Problem 1

Loss = 2.0

Gradient with respect to w_3 = 12

Gradient with respect to w_2 = 28

Gradient with respect to w_1 = -84

M8-L1 Problem 1

In this problem you will solve for $\frac{\partial L}{\partial W_2}$ and $\frac{\partial L}{\partial W_1}$ for a neural network with two input features, a hidden layer with 3 nodes, and a single output. You will use the sigmoid activation function on the hidden layer. You are provided an input sample x_0, the current weights W_1 and W_2, and the ground truth value for the sample, $t = -2$

$L = \frac{1}{2}e^T e$

In [1]:
```
import numpy as np

x0 = np.array([[-2], [-6]])

W1 = np.array([[-2, 1],[3, 8],[-12, 7]])
W2 = np.array([[-11, 2, 5]])

t = np.array([[-2]])
```

Define activation function and its derivative

First define functions for the sigmoid activation functions, as well as its derivative:

In [2]:
```
# YOUR CODE GOES HERE
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def d_sigmoid(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Forward propagation

Using your activation function, compute the output of the network y using the sample x_0 and the provided weights W_1 and W_2

In [3]:
```
# YOUR CODE GOES HERE
hidden_input = np.dot(W1, x0)

hidden_output = sigmoid(hidden_input)

output = np.dot(W2, hidden_output)
```

Backpropagation

Using your calculated value of y, the provided value of t, your σ and σ' function, and the provided weights W_1 and W_2, compute the gradients $\frac{\partial L}{\partial W_2}$ and $\frac{\partial L}{\partial W_1}$.

In [5]:
```
# YOUR CODE GOES HERE
e = output - t
L = 1/2 * np.dot(e, e)

dL_dy = e
dL_dW2 = dL_dy * hidden_output.T

dL_dhidden_output = dL_dy * W2.T
dL_dhidden_input = dL_dhidden_output * d_sigmoid(hidden_input)
dL_dW1 = np.dot(dL_dhidden_input, x0.T)

print("dL_dW1: ", dL_dW1)
print("dL_dW2: ", dL_dW2)
```

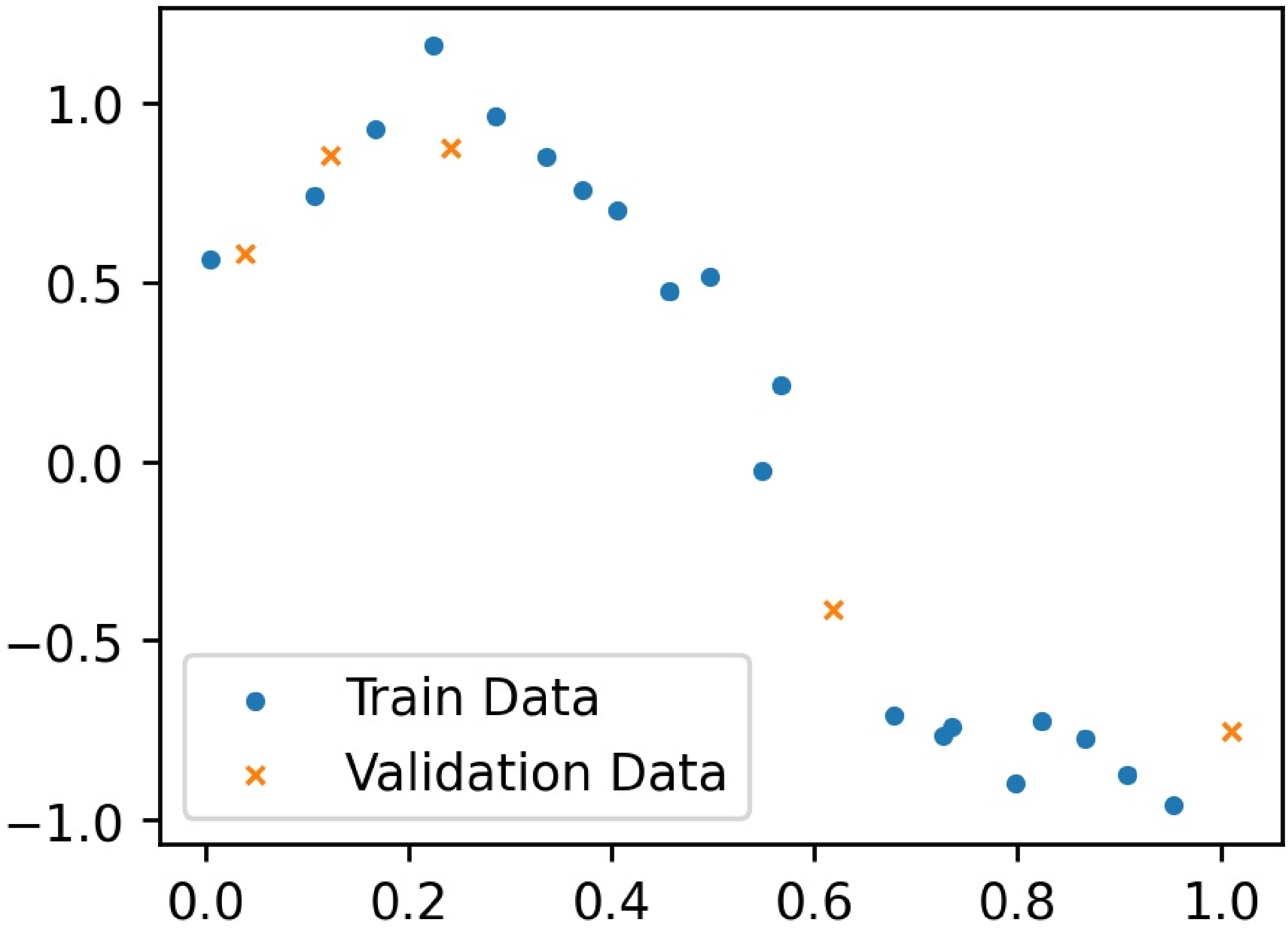
```
dL_dW1:  [[ 1.59095673e+00  4.77287018e+00]
 [-9.73264513e-24 -2.91979354e-23]
 [-1.04899214e-07 -3.14697641e-07]]
dL_dW2:  [[8.21031503e-02 2.43316128e-24 1.04899215e-08]]
```

Processing math: 100%

M8-L2 Problem 1

In this problem, you will create 3 regression networks with different complexities in PyTorch. By looking at the validation loss curves superimposed on the training loss curves, you should determine which model is optimal.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn, optim

def generate_data():
    np.random.seed(5)
    N = 25
    x = np.random.normal(np.linspace(0,1,N),0.01).reshape(-1,1)
    y = np.random.normal(np.sin(5*(x+0.082)),0.2)
    train_mask = np.zeros(N,dtype=np.bool_)
    train_mask[np.random.permutation(N)[:int(N*0.8)]] = True
    train_x, val_x = torch.Tensor(x[train_mask]), torch.Tensor(x[np.logical_not(train_mask)])
    train_y, val_y = torch.Tensor(y[train_mask]), torch.Tensor(y[np.logical_not(train_mask)])

    return train_x, val_x, train_y, val_y

def train(model, lr=0.0001, epochs=10000):
    train_x, val_x, train_y, val_y = generate_data()
    opt = optim.Adam(model.parameters(),lr=lr)
    lossfun = nn.MSELoss()
    train_hist = []
    val_hist = []

    for _ in range(epochs):
        model.train()
        loss_train = lossfun(train_y, model(train_x))
        train_hist.append(loss_train.item())

        model.eval()
        loss_val = lossfun(val_y, model(val_x))
        val_hist.append(loss_val.item())

        opt.zero_grad()
        loss_train.backward()
        opt.step()

    train_hist, val_hist = np.array(train_hist), np.array(val_hist)
    return train_hist, val_hist

def plot_loss(train_loss, val_loss):
    plt.plot(train_loss,label="Training")
    plt.plot(val_loss,label="Validation",linewidth=1)
    plt.legend()
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")

def plot_data(model = None):
    train_x, val_x, train_y, val_y = generate_data()
    plt.scatter(train_x, train_y,s=8,label="Train Data")
    plt.scatter(val_x, val_y,s=12,marker="x",label="Validation Data",linewidths=1)

    if model is not None:
        xvals = torch.linspace(0,1,1000).reshape(-1,1)
        plt.plot(xvals.detach().numpy(),model(xvals).detach().numpy(),label="Model",color="black")

    plt.legend(loc="lower left")

def get_loss(model):
    lossfun = nn.MSELoss()
    train_x, val_x, train_y, val_y = generate_data()
    loss_train = lossfun(train_y, model(train_x))
    loss_val = lossfun(val_y, model(val_x))
    return loss_train.item(), loss_val.item()


plt.figure(figsize=(4,3),dpi=250)
plot_data()
plt.show()
```



Coding neural networks for regression

Here, create 3 neural networks from scratch. You can use `nn.Sequential()` to simplify things. Each network should have 1 input and 1 output. After each hidden layer, apply ReLU activation. Name the models `model1`, `model2`, and `model3`, with architectures as follows:

- `model1`: 1 hidden layer with 4 neurons. That is, the network should have a linear transformation from size 1 to size 4. Then a ReLU activation should be applied. Finally, a linear transformation from size 4 to size 1 gives the network output. (Note: Your regression network should not have an activation after the last layer!)
- `model2`: Hidden sizes (16, 16). (Two hidden layers, each with 16 neurons)
- `model3`: Hidden sizes (128, 128, 128). (3 hidden layers, each with 128 neurons)

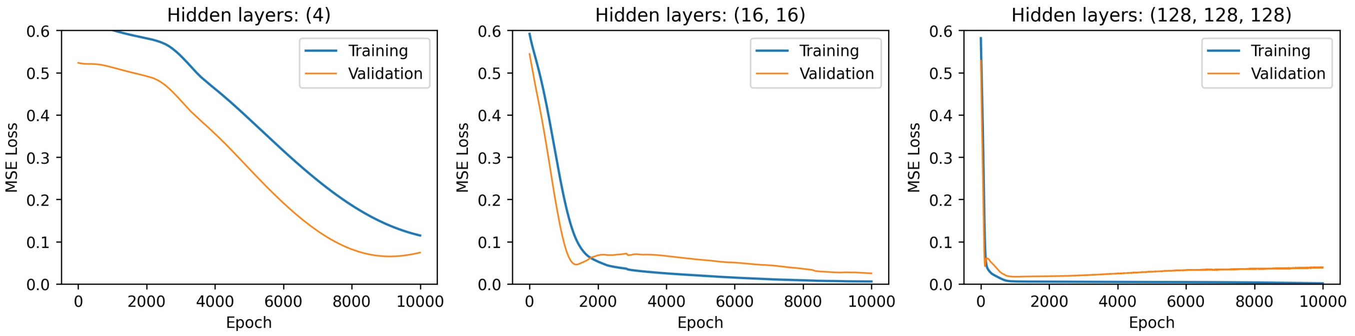
In [2]:
```
# YOUR CODE GOES HERE
model1 = nn.Sequential(nn.Linear(1, 4), nn.ReLU(), nn.Linear(4, 1))

# Define model2
model2 = nn.Sequential(nn.Linear(1, 16), nn.ReLU(), nn.Linear(16, 16), nn.ReLU(), nn.Linear(16, 1))

# Define model3
model3 = nn.Sequential(nn.Linear(1, 128), nn.ReLU(), nn.Linear(128, 128), nn.ReLU(), nn.Linear(128, 128), nn.Rel
)
```

Training and Loss curves

The following cell calls the provided function `train` to train each of your neural network models. The training and validation curves are then displayed.

In [3]:
```
hidden_layers=["(4)","(16, 16)","(128, 128, 128)"]

plt.figure(figsize=(15,3),dpi=250)
for i,model in enumerate([model1, model2, model3]):
    loss_train, loss_val = train(model)
    plt.subplot(1,3,i+1)
    plot_loss(loss_train, loss_val)
    plt.ylim(0,0.6)
    plt.title(f"Hidden layers: {hidden_layers[i]}")
```

```
plt.show()
```



Model performance

Let's print the values of MSE on the training and testing/validation data after training. Make note of which model is "best" (has lowest testing error).

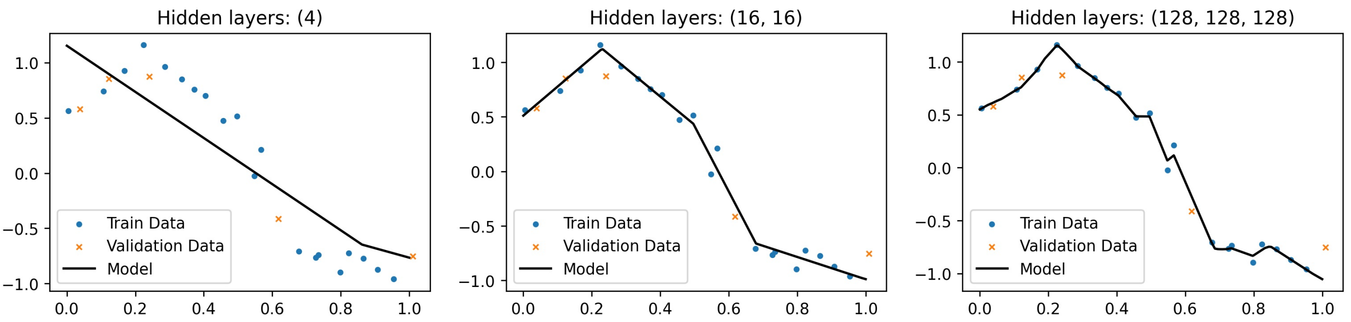
In [4]:
```
for i, model in enumerate([model1, model2, model3]):
    train_loss, val_loss = get_loss(model)
    print(f"Model {i+1}, hidden layers {hidden_layers[i]:>15}:   Train MSE: {train_loss:.4f}    Test MSE: {val_
```

```
Model 1, hidden layers             (4):   Train MSE: 0.1145    Test MSE: 0.0743
Model 2, hidden layers        (16, 16):   Train MSE: 0.0058    Test MSE: 0.0251
Model 3, hidden layers (128, 128, 128):   Train MSE: 0.0014    Test MSE: 0.0380
```

Visualization

Now we can look at how good each model's predictions are. Run the following cell to generate a visualization plot, then answer the questions.

In [5]:
```
plt.figure(figsize=(15,3),dpi=250)
for i,model in enumerate([model1, model2, model3]):
    plt.subplot(1,3,i+1)
    plot_data(model)
    plt.title(f"Hidden layers: {hidden_layers[i]}")
plt.show()
```



Questions

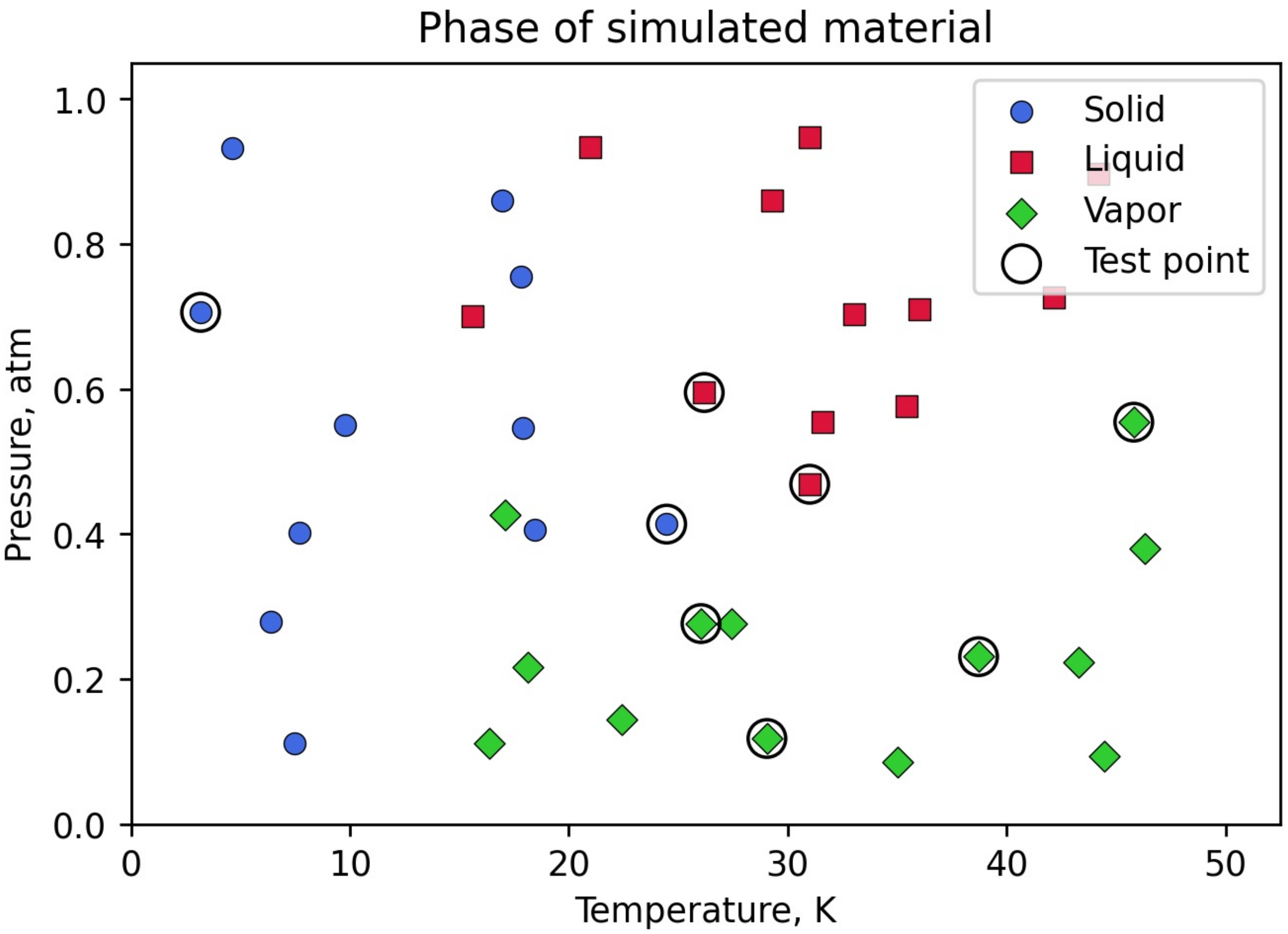
1. For the model that overfits the most, describe what happens to the loss curves while training.
2. For the model that underfits the most, describe what happens to the loss curves while training.
3. For the "best" model, what happens to the loss curves while training?

1. model3: The training loss decreases quickly and stays very low, indicating that the model is fitting well to the training data. However, the validation loss starts to increase after an initial decrease, showing that the model is fittiing to the training data too well but struggling to generalize to the validation set. This pattern is seen typically in overfitting.
2. model1: Both training and validation losses decrease, but they do so at a relatively less pace. The losses don't reach very low values, indicating that the model fails to capture the complexity of the underlying data distribution. This shows underfitting.
3. model2: Both the training and validation losses decrease and eventually stabilize at low values. The gap between training and validation loss remains small, indicating a good balance between fitting the training data and generalizing to unseen data. This suggests that model2 has the is designed well for the data complexity, making it the most appropriate model in this case.

M8-L2 Problem 2

Let's revisit the material phase prediction problem once again. You will use this problem to try multi-class classification in PyTorch. You will have to write code for a classification network and for training.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import torch
from torch import nn, optim

def plot_loss(train_loss, val_loss):
    plt.figure(figsize=(4,2),dpi=250)
    plt.plot(train_loss,label="Training")
    plt.plot(val_loss,label="Validation",linewidth=1)
    plt.legend()
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.show()

def split_data(X, Y):
    np.random.seed(100)
    N = len(Y)
    train_mask = np.zeros(N, dtype=np.bool_)
    train_mask[np.random.permutation(N)[:int(N*0.8)]] = True
    train_x, val_x = torch.Tensor(X[train_mask,:]), torch.Tensor(X[np.logical_not(train_mask),:])
    train_y, val_y = torch.Tensor(Y[train_mask]), torch.Tensor(Y[np.logical_not(train_mask)])
    return train_x, val_x, train_y, val_y

x1 = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.04883182996897,35.03654799338904,44.
x2 = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.11818202991034835,0.08595079005737
X = np.vstack([x1,x2]).T
y = np.array([0,2,2,2,2,2,0,2,2,2,2,2,0,0,2,0,1,2,0,0,1,1,1,2,0,1,0,1,1,1,0,0,1,1,1,1,])

X = torch.Tensor(X)
Y = torch.tensor(y,dtype=torch.long)

train_x, val_x, train_y, val_y = split_data(X,Y)


def plot_data(newfig=True):
    xlim = [0,52.5]
    ylim = [0,1.05]
    markers = [dict(marker="o", color="royalblue"), dict(marker="s", color="crimson"), dict(marker="D", color="
    labels = ["Solid", "Liquid", "Vapor"]

    if newfig:
        plt.figure(figsize=(6,4),dpi=250)

    x = X.detach().numpy()
    y = Y.detach().numpy().flatten()

    for i in range(1+max(y)):
        plt.scatter(x[y==i,0], x[y==i,1], s=40, **(markers[i]), edgecolor="black", linewidths=0.4,label=labels[

    plt.scatter(val_x[:,0], val_x[:,1],s=120,c="None",marker="o",edgecolors="black",label="Test point")

    plt.title("Phase of simulated material")
    plt.legend(loc="upper right")
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel("Temperature, K")
    plt.ylabel("Pressure, atm")
    plt.box(True)

def plot_model(model, res=200):
    xlim = [0,52.5]
    ylim = [0,1.05]
    xvals = np.linspace(*xlim,res)
    yvals = np.linspace(*ylim,res)
    x,y = np.meshgrid(xvals,yvals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)
    XY = torch.Tensor(XY)
    color = model.predict(XY).reshape(res,res).detach().numpy()
    cmap = ListedColormap(["lightblue","lightcoral","palegreen"])
    plt.pcolor(x, y, color, shading="nearest", zorder=-1, cmap=cmap,vmin=0,vmax=2)
    return

plot_data()
plt.show()
```



Model definition

In the cell below, complete the definition for `PhaseNet`, a classification neural network.

- The network should take in 2 inputs and return 3 outputs.
- The network size and hidden layer activations are up to you.
- Make sure to use the proper activation function (for multi-class classification) at the final layer.
- The `predict()` method has been provided, to return the integer class value. You must finish `__init__()` and `forward()`.

In [2]:
```
import torch.nn.functional as F

class PhaseNet(nn.Module):
    def __init__(self):
        super().__init__()
        # YOUR CODE GOES HERE
        self.fc1 = nn.Linear(2, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, 3)

    def predict(self,X):
        Y = self(X)
        return torch.argmax(Y,dim=1)

    def forward(self,X):
        # YOUR CODE GOES HERE
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = F.softmax(self.fc3(X), dim=1)

        return X
```

Training

Most of the training code has been provided below. Please add the following where indicated:

- Define a loss function (for multiclass classification)

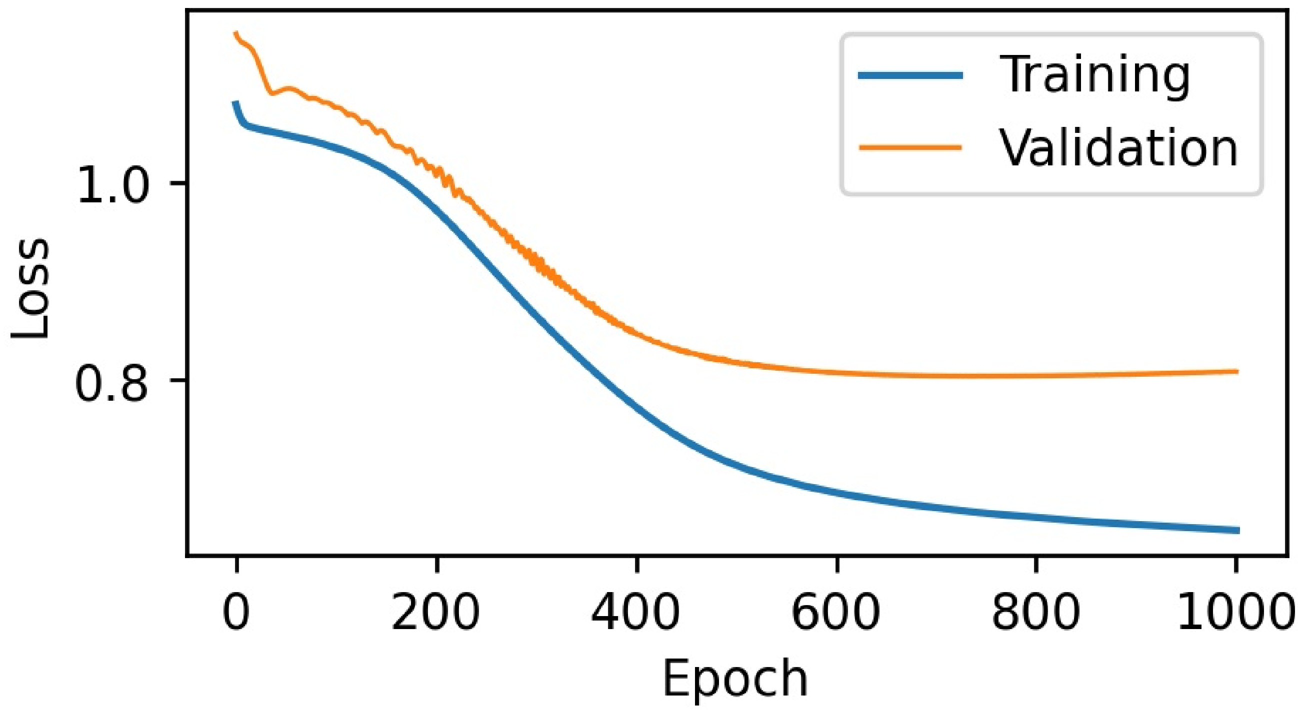
- Define an optimizer and call it `opt`. You may choose which optimizer.

Make sure the training curves you get are reasonable.

In [3]:

```
model = PhaseNet()

lr = 0.001
epochs = 1000

# Define loss function
# YOUR CODE GOES HERE
lossfun = nn.CrossEntropyLoss()

# Define an optimizer, `opt`
# YOUR CODE GOES HERE
opt = optim.Adam(model.parameters(), lr=lr)

train_hist = []
val_hist = []

for epoch in range(epochs+1):
    model.train()
    loss_train = lossfun(model(train_x), train_y)
    train_hist.append(loss_train.item())

    model.eval()
    loss_val = lossfun(model(val_x), val_y)
    val_hist.append(loss_val.item())

    opt.zero_grad()
    loss_train.backward()
    opt.step()
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch:>4} of {epochs}:   Train Loss = {loss_train.item():.4f}   Validation Loss = {loss_v


plot_loss(train_hist, val_hist)
```

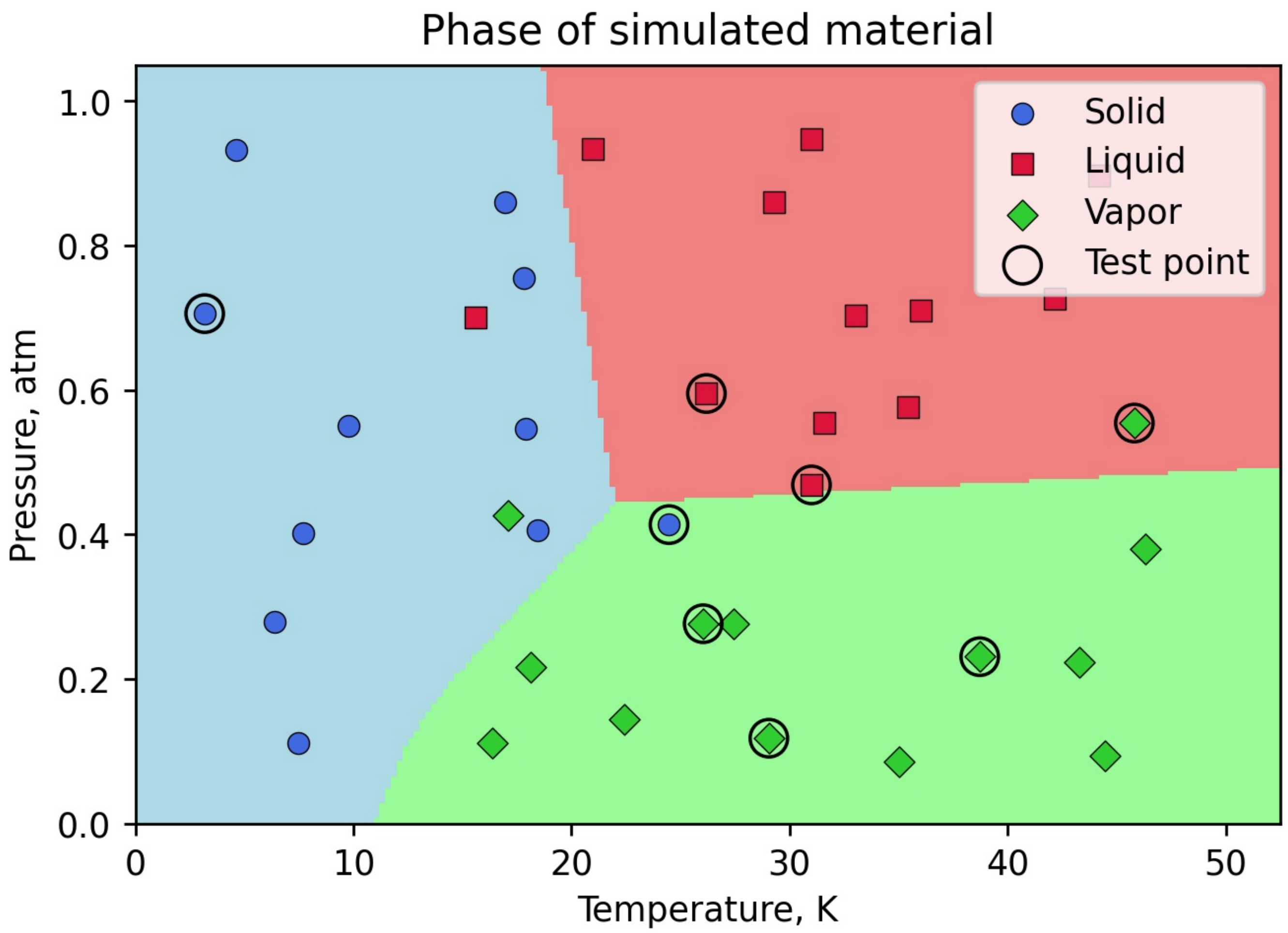
```
Epoch    0 of 1000:   Train Loss = 1.0793   Validation Loss = 1.1506
Epoch   40 of 1000:   Train Loss = 1.0505   Validation Loss = 1.0913
Epoch   80 of 1000:   Train Loss = 1.0412   Validation Loss = 1.0853
Epoch  120 of 1000:   Train Loss = 1.0279   Validation Loss = 1.0668
Epoch  160 of 1000:   Train Loss = 1.0062   Validation Loss = 1.0368
Epoch  200 of 1000:   Train Loss = 0.9725   Validation Loss = 1.0071
Epoch  240 of 1000:   Train Loss = 0.9305   Validation Loss = 0.9750
Epoch  280 of 1000:   Train Loss = 0.8866   Validation Loss = 0.9393
Epoch  320 of 1000:   Train Loss = 0.8453   Validation Loss = 0.8974
Epoch  360 of 1000:   Train Loss = 0.8072   Validation Loss = 0.8680
Epoch  400 of 1000:   Train Loss = 0.7729   Validation Loss = 0.8476
Epoch  440 of 1000:   Train Loss = 0.7442   Validation Loss = 0.8323
Epoch  480 of 1000:   Train Loss = 0.7224   Validation Loss = 0.8224
Epoch  520 of 1000:   Train Loss = 0.7066   Validation Loss = 0.8144
Epoch  560 of 1000:   Train Loss = 0.6949   Validation Loss = 0.8105
Epoch  600 of 1000:   Train Loss = 0.6860   Validation Loss = 0.8074
Epoch  640 of 1000:   Train Loss = 0.6790   Validation Loss = 0.8054
Epoch  680 of 1000:   Train Loss = 0.6732   Validation Loss = 0.8045
Epoch  720 of 1000:   Train Loss = 0.6684   Validation Loss = 0.8042
Epoch  760 of 1000:   Train Loss = 0.6643   Validation Loss = 0.8042
Epoch  800 of 1000:   Train Loss = 0.6608   Validation Loss = 0.8044
Epoch  840 of 1000:   Train Loss = 0.6577   Validation Loss = 0.8048
Epoch  880 of 1000:   Train Loss = 0.6549   Validation Loss = 0.8055
Epoch  920 of 1000:   Train Loss = 0.6524   Validation Loss = 0.8064
Epoch  960 of 1000:   Train Loss = 0.6501   Validation Loss = 0.8076
Epoch 1000 of 1000:   Train Loss = 0.6481   Validation Loss = 0.8087
```



Plot results

Plot your network predictions with the data by running the following cell. If your network has significant overfitting/underfitting, go back and retrain a new network with different layer sizes/activations.

In [4]:
```
plot_data(newfig=True)
plot_model(model)
plt.show()
```



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Problem 1

Consider a 2D robotic arm with 3 links. The position of its end-effector is governed by the arm lengths and joint angles as follows (as in the figure "data/robot-arm.png"):

$$x = L_1\cos(\theta_1) + L_2\cos(\theta_2 + \theta_1) + L_3\cos(\theta_3 + \theta_2 + \theta_1)y = L_1\sin(\theta_1) + L_2\sin(\theta_2 + \theta_1) + L_3\sin(\theta_3 + \theta_2 + \theta_1)$$

In robotics settings, inverse-kinematics problems are common for setups like this. For example, suppose all 3 arm lengths are $L_1 = L_2 = L_3 = 1$, and we want to position the end-effector at $(x, y) = (0.5, 0.5)$. What set of joint angles $(\theta_1, \theta_2, \theta_3)$ should we choose for the end-effector to reach this position?

In this problem you will train a neural network to find a function mapping from coordinates (x, y) to joint angles $(\theta_1, \theta_2, \theta_3)$ that position the end-effector at (x, y).

Summary of deliverables:

1. Neural network model
2. Generate training and validation data
3. Training function
4. 6 plots with training and validation loss
5. 6 prediction plots
6. Respond to the prompts

In [60]:

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

import torch
from torch import nn, optim

class ForwardArm(nn.Module):
    def __init__(self, L1=1, L2=1, L3=1):
        super().__init__()
        self.L1 = L1
        self.L2 = L2
        self.L3 = L3
    def forward(self, angles):
        theta1 = angles[:,0]
        theta2 = angles[:,1]
        theta3 = angles[:,2]
        x = self.L1*torch.cos(theta1) + self.L2*torch.cos(theta1+theta2) + self.L3*torch.cos(theta1+theta2+theta
        y = self.L1*torch.sin(theta1) + self.L2*torch.sin(theta1+theta2) + self.L3*torch.sin(theta1+theta2+theta
        return torch.vstack([x,y]).T

def plot_predictions(model, title=""):
    fwd = ForwardArm()

    vals = np.arange(0.1, 2.0, 0.2)
    x, y = np.meshgrid(vals,vals)
    coords = torch.tensor(np.vstack([x.flatten(),y.flatten()]).T,dtype=torch.float)
    angles = model(coords)
    preds = fwd(angles).detach().numpy()

    plt.figure(figsize=[4,4],dpi=140)

    plt.scatter(x.flatten(), y.flatten(), s=60, c="None",marker="o",edgecolors="k", label="Targets")
    plt.scatter(preds[:,0], preds[:,1], s=25, c="red", marker="o", label="Predictions")
    plt.text(0.1, 2.15, f"MSE = {nn.MSELoss()(fwd(model(coords)),coords):.1e}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim(-.1,2.1)
    plt.ylim(-.1,2.4)
    plt.legend()
    plt.title(title)
    plt.show()

def plot_arm(theta1, theta2, theta3, L1=1,L2=1,L3=1, show=True):
    x1 = L1*np.cos(theta1)
    y1 = L1*np.sin(theta1)
    x2 = x1 + L2*np.cos(theta1+theta2)
    y2 = y1 + L2*np.sin(theta1+theta2)
    x3 = x2 + L3*np.cos(theta1+theta2+theta3)
    y3 = y2 + L3*np.sin(theta1+theta2+theta3)
```

```python
    xs = np.array([0,x1,x2,x3])
    ys = np.array([0,y1,y2,y3])

    plt.figure(figsize=(5,5),dpi=140)
    plt.plot(xs, ys, linewidth=3, markersize=5,color="gray", markerfacecolor="lightgray",marker="o",markeredgec
    plt.scatter(x3,y3,s=50,color="blue",marker="P",zorder=100)
    plt.scatter(0,0,s=50,color="black",marker="s",zorder=-100)

    plt.xlim(-1.5,3.5)
    plt.ylim(-1.5,3.5)

    if show:
        plt.show()
```

End-effector position

You can use the interactive figure below to visualize the robot arm.

In [61]:

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

def plot_unit_arm(theta1, theta2, theta3):
    plot_arm(theta1, theta2, theta3)

slider1 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta1',disabled=Fa
slider2 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta2',disabled=Fa
slider3 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta3',disabled=Fa

interactive_plot = interactive(plot_unit_arm, theta1 = slider1, theta2 = slider2, theta3 = slider3)
output = interactive_plot.children[-1]
output.layout.height = '600px'

interactive_plot
```

Out[61]:

```
interactive(children=(FloatSlider(value=0.0, description='theta1', layout=Layout(width='550px'), max=2.3561944…
```

Neural Network for Inverse Kinematics

In this class we have mainly had regression problems with only one output. However, you can create neural networks with any number of outputs just by changing the size of the last layer. For this problem, we already know the function to go from joint angles (3) to end-effector coordinates (2). This is provided in neural network format as `ForwardArm()`.

If you provide an instance of `ForwardArm()` with an $N \times 3$ tensor of joint angles, and it will return an $N \times 2$ tensor of coordinates.

Here, you should create a neural network with 2 inputs and 3 outputs that, once trained, can output the joint angles (in radians) necessary to reach the input x-y coordinates.

In the cell below, complete the definition for `InverseArm()`:

- The initialization argument `hidden_layer_sizes` dictates the number of neurons per hidden layer in the network. For example, `hidden_layer_sizes=[12,24]` should create a network with 2 inputs, 12 neurons in the first hidden layer, 24 neurons in the second hidden layer, and 3 outputs.
- Use a ReLU activation at the end of each hidden layer.
- The initialization argument `max_angle` refers to the maximum bend angle of the joint. If `max_angle=None`, there should be no activation at the last layer. However, if `max_angle=1` (for example), then the output joint angles should be restricted to the interval [-1, 1] (radians). You can clamp values with the tanh function (and then scale them) to achieve this.

In [62]:

```python
class InverseArm(nn.Module):
    def __init__(self, hidden_layer_sizes=[24,24], max_angle = None):
        super().__init__()
        # YOUR CODE GOES HERE
        self.max_angle = max_angle

        self.layers = nn.ModuleList()

        for i in range(len(hidden_layer_sizes)):
            self.layers.append(nn.Linear(hidden_layer_sizes[i - 1] if i > 0 else 2, hidden_layer_sizes[i]))
        self.layers.append(nn.Linear(hidden_layer_sizes[-1], 3))

    def forward(self, xy):
        # YOUR CODE GOES HERE
        angles = xy

        for layer in self.layers[:-1]:
            angles = torch.relu(layer(angles))
        angles = self.layers[-1](angles)
```

```
        if self.max_angle is not None:
            angles = torch.tanh(angles) * self.max_angle

        return angles
```

Generate Data

In the cell below, generate a dataset of x-y coordinates. You should use a 100×100 meshgrid, for x and y each on the interval $[0, 2]$.

Randomly split your data so that 80% of points are in `X_train`, while the remaining 20% are in `X_val`. (Each of these should have 2 columns -- x and y)

In [63]:
```
# YOUR CODE GOES HERE
from sklearn.model_selection import train_test_split

x = np.linspace(0, 2, 100)
y = np.linspace(0, 2, 100)
x_grid, y_grid = np.meshgrid(x, y)
coordinates = torch.tensor(np.vstack([x_grid.ravel(), y_grid.ravel()]).T, dtype = torch.float)

X_train, X_val = train_test_split(coordinates, test_size=0.2, random_state=42)
```

Training function

Write a function `train()` below with the following specifications:

Inputs:

- `model` : `InverseArm` model to train
- `X_train` : $N \times 2$ vector of training x-y coordinates
- `X_val` : $N \times 2$ vector of validation x-y coordinates
- `lr` : Learning rate for Adam optimizer
- `epochs` : Total epoch count
- `gamma` : ExponentialLR decay rate
- `create_plot` : (`True` / `False`) Whether to display a plot with training and validation loss curves

Loss function:
The loss function you use should be based on whether the end-effector moves to the correct location. It should be the MSE between the target coordinate tensor and the coordinates that the predicted joint angles produce. In other words, if your inverse kinematics model is `model`, and `fwd` is an instance of `ForwardArm()`, then you want the MSE between input coordinates `X` and `fwd(model(X))`.

In [68]:
```
from torch.optim.lr_scheduler import ExponentialLR

def train(model, X_train, X_val, lr = 0.01, epochs = 1000, gamma = 1, create_plot = True):
    # YOUR CODE GOES HERE
    X_train = torch.tensor(X_train, dtype=torch.float32)
    X_val = torch.tensor(X_val, dtype=torch.float32)

    fwd = ForwardArm()
    lossfun = nn.MSELoss()
    opt = optim.Adam(model.parameters(), lr = lr)
    scheduler = optim.lr_scheduler.StepLR(opt, step_size = 100, gamma = gamma)

    train_hist = []
    val_hist = []

    for epoch in range(epochs):
        model.train()

        opt.zero_grad()

        angles_pred_train = model(X_train)
        coords_pred_train = fwd(angles_pred_train)

        loss_train = lossfun(coords_pred_train, X_train)
        loss_train.backward()

        opt.step()

        train_hist.append(loss_train.item())

        model.eval()
        with torch.no_grad():
            predicted_angles_val = model(X_val)
            predicted_coords_val = fwd(predicted_angles_val)
```

```
            loss_val = lossfun(predicted_coords_val, X_val)
            val_hist.append(loss_val.item())

        scheduler.step()

        if epoch % (epochs // 25) == 0:
            print(f"Epoch {epoch:>4}/{epochs}: Train Loss = {loss_train.item():.4f}, Validation Loss = {los

    if create_plot:
        plt.figure(figsize=(10, 5))
        plt.plot(train_hist, label='Training Loss')
        plt.plot(val_hist, label='Validation Loss')
        plt.xlabel('Epoch')
        plt.ylabel('MSE Loss')
        plt.title('Training and Validation Loss')
        plt.legend()
        plt.show()

    return model
```

Training a model

Create 3 models of different complexities (with `max_angle=None`):

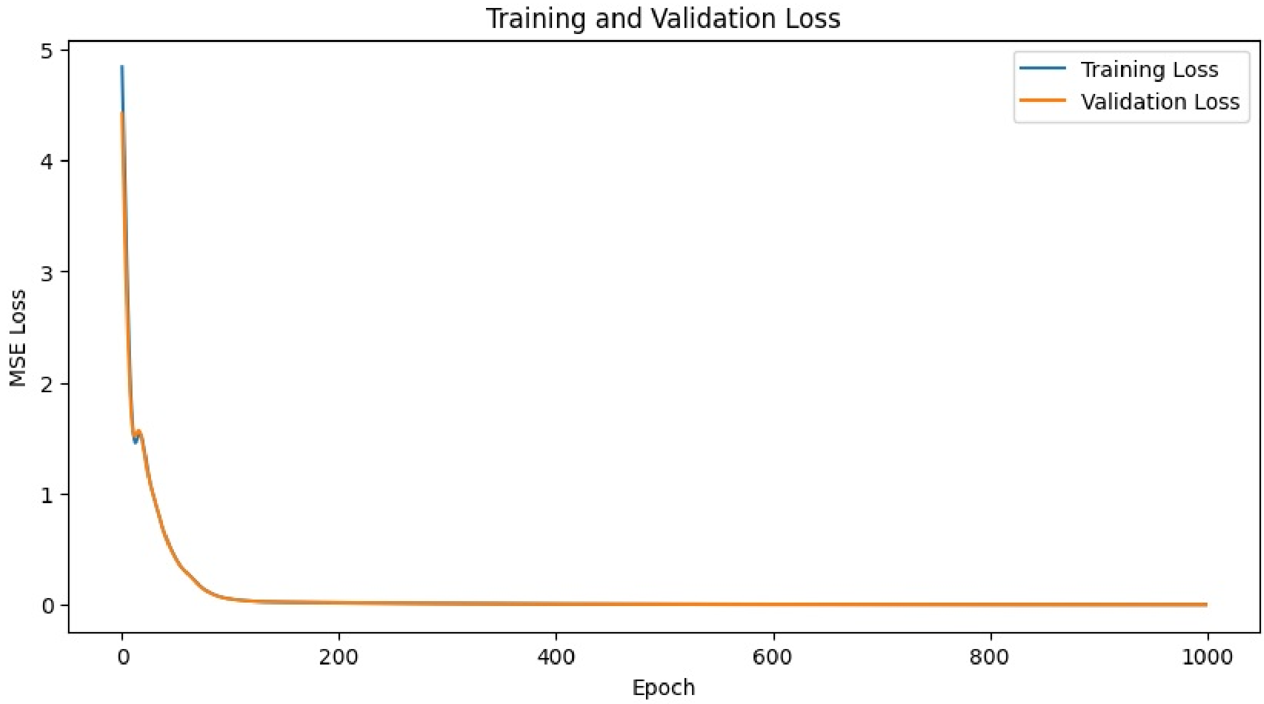
- `hidden_layer_sizes=[12]`
- `hidden_layer_sizes=[24,24]`
- `hidden_layer_sizes=[48,48,48]`

Train each model for 1000 epochs, learning rate 0.01, and gamma 0.995. Show the plot for each.

In [69]:

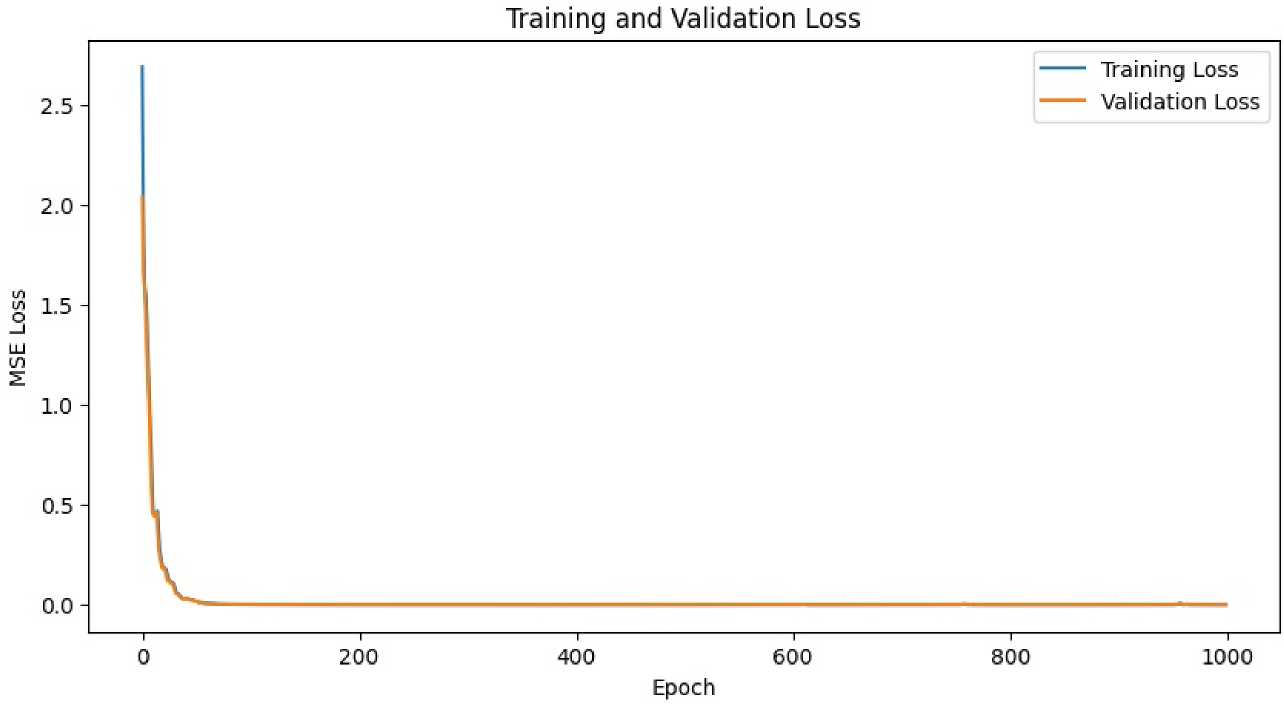
```
# YOUR CODE GOES HERE
model1 = InverseArm(hidden_layer_sizes = [12], max_angle = None)
model2 = InverseArm(hidden_layer_sizes = [24, 24], max_angle = None)
model3 = InverseArm(hidden_layer_sizes = [48, 48, 48], max_angle = None)

model1 = train(model1, X_train, X_val, epochs = 1000, lr = 0.01, gamma = 0.995, create_plot = True)
model2 = train(model2, X_train, X_val, epochs = 1000, lr = 0.01, gamma = 0.995, create_plot = True)
model3 = train(model3, X_train, X_val, epochs = 1000, lr = 0.01, gamma = 0.995, create_plot = True)
```

```
C:\Users\barat\AppData\Local\Temp\ipykernel_15088\664775586.py:5: UserWarning: To copy construct from a tensor,
it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
  X_train = torch.tensor(X_train, dtype=torch.float32)
C:\Users\barat\AppData\Local\Temp\ipykernel_15088\664775586.py:6: UserWarning: To copy construct from a tensor,
it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
  X_val = torch.tensor(X_val, dtype=torch.float32)
Epoch    0/1000: Train Loss = 4.8425, Validation Loss = 4.4216
Epoch   40/1000: Train Loss = 0.6166, Validation Loss = 0.6107
Epoch   80/1000: Train Loss = 0.1139, Validation Loss = 0.1108
Epoch  120/1000: Train Loss = 0.0319, Validation Loss = 0.0315
Epoch  160/1000: Train Loss = 0.0223, Validation Loss = 0.0227
Epoch  200/1000: Train Loss = 0.0185, Validation Loss = 0.0187
Epoch  240/1000: Train Loss = 0.0155, Validation Loss = 0.0156
Epoch  280/1000: Train Loss = 0.0129, Validation Loss = 0.0131
Epoch  320/1000: Train Loss = 0.0108, Validation Loss = 0.0109
Epoch  360/1000: Train Loss = 0.0091, Validation Loss = 0.0092
Epoch  400/1000: Train Loss = 0.0077, Validation Loss = 0.0077
Epoch  440/1000: Train Loss = 0.0066, Validation Loss = 0.0066
Epoch  480/1000: Train Loss = 0.0056, Validation Loss = 0.0056
Epoch  520/1000: Train Loss = 0.0048, Validation Loss = 0.0048
Epoch  560/1000: Train Loss = 0.0042, Validation Loss = 0.0042
Epoch  600/1000: Train Loss = 0.0036, Validation Loss = 0.0037
Epoch  640/1000: Train Loss = 0.0032, Validation Loss = 0.0032
Epoch  680/1000: Train Loss = 0.0028, Validation Loss = 0.0028
Epoch  720/1000: Train Loss = 0.0025, Validation Loss = 0.0025
Epoch  760/1000: Train Loss = 0.0022, Validation Loss = 0.0023
Epoch  800/1000: Train Loss = 0.0020, Validation Loss = 0.0021
Epoch  840/1000: Train Loss = 0.0018, Validation Loss = 0.0019
Epoch  880/1000: Train Loss = 0.0017, Validation Loss = 0.0017
Epoch  920/1000: Train Loss = 0.0015, Validation Loss = 0.0016
Epoch  960/1000: Train Loss = 0.0014, Validation Loss = 0.0015
```




```
Epoch    0/1000: Train Loss = 2.6520, Validation Loss = 1.7917
Epoch   40/1000: Train Loss = 0.0806, Validation Loss = 0.0777
Epoch   80/1000: Train Loss = 0.0179, Validation Loss = 0.0185
Epoch  120/1000: Train Loss = 0.0078, Validation Loss = 0.0081
Epoch  160/1000: Train Loss = 0.0030, Validation Loss = 0.0032
Epoch  200/1000: Train Loss = 0.0017, Validation Loss = 0.0017
Epoch  240/1000: Train Loss = 0.0012, Validation Loss = 0.0013
Epoch  280/1000: Train Loss = 0.0009, Validation Loss = 0.0010
Epoch  320/1000: Train Loss = 0.0008, Validation Loss = 0.0008
Epoch  360/1000: Train Loss = 0.0007, Validation Loss = 0.0007
Epoch  400/1000: Train Loss = 0.0007, Validation Loss = 0.0007
Epoch  440/1000: Train Loss = 0.0006, Validation Loss = 0.0006
Epoch  480/1000: Train Loss = 0.0006, Validation Loss = 0.0006
Epoch  520/1000: Train Loss = 0.0005, Validation Loss = 0.0005
Epoch  560/1000: Train Loss = 0.0005, Validation Loss = 0.0005
Epoch  600/1000: Train Loss = 0.0004, Validation Loss = 0.0004
Epoch  640/1000: Train Loss = 0.0004, Validation Loss = 0.0004
Epoch  680/1000: Train Loss = 0.0004, Validation Loss = 0.0004
Epoch  720/1000: Train Loss = 0.0003, Validation Loss = 0.0003
Epoch  760/1000: Train Loss = 0.0003, Validation Loss = 0.0004
Epoch  800/1000: Train Loss = 0.0003, Validation Loss = 0.0003
Epoch  840/1000: Train Loss = 0.0003, Validation Loss = 0.0003
Epoch  880/1000: Train Loss = 0.0002, Validation Loss = 0.0002
Epoch  920/1000: Train Loss = 0.0002, Validation Loss = 0.0002
Epoch  960/1000: Train Loss = 0.0005, Validation Loss = 0.0003
```

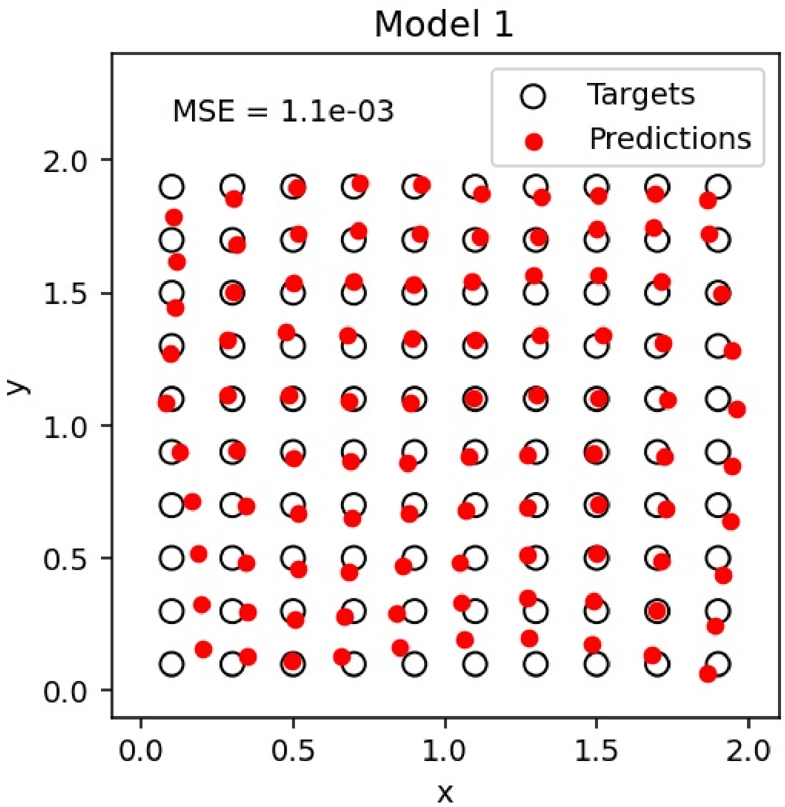

```
Epoch    0/1000: Train Loss = 2.6872, Validation Loss = 2.0361
Epoch   40/1000: Train Loss = 0.0297, Validation Loss = 0.0326
Epoch   80/1000: Train Loss = 0.0021, Validation Loss = 0.0020
Epoch  120/1000: Train Loss = 0.0008, Validation Loss = 0.0008
Epoch  160/1000: Train Loss = 0.0005, Validation Loss = 0.0005
Epoch  200/1000: Train Loss = 0.0003, Validation Loss = 0.0003
Epoch  240/1000: Train Loss = 0.0002, Validation Loss = 0.0002
Epoch  280/1000: Train Loss = 0.0002, Validation Loss = 0.0002
Epoch  320/1000: Train Loss = 0.0002, Validation Loss = 0.0002
Epoch  360/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  400/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  440/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  480/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  520/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  560/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  600/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  640/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  680/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  720/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  760/1000: Train Loss = 0.0011, Validation Loss = 0.0002
Epoch  800/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  840/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  880/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  920/1000: Train Loss = 0.0001, Validation Loss = 0.0001
Epoch  960/1000: Train Loss = 0.0004, Validation Loss = 0.0004
```

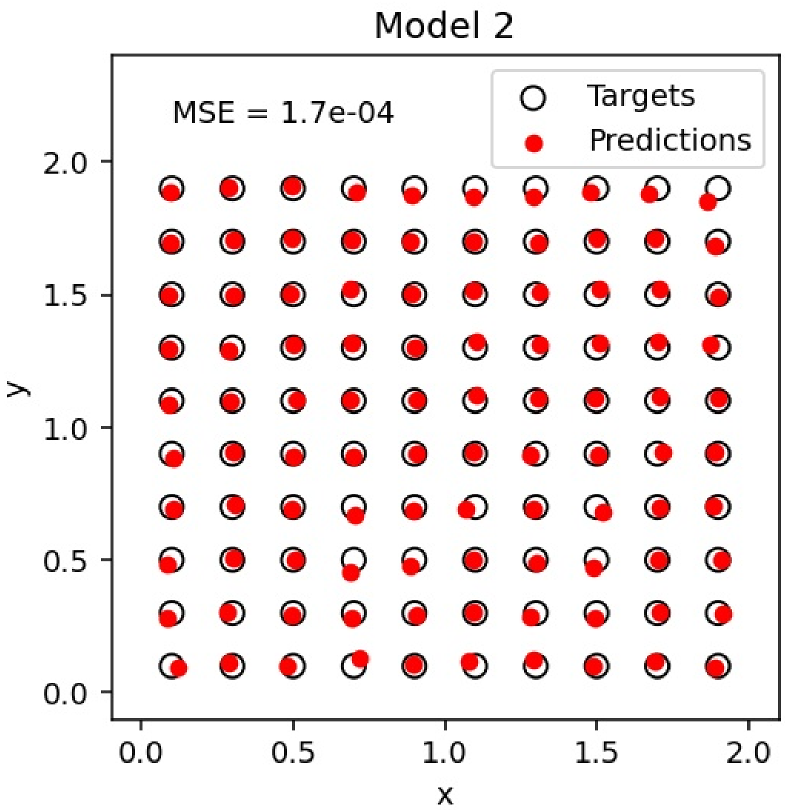


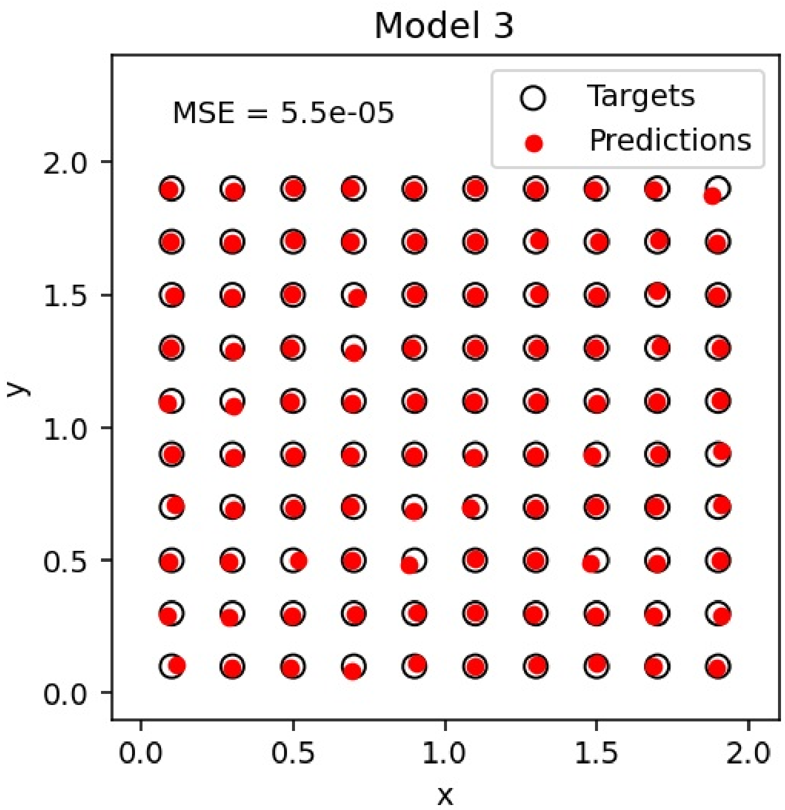
Visualizations

For each of your models, use the function `plot_predictions` to visualize model predictions on the domain. You should observe improvements with increasing network size.

In [73]:
```
# YOUR CODE GOES HERE

plot_predictions(model1, title = "Model 1")
plot_predictions(model2, title = "Model 2")
plot_predictions(model3, title = "Model 3")
```







Interactive Visualization

You can use the interactive plot below to look at the performance of your model. (The model used must be named `model`.)

In [72]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, FloatSlider, Dropdown

def plot_inverse(x, y):
    xy = torch.Tensor([[x,y]])
    theta1, theta2, theta3 = model1(xy).detach().numpy().flatten().tolist()
    plot_arm(theta1, theta2, theta3, show=False)
    plt.scatter(x, y, s=100, c="red",zorder=1000,marker="x")
    plt.plot([0,2,2,0,0],[0,0,2,2,0],c="lightgray",linewidth=1,zorder=-1000)
    plt.show()
```

```
slider1 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100, description='x', disabled=False, continuous_update=
slider2 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100, description='y', disabled=False, continuous_update=

interactive_plot = interactive(plot_inverse, x = slider1, y = slider2)
output = interactive_plot.children[-1]
output.layout.height = '600px'

interactive_plot
```

Out[72]:
```
interactive(children=(FloatSlider(value=1.0, description='x', layout=Layout(width='550px'), max=2.5, min=-0.5,…
```

Training more neural networks

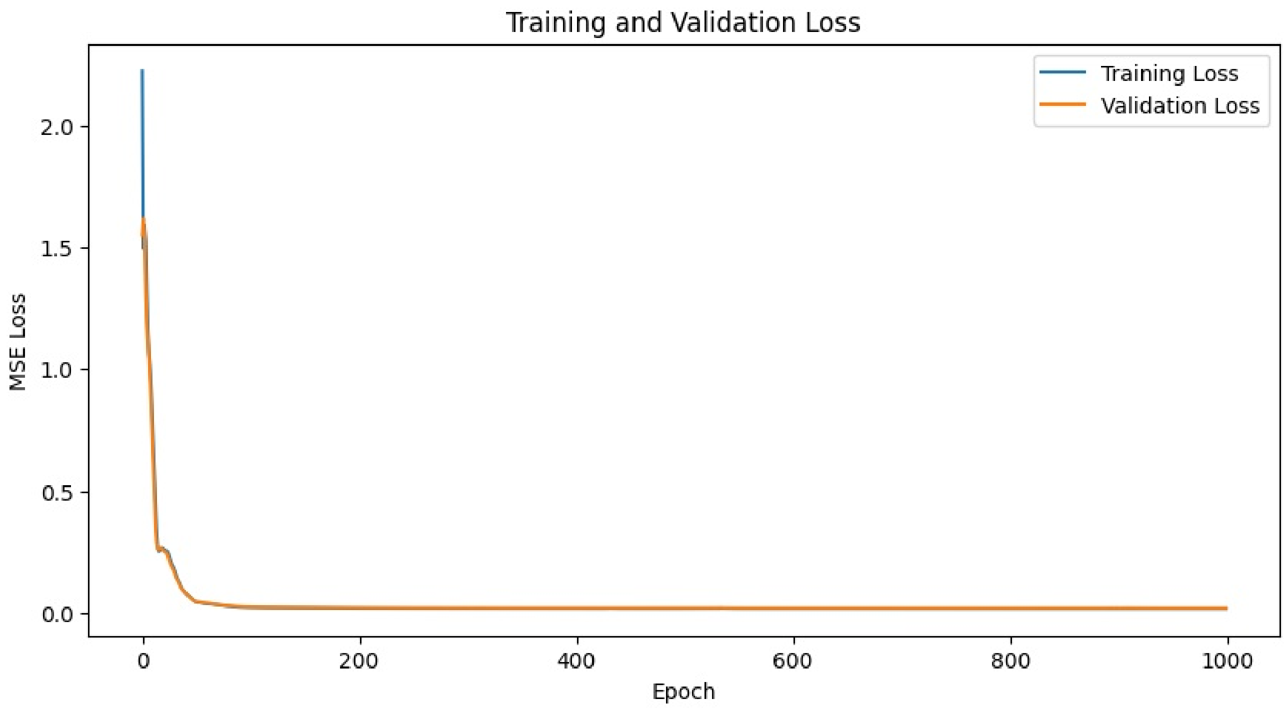
Now train more networks with the following details:

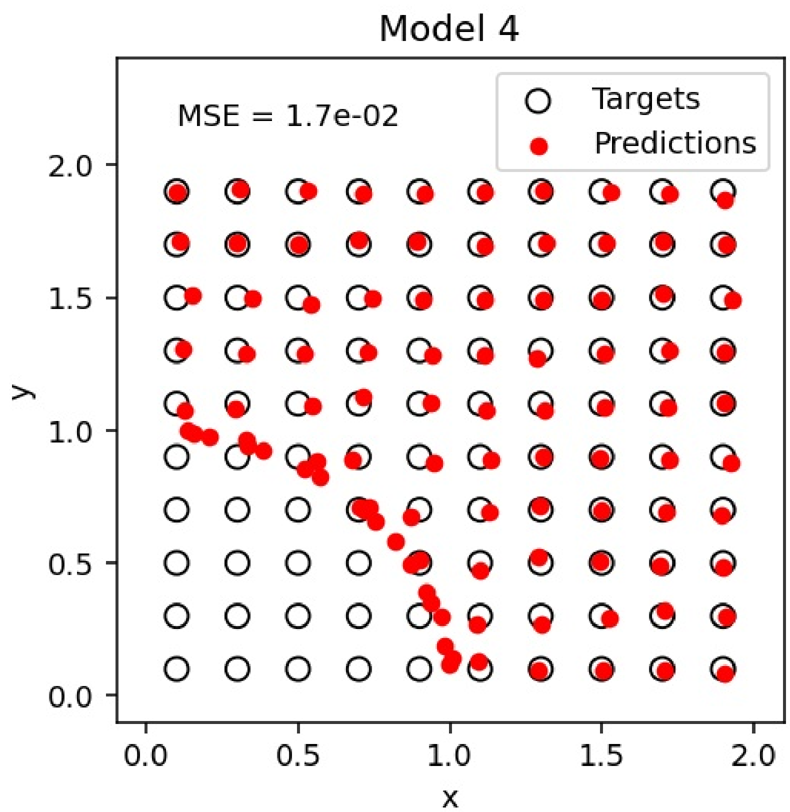
1. `hidden_layer_sizes=[48,48], max_angle=torch.pi/2` , train with `lr=0.01, epochs=1000, gamma=.995`
2. `hidden_layer_sizes=[48,48], max_angle=None` , train with `lr=1, epochs=1000, gamma=1`
3. `hidden_layer_sizes=[48,48], max_angle=2` , train with `lr=0.0001, epochs=300, gamma=1`

For each network, show a loss curve plot and a `plot_predictions` plot.

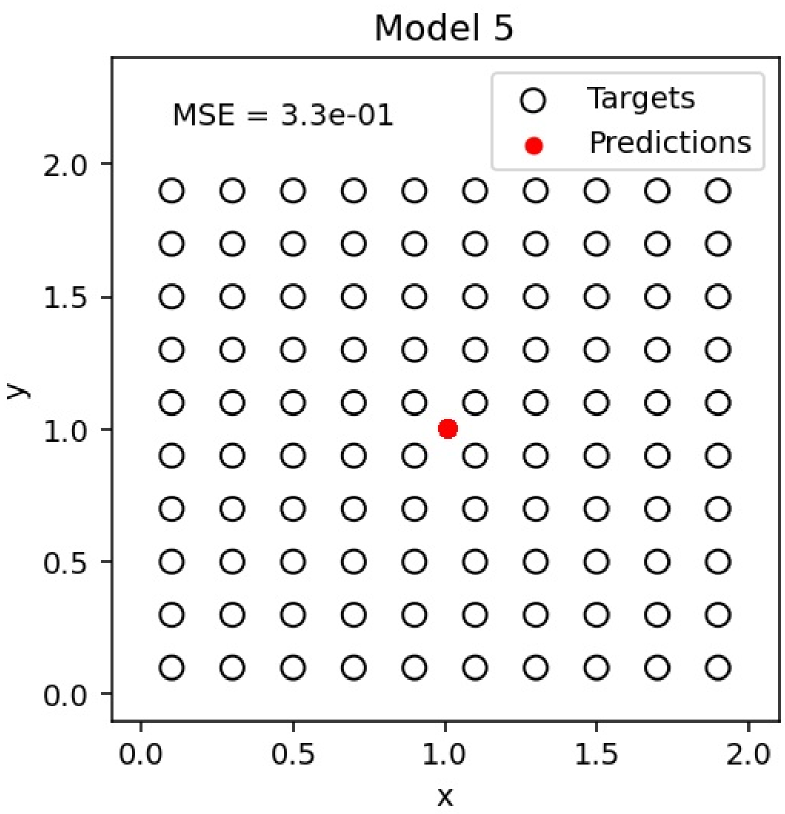
In [74]:
```
# YOUR CODE GOES HERE
model4 = InverseArm(hidden_layer_sizes=[48, 48], max_angle = np.pi/2)
model4 = train(model4, X_train, X_val, epochs = 1000, lr = 0.01, gamma = 0.995, create_plot = True)
plot_predictions(model4, title = "Model 4")

model5 = InverseArm(hidden_layer_sizes = [48,48], max_angle = None)
model5 = train(model5, X_train, X_val, epochs = 1000, lr = 1, gamma = 1, create_plot = True)
plot_predictions(model5, title="Model 5")

model6 = InverseArm(hidden_layer_sizes = [48,48], max_angle = 2)
model6 = train(model6, X_train, X_val, epochs = 300, lr = 0.0001, gamma = 1, create_plot = True)
plot_predictions(model6, title = "Model 6")
```

```
C:\Users\barat\AppData\Local\Temp\ipykernel_15088\664775586.py:5: UserWarning: To copy construct from a tensor,
it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
  X_train = torch.tensor(X_train, dtype=torch.float32)
C:\Users\barat\AppData\Local\Temp\ipykernel_15088\664775586.py:6: UserWarning: To copy construct from a tensor,
it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
  X_val = torch.tensor(X_val, dtype=torch.float32)
Epoch    0/1000: Train Loss = 2.2229, Validation Loss = 1.5491
Epoch   40/1000: Train Loss = 0.0807, Validation Loss = 0.0795
Epoch   80/1000: Train Loss = 0.0265, Validation Loss = 0.0287
Epoch  120/1000: Train Loss = 0.0203, Validation Loss = 0.0227
Epoch  160/1000: Train Loss = 0.0190, Validation Loss = 0.0214
Epoch  200/1000: Train Loss = 0.0183, Validation Loss = 0.0207
Epoch  240/1000: Train Loss = 0.0179, Validation Loss = 0.0204
Epoch  280/1000: Train Loss = 0.0177, Validation Loss = 0.0202
Epoch  320/1000: Train Loss = 0.0175, Validation Loss = 0.0200
Epoch  360/1000: Train Loss = 0.0174, Validation Loss = 0.0198
Epoch  400/1000: Train Loss = 0.0172, Validation Loss = 0.0197
Epoch  440/1000: Train Loss = 0.0173, Validation Loss = 0.0197
Epoch  480/1000: Train Loss = 0.0171, Validation Loss = 0.0196
Epoch  520/1000: Train Loss = 0.0170, Validation Loss = 0.0195
Epoch  560/1000: Train Loss = 0.0170, Validation Loss = 0.0195
Epoch  600/1000: Train Loss = 0.0169, Validation Loss = 0.0194
Epoch  640/1000: Train Loss = 0.0170, Validation Loss = 0.0195
Epoch  680/1000: Train Loss = 0.0169, Validation Loss = 0.0193
Epoch  720/1000: Train Loss = 0.0169, Validation Loss = 0.0193
Epoch  760/1000: Train Loss = 0.0169, Validation Loss = 0.0193
Epoch  800/1000: Train Loss = 0.0169, Validation Loss = 0.0194
Epoch  840/1000: Train Loss = 0.0168, Validation Loss = 0.0192
Epoch  880/1000: Train Loss = 0.0168, Validation Loss = 0.0192
Epoch  920/1000: Train Loss = 0.0170, Validation Loss = 0.0193
Epoch  960/1000: Train Loss = 0.0167, Validation Loss = 0.0192
```


Training and Validation Loss
Training Loss
Validation Loss
2.0
1.5
1.0
0.5
0.0
MSE Loss
0
200
400
600
800
1000
Epoch

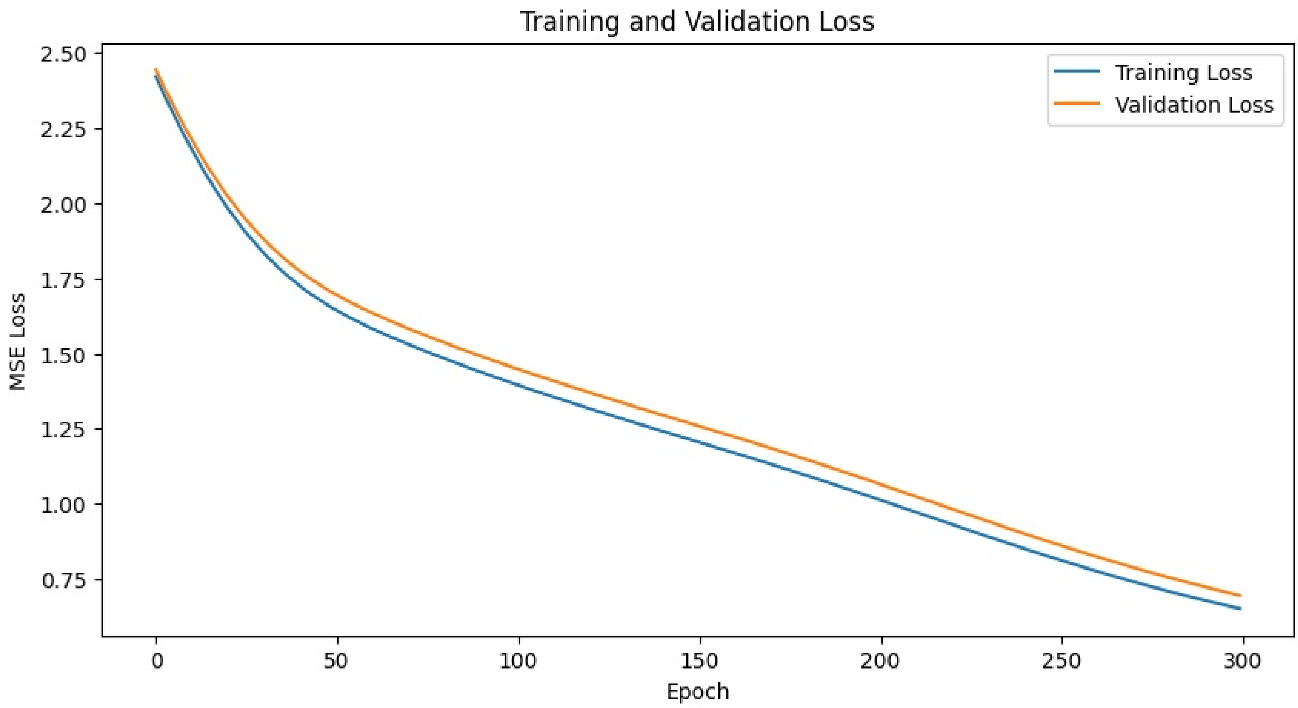

Model 4
MSE = 1.7e-02
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x

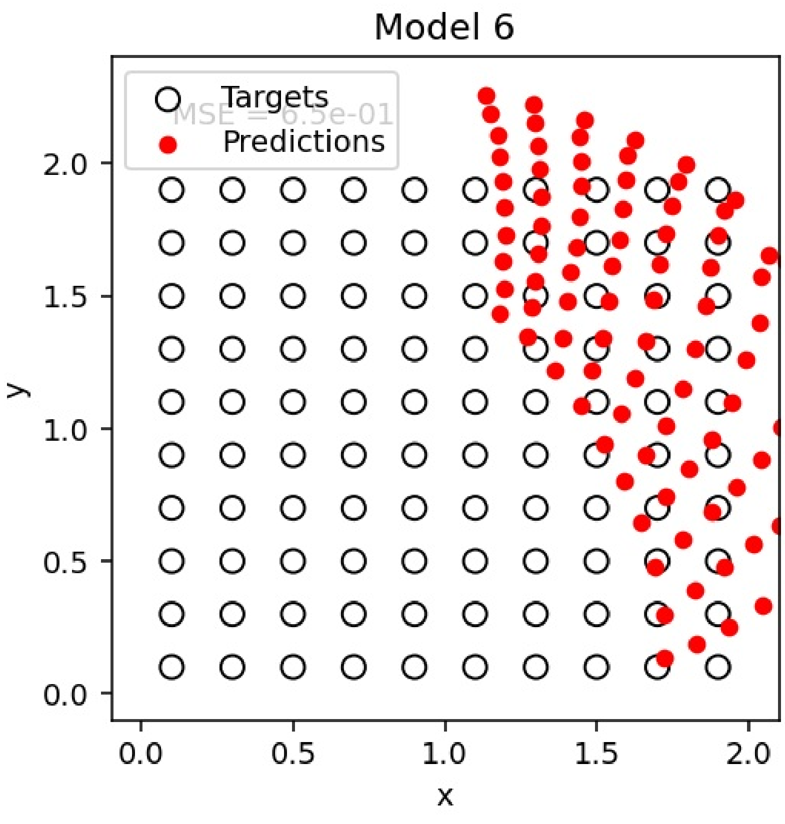
```
Epoch    0/1000: Train Loss = 2.6804, Validation Loss = 2.8504
Epoch   40/1000: Train Loss = 0.3415, Validation Loss = 0.8446
Epoch   80/1000: Train Loss = 0.5532, Validation Loss = 0.4691
Epoch  120/1000: Train Loss = 0.3397, Validation Loss = 0.3425
Epoch  160/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  200/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  240/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  280/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  320/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  360/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  400/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  440/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  480/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  520/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  560/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  600/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  640/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  680/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  720/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  760/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  800/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  840/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  880/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  920/1000: Train Loss = 0.3395, Validation Loss = 0.3426
Epoch  960/1000: Train Loss = 0.3395, Validation Loss = 0.3426
```



Model 5
MSE = 3.3e-01
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x

```
Epoch   0/300: Train Loss = 2.4209, Validation Loss = 2.4439
Epoch  12/300: Train Loss = 2.1337, Validation Loss = 2.1677
Epoch  24/300: Train Loss = 1.9155, Validation Loss = 1.9580
Epoch  36/300: Train Loss = 1.7620, Validation Loss = 1.8101
Epoch  48/300: Train Loss = 1.6569, Validation Loss = 1.7079
Epoch  60/300: Train Loss = 1.5812, Validation Loss = 1.6333
Epoch  72/300: Train Loss = 1.5192, Validation Loss = 1.5718
Epoch  84/300: Train Loss = 1.4634, Validation Loss = 1.5162
Epoch  96/300: Train Loss = 1.4113, Validation Loss = 1.4644
Epoch 108/300: Train Loss = 1.3624, Validation Loss = 1.4158
Epoch 120/300: Train Loss = 1.3157, Validation Loss = 1.3692
Epoch 132/300: Train Loss = 1.2704, Validation Loss = 1.3239
Epoch 144/300: Train Loss = 1.2264, Validation Loss = 1.2800
Epoch 156/300: Train Loss = 1.1829, Validation Loss = 1.2364
Epoch 168/300: Train Loss = 1.1385, Validation Loss = 1.1918
Epoch 180/300: Train Loss = 1.0926, Validation Loss = 1.1456
Epoch 192/300: Train Loss = 1.0450, Validation Loss = 1.0977
Epoch 204/300: Train Loss = 0.9961, Validation Loss = 1.0483
Epoch 216/300: Train Loss = 0.9467, Validation Loss = 0.9982
Epoch 228/300: Train Loss = 0.8975, Validation Loss = 0.9483

Epoch 240/300: Train Loss = 0.8496, Validation Loss = 0.8995
Epoch 252/300: Train Loss = 0.8037, Validation Loss = 0.8526
Epoch 264/300: Train Loss = 0.7604, Validation Loss = 0.8082
Epoch 276/300: Train Loss = 0.7202, Validation Loss = 0.7667
Epoch 288/300: Train Loss = 0.6831, Validation Loss = 0.7282
```





Prompts

None of these 3 models should have great performance. Describe what went wrong in each case.

Model 4 is performing poorly because the max_angle parameter is incorrectly set. Since max_angle is set to pi/2, the predictions for the lower half of the x and y coordinates are incorrect.

Model 5 is underperforming due to an excessively high learning rate of 1. This causes the model to oscillate around the global minimum without converging.

Model 6 is underfitting because the number of epochs is too low for the model to reach the global optimum. As a result, the loss function does not fully converge, leading to poor performance.