[illegible]

SRECHARAN
SSELVAM

Homework 7

Instructions

This homework contains **4** concepts and **6** programming questions. In MS word or a similar text editor, write down the problem number and your answer for each problem. Combine all answers for concept questions in a single PDF file. Export/print the Jupyter notebook as a PDF file including the code you implemented and the outputs of the program. Make sure all plots and outputs are visible in the PDF.

Combine all answers into a single PDF named andrewID_hw7.pdf and submit it to Gradescope before the due date. Refer to the syllabus for late homework policy. Please assign each question a page by using the “Assign Questions and Pages” feature in Gradescope.

Here is a breakdown of the points for programming questions:

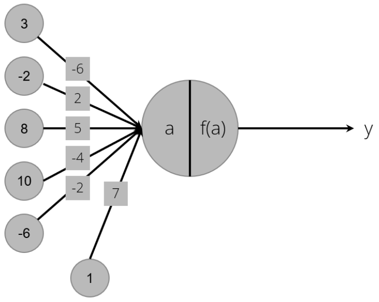
Name	Points
M7-L1-P1	10
M7-L1-P2	5
M7-L2-P1	5
M7-L2-P2	10
M7-HW1	30
M7-HW2	30

☐

Problem 1 (2.5 points)

Consider the following perceptron. Compute the output y, using σ(a), the sigmoid activation function

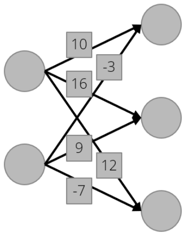
$$\sigma(a) = \frac{1}{1 + \exp(\text{-a})}$$



☐

Problem 2 (2.5 Points)

(Multiple choice)
Which of the following weight matrices is correct for the provided fully connected layers?



$$W_1 = \begin{bmatrix} 10 & 16 & 12 \\ -3 & 9 & -7 \end{bmatrix} \quad W_2 = \begin{bmatrix} 10 & -3 & 16 \\ -3 & 9 & -7 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} 10 & -3 \\ 16 & 9 \\ 12 & -7 \end{bmatrix} \quad W_4 = \begin{bmatrix} 10 & 9 \\ -3 & 12 \\ 16 & -7 \end{bmatrix}$$

☐

Problem 3 (2.5 Points)

The following question concerns the sigmoid, tanh, and softmax activation functions.
(Multiple choice choose one)
Select the true statement:

1. The tanh activation function is suitable in the output layer for binary classification problems because its output has a probabilistic interpretation
2. The softmax activation function is used in the output layer for multi-class classification problems to produce a probability distribution over multiple classes
3. Unlike other activation functions, the sigmoid activation function does not suffer from the problem of vanishing gradients in networks with many hidden layers
4. All of the above

□

Problem 4 (2.5 Points)

The following question considers the ReLU, Leaky ReLU and GELU activation functions.
(Multiple choice choose one)
Select the true statement:

1. The GELU activation function is a smooth approximation of the ReLU function, which means its derivative is continuous
2. The derivatives of ReLU and Leaky ReLU have a discontinuity at x = 0
3. Leaky ReLU is a variant of ReLU that allows a small non-zero gradient for negative input values,
4. All of the above

ANSWERS:

PROBLEM 1:-

Step 1:- Compute α ;

The net input a is,

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

$$a = -18 - 4 + 40 - 40 + 12 + 7$$

$$a = -3$$

Step 2:- compute $\sigma(a)$;

we know $a = -3$, so by plugging in,

$$\sigma(-3) = \frac{1}{1+e^{3}} \quad \{e \simeq 2.718\}$$

$$\Rightarrow \sigma(-3) = \frac{1}{1+20.855} \Rightarrow \sigma(-3) = \frac{1}{21.0855}$$

$$\Rightarrow \sigma(-3) = 0.0474$$

Thus, the output y of the perceptron using sigmoid activation function is 0.0474

PROBLEM 2:-

From the first node in the input layer;

→ Top node in the second layer = 10

→ middle node in the second layer = 16

→ Bottom node in the second layer = 12

From the second node in the input layer;

→ Top node in the second layer = -3

→ Middle node in the second layer = 9

→ Bottom node in the second layer = -7

From the third node in the input layer;

→ This node does not have outgoing connections to the second layer based on the given image.

Answer : $W_1 = \begin{bmatrix} 10 & 16 & 12 \\ -3 & 9 & -7 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 10 & 16 & 12 \\ -3 & 9 & -7 \end{bmatrix}$

PROBLEM 3 :-

ANSWER :

OPTION 2 = 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 :-

ANSWER :

OPTION 4 = ALL OF THE ABOVE

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.

In [6]:
```
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.

In [7]:
```
def perceptron_layer(x, weight, bias):
    # YOUR CODE GOES HERE
    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.

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.

In [8]:
```
def sigmoid(x):
    return 1./(1.+np.exp(-x))

def MLP(x, weights, biases):
    # YOUR CODE GOES HERE
    n = len(weights)
    for i in range(n-1):
        x = sigmoid(perceptron_layer(x,weights[i],biases[i]))
    x = perceptron_layer(x,weights[-1],biases[-1])
    return x
```

In [9]:
```
# 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],
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
```

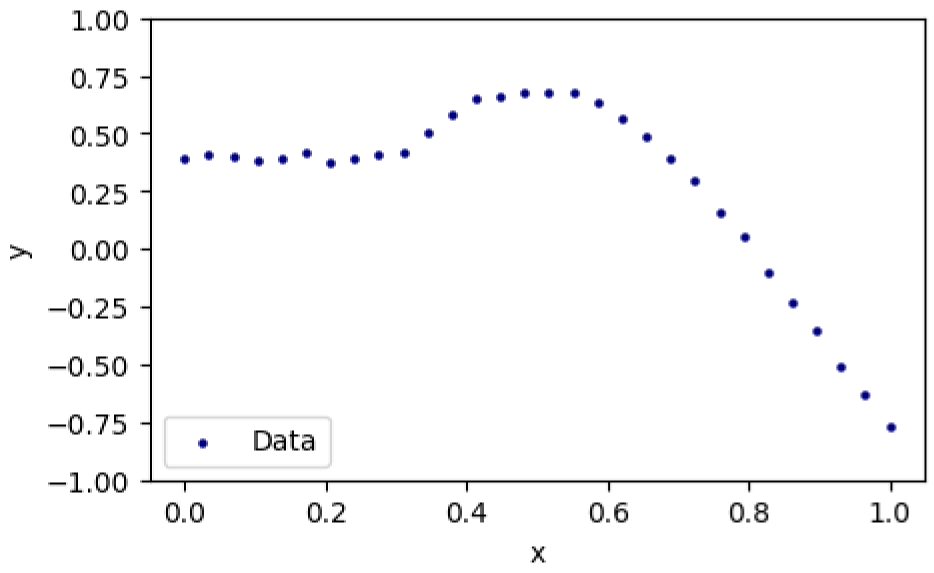
M7-L1 Problem 2

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 [8]:
```
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, 0.275862
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437,  0.37573321,  0.39571087,

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
           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.ar

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()
```

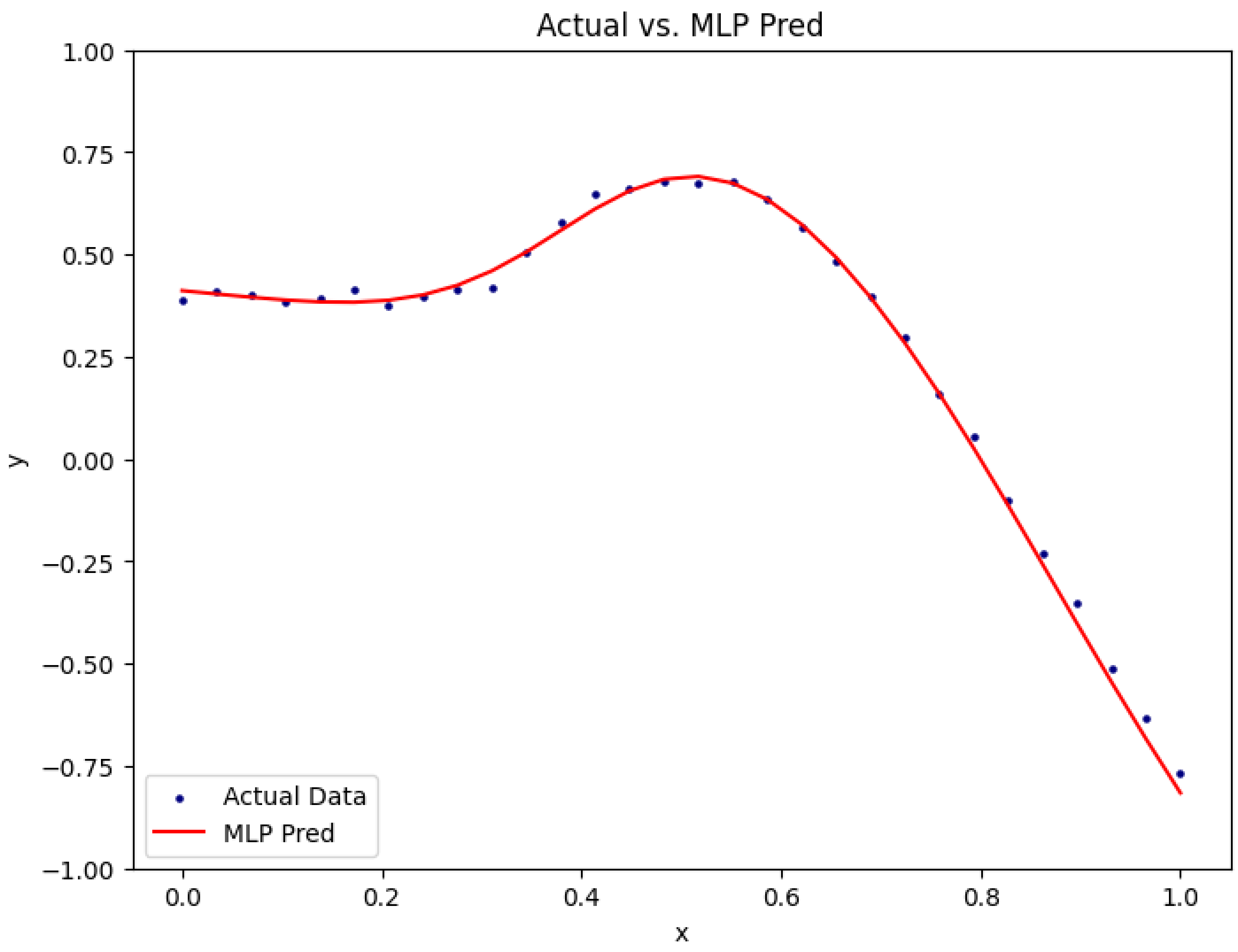


MLP Function

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 [12]:
```
# YOUR CODE GOES HERE
def sigmoid(x):
    return 1./(1.+np.exp(-x))

def MLP(x, weights, biases):
    assert len(weights) == len(biases)

    for i in range(len(weights)):
        x = np.dot(x, weights[i].T) + biases[i]
        if i < len(weights) - 1:
            x = sigmoid(x)
    return x

y_pred = MLP(x, weights, biases)
```

```
# Plotting
plt.figure(figsize=(8,6))
plt.scatter(x, y, s=5, c="navy", label="Actual Data")
plt.plot(x, y_pred, color='r', label="MLP Pred")
plt.legend(loc="lower left")
plt.ylim(-1, 1)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Actual vs. MLP Pred")
plt.show()
```



Varying weights

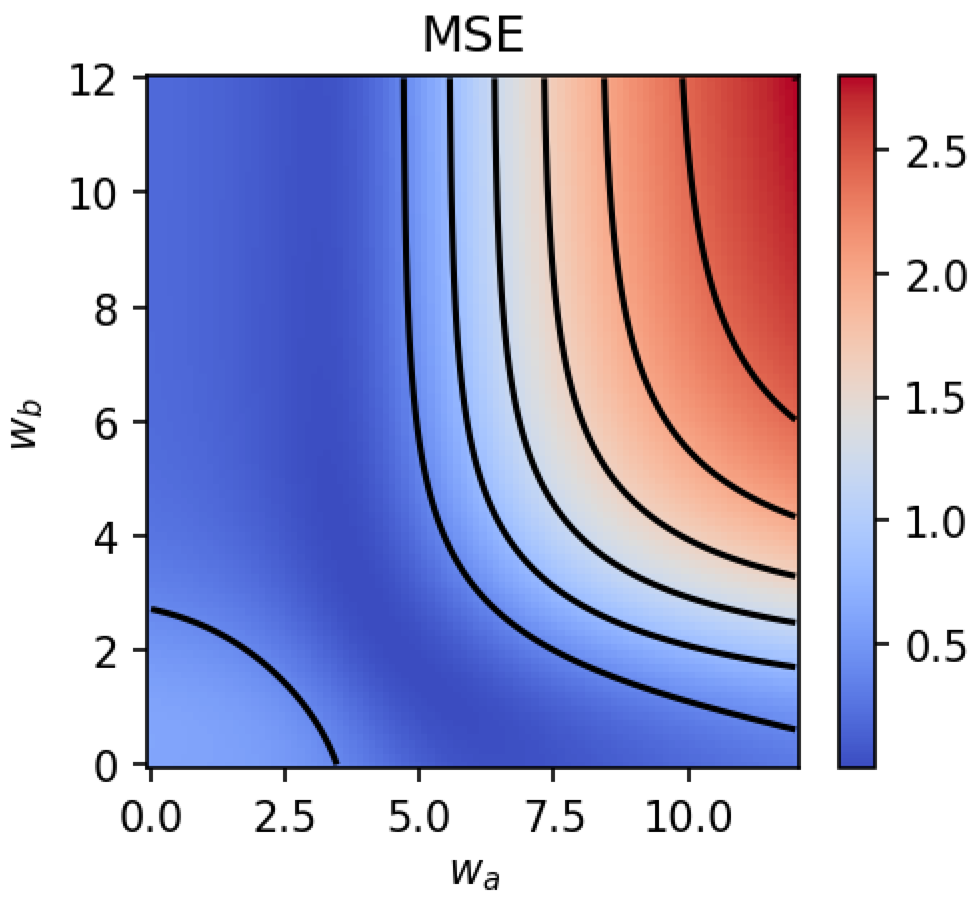
The provided network has 2 hidden layers, each with 3 neurons. The weights and biases are shown below. Note the weights w_a and w_b -- these are left for you to investigate:

$$\underline{x\ (N\times 1)} \rightarrow \sigma\left(w = \begin{bmatrix} -5.9 \\ \boldsymbol{w_a} \\ \boldsymbol{w_b} \end{bmatrix}; b = \begin{bmatrix} 2.02 \\ -3.48 \\ -1.12 \end{bmatrix}'\right) \rightarrow \underline{(N\times 3)} \rightarrow \sigma\left(w = \begin{bmatrix} 0.9 & -1. & -1.65 \\ 4.76 & -0.89 & -2.93 \\ -0.95 & 3.19 & 2.61 \end{bmatrix}; b = \begin{bmatrix} 1.35 \\ -0.11 \\ -4.03 \end{bmatrix}'\right) \rightarrow \underline{(I}$$

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

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

In [10]:
```
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 [11]:
```
%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
```

```
Out[11]: interactive(children=(FloatSlider(value=0.0, description='wa', layout=Layout(width='550px'), max=12.0, readout…
```

Questions

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

- *ANSWER:* wb = 3

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

- *ANSWER:* If wa and wb w bare The MSE increases both large:

The sigmoid function will saturate, meaning it will produce outputs very close to 0 or 1. The MLP's predictions will exhibit sharper transitions between values as the input changes. Depending on the sign and exact magnitude of the weights, the predictions could become highly non-linear, or even seem to overfit the data by showing rapid oscillations or extreme values.

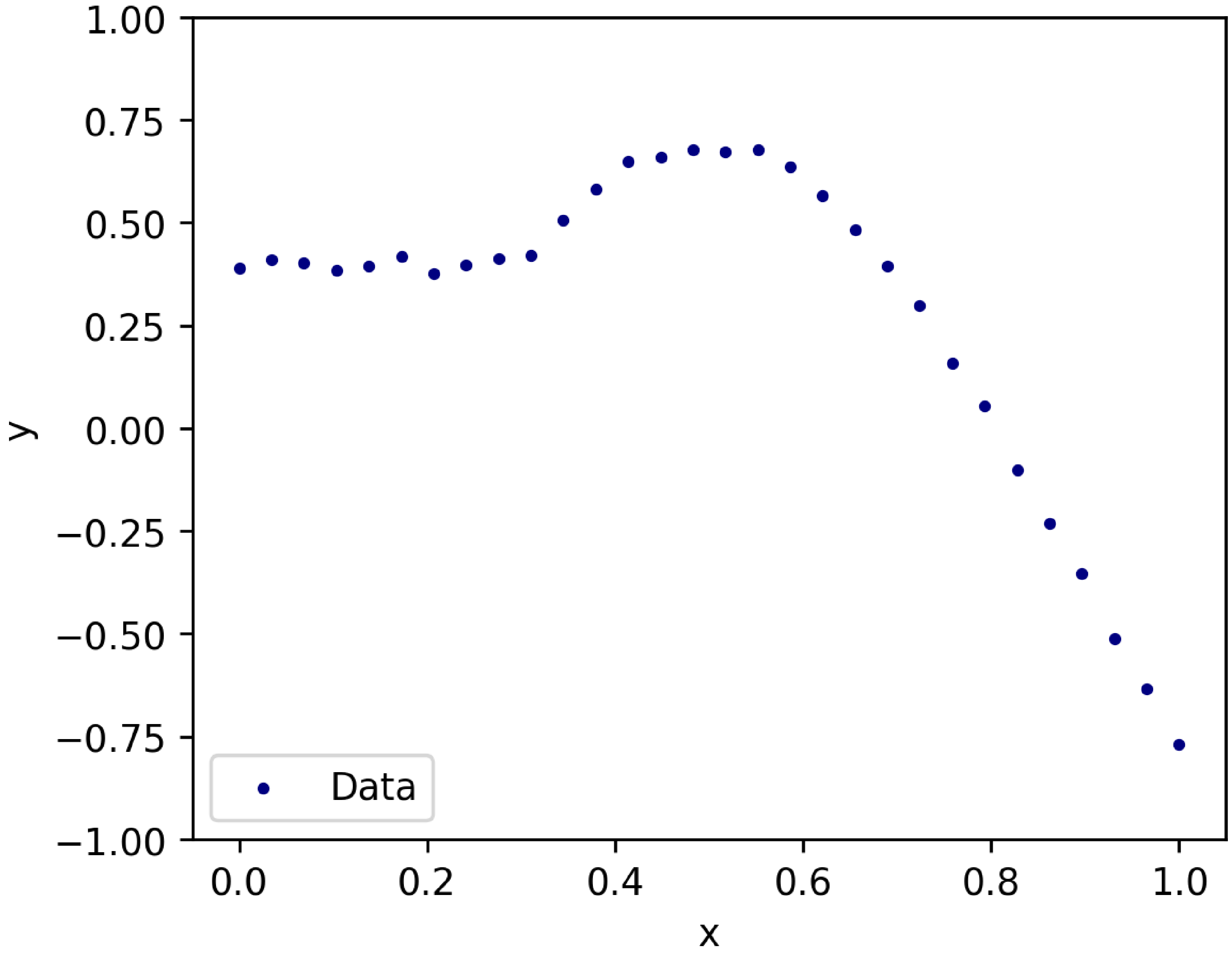
M7-L2 Problem 1

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

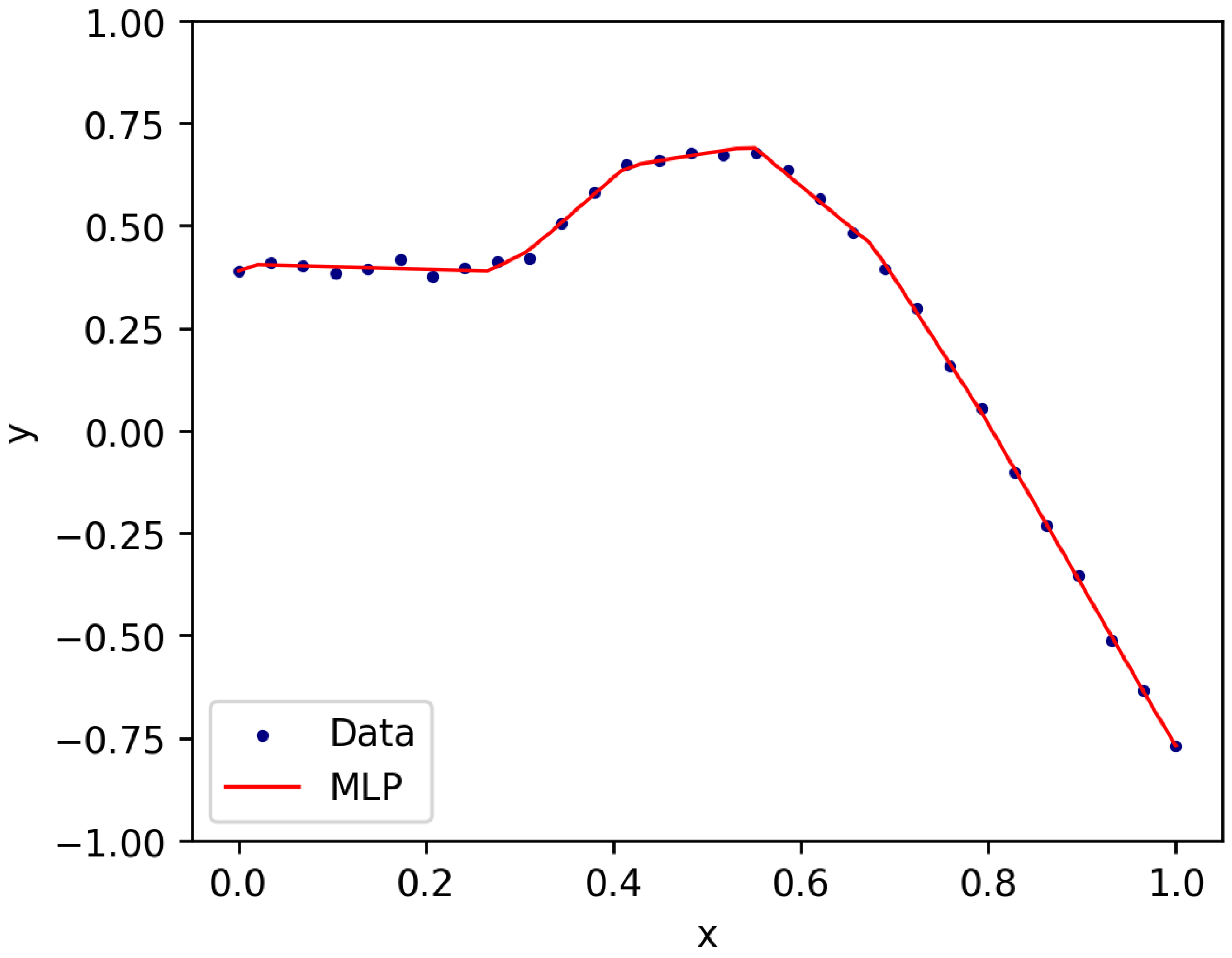
In [44]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor

x = np.array([0.        , 0.03448276, 0.06896552, 0.10344828, 0.13793103,0.17241379, 0.20689655, 0.24137931, 0.27586
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437,  0.37573321,  0.39571087,


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()
```


1.00
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

`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.

In [45]:
```
mlp = MLPRegressor(hidden_layer_sizes=[10,10], max_iter = 5000, 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?

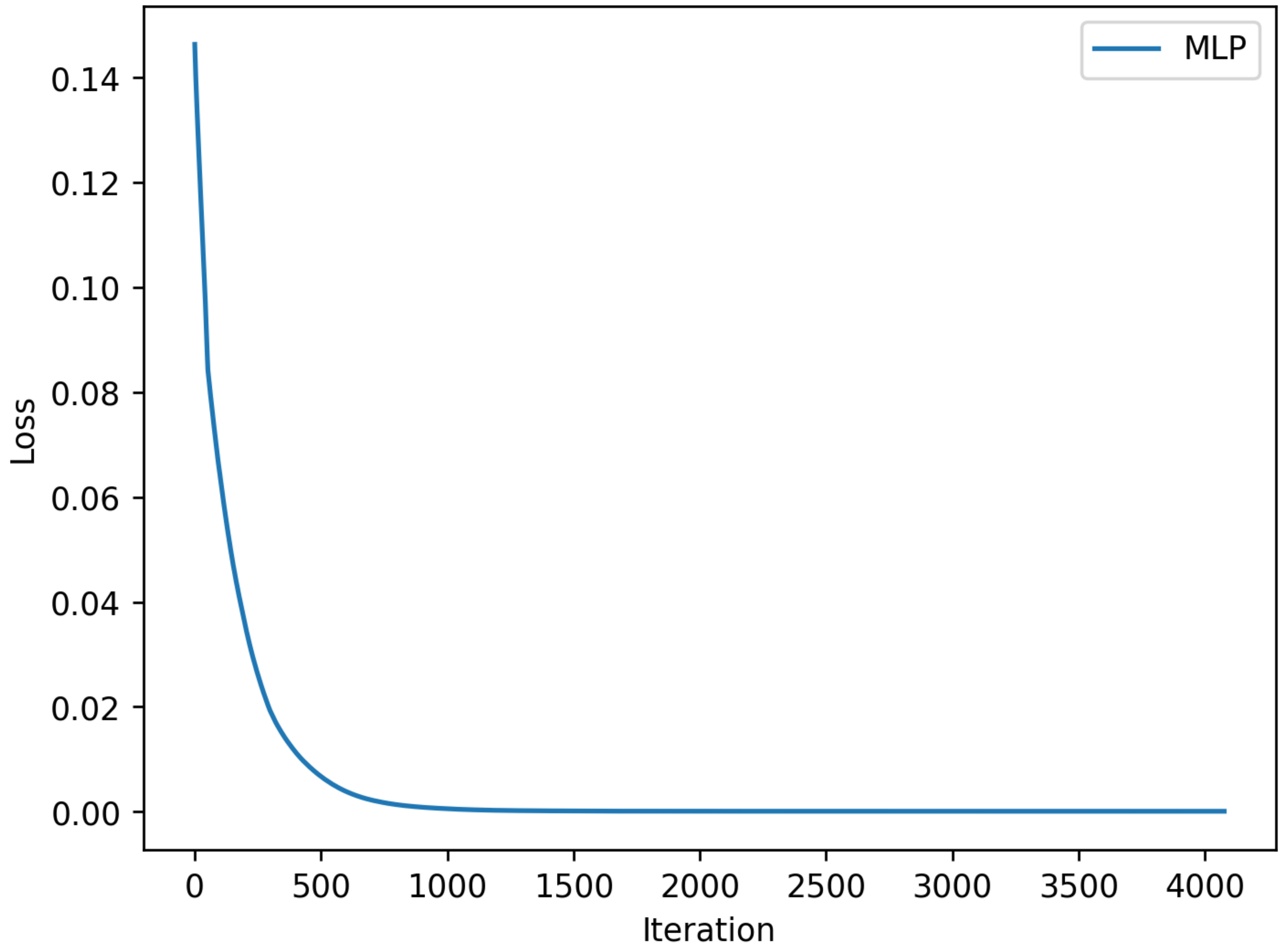
In [46]:
```
print("The chosen values for the hyperparameters are:\nmax_iter = 5000 \ntol = 1e-10")
```

```
The chosen values for the hyperparameters are:
max_iter = 5000
tol = 1e-10
```

Loss Curve

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

In [47]:
```
loss = mlp.loss_curve_
plt.figure(dpi=250)
plt.plot(loss,label="MLP")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



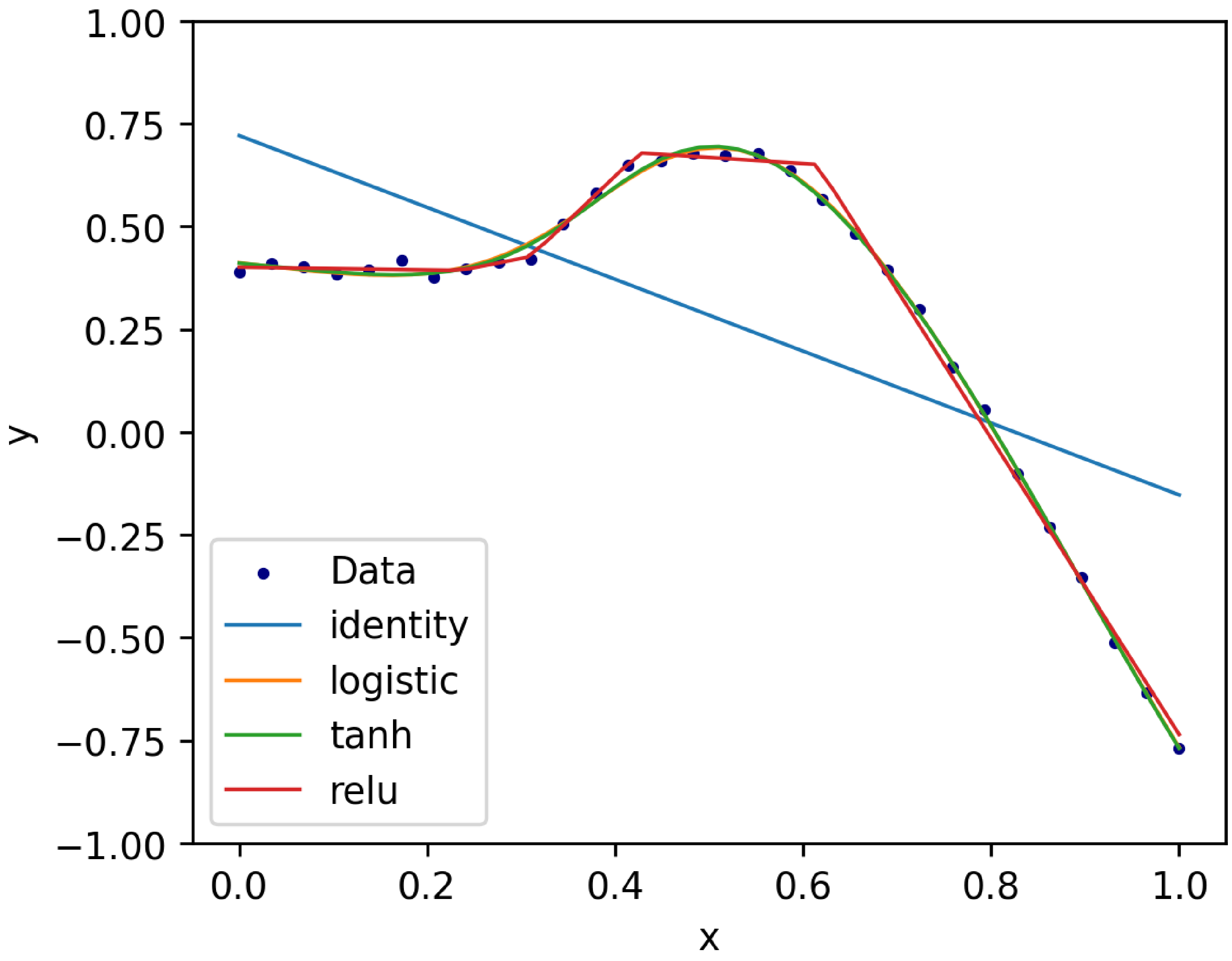
Activation Functions

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

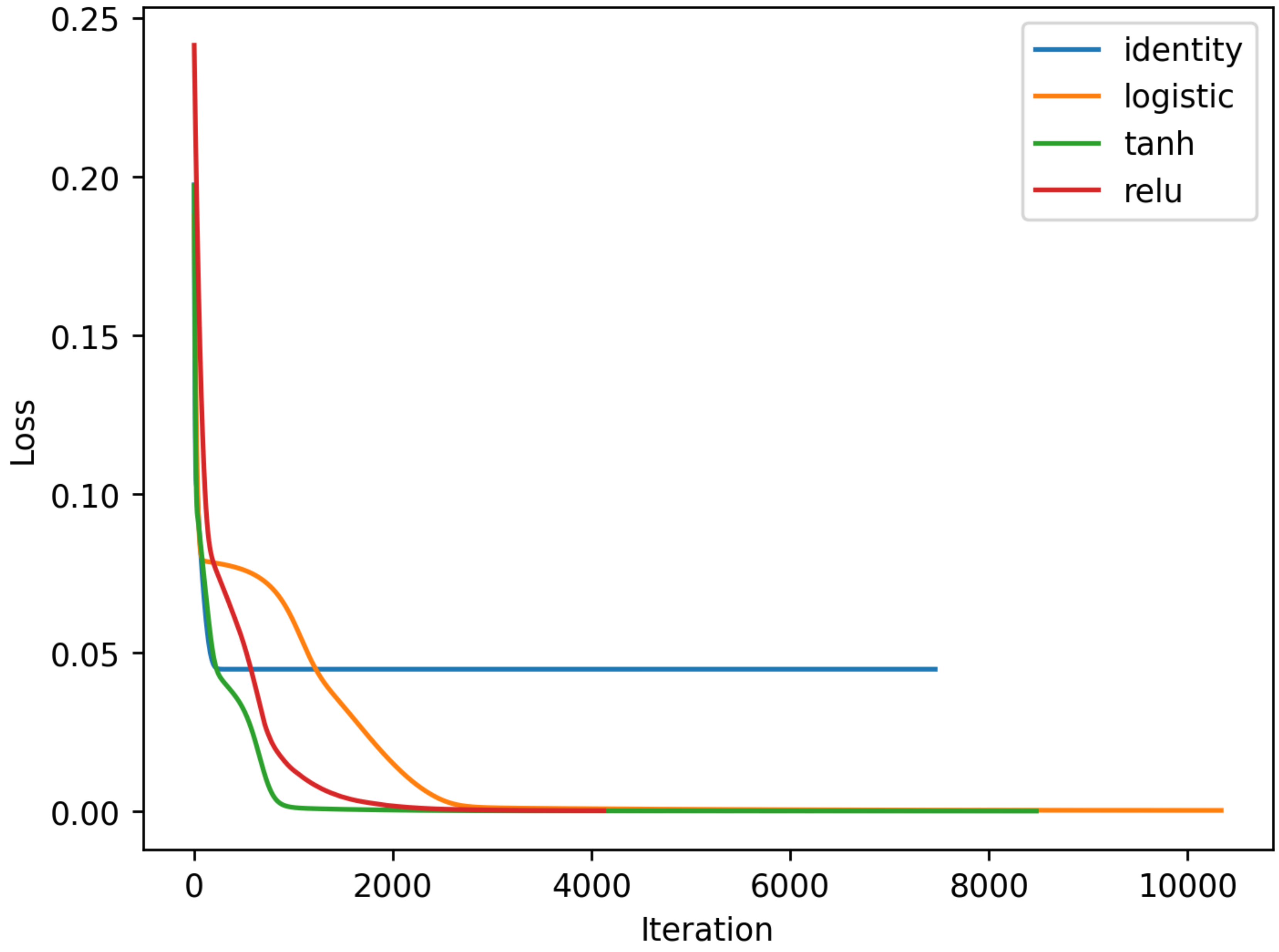
In [48]:
```
activations = ["identity","logistic","tanh","relu"]
models = dict()

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()
```


1.00
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
identity
logistic
tanh
relu

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.

```
In [49]:
# YOUR CODE GOES HERE
plt.figure(dpi=250)
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()
```


identity
logistic
tanh
relu
Loss
Iteration
0.25
0.20
0.15
0.10
0.05
0.00
0
2000
4000
6000
8000
10000

Questions

2. Which activation functions produced a good fit?

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

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

In [50]:
```
print("1) A good fit can be interpreted by the lowest loss. The activation functions 'tanh' and 'relu' both have conv
print("2) The 'identity' activation function model seems to have the highest loss and does not seem to have convergec
print("3) Comparing 'tanh' and 'relu', which both fit well, the 'relu' model appears to converge faster as its loss c
```

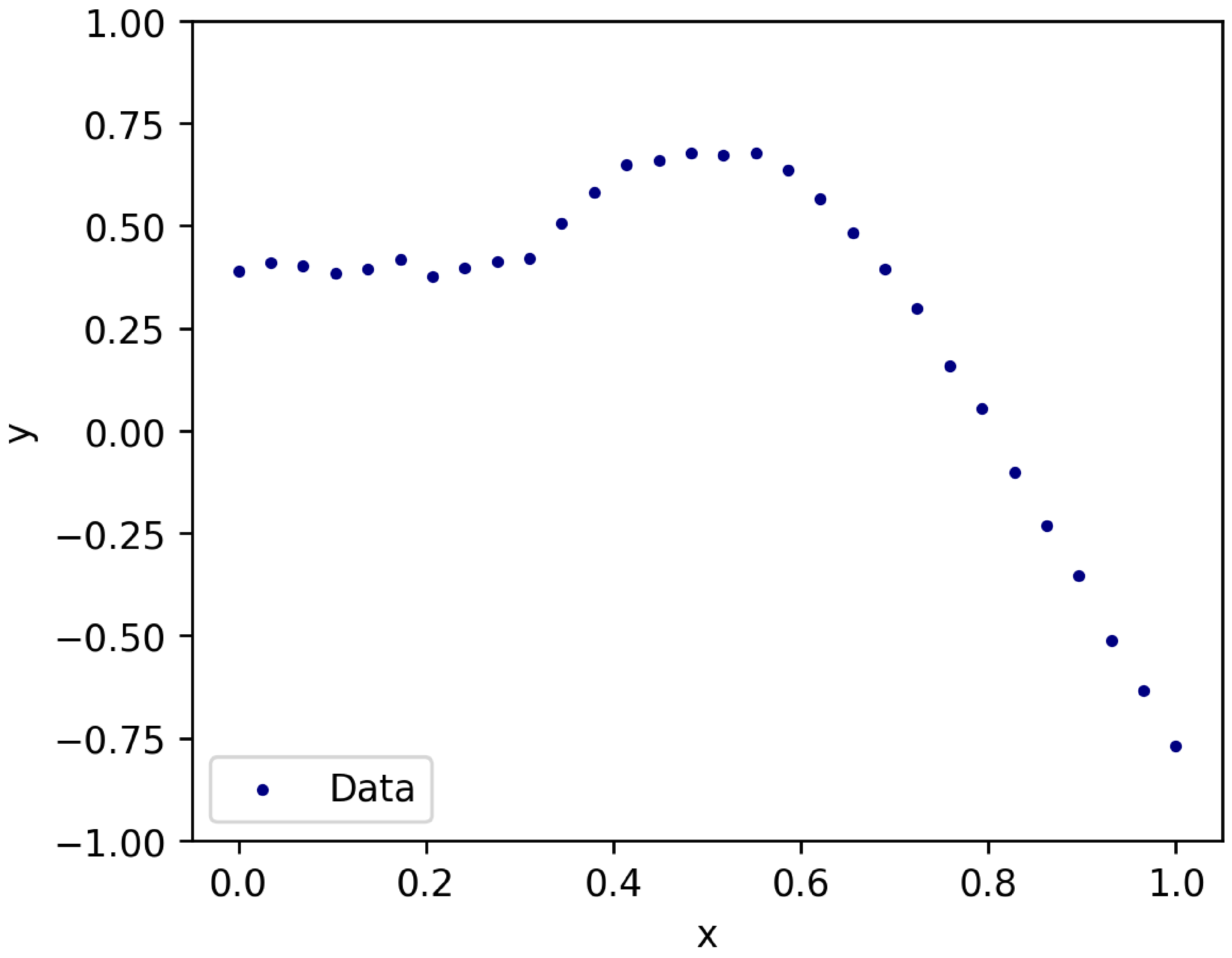
```
1) A good fit can be interpreted by the lowest loss. The activation functions 'tanh' and 'relu' both have converged t
o a very low loss, indicating they have produced a good fit.

2) The 'identity' activation function model seems to have the highest loss and does not seem to have converged much a
t all within the given number of iterations. Thus, it can be considered the slowest in terms of convergence.

3) Comparing 'tanh' and 'relu', which both fit well, the 'relu' model appears to converge faster as its loss decrease
s rapidly and becomes stable in fewer iterations compared to 'tanh'.
```

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:

In [39]:
```
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, 0.275862
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437,  0.37573321,  0.39571087,


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()`

In [40]:
```
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.

In [41]:
```
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.

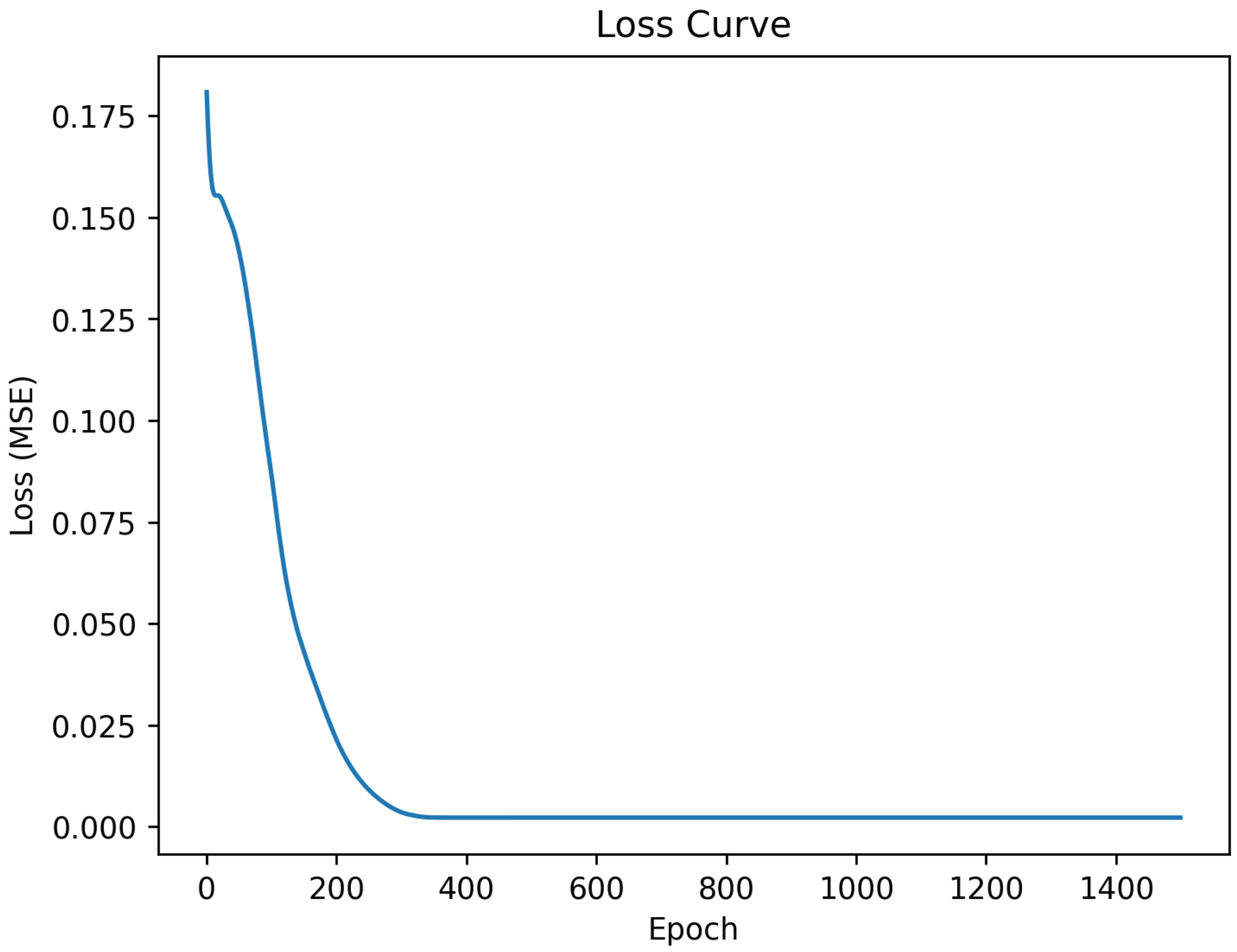
In [42]:
```
model = Net_2_layer(N_hidden = 6, activation = F.relu)
loss_curve = []
```

Training a model

In [43]:
```
# 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()
```

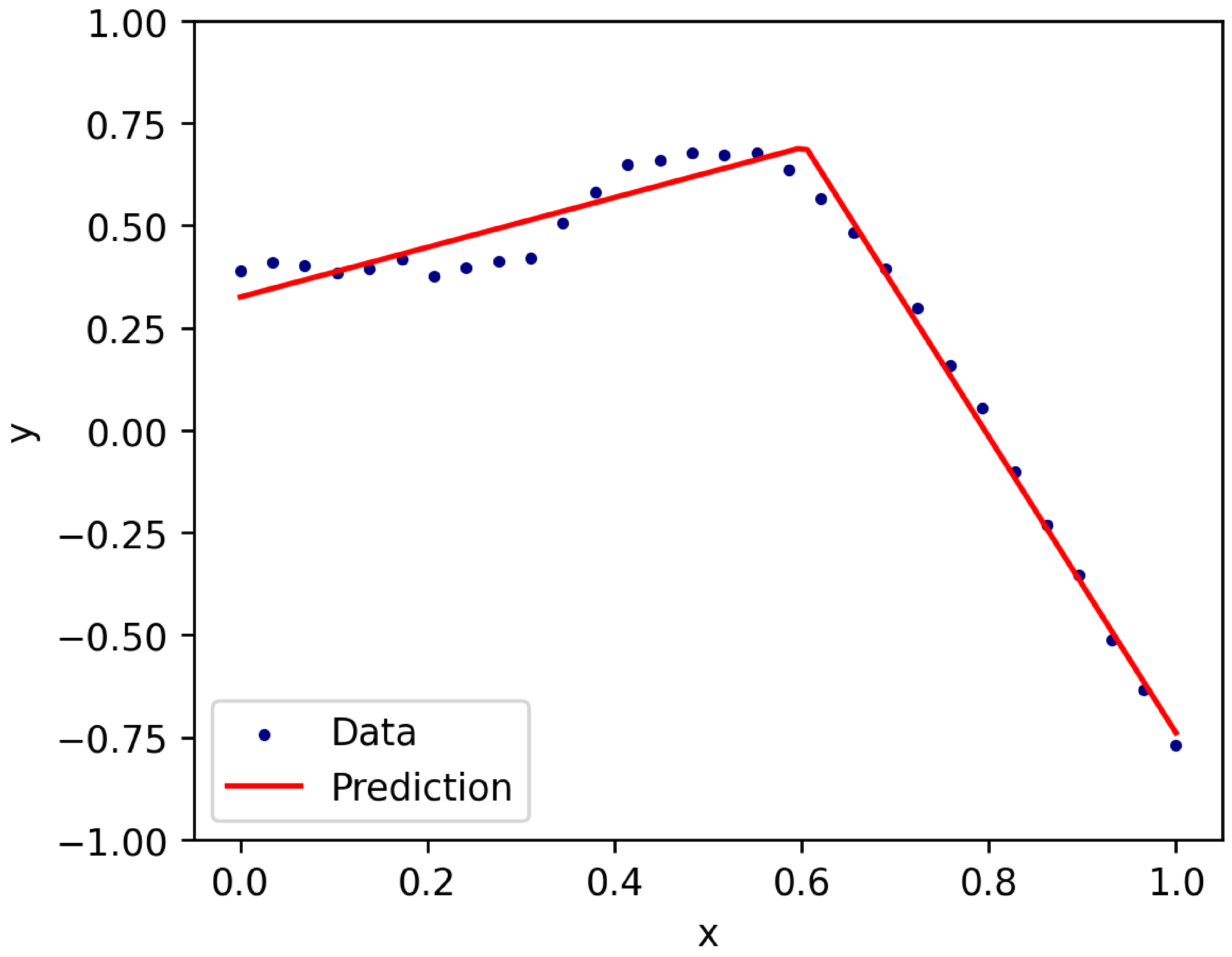
```
Epoch 0 of 1500...      Average loss: 0.18086811900138855
Epoch 60 of 1500...     Average loss: 0.13289208710193634
Epoch 120 of 1500...    Average loss: 0.0629546120762825
Epoch 180 of 1500...    Average loss: 0.0293815229088068
Epoch 240 of 1500...    Average loss: 0.010744189843535423
Epoch 300 of 1500...    Average loss: 0.0035192694049328566
Epoch 360 of 1500...    Average loss: 0.002213196363300085
Epoch 420 of 1500...    Average loss: 0.002206931123510003
Epoch 480 of 1500...    Average loss: 0.0022069127298891544
Epoch 540 of 1500...    Average loss: 0.002206912962719798
Epoch 600 of 1500...    Average loss: 0.00220691179856658
Epoch 660 of 1500...    Average loss: 0.002206912962719798
Epoch 720 of 1500...    Average loss: 0.00220691179856658
Epoch 780 of 1500...    Average loss: 0.0022069120313972235
Epoch 840 of 1500...    Average loss: 0.002206911565735936
Epoch 900 of 1500...    Average loss: 0.00220691179856658
Epoch 960 of 1500...    Average loss: 0.002206911565735936
Epoch 1020 of 1500...   Average loss: 0.0022069124970585108
Epoch 1080 of 1500...   Average loss: 0.002206911565735936
Epoch 1140 of 1500...   Average loss: 0.002206911565735936
Epoch 1200 of 1500...   Average loss: 0.002206911565735936
Epoch 1260 of 1500...   Average loss: 0.0022069124970585108
Epoch 1320 of 1500...   Average loss: 0.00220691179856658
Epoch 1380 of 1500...   Average loss: 0.002206911565735936
Epoch 1440 of 1500...   Average loss: 0.002206911565735936
```

In [44]:
```
plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```



In [45]:
```
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

Your Turn

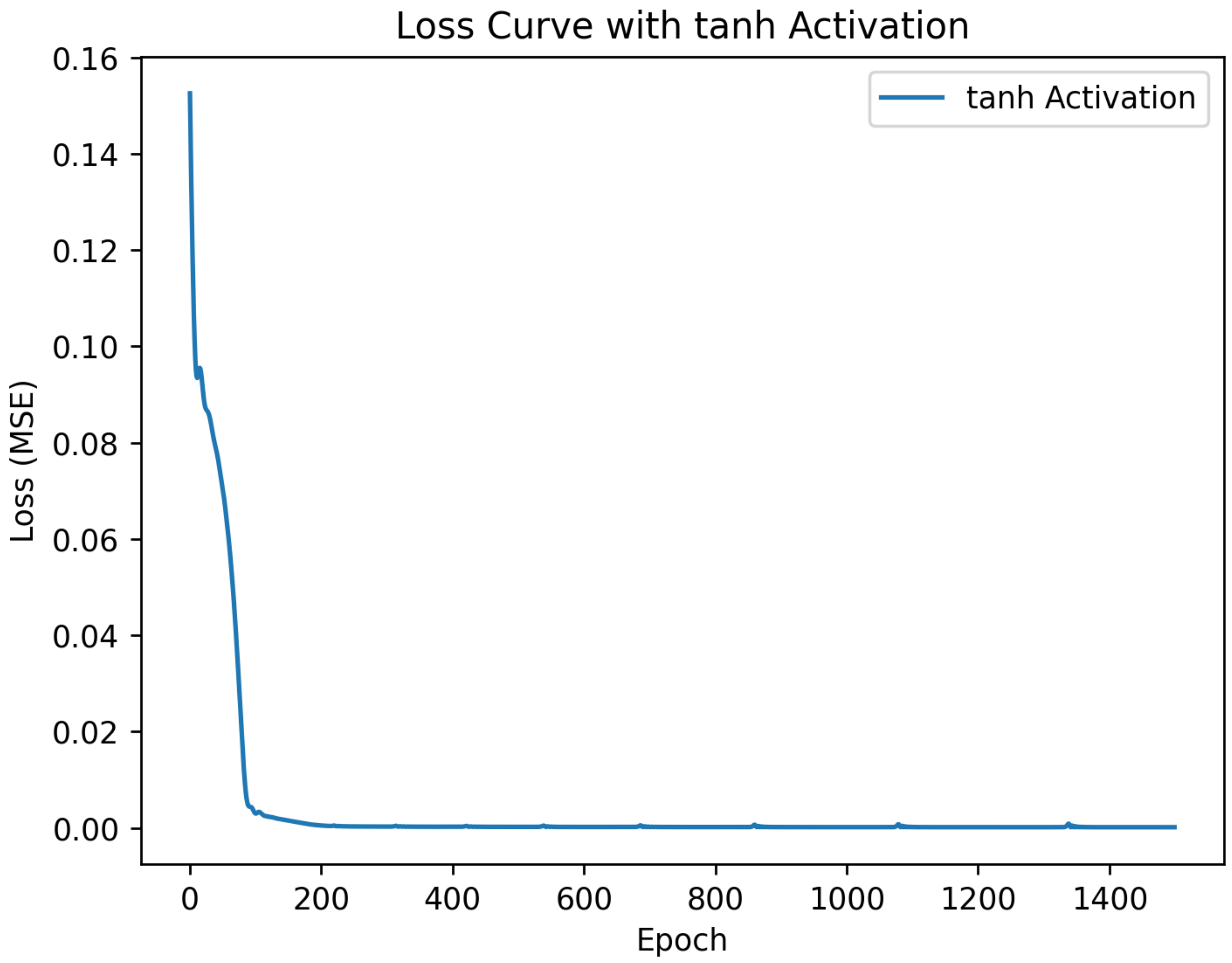
In the cells below, create a new instance of `Net_2_layer`. This time, use 20 neurons per hidden layer, and an activation of `F.tanh`. Plot the loss curve and a visualization of the prediction with the data.

In [46]:
```
# YOUR CODE GOES HERE
model_tanh = Net_2_layer(N_hidden=20,activation=F.tanh)
lr_tanh = 0.005
epochs_tanh = 1500
loss_fcn_tanh = F.mse_loss
loss_curve_tanh = []
opt_tanh = optim.Adam(params=model_tanh.parameters(),lr=lr_tanh)

for i in range(epochs_tanh):
    out_tanh = model_tanh(x)
    loss_tanh = loss_fcn_tanh(out_tanh,y)

    loss_curve_tanh.append(loss_tanh.item())

    if i % int(epochs_tanh/25) == 0:
        print(f"Epoch {i} of {epochs_tanh}... \tAverage loss: {loss_tanh.item()}")

    opt_tanh.zero_grad()
    loss_tanh.backward()
    opt_tanh.step()
```

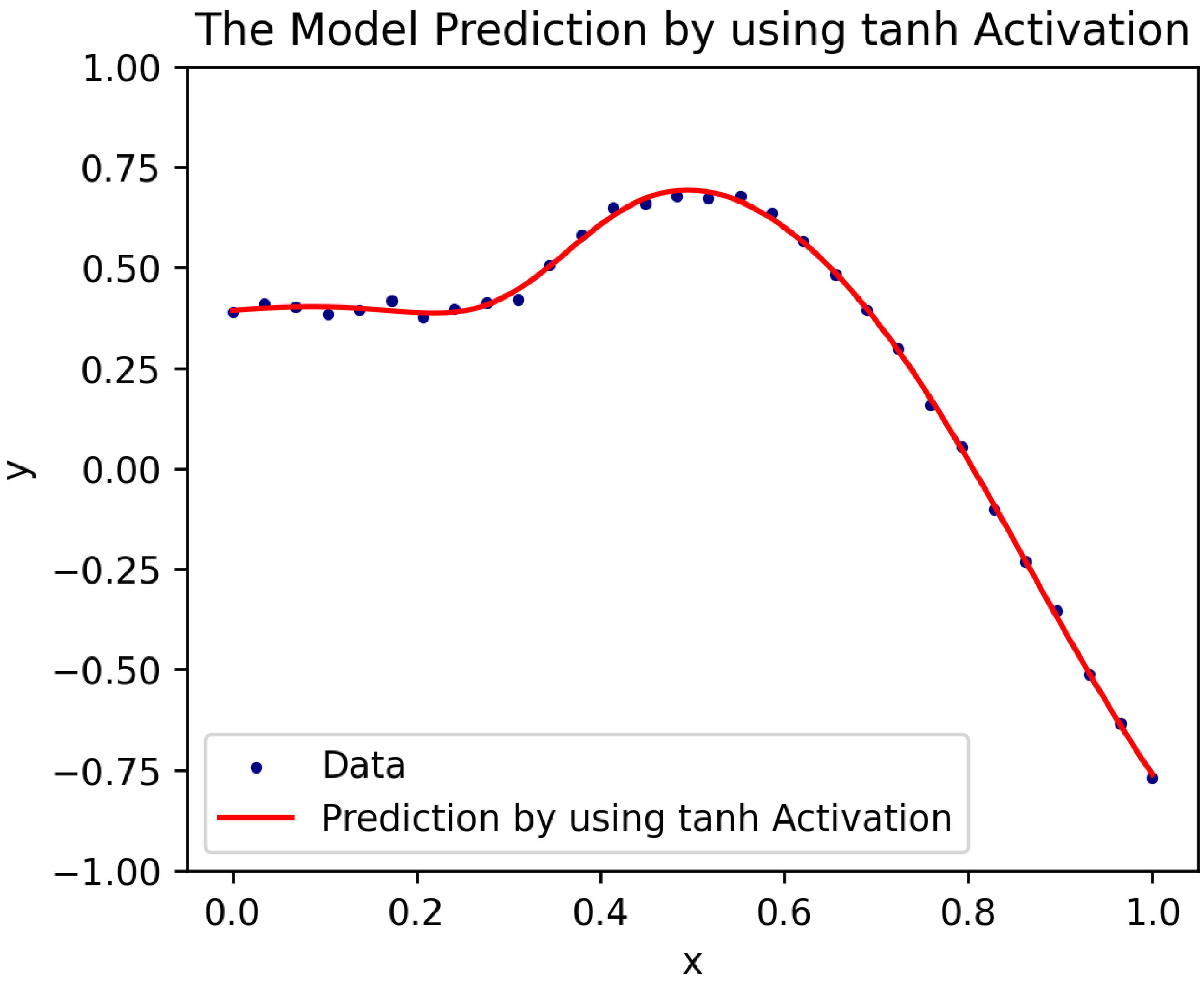
```
Epoch 0 of 1500...      Average loss: 0.152520552277565
Epoch 60 of 1500...     Average loss: 0.058475978672504425
Epoch 120 of 1500...    Average loss: 0.0023163536097854376
Epoch 180 of 1500...    Average loss: 0.0008001626119948924
Epoch 240 of 1500...    Average loss: 0.00032572229974903166
Epoch 300 of 1500...    Average loss: 0.0002710708067752421
Epoch 360 of 1500...    Average loss: 0.0002485666482243687
Epoch 420 of 1500...    Average loss: 0.00039799505611881614
Epoch 480 of 1500...    Average loss: 0.0002188062499044463
Epoch 540 of 1500...    Average loss: 0.00036571003147400916
Epoch 600 of 1500...    Average loss: 0.00019941313075833023
Epoch 660 of 1500...    Average loss: 0.0001909386191982776
Epoch 720 of 1500...    Average loss: 0.0001848832325777039
Epoch 780 of 1500...    Average loss: 0.00017721555195748806
Epoch 840 of 1500...    Average loss: 0.00017133410437963903
Epoch 900 of 1500...    Average loss: 0.00016665832663420588
Epoch 960 of 1500...    Average loss: 0.00016233448695857078
Epoch 1020 of 1500...   Average loss: 0.00015852322394493967
Epoch 1080 of 1500...   Average loss: 0.0005748569965362549
Epoch 1140 of 1500...   Average loss: 0.00015301753592211753
Epoch 1200 of 1500...   Average loss: 0.00015071888628881425
Epoch 1260 of 1500...   Average loss: 0.00014871552411932498
Epoch 1320 of 1500...   Average loss: 0.0001470085553592071
Epoch 1380 of 1500...   Average loss: 0.00014779217599425465
Epoch 1440 of 1500...   Average loss: 0.00014448245929088444
```

In [47]:

```
plt.figure(dpi=250)
plt.plot(loss_curve_tanh, label="tanh Activation")
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve with tanh Activation')
plt.legend()
plt.show()
```


Loss Curve with tanh Activation
tanh Activation
Loss (MSE)
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
200
400
600
800
1000
1200
1400
Epoch

```
In [48]: xs_tanh = torch.linspace(0, 1, 100).reshape(-1, 1)
         ys_tanh = model_tanh(xs_tanh)
         plt.figure(figsize=(5, 4), dpi=250)
         plt.scatter(x, y, s=5, c="navy", label="Data")
         plt.plot(xs_tanh.detach().numpy(), ys_tanh.detach().numpy(), "r-", label="Prediction by using tanh Activation")
         plt.legend(loc="lower left")
         plt.ylim(-1, 1)
         plt.xlabel("x")
         plt.ylabel("y")
         plt.title("The Model Prediction by using tanh Activation")
         plt.show()
```


The Model Prediction by using tanh Activation
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 by using tanh Activation

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

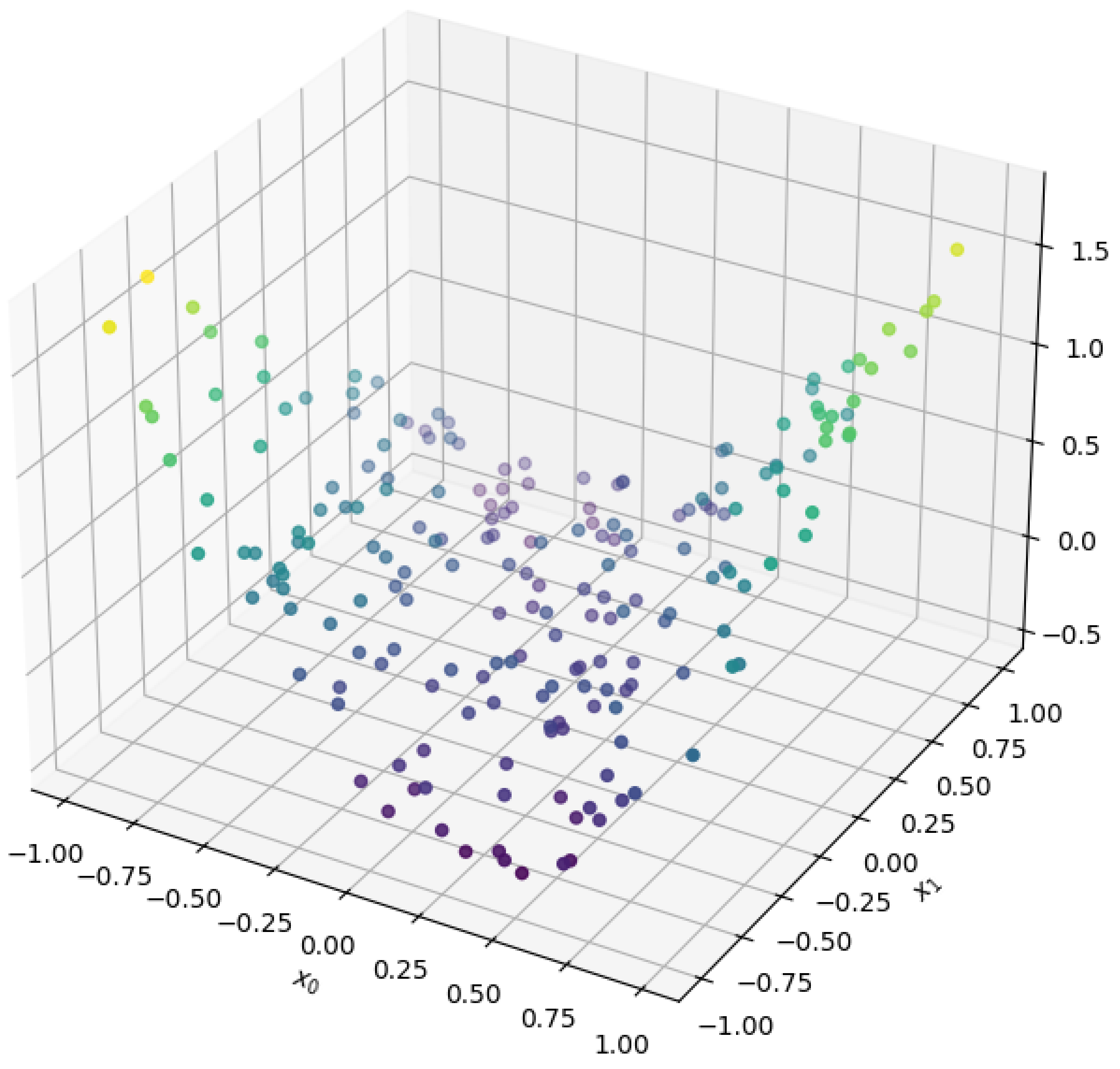
Imports and Utility Functions:

In [2]:
```
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.

In [17]:
```
# YOUR CODE GOES HERE
x_data, y_data = dataGen()
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(111,projection='3d')
ax.scatter(x_data[:,0],x_data[:, 1],y_data,c=y_data,cmap='viridis',marker='o')
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.

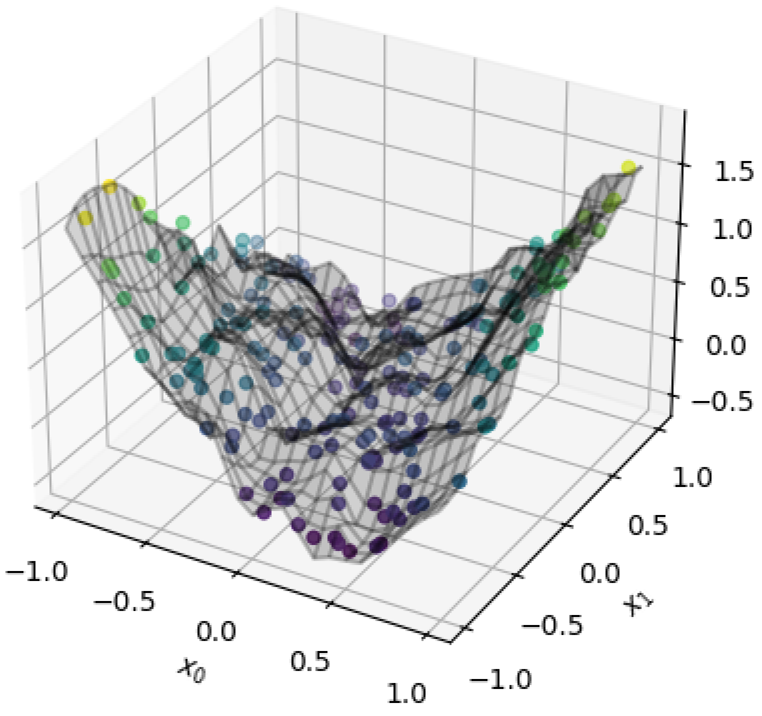
In [25]:
```
# YOUR CODE GOES HERE
import torch.nn.functional as F
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Linear(2,128)
        self.layer2 = nn.Linear(128,64)
        self.layer3 = nn.Linear(64,1)
    def forward(self,x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = self.layer3(x)
        return x
model = NeuralNetwork()
cr = nn.MSELoss()
opt = torch.optim.Adam(model.parameters(), lr=0.01)

x_ten = torch.tensor(x_data,dtype=torch.float32)
y_ten = torch.tensor(y_data,dtype=torch.float32).unsqueeze(1)
epochs = 5000
losses = []
for i in range(epochs):
    ops = model(x_ten)
    loss = cr(ops,y_ten)
    losses.append(loss.item())
    opt.zero_grad()
    loss.backward()
    opt.step()
    if (i+1)%1000 == 0:
        print(f'Epoch [{i+1}/{epochs}] | Loss is = {loss.item():.4f}')

print(f" The Final MSE is =  {losses[-1]:.4f}")
```

```
Epoch [1000/5000] | Loss is = 0.0059
Epoch [2000/5000] | Loss is = 0.0031
Epoch [3000/5000] | Loss is = 0.0021
Epoch [4000/5000] | Loss is = 0.0019
Epoch [5000/5000] | Loss is = 0.0015
 The Final MSE is =  0.0015
```

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

In [8]:
```
# YOUR CODE GOES HERE
# Visualize the trained model
visualizeModel(model)
```



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 cantly from further training.

In [1]:
```
print("""
Mean Squared Error (MSE) of the Trained Model:
The MSE of the trained model on the generated data is 0.0015.

Structure of the Network:
Input Layer: The input layer has 2 nodes corresponding to the two input features x0 and x1.
Hidden Layers: The network has two extra layers apart from the main input and output layers. The first extra layer ha
Output Layer: The output layer has 1 node as the problem is a regression problem and we need to predict a single valu

Activation Function:
The ReLU (Rectified Linear Unit) activation function was used for the hidden layers. ReLU is a popular choice for dee

Loss Function:
Mean Squared Error (MSE) was used as the loss function. MSE is commonly used for regression problems as it quantifies

Optimizer:
The Adam optimizer was used with a learning rate of 0.01. Adam combines the best properties of the AdaGrad and RMSPro

Number of Training Epochs:
The model was trained for 5000 epochs. This choice was made based on observing the convergence of the loss. Training 

Reasoning for Choices:
The choice of 2 hidden layers with 128 and 64 neurons was made to provide the model with enough capacity to learn the
The ReLU activation function is computationally efficient and helps in preventing the vanishing gradient problem, esp
Adam optimizer was chosen because of its adaptive learning rate properties, making the training process faster and mo
A learning rate of 0.01 was found to be effective through experimentation. It was neither too high nor too low.
5000 epochs ensured that the model had sufficient iterations to converge to a low error.
""")
```

```
Mean Squared Error (MSE) of the Trained Model:
The MSE of the trained model on the generated data is 0.0015.

Structure of the Network:
Input Layer: The input layer has 2 nodes corresponding to the two input features x0 and x1.
Hidden Layers: The network has two extra layers apart from the main input and output layers. The first extra layer ha
s 128 points and the second one has 64 points. I picked these numbers by testing different options. Generally, in dee
p learning, more layers and points can better understand complex data, but too many can cause issues. So, I began wit
h a reasonable number and adjusted based on results.
Output Layer: The output layer has 1 node as the problem is a regression problem and we need to predict a single valu
e.

Activation Function:
The ReLU (Rectified Linear Unit) activation function was used for the hidden layers. ReLU is a popular choice for dee
p neural networks as it helps in introducing non-linearity to the model without suffering from the vanishing gradient
problem. It also computationally more efficient compared to other activation functions like sigmoid or tanh.

Loss Function:
Mean Squared Error (MSE) was used as the loss function. MSE is commonly used for regression problems as it quantifies
the difference between the predicted and actual values.

Optimizer:
The Adam optimizer was used with a learning rate of 0.01. Adam combines the best properties of the AdaGrad and RMSPro
p algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. The choice of a
learning rate is often empirical, but 0.01 was found to provide a good balance between fast convergence and model sta
bility for this dataset.

Number of Training Epochs:
The model was trained for 5000 epochs. This choice was made based on observing the convergence of the loss. Training
was stopped once the loss plateaued, indicating that the model might not benefit significantly from further training.

Reasoning for Choices:
The choice of 2 hidden layers with 128 and 64 neurons was made to provide the model with enough capacity to learn the
complex relationship between the inputs and the outputs.
The ReLU activation function is computationally efficient and helps in preventing the vanishing gradient problem, esp
ecially in deep networks.
Adam optimizer was chosen because of its adaptive learning rate properties, making the training process faster and mo
re robust.
A learning rate of 0.01 was found to be effective through experimentation. It was neither too high nor too low.
5000 epochs ensured that the model had sufficient iterations to converge to a low error.
```

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

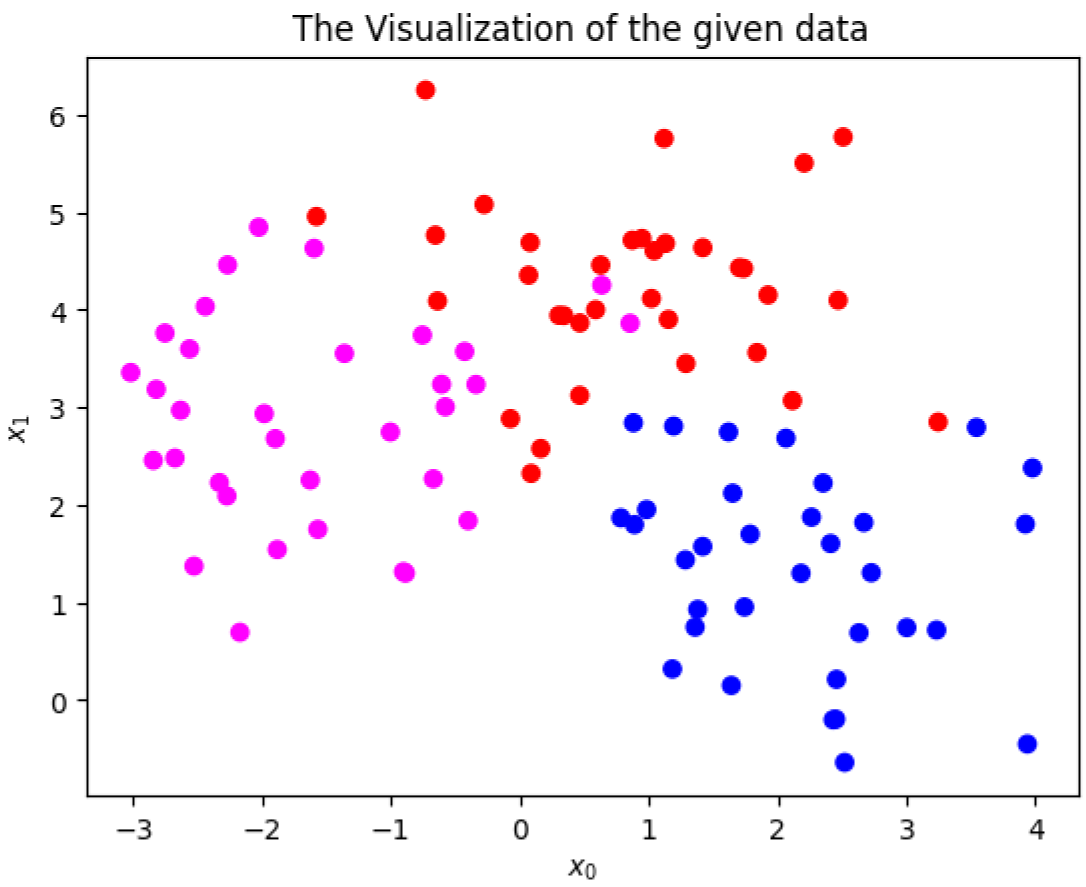
Imports and Utility Functions:

In [56]:
```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_logits = model(Xgrid)
    Ypred_probs = F.softmax(Ypred_logits, dim=1)
    Ypred = torch.argmax(Ypred_probs, dim=1)
    # Plot data
    plt.scatter(x[:,0], x[:,1], c = y, cmap = ListedColormap(['red','blue','magenta']))
    # Plot model
    plt.contourf(X0.detach().numpy(), X1.detach().numpy(), Ypred.reshape(n,n).detach().numpy(), cmap = ListedColormap
    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.

In [57]:
```
# YOUR CODE GOES HERE
x, y = dataGen()
plt.scatter(x[:, 0],x[:, 1],c=y,cmap=ListedColormap(['red','blue','magenta']))
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.title('The Visualization of the given data')
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.

In [58]:
```
# YOUR CODE GOES HERE
import torch.nn.functional as F
```

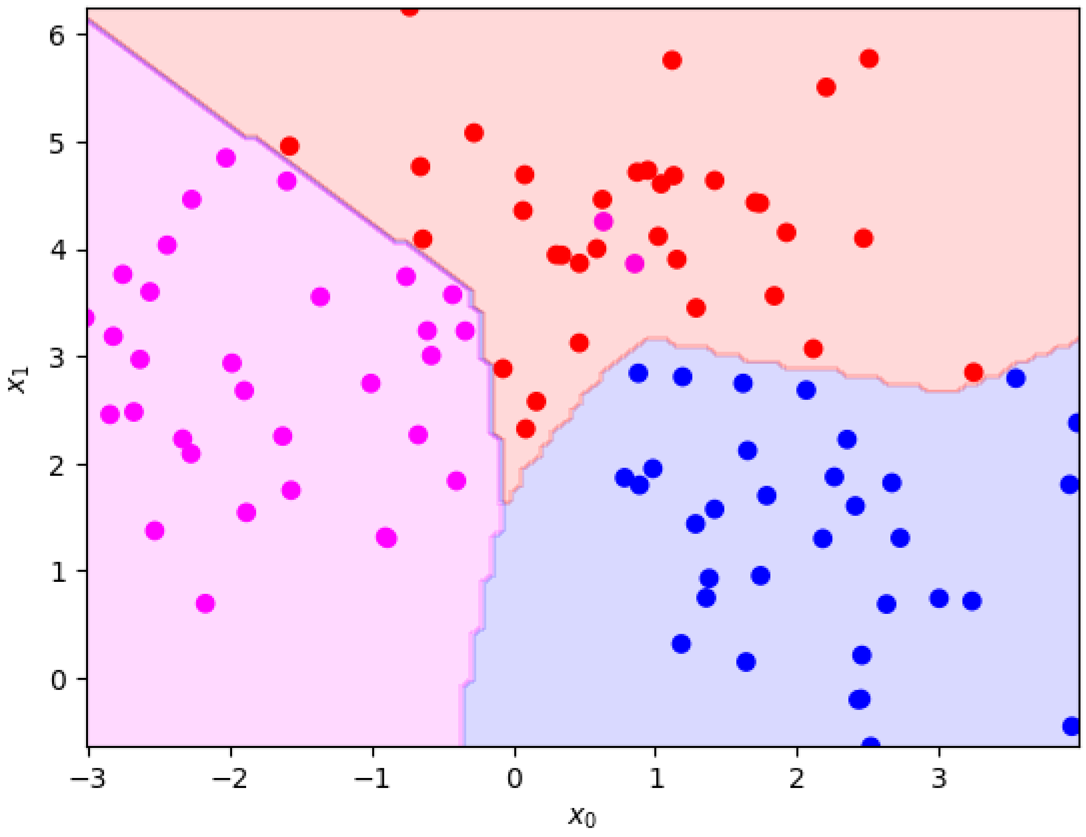
```
class ClassificationNN(nn.Module):
    def __init__(self):
        super(ClassificationNN, self).__init__()
        self.layer1 = nn.Linear(2,128)
        self.layer2 = nn.Linear(128,64)
        self.layer3 = nn.Linear(64,3)
    def forward(self,x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.softmax(self.layer3(x),dim=1)
        return x

model = ClassificationNN()
cr = nn.CrossEntropyLoss()
opt = torch.optim.Adam(model.parameters(),lr=0.01)
x_ten = torch.tensor(x,dtype=torch.float32)
y_ten = torch.tensor(y,dtype=torch.long)
epochs = 1000
for i in range(epochs):
    ops = model(x_ten)
    loss = cr(ops,y_ten)
    opt.zero_grad()
    loss.backward()
    opt.step()
    if (i + 1)%100 == 0:
        _,predicted = torch.max(ops,1)
        acc = (predicted == y_ten).sum().item()/len(y_ten)
        print(f'Epoch [{i + 1}/{epochs}] | Loss is = {loss.item():.4f} | Accuracy is = {acc:.4f}')
```

```
Epoch [100/1000] | Loss is = 0.5934 | Accuracy is = 0.9600
Epoch [200/1000] | Loss is = 0.5723 | Accuracy is = 0.9800
Epoch [300/1000] | Loss is = 0.5718 | Accuracy is = 0.9800
Epoch [400/1000] | Loss is = 0.5716 | Accuracy is = 0.9800
Epoch [500/1000] | Loss is = 0.5716 | Accuracy is = 0.9800
Epoch [600/1000] | Loss is = 0.5715 | Accuracy is = 0.9800
Epoch [700/1000] | Loss is = 0.5715 | Accuracy is = 0.9800
Epoch [800/1000] | Loss is = 0.5715 | Accuracy is = 0.9800
Epoch [900/1000] | Loss is = 0.5715 | Accuracy is = 0.9800
Epoch [1000/1000] | Loss is = 0.5715 | Accuracy is = 0.9800
```

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

In [59]:
```
# YOUR CODE GOES HERE
visualizeModel(model)
```



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.

YOUR ANSWER GOES HERE of the data points correctly.

In [60]:
```
print("""
Accuracy is = 0.98

Network Structure:
The network consists of three layers:
An input layer with 2 nodes, reflecting the two features of our data.
Two hidden layers: The first hidden layer has 128 nodes and the second one has 64 nodes.
An output layer with 3 nodes. This is because we have three classes in our classification problem.

Activation Functions:
For the hidden layers, I used the Rectified Linear Unit (ReLU) activation function. ReLU is a popular choice for feed

Loss Function:
I used the Cross-Entropy Loss for this multi-class classification problem. This loss function is suitable for classif

Optimizer:
The optimizer I used is the Adam optimizer with a learning rate of 0.01. Adam is an efficient gradient descent optimi

Learning Rate:
The learning rate was set to 0.01. The learning rate determines the step size during gradient descent. A smaller lear

Training Epochs:
The model was trained for 1000 epochs. The accuracy plateaued after around 200 epochs, indicating that further traini

Reasoning:
Given the scatter plot of the data, it was clear that the data points for the three classes were somewhat clustered b
The high accuracy achieved (98%) indicates that the network was able to effectively classify the majority of the data
""")
```

```
Accuracy is = 0.98

Network Structure:
The network consists of three layers:
An input layer with 2 nodes, reflecting the two features of our data.
Two hidden layers: The first hidden layer has 128 nodes and the second one has 64 nodes.
An output layer with 3 nodes. This is because we have three classes in our classification problem.

Activation Functions:
For the hidden layers, I used the Rectified Linear Unit (ReLU) activation function. ReLU is a popular choice for feed
forward neural networks due to its non-linearity and the fact that it can help prevent the vanishing gradient proble
m. For the output layer, I used the softmax function to obtain the probabilities for each of the three classes.

Loss Function:
I used the Cross-Entropy Loss for this multi-class classification problem. This loss function is suitable for classif
ication problems as it measures the performance of a classification model whose output is a probability value between
0 and 1.

Optimizer:
The optimizer I used is the Adam optimizer with a learning rate of 0.01. Adam is an efficient gradient descent optimi
zation algorithm that can handle sparse gradients and is well suited for problems that are large in terms of data and
parameters.

Learning Rate:
The learning rate was set to 0.01. The learning rate determines the step size during gradient descent. A smaller lear
ning rate might converge to a solution more reliably but could be slower, while a larger learning rate can speed up t
raining but may cause the model to overshoot the minimum.

Training Epochs:
The model was trained for 1000 epochs. The accuracy plateaued after around 200 epochs, indicating that further traini
ng might not significantly improve the performance. This is evident from the consistent accuracy of 0.9800 from epoch
200 onwards.

Reasoning:
Given the scatter plot of the data, it was clear that the data points for the three classes were somewhat clustered b
ut also had some overlapping regions. This motivated the use of a neural network with multiple hidden layers to captu
re this complexity. The model was designed to gradually decrease the number of nodes from the input layer to the outp
ut layer. This is a common practice to prevent overfitting and reduce the computational demand.
The high accuracy achieved (98%) indicates that the network was able to effectively classify the majority of the data
points correctly.
```