Problem 1
0.0474

Problem 2
W_3

Problem 3
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
3 .Leaky ReLU is a variant of ReLU that allows a small non-zero gradient for negative input values

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 [1]:
```
import numpy as np
np.set_printoptions(precision=3)
```

Out[1]:
```
<Token var=<ContextVar name='format_options' default={'edgeitems': 3, 'threshold': 1000, 'floatmode': 'maxprec'
, 'precision': 8, 'suppress': False, 'linewidth': 75, 'nanstr': 'nan', 'infstr': 'inf', 'sign': '-', 'formatter
': None, 'legacy': 9223372036854775807, 'override_repr': None} at 0x0000022BE74DE700> at 0x0000022BE74BBF80>
```

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 [2]:
```
def perceptron_layer(x, weight, bias):
    # YOUR CODE GOES HERE
    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. ]]
```

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 [26]:
```
def sigmoid(x):
    return 1./(1.+np.exp(-x))
```

```python
def MLP(x, weights, biases):
    mlp = x

    # YOUR CODE GOES HERE
    for i in range(len(weights)):
        mlp = mlp @ weights[i].T + biases[i]
        if (i < len(weights) - 1):
            mlp = sigmoid(mlp)

    return mlp
```

In [27]:

```python
# 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
```

Processing math: 100%

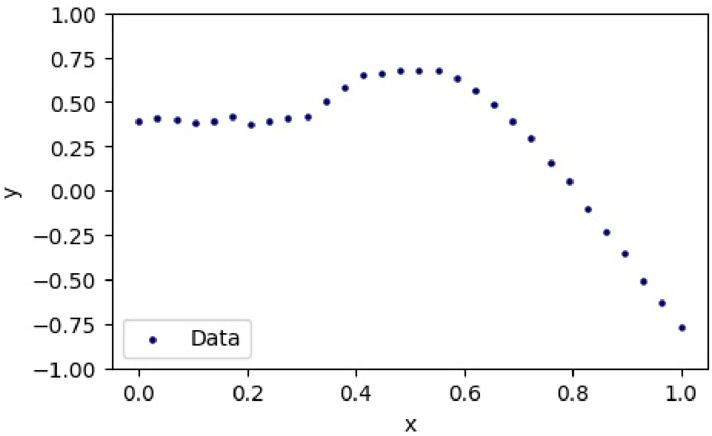
M7-L1 Problem 2

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 [5]:
```
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, 0.2
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437,  0.37573321,  0.39571

weights = [np.array([[-5.90378086,  0,  0 ]]).T,
           np.array([[ 0.8996511 ,  4.75805319, -0.95266992],[-0.99667812, -0.89303165,  3.19020423],[-1.652134
           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 ]),

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 [6]:
```
# YOUR CODE GOES HERE
def sigmoid(x):
    return 1./(1.+np.exp(-x))

def MLP(x, weights, biases):
    mlp = x

    # YOUR CODE GOES HERE
    for i in range(len(weights)):
        mlp = mlp @ weights[i].T + biases[i]
        if (i < len(weights) - 1):
            mlp = sigmoid(mlp)

    return mlp
```

Varying weights

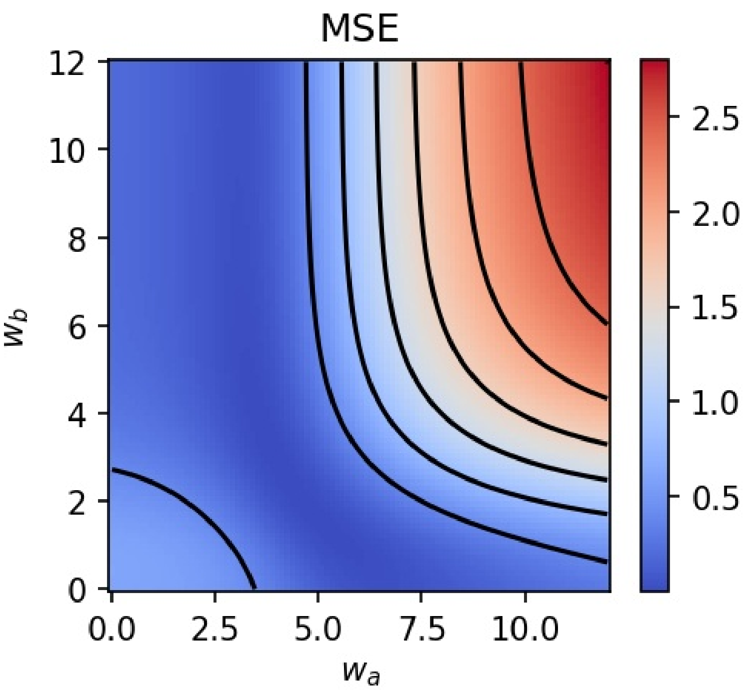
The provided network has 2 hidden layers, each with 3 neurons. The weights and biases are shown below. Note the weights w_a and w_b --

these are left for you to investigate:

$$x\ (N \times 1) \to \sigma\left(w = \begin{bmatrix}-5.9\\ w_a\\ w_b\end{bmatrix}; b = \begin{bmatrix}2.02\\ -3.48\\ -1.12\end{bmatrix}'\right) \to (N \times 3) \to \sigma\left(w = \begin{bmatrix}0.9 & -1. & -1.65\\ 4.76 & -0.89 & -2.93\\ -0.95 & 3.19 & 2.61\end{bmatrix}; b = \begin{bmatrix}1.35\\ -0.11\\ -4.03\end{bmatrix}'\right) \to (N \times 3) \to \sigma\left(w = \begin{bmatrix}1.72\\ -1.56\\ -3.31\end{bmatrix}'; b = \right.$$

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

In [7]:
```
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 [8]:
```
%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("$w_a$")
```

```python
    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[8]:
```
interactive(children=(FloatSlider(value=0.0, description='wa', layout=Layout(width='550px'), 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)?

- *ANSWER:* wb = 3.0 gives an MSE value of 0.001 at wa = 4.0

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

- *ANSWER:* For large wa and wb, we can see areas of high error, suggesting that such weight values do not lead to a good fit for the data. This is likely due to high weights, causing extreme outputs that do not match the small variations in the target data. The high Mean Squared Error (MSE = 2.801) also reflects this mismatch, suggesting that the model is overfitting or saturating due to these large weights.

Processing math: 100%

M7-L2 Problem 1

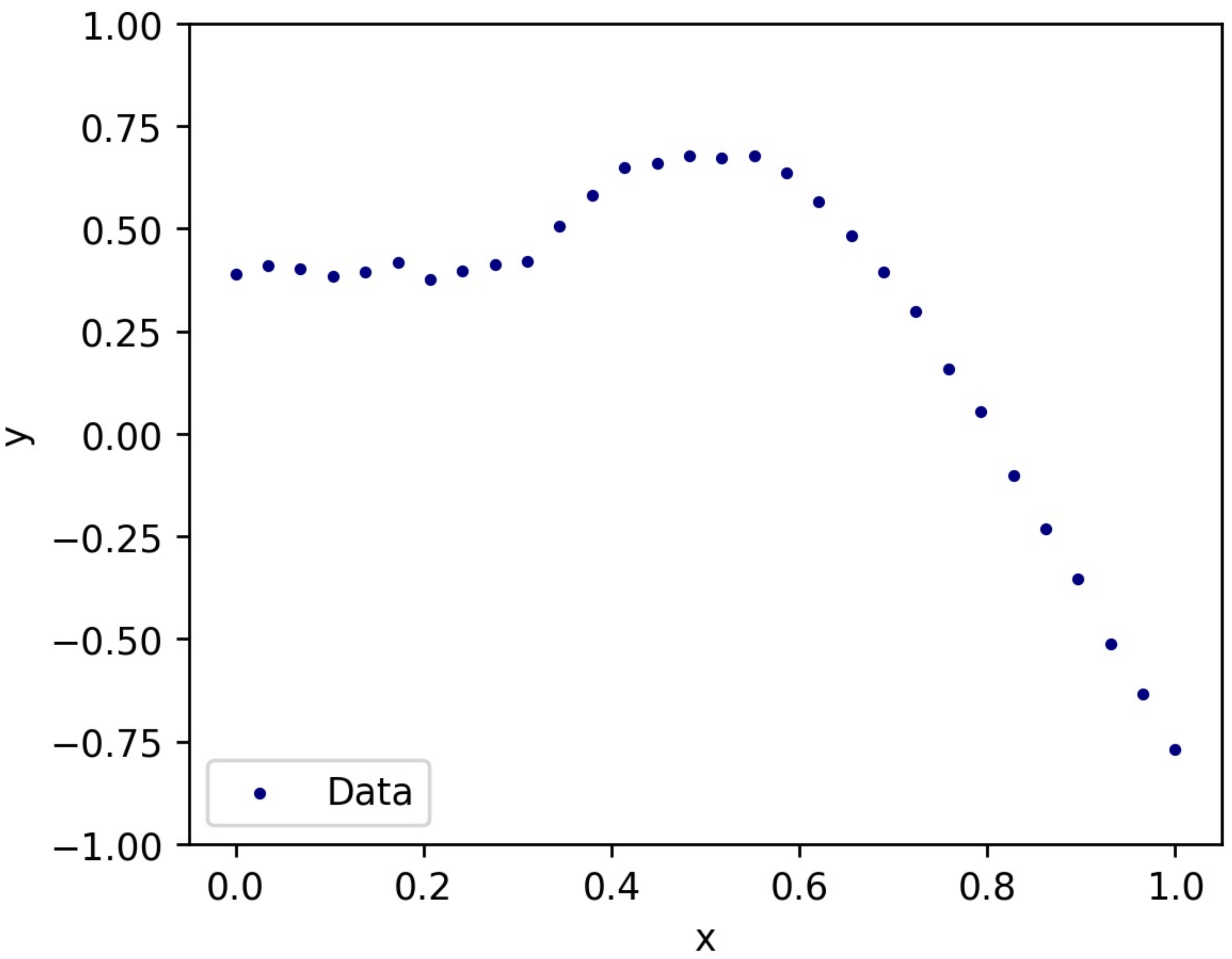
In this function you will:

- 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 [29]:

```
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, 0.2
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437,  0.37573321,  0.39571


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



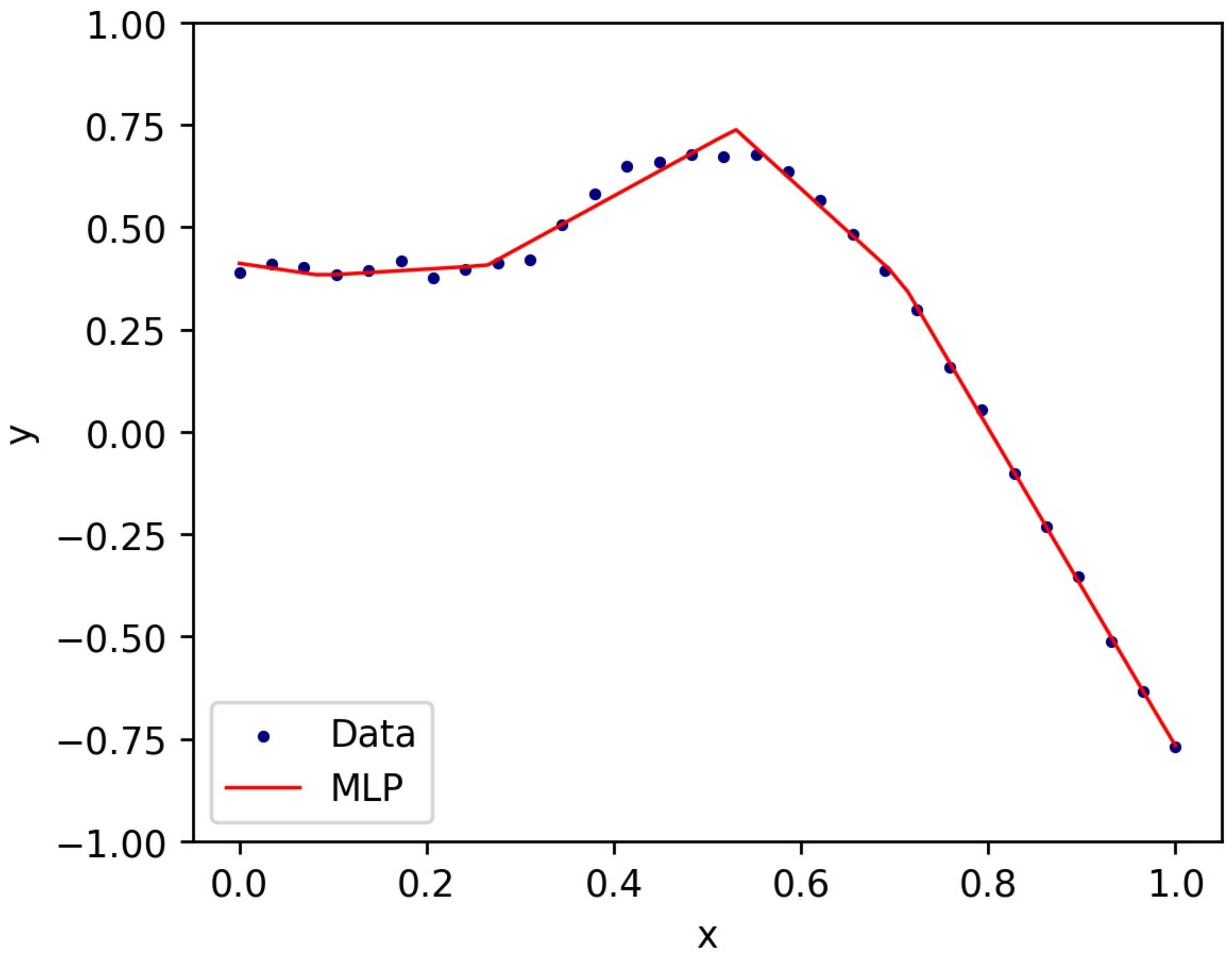
`MLPRegressor()`

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

In [30]:

```
mlp = MLPRegressor(hidden_layer_sizes=[10,10], max_iter = 10000, tol = 1e-9) # 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?

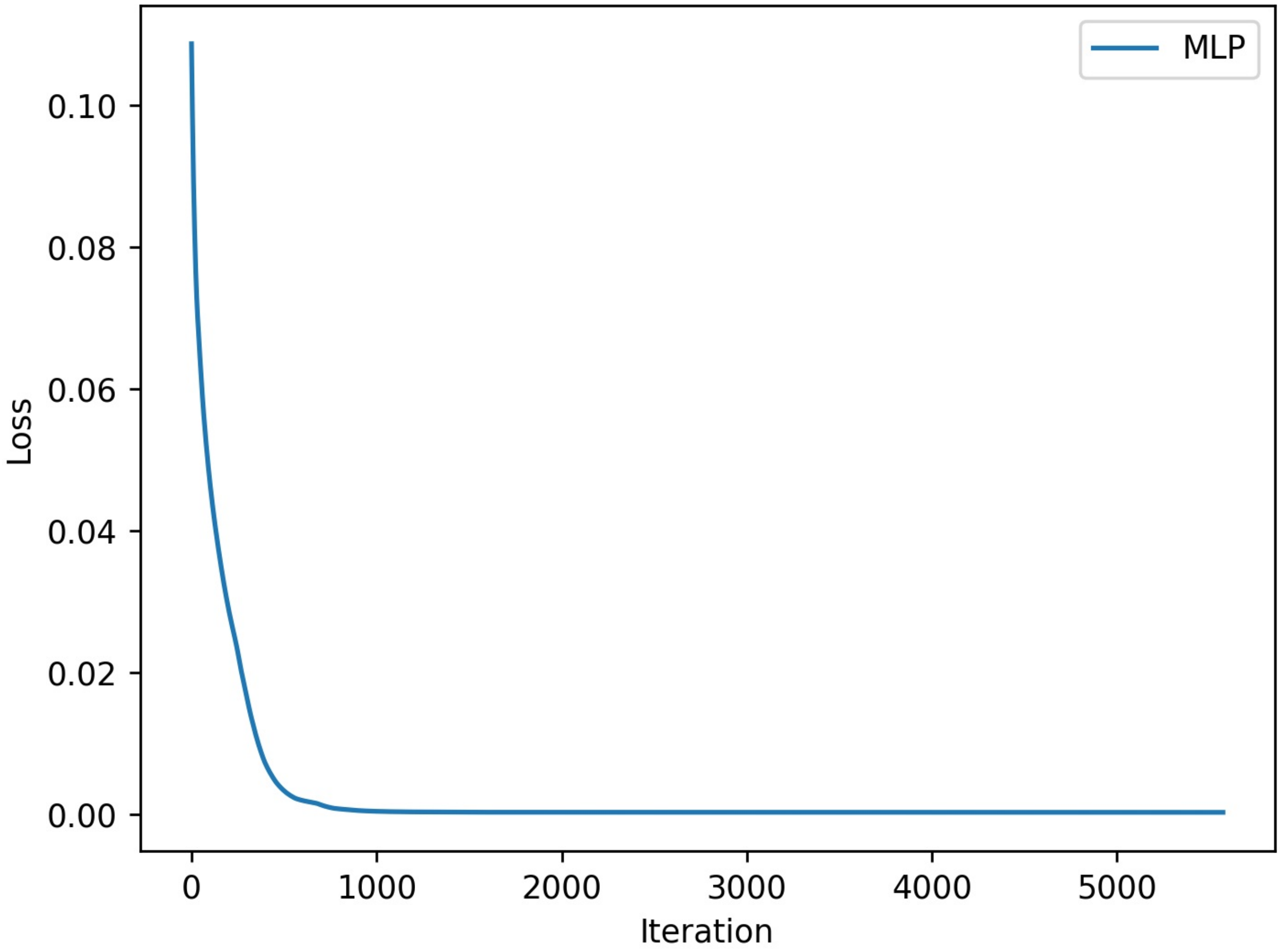
Ans: For max_iter = 10000 and tol = 1e-9, we get a good fit to the data. However, it is probable that the model is overfit to the sample data

Loss Curve

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

In [31]:
```
loss = mlp.loss_curve_
```

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



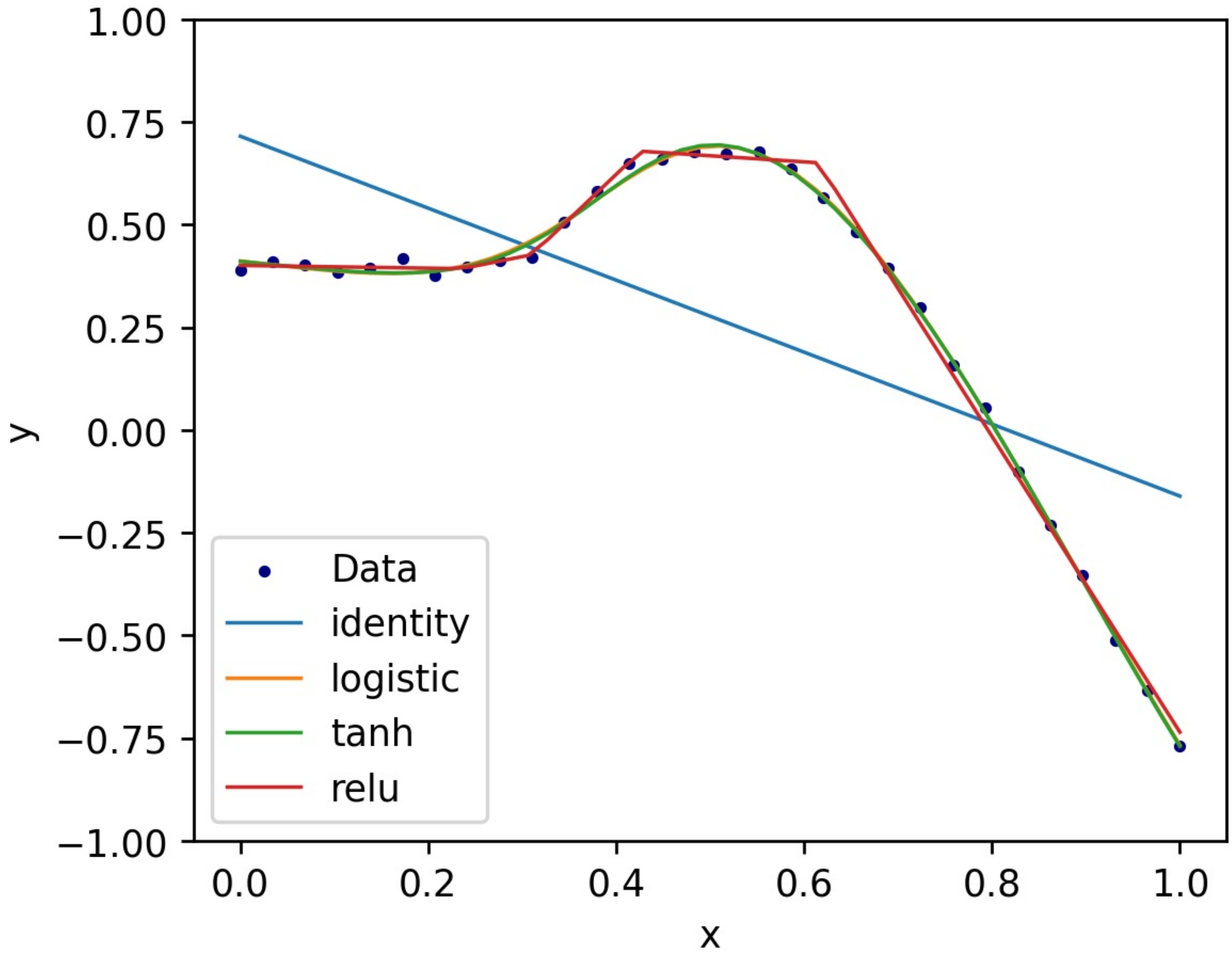
Activation Functions

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 [32]:
```
activations = ["identity","logistic","tanh","relu"]
models = dict()

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

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

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

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

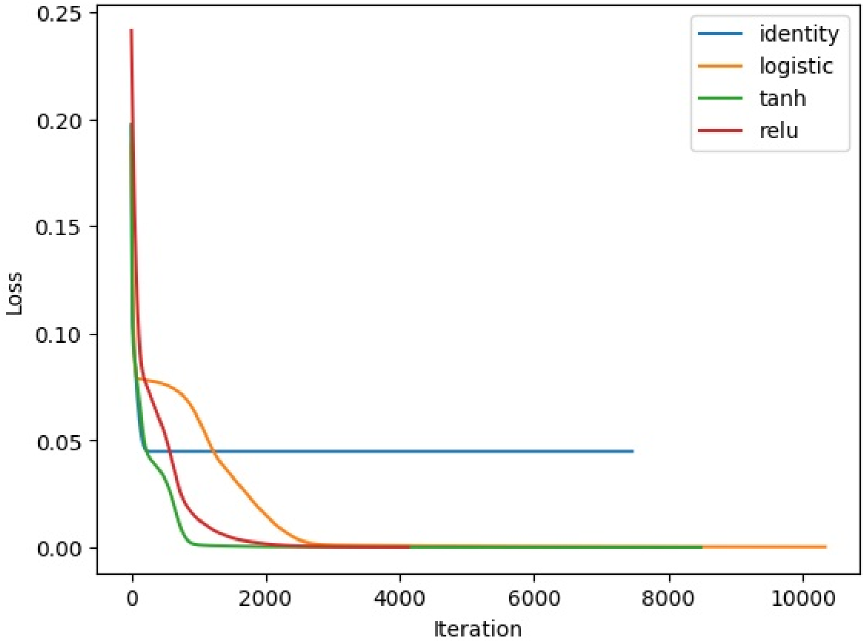


Loss curves

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

In [36]:

```
# YOUR CODE GOES HERE
losses = dict()

for act in activations:
    losses[act] = models[act].loss_curve_

    plt.plot(losses[act], label = act)

# plt.xlim([0, 2000])
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Questions

2. Which activation functions produced a good fit?

Ans: The tanh and logistic functions created good fits. The relu function also produces a reasonable fit to the data, but the curve is not very smooth.

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

Ans: The logistic activation function takes the longest to converge.

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

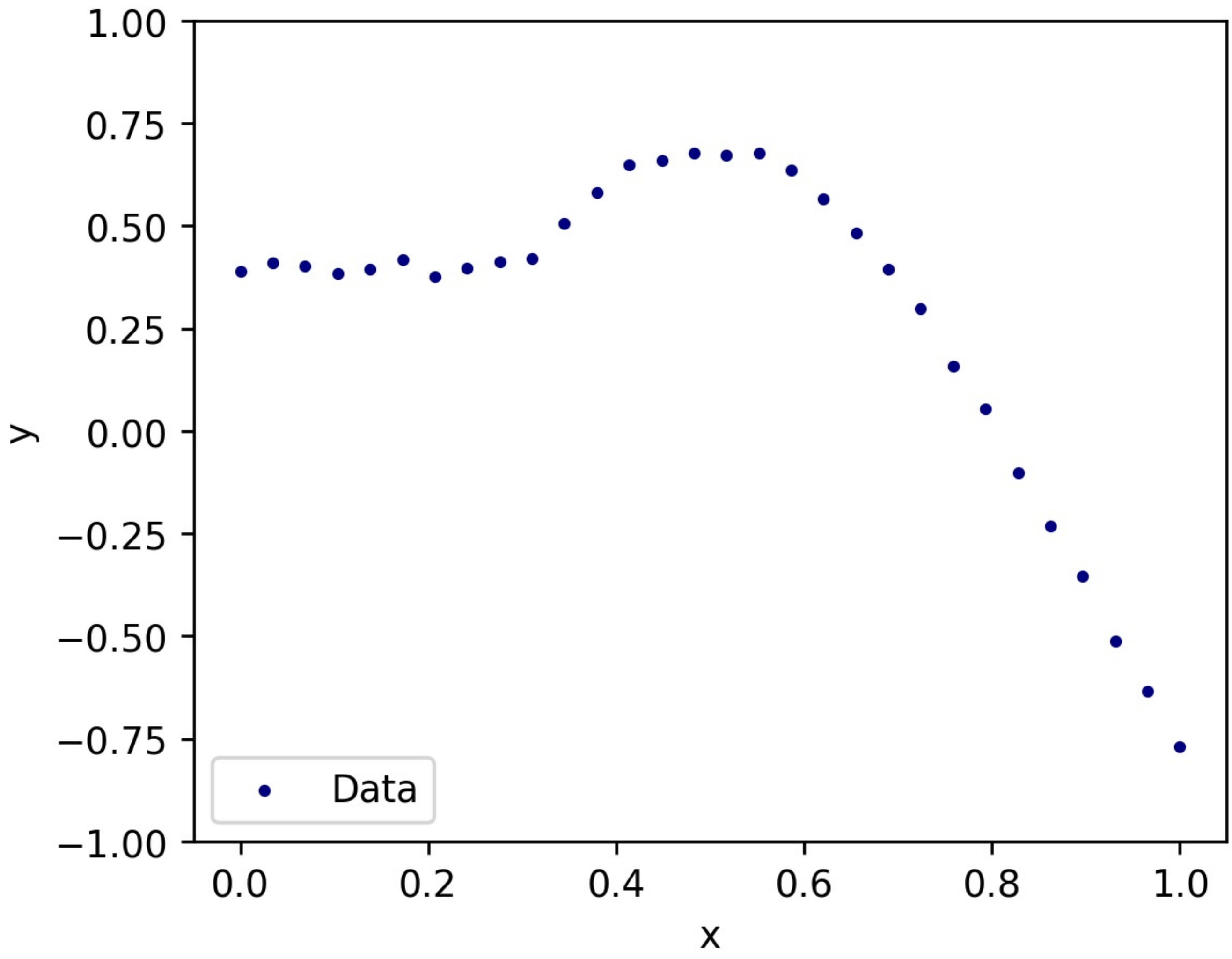
Ans: The tanh function converged in the least number of iterations among the activation functions that produced a "good" fit

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 [2]:
```
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, 0.1
y = np.array([ 0.38914369,  0.40997345,  0.40282978,  0.38493705,  0.394214  ,0.41651437,  0.37573321,  0.39571


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 [3]:
```
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 [4]:
```
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 [5]:
```
model = Net_2_layer(N_hidden = 6, activation = F.relu)
loss_curve = []
```

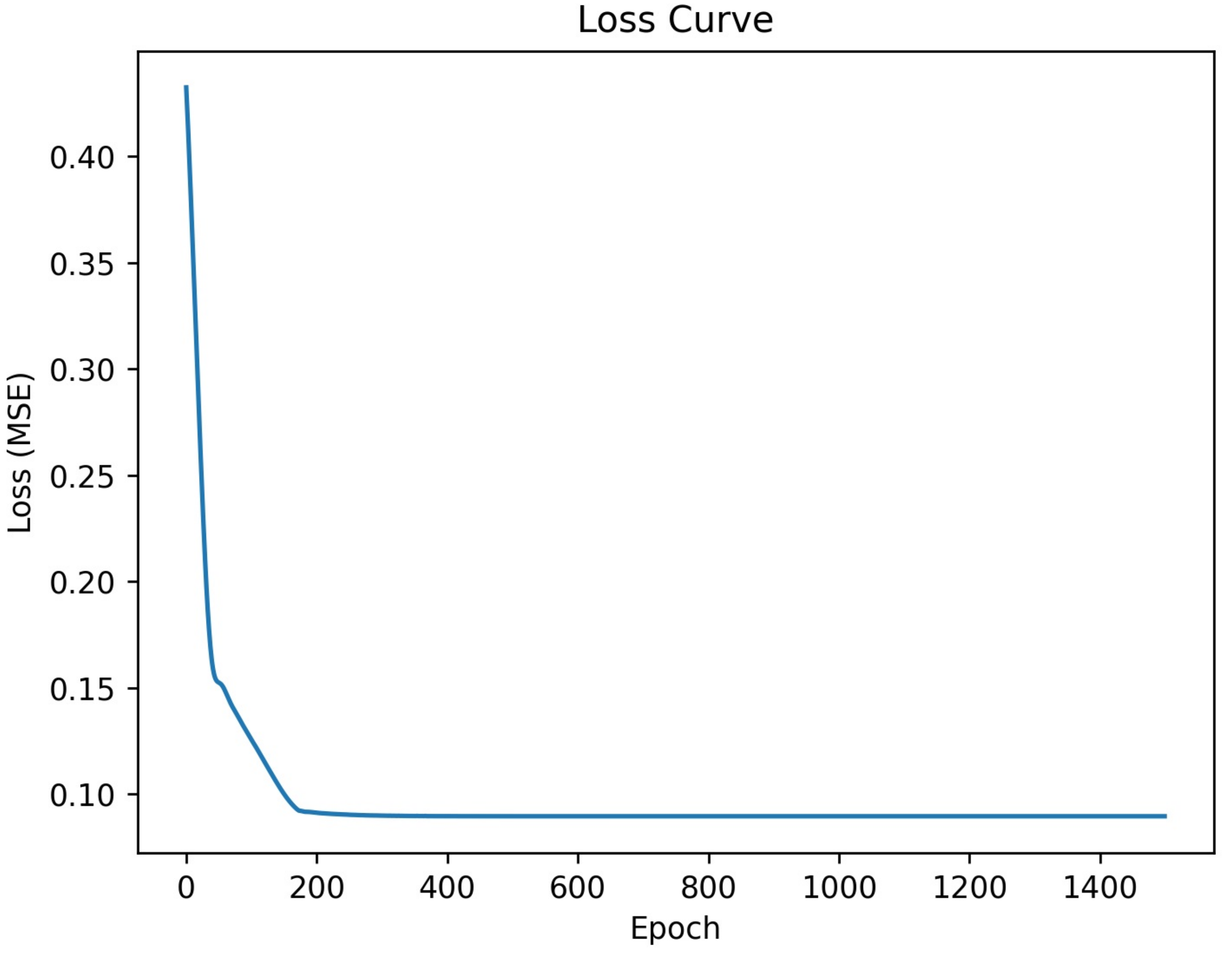
Training a model

In [6]:
```
# 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: 0.43229278922080994
Epoch 60 of 1500...     Average loss: 0.14824233949184418
Epoch 120 of 1500...    Average loss: 0.11515723913908005
Epoch 180 of 1500...    Average loss: 0.09178225696086884
Epoch 240 of 1500...    Average loss: 0.09043918550014496
Epoch 300 of 1500...    Average loss: 0.08989021182060242
Epoch 360 of 1500...    Average loss: 0.08975287526845932
Epoch 420 of 1500...    Average loss: 0.0896228551864624
Epoch 480 of 1500...    Average loss: 0.08960828185081482
Epoch 540 of 1500...    Average loss: 0.08960011601448059
Epoch 600 of 1500...    Average loss: 0.08959943801164627
Epoch 660 of 1500...    Average loss: 0.08959944546222687
Epoch 720 of 1500...    Average loss: 0.08959944546222687
Epoch 780 of 1500...    Average loss: 0.08959943801164627
Epoch 840 of 1500...    Average loss: 0.08959944546222687
Epoch 900 of 1500...    Average loss: 0.08959944546222687
Epoch 960 of 1500...    Average loss: 0.08959944546222687
Epoch 1020 of 1500...   Average loss: 0.08959944546222687
Epoch 1080 of 1500...   Average loss: 0.08959944546222687
Epoch 1140 of 1500...   Average loss: 0.08959944546222687
Epoch 1200 of 1500...   Average loss: 0.08959944546222687
Epoch 1260 of 1500...   Average loss: 0.08959944546222687
Epoch 1320 of 1500...   Average loss: 0.08959944546222687
Epoch 1380 of 1500...   Average loss: 0.08959944546222687
Epoch 1440 of 1500...   Average loss: 0.08959944546222687
```

In [7]:

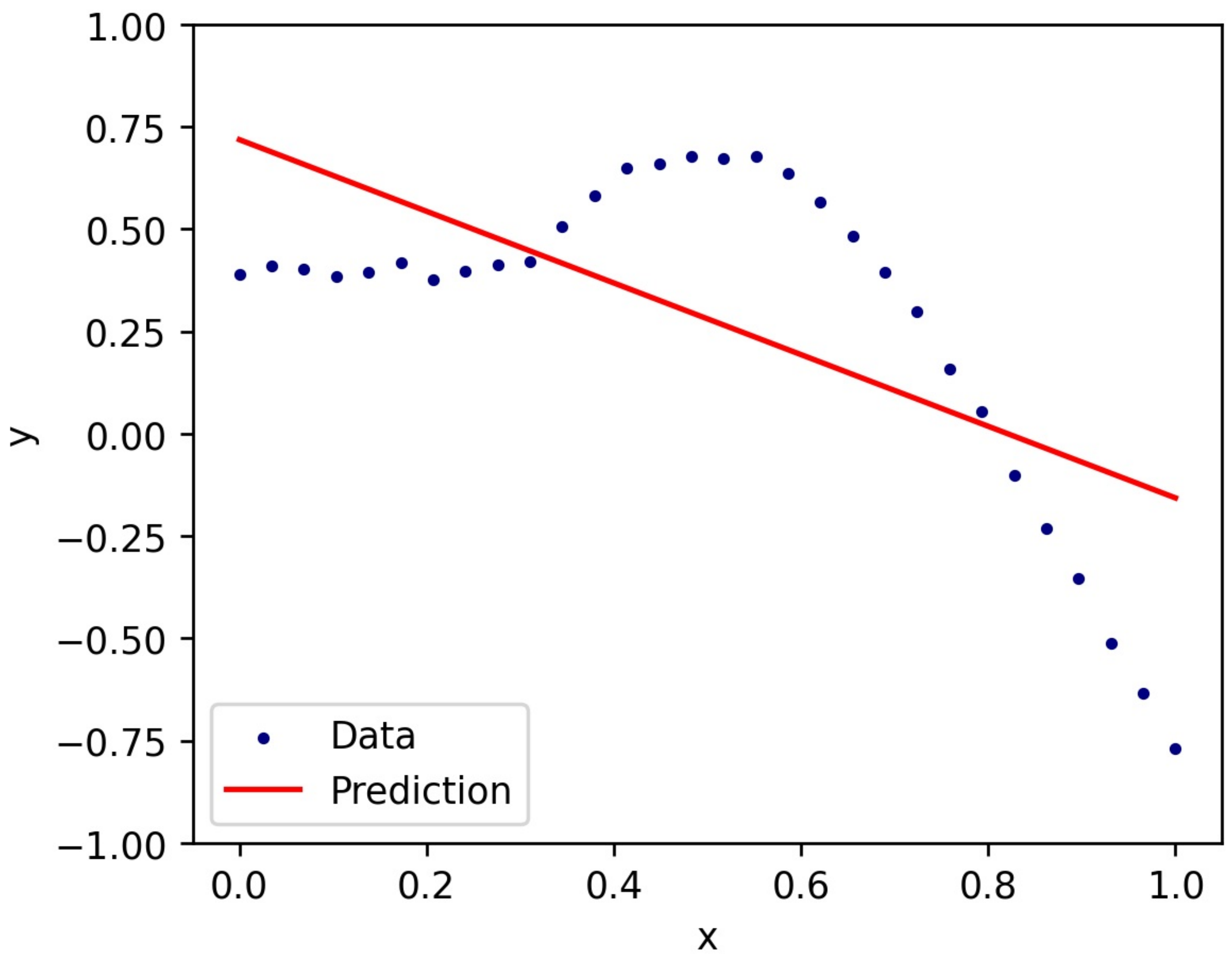
```
plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```



In [8]:

```
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")
```

```python
plt.ylabel("y")
plt.show()
```



Your Turn

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

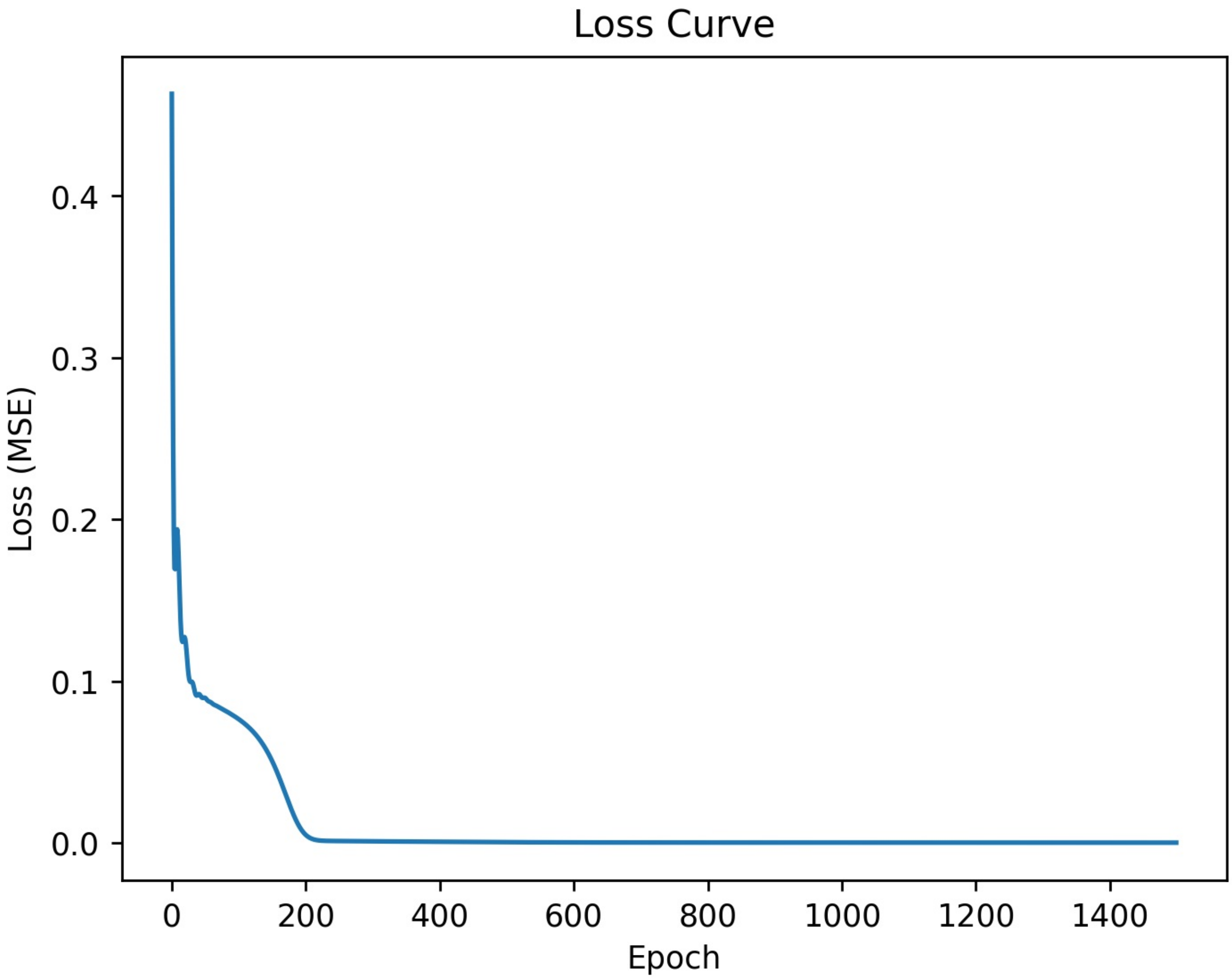
In [9]:
```python
# YOUR CODE GOES HERE
model_ = Net_2_layer(N_hidden = 20, activation = F.tanh)
loss_curve_ = []
```

In [10]:
```python
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: 0.46320417523384094
Epoch 60 of 1500...     Average loss: 0.08640876412391663
Epoch 120 of 1500...    Average loss: 0.06939766556024551
Epoch 180 of 1500...    Average loss: 0.019154317677021027
Epoch 240 of 1500...    Average loss: 0.0012369528412818909
Epoch 300 of 1500...    Average loss: 0.0010122036328539252
Epoch 360 of 1500...    Average loss: 0.0008473008638247848
Epoch 420 of 1500...    Average loss: 0.0006861641886644065
Epoch 480 of 1500...    Average loss: 0.0005353608867153525
Epoch 540 of 1500...    Average loss: 0.0004182912816759199
Epoch 600 of 1500...    Average loss: 0.00034434444387443364
Epoch 660 of 1500...    Average loss: 0.00030389547464437783
Epoch 720 of 1500...    Average loss: 0.00028214746271260083
Epoch 780 of 1500...    Average loss: 0.000268586038146168
Epoch 840 of 1500...    Average loss: 0.0002582859306130558
Epoch 900 of 1500...    Average loss: 0.00024952113744802773
Epoch 960 of 1500...    Average loss: 0.0002417687646811828
Epoch 1020 of 1500...   Average loss: 0.00023485107521992177
Epoch 1080 of 1500...   Average loss: 0.00022866240760777146
Epoch 1140 of 1500...   Average loss: 0.0002231192629551515
Epoch 1200 of 1500...   Average loss: 0.00021815820946358144
Epoch 1260 of 1500...   Average loss: 0.00021374403149820864
Epoch 1320 of 1500...   Average loss: 0.00020985915034543723
Epoch 1380 of 1500...   Average loss: 0.000206483862712048
Epoch 1440 of 1500...   Average loss: 0.00020357657922431827
```

In [11]:

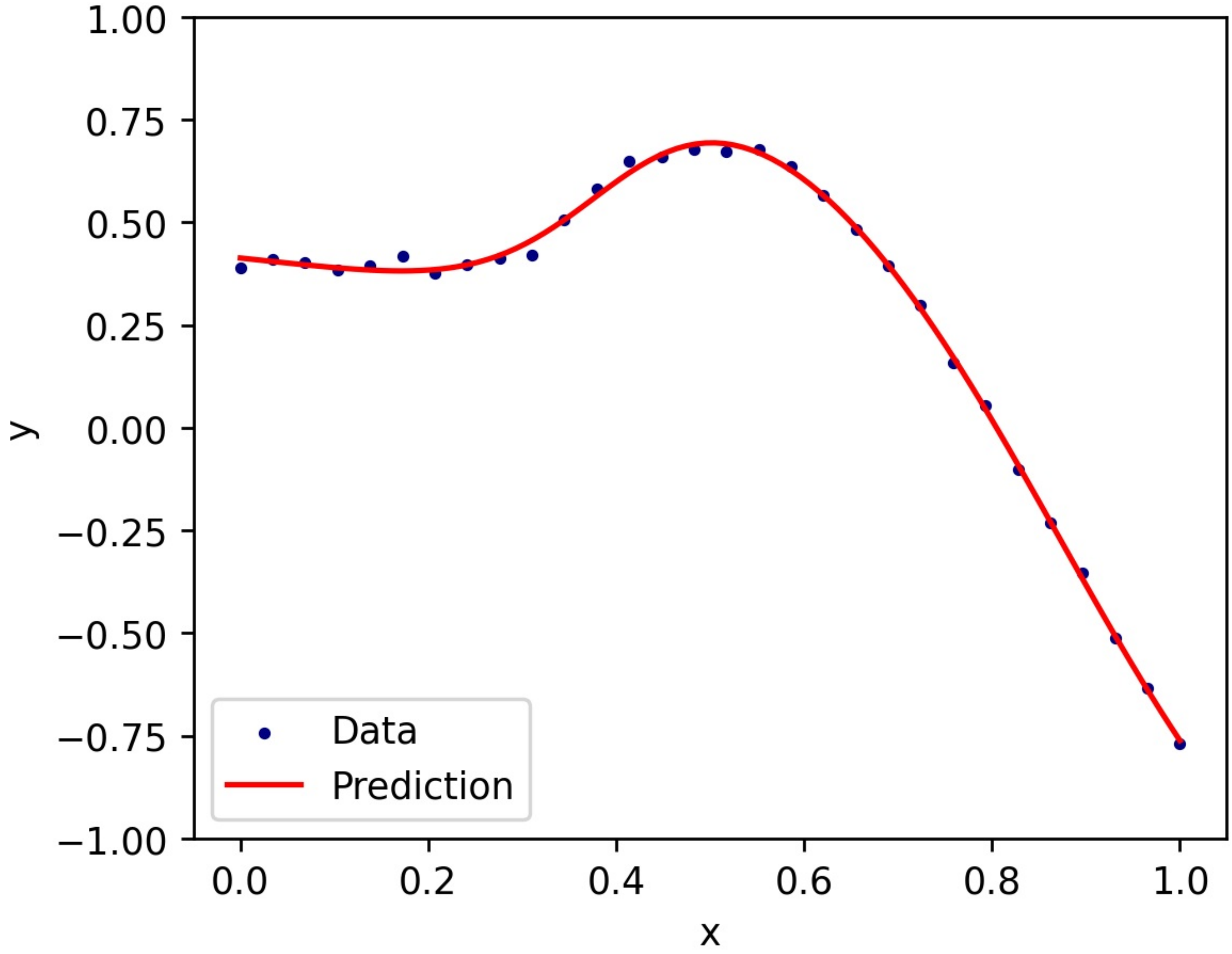
```
plt.figure(dpi=250)
plt.plot(loss_curve_)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
plt.show()
```



In [12]:

```
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()
```



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 [22]:

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from torch import optim

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 = 0.2!
    ax.plot_wireframe(X0.detach().numpy(),X1.detach().numpy(),Ypred.detach().numpy(),color = 'black', alpha = 0
    ax.set_xlabel('$x_0$')
    ax.set_ylabel('$x_1$')
    ax.set_zlabel('$y$')
    plt.show()
```

Generate and visualize the data

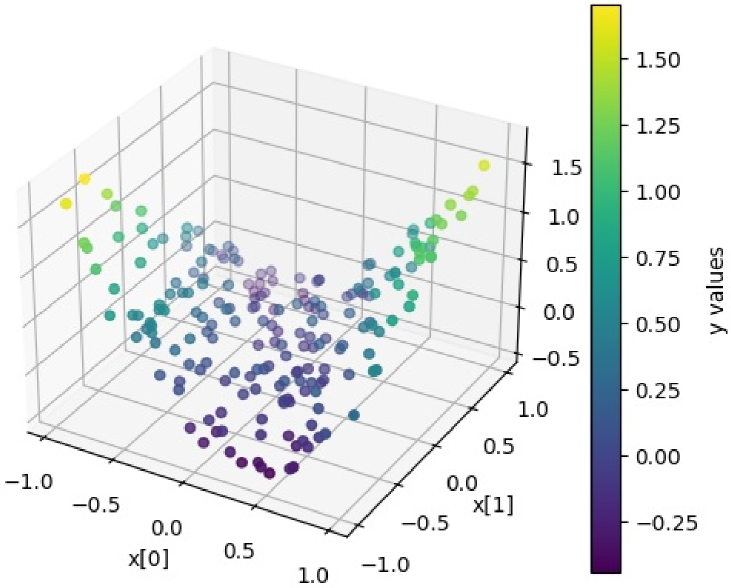
Use the `dataGen()` function to generate the x and y data, then visualize with a 3D scatter plot.

In [23]:

```
# YOUR CODE GOES HERE
[x, y] = dataGen()

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
sc = ax.scatter(x[:, 0], x[:, 1], y, c = y, cmap = 'viridis')
plt.colorbar(sc, ax = ax, label='y values')
ax.set_xlabel("x[0]")
ax.set_ylabel("x[1]")
```

```
ax.set_zlabel("y")
```

Out[23]: Text(0.5, 0, 'y')



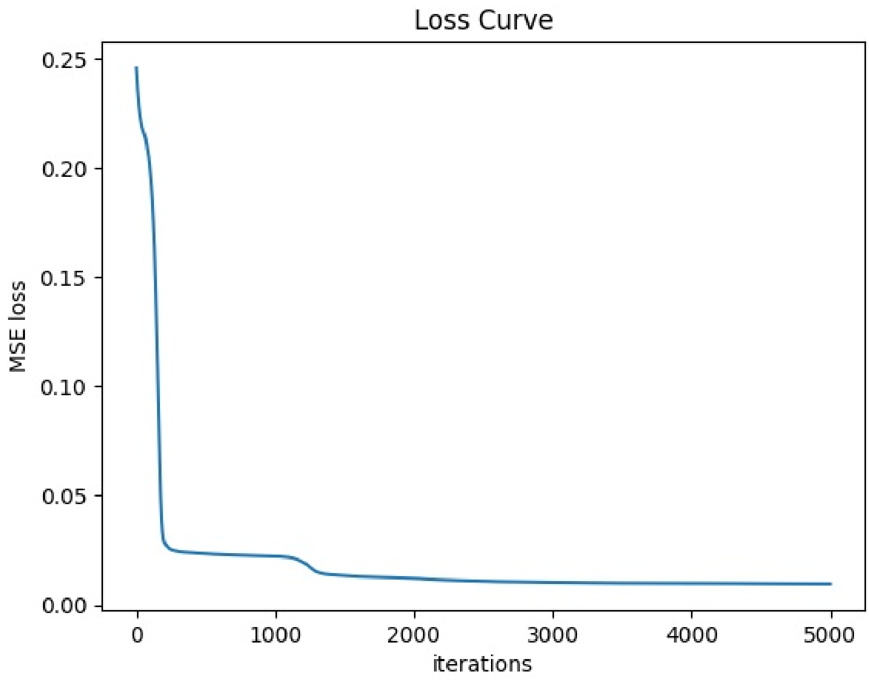
Create and train a neural network using PyTorch

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 [36]:
```
# YOUR CODE GOES HERE
lr = 0.001
epochs = 5000

class neural_network(nn.Module):
    def __init__(self, n_hidden = 6, n_in = 2, n_out = 1, activation = nn.functional.tanh):
        super().__init__()

        self.linear_1 = nn.Linear(n_in, n_hidden)
        self.linear_2 = nn.Linear(n_hidden, n_hidden)
        self.linear_3 = nn.Linear(n_hidden, n_hidden)
        self.linear_4 = nn.Linear(n_hidden, n_hidden)
        self.linear_5 = nn.Linear(n_hidden, n_out)

        self.activation = activation

    def forward(self, x):
        x = self.linear_1(x)
        x = self.activation(x)

        x = self.linear_2(x)
        x = self.activation(x)

        x = self.linear_3(x)
        x = self.activation(x)

        x = self.linear_4(x)
        x = self.activation(x)

        x = self.linear_5(x)

        return x

model = neural_network(n_hidden = 8, n_in = 2, n_out = 1, activation = nn.functional.relu)
x = torch.Tensor(x)
y = torch.Tensor(y).reshape(-1, 1)

loss_curve = []

print("Model details: \n", model)


loss_function = nn.functional.mse_loss

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

for i in range(epochs):
    output = model(x)
```

```python
    loss = loss_function(output, y)

    loss_curve.append(loss.item())

    if(i % 100 == 0):
        print("Iteration: ", i, ", Loss: ", loss.item())

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

plt.figure()
plt.plot(loss_curve)
plt.xlabel("iterations")
plt.ylabel("MSE loss")
plt.title("Loss Curve")
plt.show()
```

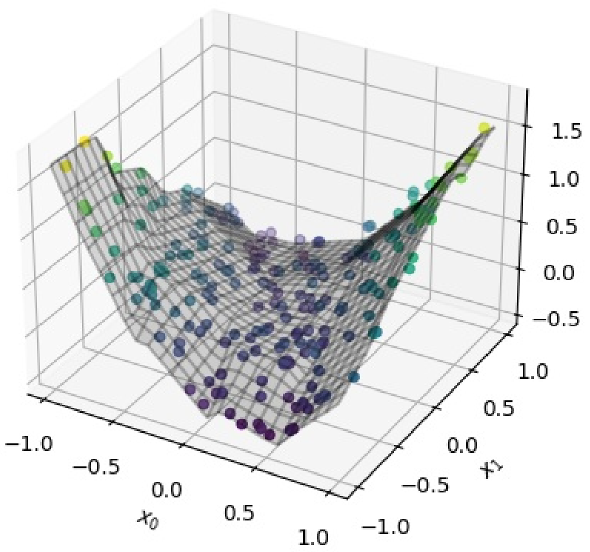
```
Model details:
 neural_network(
  (linear_1): Linear(in_features=2, out_features=8, bias=True)
  (linear_2): Linear(in_features=8, out_features=8, bias=True)
  (linear_3): Linear(in_features=8, out_features=8, bias=True)
  (linear_4): Linear(in_features=8, out_features=8, bias=True)
  (linear_5): Linear(in_features=8, out_features=1, bias=True)
)
Iteration:  0 , Loss:  0.245836541056633
Iteration:  100 , Loss:  0.19771771132946014
Iteration:  200 , Loss:  0.028797900304198265
Iteration:  300 , Loss:  0.02437671832740307
Iteration:  400 , Loss:  0.023777756839990616
Iteration:  500 , Loss:  0.023349428549408913
Iteration:  600 , Loss:  0.022998562082648277
Iteration:  700 , Loss:  0.02271977812051773
Iteration:  800 , Loss:  0.022491293027997017
Iteration:  900 , Loss:  0.022352388128638268
Iteration:  1000 , Loss:  0.022197151556611066
Iteration:  1100 , Loss:  0.0216795913875103
Iteration:  1200 , Loss:  0.019483935087919235
Iteration:  1300 , Loss:  0.015042351558804512
Iteration:  1400 , Loss:  0.013774579390883446
Iteration:  1500 , Loss:  0.013384107500314713
Iteration:  1600 , Loss:  0.01291478518396616
Iteration:  1700 , Loss:  0.012698727659881115
Iteration:  1800 , Loss:  0.012502877973020077
Iteration:  1900 , Loss:  0.012300939299166203
Iteration:  2000 , Loss:  0.012080254964530468
Iteration:  2100 , Loss:  0.011680684052407742
Iteration:  2200 , Loss:  0.011361232027411461
Iteration:  2300 , Loss:  0.011122622527182102
Iteration:  2400 , Loss:  0.01094749104231596
Iteration:  2500 , Loss:  0.010772015899419785
Iteration:  2600 , Loss:  0.010603544302284718
Iteration:  2700 , Loss:  0.010494479909539223
Iteration:  2800 , Loss:  0.010414516553282738
Iteration:  2900 , Loss:  0.01031608134508133
Iteration:  3000 , Loss:  0.010221059434115887
Iteration:  3100 , Loss:  0.010144032537937164
Iteration:  3200 , Loss:  0.01005033403635025
Iteration:  3300 , Loss:  0.010004525072872639
Iteration:  3400 , Loss:  0.009938404895365238
Iteration:  3500 , Loss:  0.00990714505314827
Iteration:  3600 , Loss:  0.009881635196506977
Iteration:  3700 , Loss:  0.009868196211755276
Iteration:  3800 , Loss:  0.00983534287661314
Iteration:  3900 , Loss:  0.009824545122683048
Iteration:  4000 , Loss:  0.009803889319300652
Iteration:  4100 , Loss:  0.009782475419342518
Iteration:  4200 , Loss:  0.009766326285898685
Iteration:  4300 , Loss:  0.009753881953656673
Iteration:  4400 , Loss:  0.009710527025163174
Iteration:  4500 , Loss:  0.009675389155745506
Iteration:  4600 , Loss:  0.009655340574681759
Iteration:  4700 , Loss:  0.009620008058845997
Iteration:  4800 , Loss:  0.009619485586881638
Iteration:  4900 , Loss:  0.009576338343322277
```



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 [37]:
```
# YOUR CODE GOES HERE
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

Final MSE value: 0.00958

Structure of the network

- Input layer: 2 neurons
- Hidden layer 1: 8 neurons
- Hidden layer 2: 8 neurons
- Hidden layer 3: 8 neurons
- Hidden layer 4: 8 neurons
- Output layer: 1 neuron
- Activation function: relu

The model has 5 layers, with the first 4 layers having 10 neurons each. Each layer has a relu activation function, and the last layer has 1 neurons, with no activation function.

- Optimizer: Adam
- Learning rate: 0.001
- Max epochs: 5000

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 [1]:

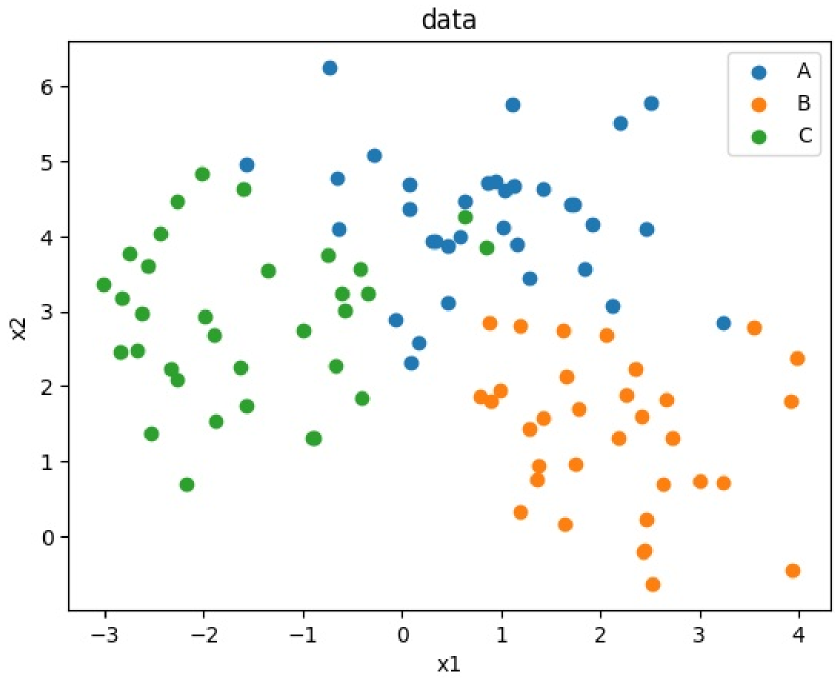
```
import torch
import torch.nn as nn
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from torch import optim


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(['
    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 [2]:

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

fig = plt.figure()
x1 = x[:,0]
x2 = x[:,1]
classes = ['A', 'B', 'C']

for i in range(3):
    plt.scatter(x1[y == i], x2[y == i], label = classes[i])

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('data')
plt.legend()
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.

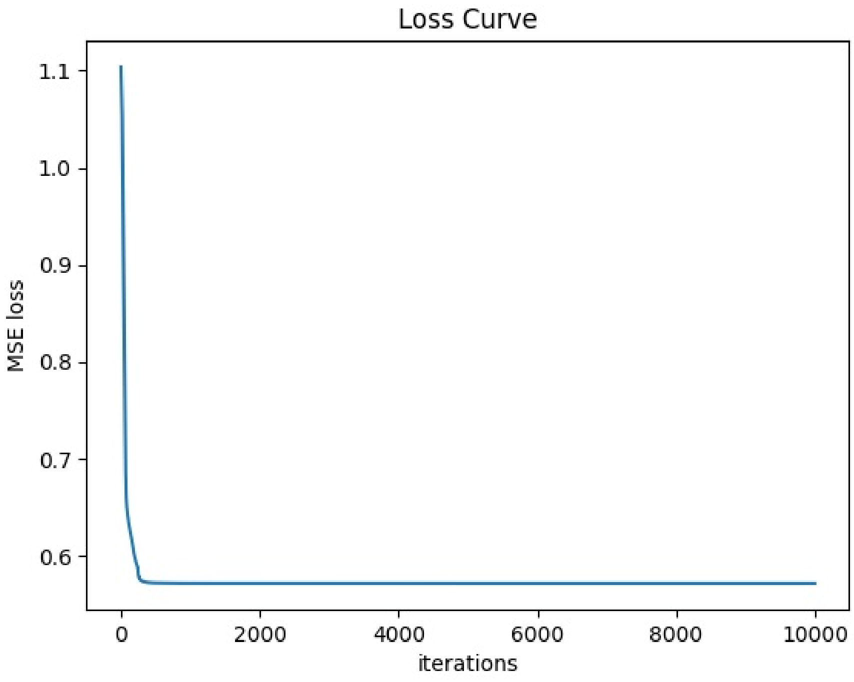
In [6]:

```
# YOUR CODE GOES HERE
lr = 0.005
epochs = 10000

class neural_network(nn.Module):
    def __init__(self, n_hidden = 6, n_in = 2, n_out = 3, activation = nn.functional.relu):
        super().__init__()

        self.linear_1 = nn.Linear(n_in, n_hidden)
        self.linear_2 = nn.Linear(n_hidden, n_hidden)
        self.linear_3 = nn.Linear(n_hidden, n_hidden)
        self.linear_4 = nn.Linear(n_hidden, n_hidden)
        self.linear_5 = nn.Linear(n_hidden, n_out)
        self.softmax = nn.Softmax(dim = 1)
        self.activation = activation

    def forward(self, x):
        x = self.activation(self.linear_1(x))
        x = self.activation(self.linear_2(x))
        x = self.activation(self.linear_3(x))
        x = self.activation(self.linear_4(x))
        x = self.linear_5(x)

        x = self.softmax(x)

        return x

model = neural_network(n_hidden = 10, n_in = 2, n_out = 3, activation = nn.functional.relu)
opt = optim.Adam(params = model.parameters(), lr = lr)
loss_function = nn.CrossEntropyLoss()

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

loss_curve = []

print("Model details: \n", model)

for i in range(epochs):
    output = model(x)
    loss = loss_function(output, y)

    loss_curve.append(loss.item())
    accuracy = (torch.argmax(output, dim = 1) == y).float().mean() * 100
```

```python
    if(i % 100 == 0):
        print(f"Iteration: {i}, Loss: {loss.item():.4f}, Accuracy: {accuracy.item():.2f}%")

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

plt.figure()
plt.plot(loss_curve)
plt.xlabel("iterations")
plt.ylabel("MSE loss")
plt.title("Loss Curve")
plt.show()
```

```
Model details:
 neural_network(
  (linear_1): Linear(in_features=2, out_features=10, bias=True)
  (linear_2): Linear(in_features=10, out_features=10, bias=True)
  (linear_3): Linear(in_features=10, out_features=10, bias=True)
  (linear_4): Linear(in_features=10, out_features=10, bias=True)
  (linear_5): Linear(in_features=10, out_features=3, bias=True)
  (softmax): Softmax(dim=1)
)
Iteration: 0, Loss: 1.1041, Accuracy: 33.00%
Iteration: 100, Loss: 0.6375, Accuracy: 92.00%
Iteration: 200, Loss: 0.5975, Accuracy: 97.00%
Iteration: 300, Loss: 0.5738, Accuracy: 98.00%
Iteration: 400, Loss: 0.5722, Accuracy: 98.00%
Iteration: 500, Loss: 0.5718, Accuracy: 98.00%
Iteration: 600, Loss: 0.5717, Accuracy: 98.00%
Iteration: 700, Loss: 0.5716, Accuracy: 98.00%
Iteration: 800, Loss: 0.5715, Accuracy: 98.00%
Iteration: 900, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1000, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1100, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1200, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1300, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1400, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1500, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1600, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1700, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1800, Loss: 0.5715, Accuracy: 98.00%
Iteration: 1900, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2000, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2100, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2200, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2300, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2400, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2500, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2600, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2700, Loss: 0.5715, Accuracy: 98.00%
Iteration: 2800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 2900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3000, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 3900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4000, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 4900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5000, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 5900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6000, Loss: 0.5714, Accuracy: 98.00%
```

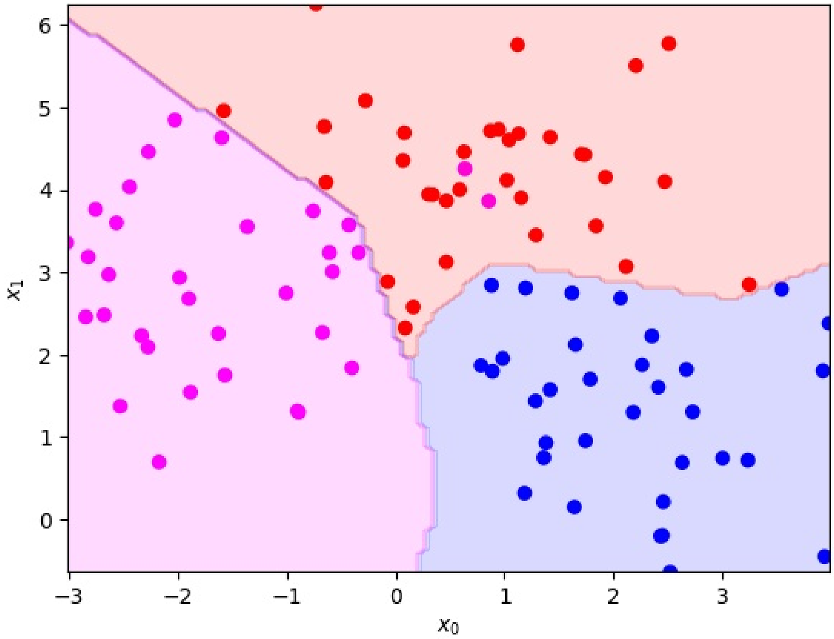
```
Iteration: 6100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 6900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7000, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 7900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8000, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 8900, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9000, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9100, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9200, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9300, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9400, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9500, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9600, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9700, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9800, Loss: 0.5714, Accuracy: 98.00%
Iteration: 9900, Loss: 0.5714, Accuracy: 98.00%
```



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 [7]:
```
# 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.

Final cross entropy loss value: 0.5714, Accuracy: 98%

Structure of the network

- Input layer: 2 neurons
- Hidden layer 1: 10 neurons
- Hidden layer 2: 10 neurons
- Hidden layer 3: 10 neurons
- Hidden layer 4: 10 neurons
- Output layer: 3 neuron
- Activation function: relu
- Activation function for last layer: softmax

The model has 5 layers, with the first 4 layers having 10 neurons each. Each layer has a relu activation function, and the last layer has 3 neurons, and a softmax activation function is used on the last layer. Here, softmax is chosen since the task involves classification.

- Optimizer: Adam
- Learning rate: 0.005
- Max epochs: 10000

Processing math: 100%