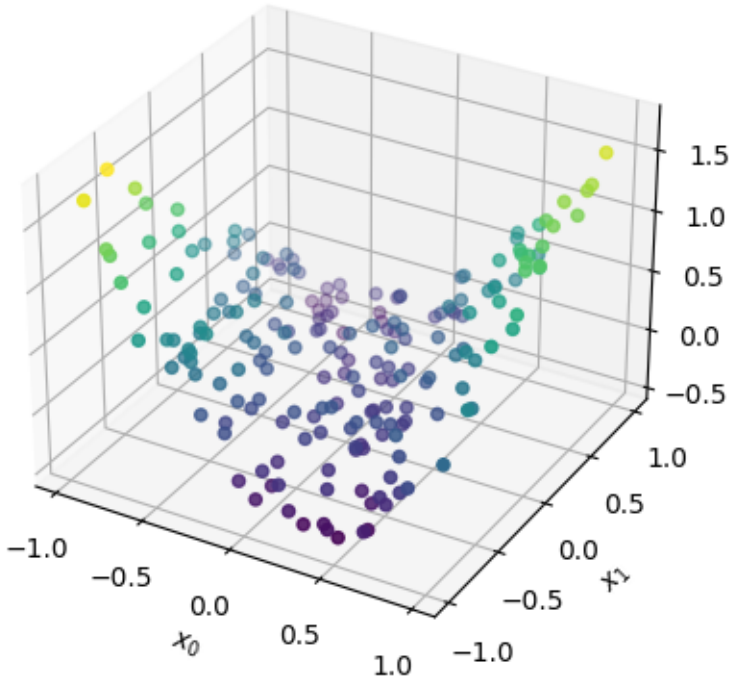
[ ]:
```
import torch
import torch.nn as nn
from torch import optim, nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

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()
```

1.2 Generate and visualize the data

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

[ ]:
```
x, y = dataGen()

fig,ax = plt.subplots(subplot_kw={"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()
```



1.3 Create and train a neural network using PyTorch

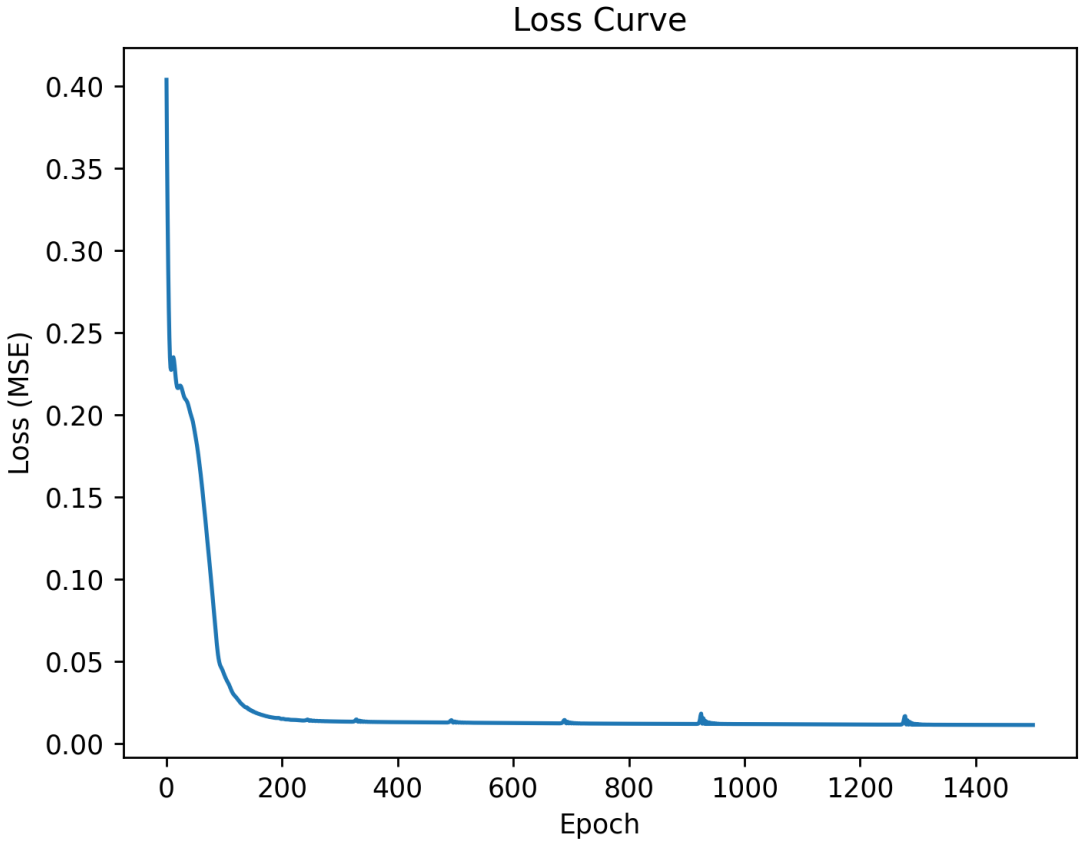
Choice of structure and training parameters are entirely up to you, however you will need to provide reasoning for your choices. An MSE smaller than 0.02 is reasonable.

[ ]:
```
class NNet(nn.Module):
    def __init__(self, N_hidden=6, N_in=2, N_out=1, activation = F.relu):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(N_in, N_hidden),
            nn.Tanh(),
            nn.Linear(N_hidden, N_hidden),
            nn.Tanh(),
            nn.Linear(N_hidden, N_hidden),
            nn.Tanh(),
            nn.Linear(N_hidden, N_hidden),
            nn.Tanh(),
            nn.Linear(N_hidden, N_out)
        )

    def forward(self,x):
        return self.seq(x)
```

```
x = torch.Tensor(x)
y = torch.Tensor(y.reshape(-1,1))

model = NNet(N_hidden = 10)
loss_curve = []

lr = 0.005
epochs = 1500
loss_fcn = nn.MSELoss()

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)

    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()}")

    opt.zero_grad()
    loss.backward()
    opt.step()

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

```
Epoch 0 of 1500…      Average loss: 0.40396177768707275
Epoch 60 of 1500…     Average loss: 0.16132377088069916
Epoch 120 of 1500…    Average loss: 0.02841203287243843
Epoch 180 of 1500…    Average loss: 0.016156360507011414
Epoch 240 of 1500…    Average loss: 0.014165695756673813
Epoch 300 of 1500…    Average loss: 0.013476306572556496
Epoch 360 of 1500…    Average loss: 0.013207373209297657
Epoch 420 of 1500…    Average loss: 0.013016367331147194
Epoch 480 of 1500…    Average loss: 0.012853248976171017
Epoch 540 of 1500…    Average loss: 0.012704039923846722
Epoch 600 of 1500…    Average loss: 0.012557604350149632
Epoch 660 of 1500…    Average loss: 0.012407958507537842
Epoch 720 of 1500…    Average loss: 0.012280775234103203
Epoch 780 of 1500…    Average loss: 0.01217031478881836
```

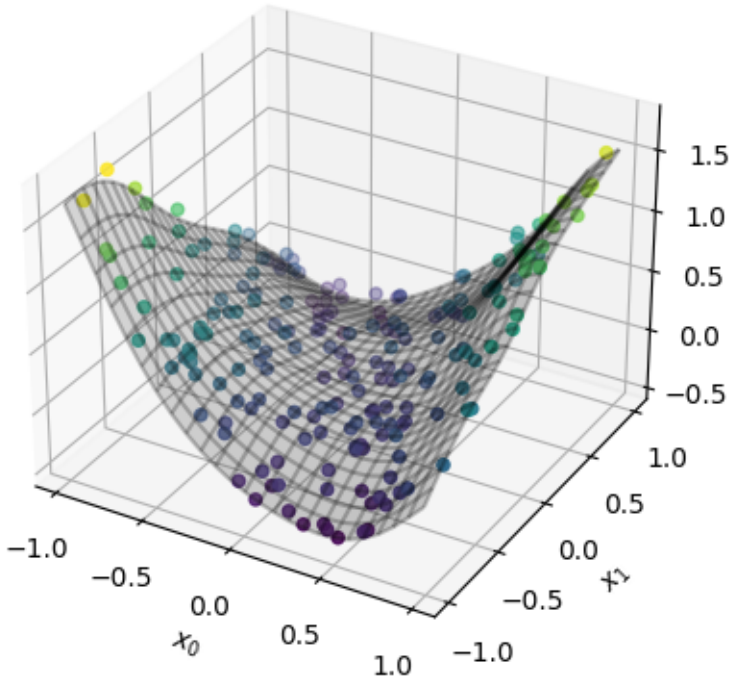
```
Epoch 840 of 1500…    Average loss: 0.012086634524166584
Epoch 900 of 1500…    Average loss: 0.01201742421835661
Epoch 960 of 1500…    Average loss: 0.012036304920911789
Epoch 1020 of 1500…   Average loss: 0.01190644409507513
Epoch 1080 of 1500…   Average loss: 0.011832568794488907
Epoch 1140 of 1500…   Average loss: 0.011747300624847412
Epoch 1200 of 1500…   Average loss: 0.011655702255666256
Epoch 1260 of 1500…   Average loss: 0.011568091809749603
Epoch 1320 of 1500…   Average loss: 0.011555836535990238
Epoch 1380 of 1500…   Average loss: 0.011462978087365627
Epoch 1440 of 1500…   Average loss: 0.011418207548558712
```



1.4 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

[ ]:
```
visualizeModel(model)
```



1.5 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.

Final MSE: 0.011418207548558712

The number of nodes per layer was chosen by increasing the number until there wasn't a significant increase in performance. With this, the lowest number of nodes per layer was chosen to ensure faster training. A similar logic was applied to chosing the number of hidden layers. I went with 5 hidden layers to improve the accuracy of the fit without overfitting choosing a higher number like 10 or 15. In addition, 5 seemed to produce the best results. Tanh was chosen as an activation function because it has a fast convergence with low loss. It also produced smoother results in the output because of it's continuous nature as compared to relu. The mse loss function was used because it is a pretty standard mse function for regression problems like this one. It provides an overview of how well our network approximates as a function the data which is exactly what we are trying to accomplish with this regression problem. The Adam optimizer was chosen to hopefully help improve the speed of training. Since the learning rate dynamically adjusts, the hope was that it would produce a more accurate result than traditional stochastic gradient descent and also take more optimal steps to get there. The learning rate was chosen after a few iterations to optimize the speed of training while also reducing the final loss as much as possible. Number of epochs was chosen because it reduced the loss function to a point where it wouldn't reduce by a significant enough factor if training continued.

M7-HW2

October 30, 2023

1 Problem 2

1.1 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:

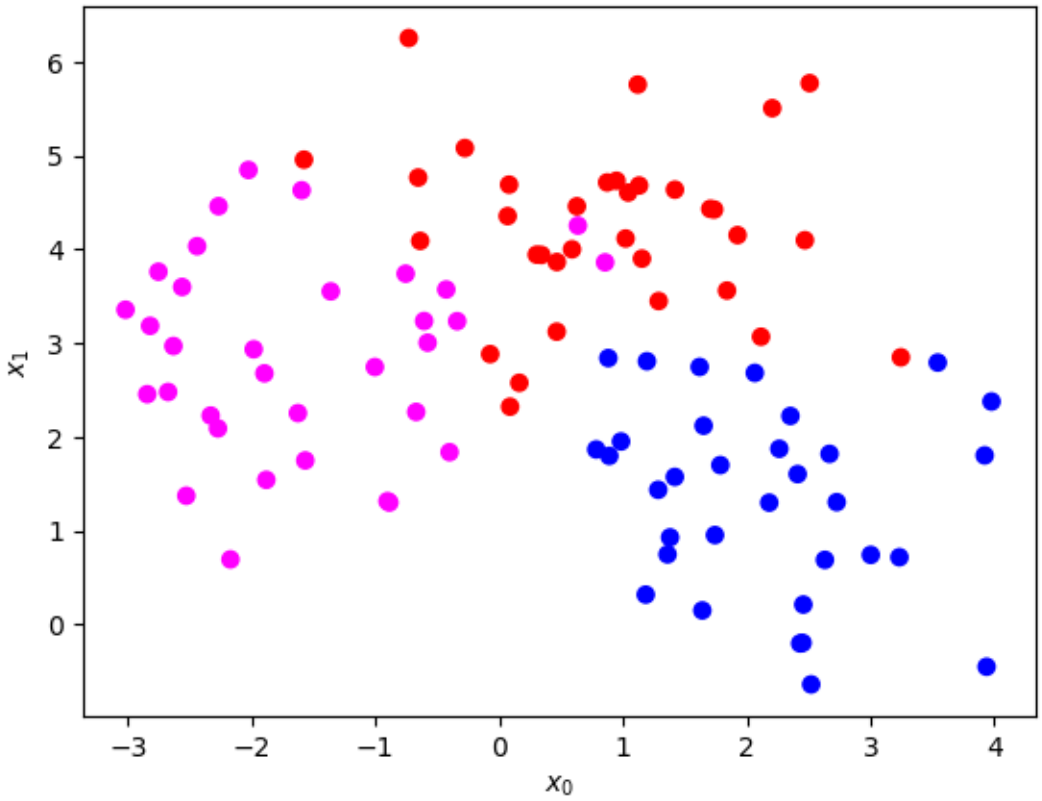
[ ]:
```
import torch
import torch.nn as nn
from torch import optim, nn
import torch.nn.functional as F
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

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()
```

1.2 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.

[ ]:
```
x, y = dataGen()
def getArray(index):
    arr = np.zeros(3)
    arr[index] = 1
    return arr
y_new = np.array([getArray(i) for i in y])

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



1.3 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.

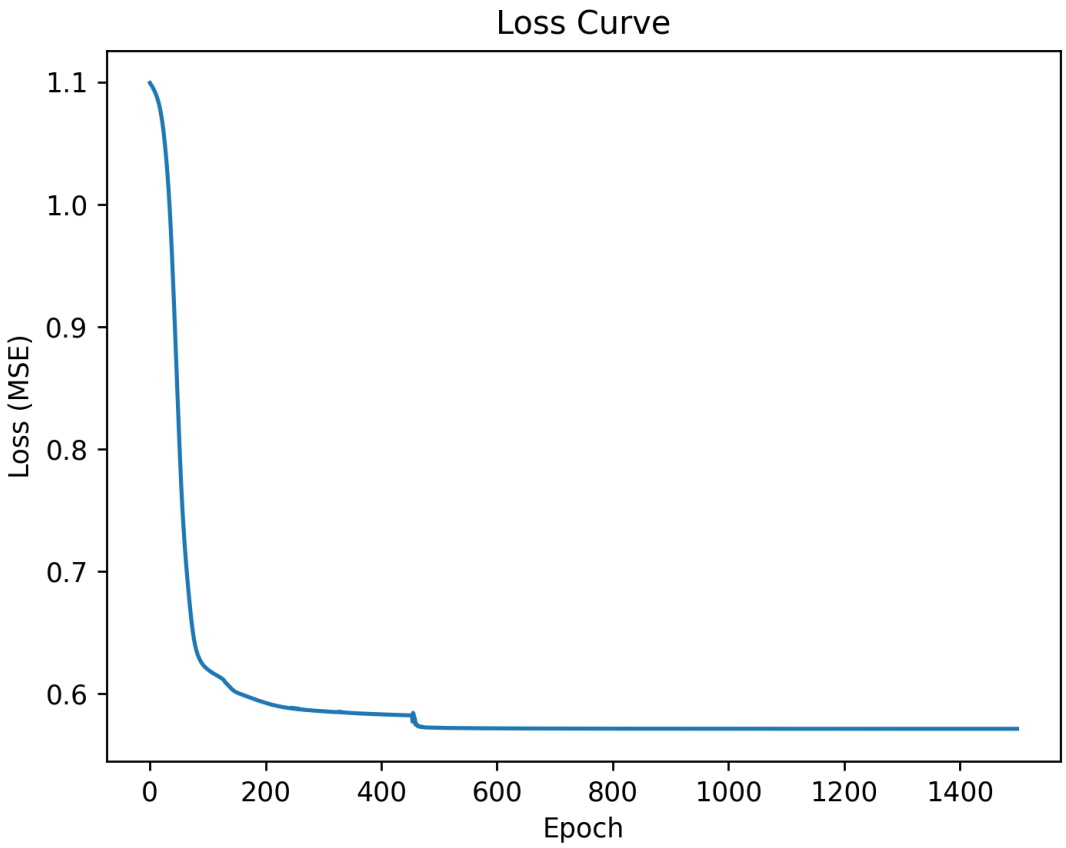
[ ]:
```
class NNet(nn.Module):
    def __init__(self, N_hidden=6, N_in=2, N_out=3):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(N_in, N_hidden),
            nn.LeakyReLU(),
            nn.Linear(N_hidden, N_hidden),
            nn.Tanh(),
            nn.Linear(N_hidden, N_hidden),
            nn.LeakyReLU(),
            nn.Linear(N_hidden, N_hidden),
            nn.LeakyReLU(),
            nn.Linear(N_hidden, N_out),
            nn.Sigmoid()
```

```
        )

    def forward(self,x):
        return self.seq(x)

x = torch.Tensor(x)
y = torch.Tensor(y.reshape(-1,1))
y_new = torch.Tensor(y_new)

model = NNet(N_hidden = 10)
loss_curve = []
accuracy = []

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

# 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_new)

    loss_curve.append(loss.item())
    acc = out.max(axis=1).indices == y.squeeze(1)

    # Print loss progress info 25 times during training
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch} of {epochs}... \tAverage loss: {loss.item()}␣
 ↪\tAccuracy: {acc.float().sum() / y.squeeze(1).shape[0]}")

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

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

```
plt.show()
```

```
Epoch 0 of 1500…      Average loss: 1.0993950366973877        Accuracy:
0.3400000035762787
Epoch 60 of 1500…     Average loss: 0.7226464748382568        Accuracy:
0.9200000166893005
Epoch 120 of 1500…    Average loss: 0.6139636635780334        Accuracy:
0.9399999976158142
Epoch 180 of 1500…    Average loss: 0.5958366990089417        Accuracy:
0.9700000286102295
Epoch 240 of 1500…    Average loss: 0.5884358286857605        Accuracy:
0.9700000286102295
Epoch 300 of 1500…    Average loss: 0.5857990980148315        Accuracy:
0.9700000286102295
Epoch 360 of 1500…    Average loss: 0.5840986967086792        Accuracy:
0.9700000286102295
Epoch 420 of 1500…    Average loss: 0.5829318165779114        Accuracy:
0.9700000286102295
Epoch 480 of 1500…    Average loss: 0.5725138783454895        Accuracy:
0.9800000190734863
Epoch 540 of 1500…    Average loss: 0.5720308423042297        Accuracy:
0.9800000190734863
Epoch 600 of 1500…    Average loss: 0.5718442797660828        Accuracy:
0.9800000190734863
Epoch 660 of 1500…    Average loss: 0.5717346668243408        Accuracy:
0.9800000190734863
Epoch 720 of 1500…    Average loss: 0.5716642737388611        Accuracy:
0.9800000190734863
Epoch 780 of 1500…    Average loss: 0.5716164112091064        Accuracy:
0.9800000190734863
Epoch 840 of 1500…    Average loss: 0.5715824365615845        Accuracy:
0.9800000190734863
Epoch 900 of 1500…    Average loss: 0.5715575814247131        Accuracy:
0.9800000190734863
Epoch 960 of 1500…    Average loss: 0.5715387463569641        Accuracy:
0.9800000190734863
Epoch 1020 of 1500…   Average loss: 0.5715242624282837        Accuracy:
0.9800000190734863
Epoch 1080 of 1500…   Average loss: 0.5715128183364868        Accuracy:
0.9800000190734863
Epoch 1140 of 1500…   Average loss: 0.5715035796165466        Accuracy:
0.9800000190734863
Epoch 1200 of 1500…   Average loss: 0.5714960694313049        Accuracy:
0.9800000190734863
Epoch 1260 of 1500…   Average loss: 0.5714898705482483        Accuracy:
0.9800000190734863
Epoch 1320 of 1500…   Average loss: 0.5714847445487976        Accuracy:
0.9800000190734863
```

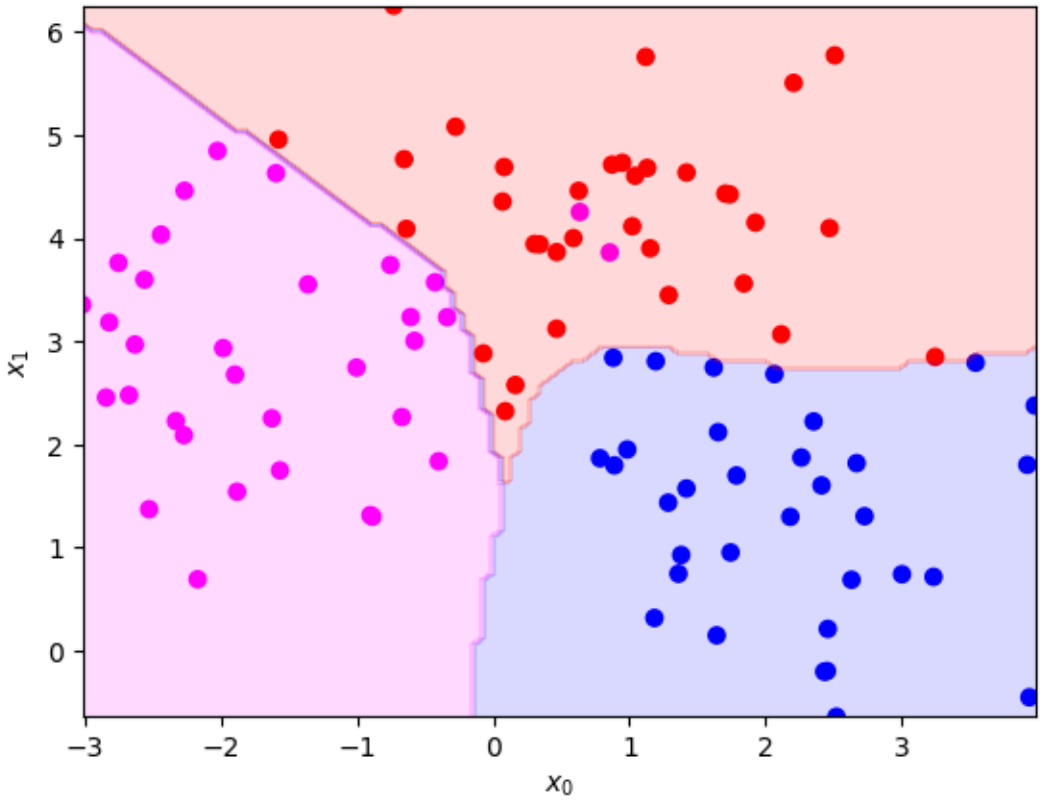
```
Epoch 1380 of 1500…   Average loss: 0.5714803338050842        Accuracy:
0.9800000190734863
Epoch 1440 of 1500…   Average loss: 0.5714766979217529        Accuracy:
0.9800000190734863
```



1.4 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

[ ]:
```
visualizeModel(model)
```



1.5 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.

number and size of hidden layers: The number of hidden layers was chosen to minimize the number of nodes and layers while still producing consistently smooth and accurate results. This was iterated on until a final number of layers and nodes per layer was decided on. The activation functions were chosen to produce smooth output results while also eliminating unnecessary nodes. For this reason the LeakyReLU function was used for most of the inner layers with the second layer having an activation function of tanh. This was done to smooth out the final results a bit more as the LeakyReLU function produced very jagged dividing lines. Cross entropy loss was used for the loss function because it measures the performance od probability classification models and is useful for multi-class classification problems such as this one. The Adam optimizer was used as the optimizer because it provides a varying gradient to speed up or slow down training when necessary. This provides faster and more accurate training results. The learning rate used was determined after trying out a few in the range of 0.05 - 0.0005. 0.005 was found to speed up training enough while also producing accurate and not over trained results. A similar method was used for determining the number of epochs while making sure that training had slowed down enough but not allowed to go on for too long and risk over fitting.

M7-L1-P1

October 30, 2023

0.1 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)
```

0.2 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):
    return x@weight.T + bias

# 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. ]]
```

0.3 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):
    a = x
    for i,weight in enumerate(weights[:-1]):
        a = sigmoid(perceptron_layer(a, weight, biases[i]))

    return perceptron_layer(a, weights[-1], biases[-1])
```

[ ]:
```
# 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
```

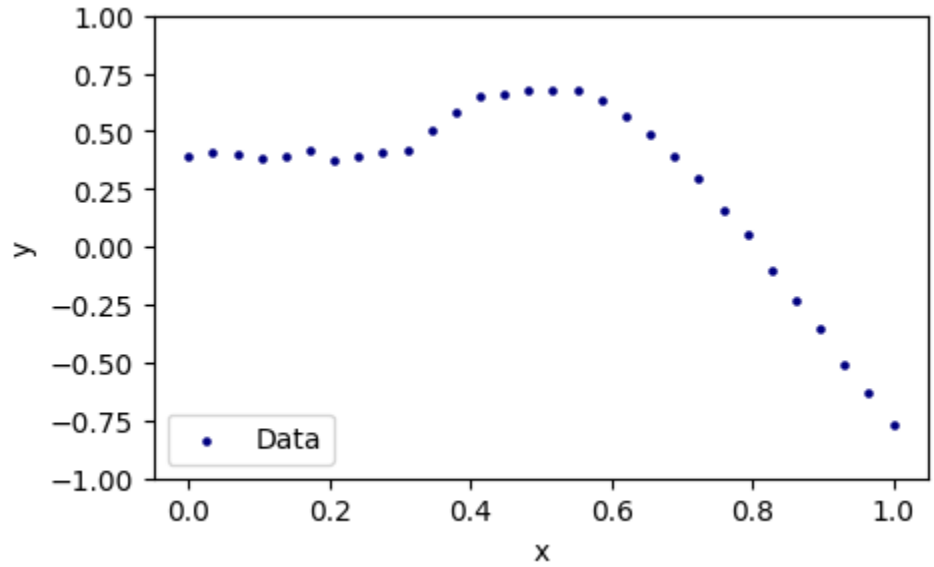
M7-L1 Problem 2

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:

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0.        , 0.03448276, 0.06896552, 0.10344828, 0.13793103,0.17241379, 0.2
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437

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

biases = [np.array([ 2.02112296, -3.47589349, -1.11586831]), np.array([ 1.35350721, -0.1

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).

In [ ]:
```
def perceptron_layer(x, weight, bias):
    return x@weight.T + bias

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

def MLP(x, weights, biases):
    a = x
    for i,weight in enumerate(weights[:-1]):
        a = sigmoid(perceptron_layer(a, weight, biases[i]))

    return perceptron_layer(a, weights[-1], biases[-1])
```

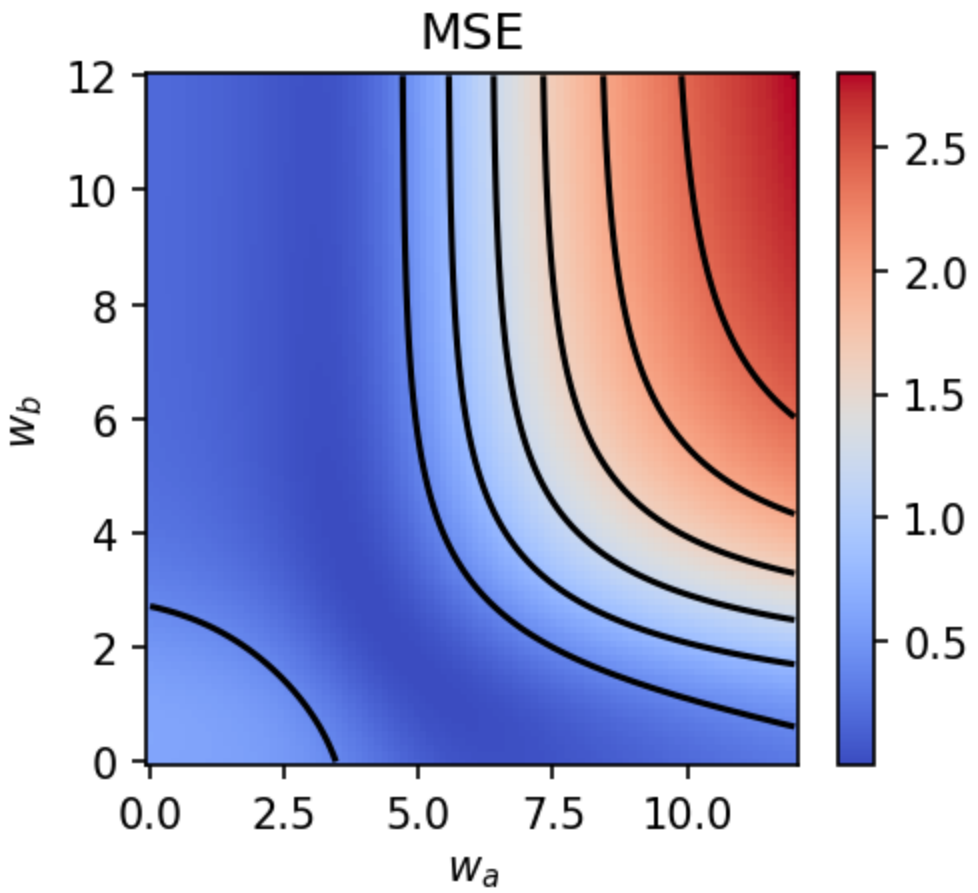
Varying weights

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:

$$\underline{x\ (N\times 1)} \rightarrow \sigma\left(w=\begin{bmatrix}-5.9\\ \boldsymbol{w_a}\\ \boldsymbol{w_b}\end{bmatrix}; b=\begin{bmatrix}2.02\\ -3.48\\ -1.12\end{bmatrix}'\right) \rightarrow \underline{(N\times 3)} \rightarrow \sigma\left(w=\begin{bmatrix}0.9 & -1. & -1.6\\ 4.76 & -0.89 & -2.9\\ -0.95 & 3.19 & 2.61\end{bmatrix}\right.$$

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

In [ ]:
```
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()
```



In [ ]:

```
# %matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, FloatSlide


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
```

Out[ ]:
```
interactive(children=(FloatSlider(value=0.0, description='wa', layout=Layout(width='550p
x'), max=12.0, readout…
```

Questions

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

- 3.0

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

- For large values of w_a and w_b, the MLP's prediction gets much worse. It starts to not fit the end points very well.

M7-L2-P1

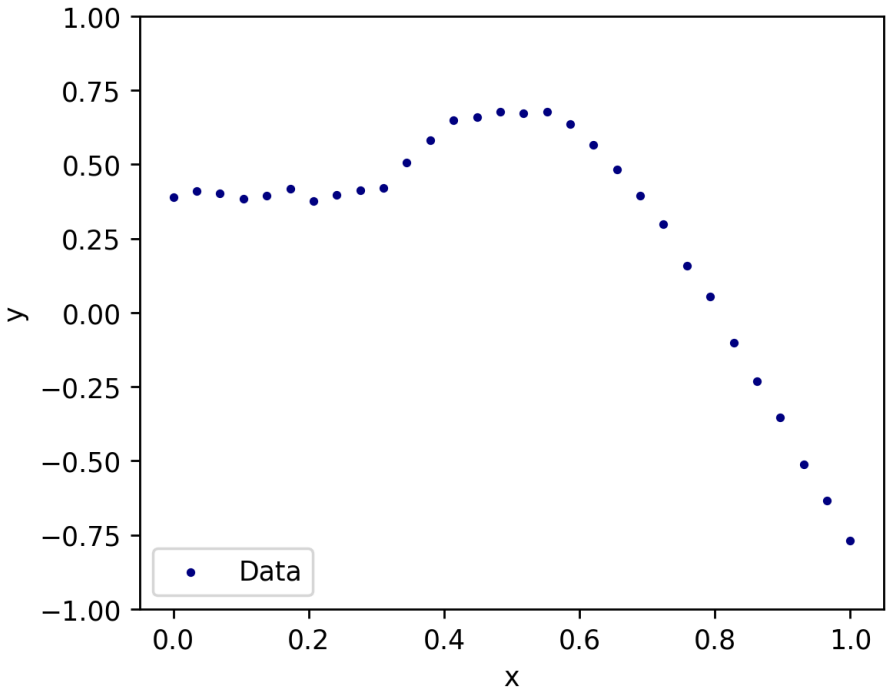
October 28, 2023

1 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

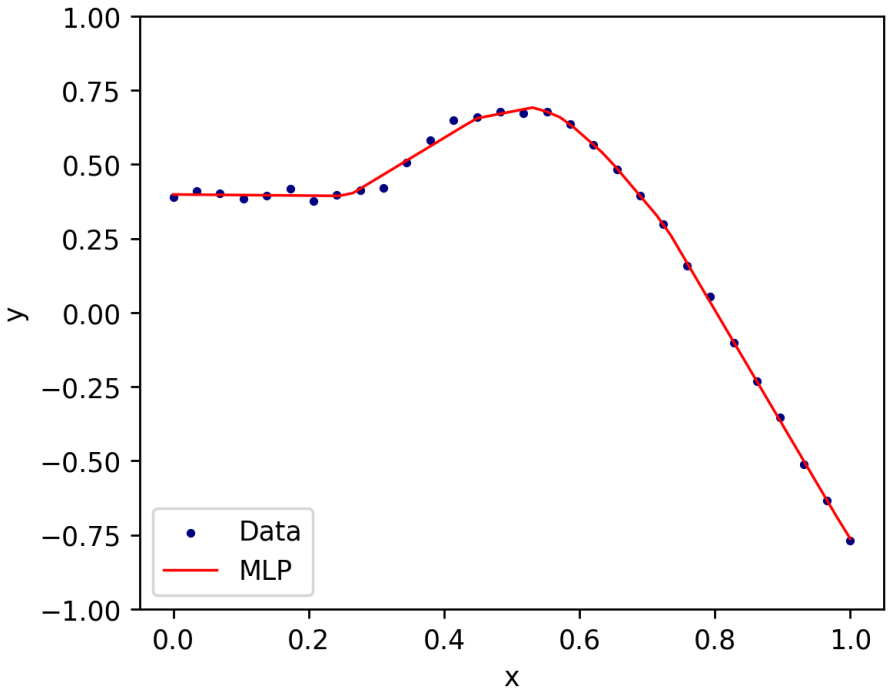
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor

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()
```



1.1 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=50000, tol=1e-7) # 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()
```



1.2 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.M

1.3 Question

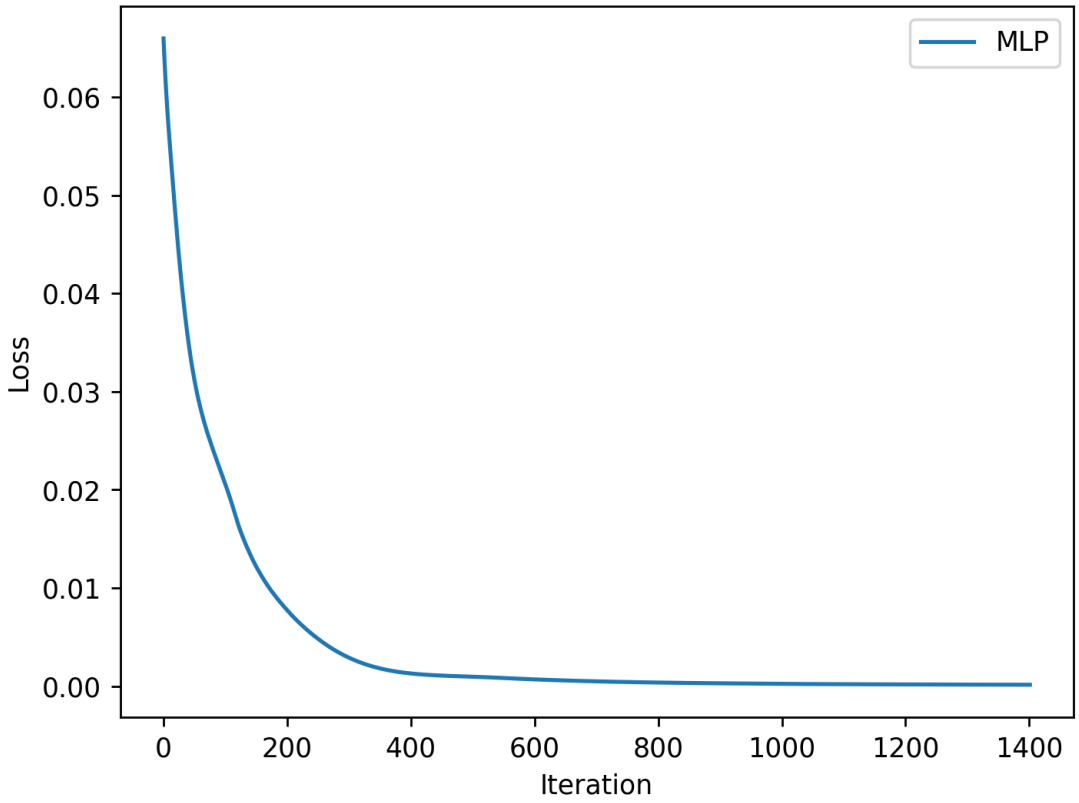
1. What values of `max_iter` and `tol` gave you a reasonable fit?

50,000 iterations and tolerance of 1e-7

1.4 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()
```



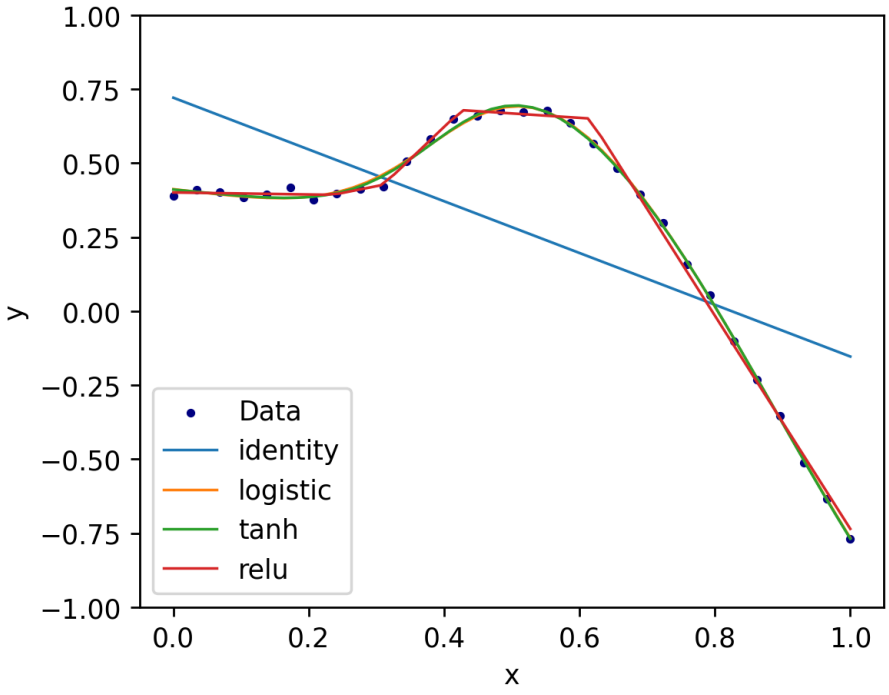
1.5 Activation Functions

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()
```

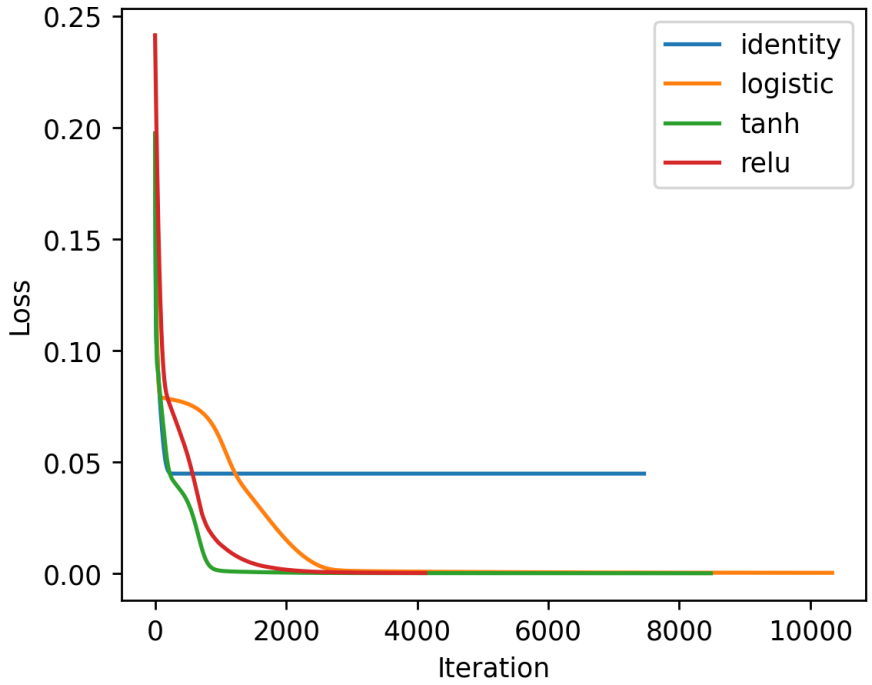


1.6 Loss curves

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.

[ ]:
```
plt.figure(figsize=(5,4),dpi=250)

for act in activations:
    plt.plot(models[act].loss_curve_,label=act)

plt.legend(loc="lower left")
plt.xlabel("Iteration")
plt.ylabel("Loss")
```

```
plt.legend()
plt.show()
```



1.7 Questions

2. Which activation functions produced a good fit?

Logistic and tanh produced the best fits.

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

The logistic converged the slowest but had a much better fit thanones like identity that converged very quickly.

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

Tanh converged the quickest and had one of the best fits.

M7-L2-P2

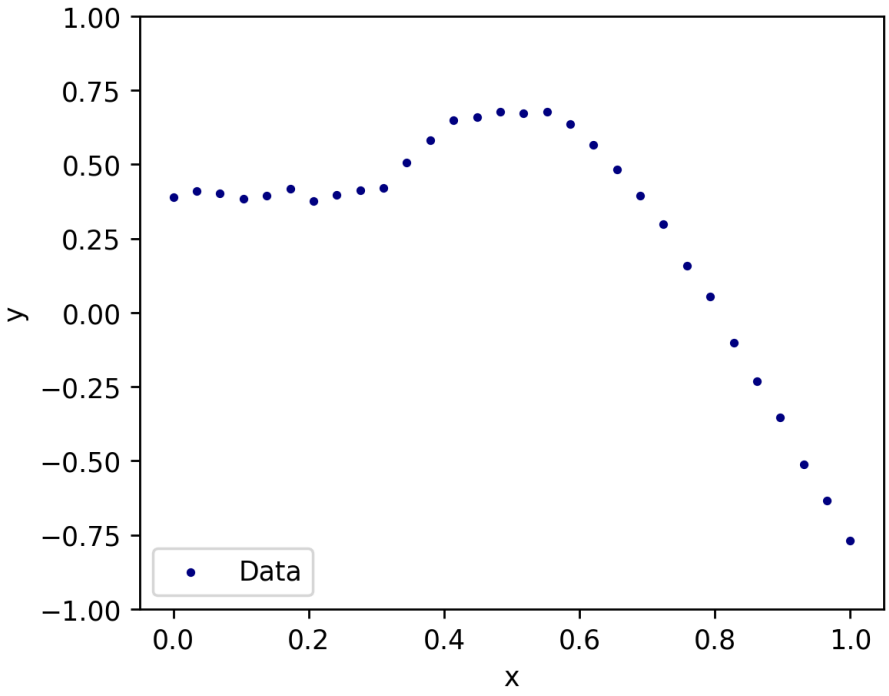
October 28, 2023

1 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()
```



1.1 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)
```

1.2 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
```

1.3 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 = []
```

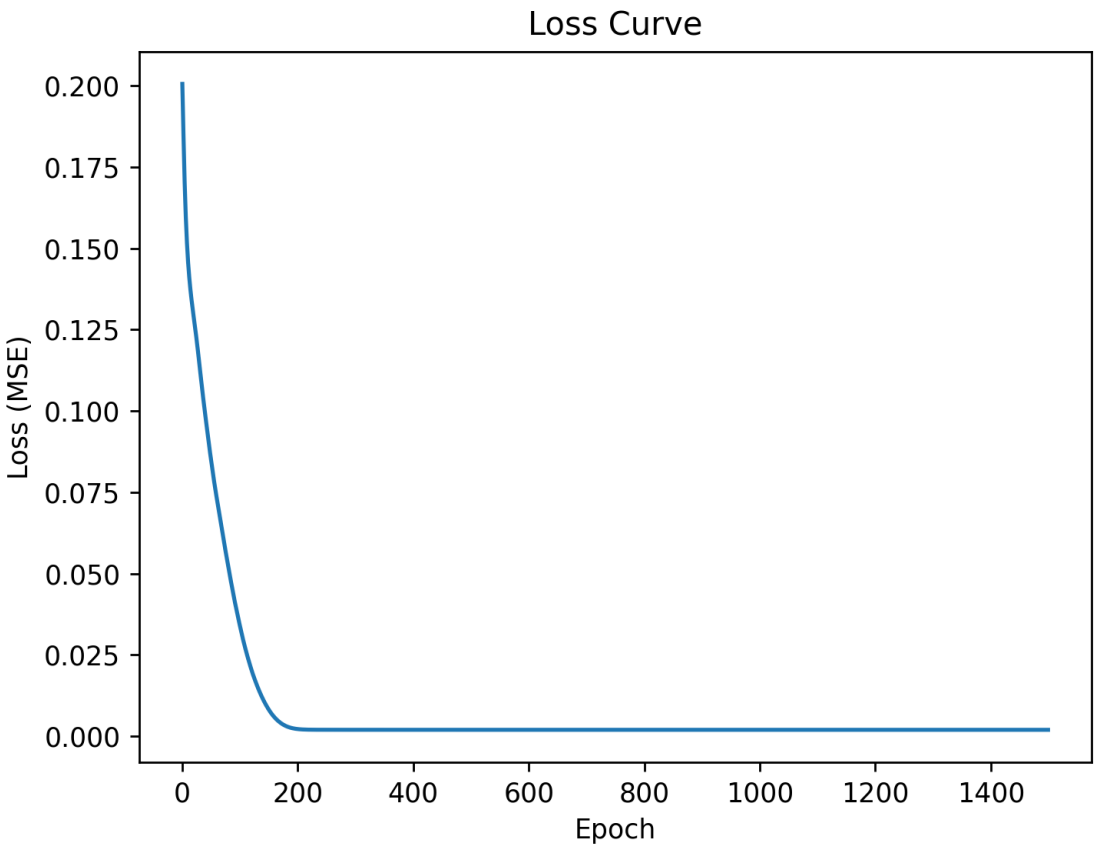
1.4 Training a model

[ ]:
```
# 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()}")
```

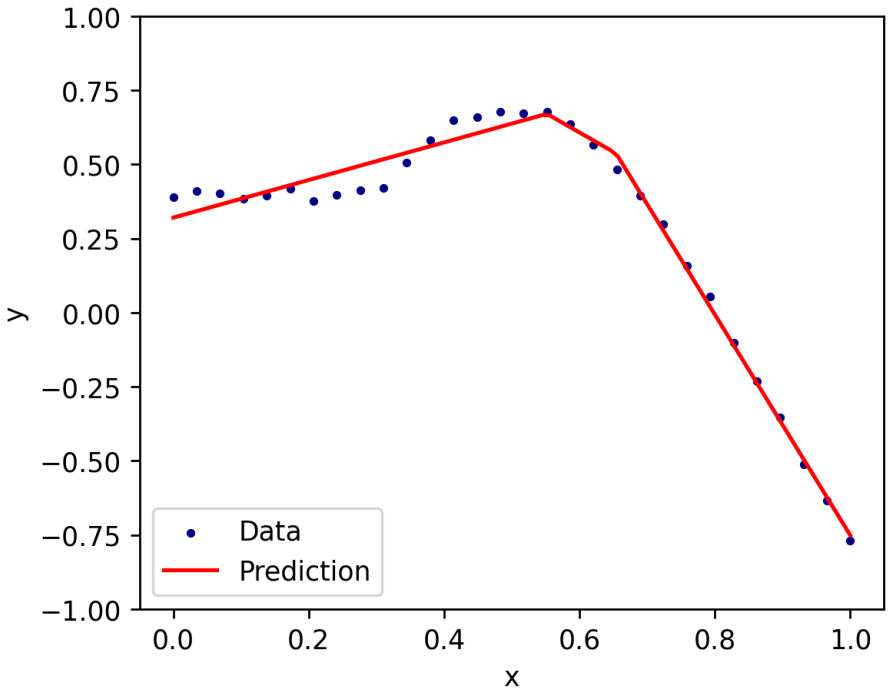
```
    # Move the model parameters 1 step closer to their optima:
    opt.zero_grad()
    loss.backward()
    opt.step()
```

```
Epoch 0 of 1500…      Average loss: 0.20060913264751434
Epoch 60 of 1500…     Average loss: 0.07273828983306885
Epoch 120 of 1500…    Average loss: 0.020207548514008522
Epoch 180 of 1500…    Average loss: 0.0030261396896094084
Epoch 240 of 1500…    Average loss: 0.0019302271539345384
Epoch 300 of 1500…    Average loss: 0.0019199324306100607
Epoch 360 of 1500…    Average loss: 0.0019171799067407846
Epoch 420 of 1500…    Average loss: 0.0019168586004525423
Epoch 480 of 1500…    Average loss: 0.001916774665005505
Epoch 540 of 1500…    Average loss: 0.0019166858401149511
Epoch 600 of 1500…    Average loss: 0.0019165772246196866
Epoch 660 of 1500…    Average loss: 0.0019164994591847062
Epoch 720 of 1500…    Average loss: 0.0019163553370162845
Epoch 780 of 1500…    Average loss: 0.0019162431126460433
Epoch 840 of 1500…    Average loss: 0.0019161227392032743
Epoch 900 of 1500…    Average loss: 0.0019159989897161722
Epoch 960 of 1500…    Average loss: 0.001915863249450922
Epoch 1020 of 1500…   Average loss: 0.0019157796632498503
Epoch 1080 of 1500…   Average loss: 0.0019156151684001088
Epoch 1140 of 1500…   Average loss: 0.0019154789624735713
Epoch 1200 of 1500…   Average loss: 0.0019153128378093243
Epoch 1260 of 1500…   Average loss: 0.0019152022432535887
Epoch 1320 of 1500…   Average loss: 0.001915033208206296
Epoch 1380 of 1500…   Average loss: 0.0019148987485095859
Epoch 1440 of 1500…   Average loss: 0.0019147639395669103
```

[ ]:
```
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()
```



1.5 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.

[ ]:
```
model = Net_2_layer(N_hidden = 20, activation = F.tanh)
loss_curve = []
```

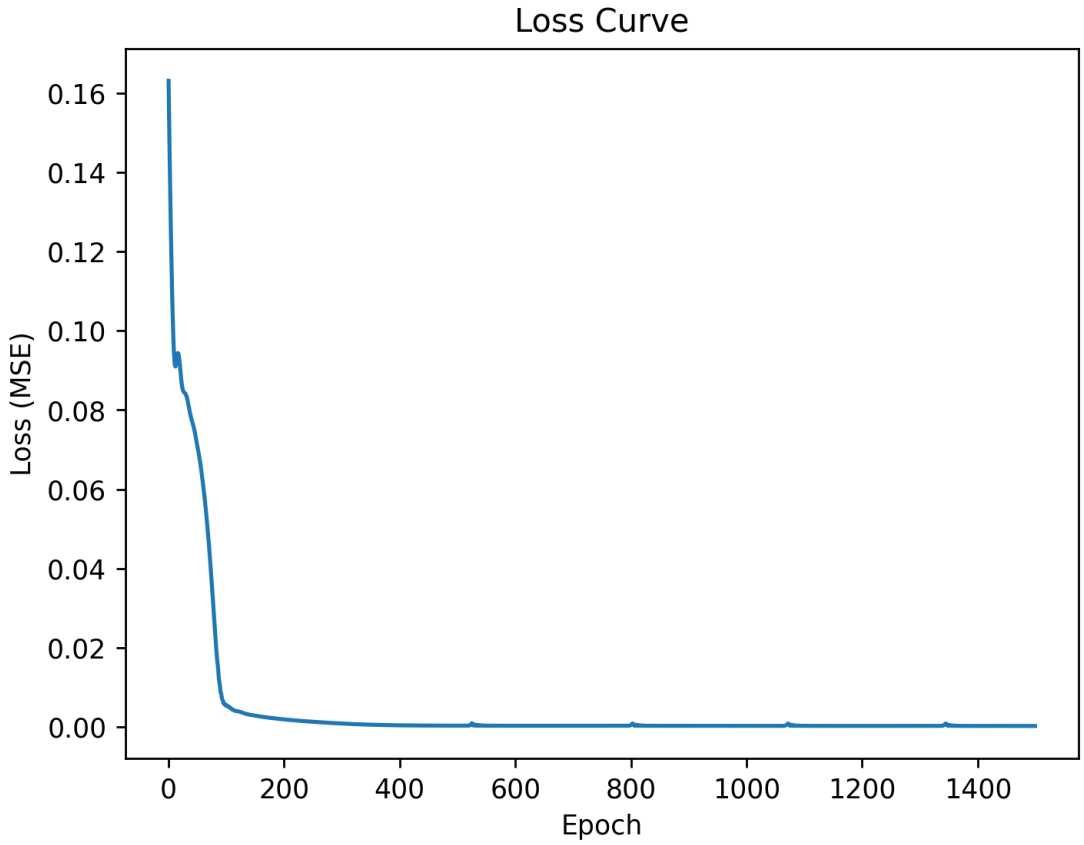
[ ]:
```
# Training parameters: Learning rate, number of epochs, loss function
# (These can be tuned)
lr = 0.005
epochs = 1500
loss_fcn = F.mse_loss

# 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
```

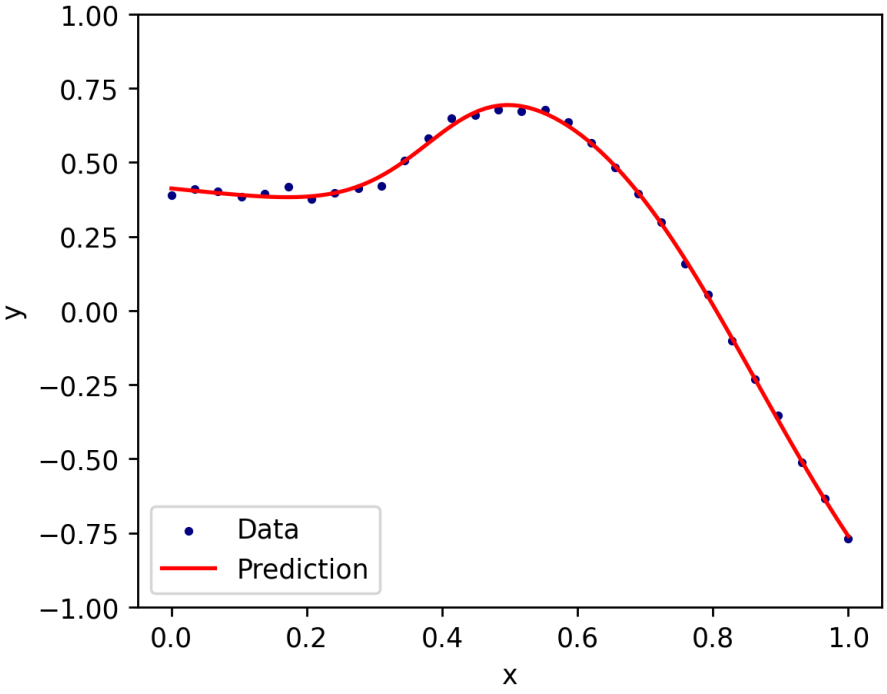
```
    loss_curve.append(loss.item())

    # 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: 0.16317762434482574
Epoch 60 of 1500…     Average loss: 0.06078961864113808
Epoch 120 of 1500…    Average loss: 0.003894438734278083
Epoch 180 of 1500…    Average loss: 0.0021709902212023735
Epoch 240 of 1500…    Average loss: 0.0013601431855931878
Epoch 300 of 1500…    Average loss: 0.0008099587867036462
Epoch 360 of 1500…    Average loss: 0.0004718304262496531
Epoch 420 of 1500…    Average loss: 0.00032416018075309694
Epoch 480 of 1500…    Average loss: 0.0002800915972329676
Epoch 540 of 1500…    Average loss: 0.00028401173767633736
Epoch 600 of 1500…    Average loss: 0.000254952086834237
Epoch 660 of 1500…    Average loss: 0.00024991866666823626
Epoch 720 of 1500…    Average loss: 0.0002460372052155435
Epoch 780 of 1500…    Average loss: 0.0002424498670734465
Epoch 840 of 1500…    Average loss: 0.00024211336858570576
Epoch 900 of 1500…    Average loss: 0.00023554654035251588
Epoch 960 of 1500…    Average loss: 0.0002320874627912417
Epoch 1020 of 1500…   Average loss: 0.00022848552907817066
Epoch 1080 of 1500…   Average loss: 0.0002608615905046463
Epoch 1140 of 1500…   Average loss: 0.00022104621166363358
Epoch 1200 of 1500…   Average loss: 0.0002172699460061267
Epoch 1260 of 1500…   Average loss: 0.0002132928348146379
Epoch 1320 of 1500…   Average loss: 0.00020909849263262004
Epoch 1380 of 1500…   Average loss: 0.00021068908972665668
Epoch 1440 of 1500…   Average loss: 0.000200624592253007
```

[ ]:
```
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()
```




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