SRECHARAN SSELVAM

Homework 8

Instructions
This homework contains **1** concepts and **4** 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:

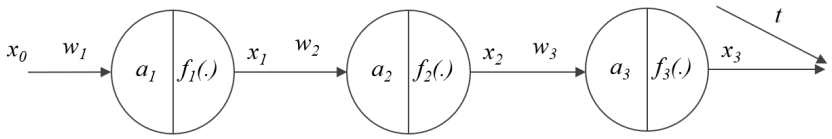
Name	Points
M8-L1-P1	10
M8-L2-P1	10
M8-L2-P2	10
M8-HW1	60

Problem 1 (10 points)

Consider the following network, with $x_0 = 2$, $w_1 = -1$, $w_2 = 3$, $w_3 = 7$, and linear (identity) activation functions.

Compute $\partial L/\partial w_3$, $\partial L/\partial w_2$, $\partial L/\partial w_1$ provided that t = -40

L=



ANSWER:

Given; $x_0 = 2$

$w_1 = 4$

$w_2 = 3$

$w_3 = 7$

$t = -40$

Linear activation function $f_i'(a_i) = 1$

Now let's calculate the forward pass to find the outputs of each node;

$\Rightarrow a_1 = x_0 \cdot w_1 = -2$

$\Rightarrow x_1 = f_1(a_1) = -2$

$\Rightarrow \quad a_2 = x_1 \cdot w_2 = -6$

$\Rightarrow \quad x_2 = f_2(a_2) = -6$

$\Rightarrow \quad a_3 = x_2 \cdot w_3 = -42$

$\Rightarrow \quad x_3 = f_3(a_3) = -42$

Now, we are calculating the error;

$\Rightarrow e = t - x_3$

$\Rightarrow e = -40 - (-42) = 2$

For w_3,

$\delta_3 = -(t - x_3) = -2$

$$\frac{\partial L}{\partial w_3} = \delta_3 \cdot x_2 = 12$$

For w_2,

$\delta_2 = \delta_3 \cdot w_3 \cdot f_2'(a_2) = -14$

$$\frac{\partial L}{\partial w_2} = \delta_2 x_1 = 28$$

For w_1,

$$\delta_1 = \delta_2 \cdot w_2 \cdot f_1'(a_1) = -12$$

$$\frac{\partial L}{\partial w_1} = \delta_1 \cdot x_0 = -84$$

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 [35]:
```
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 [36]:
```
# YOUR CODE GOES HERE
def sigmoid(x):
    s = 1/(1+np.exp(-x))
    return s

def sigmoid_derivative(x):
    ds = sigmoid(x)*(1-sigmoid(x))
    return ds
```

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 [44]:
```
# YOUR CODE GOES HERE
z1 = np.dot(W1, x0)
a1 = sigmoid(z1)
z2 = np.dot(W2, a1)
y = z2
loss = 0.5 * (y - t) ** 2
print("The Output of the neural network is =", y)
```

```
The Output of the neural network is = [[-1.31123207]]
```

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 [45]:
```
# YOUR CODE GOES HERE
delta2 = (y - t)
dw2 = np.dot(delta2, a1.T)
delta1 = np.dot(W2.T, delta2) * sigmoid_derivative(z1)
dw1 = np.dot(delta1, x0.T)
print("The Gradient dL/dW2 is = ", dw2)
print("The Gradient dL/dW1 is = \n", dw1)
```

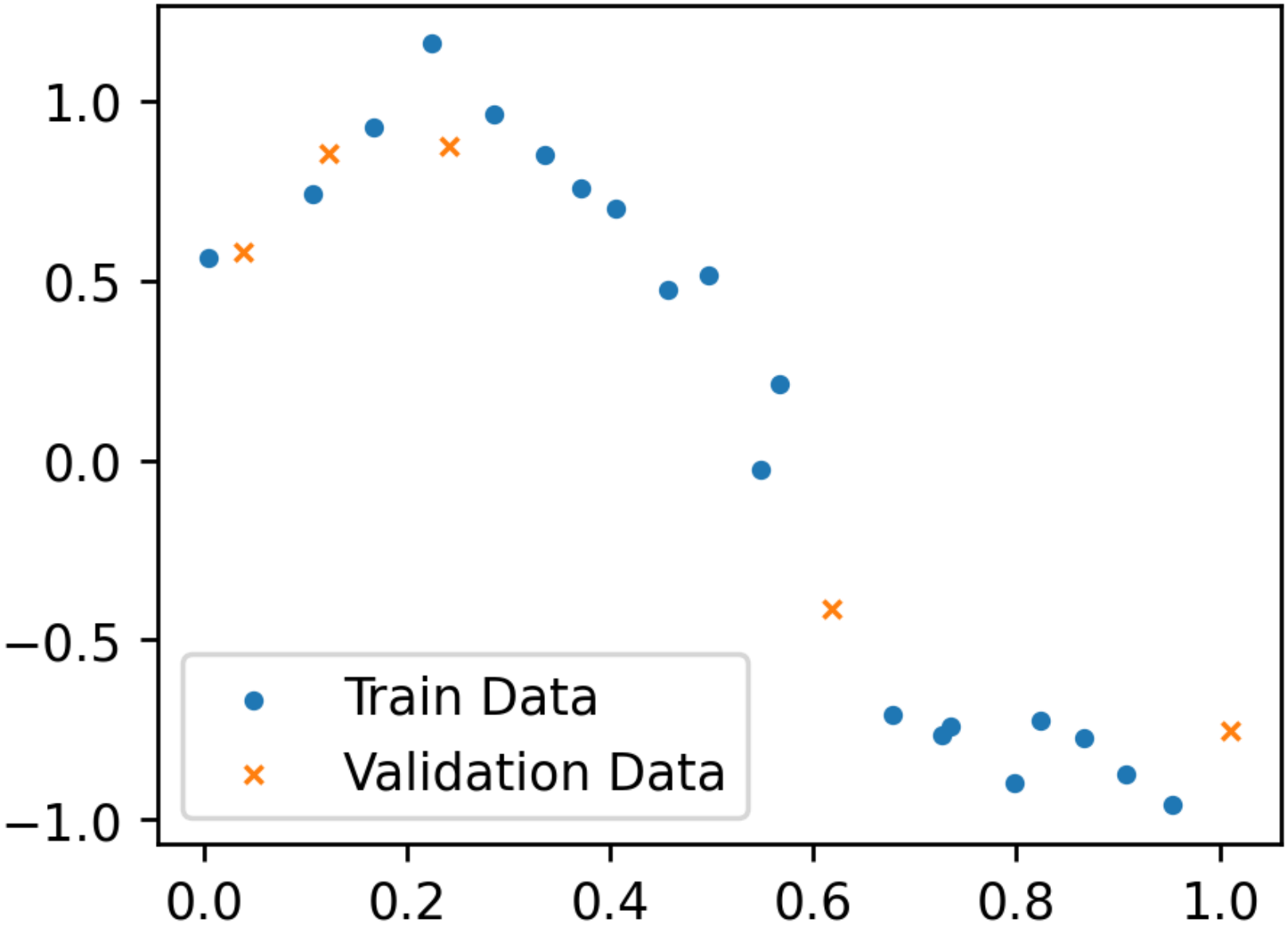
```
The Gradient dL/dW2 is =  [[8.21031503e-02 2.43316128e-24 1.04899215e-08]]
The Gradient dL/dW1 is =
 [[ 1.59095673e+00  4.77287018e+00]
 [-9.73264513e-24 -2.91979354e-23]
 [-1.04899214e-07 -3.14697641e-07]]
```

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 [21]:
```
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:

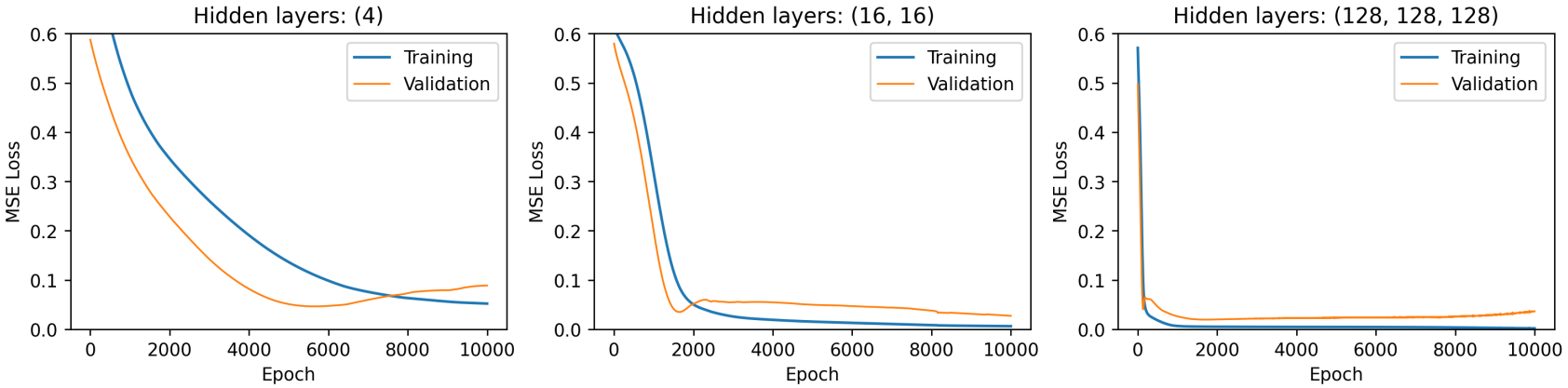
- `model1`: 1 hidden layer with 4 neurons. That is, the network should have a linear transformation from size 1 to size 4. Then a ReLU activation should be applied. Finally, a linear transformation from size 4 to size 1 gives the network output. (Note: Your regression network should not have an activation 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 [23]:
```
# YOUR CODE GOES HERE
model1 = nn.Sequential(
    nn.Linear(1, 4),
    nn.ReLU(),
    nn.Linear(4, 1))

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

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

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 [24]:
```
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).

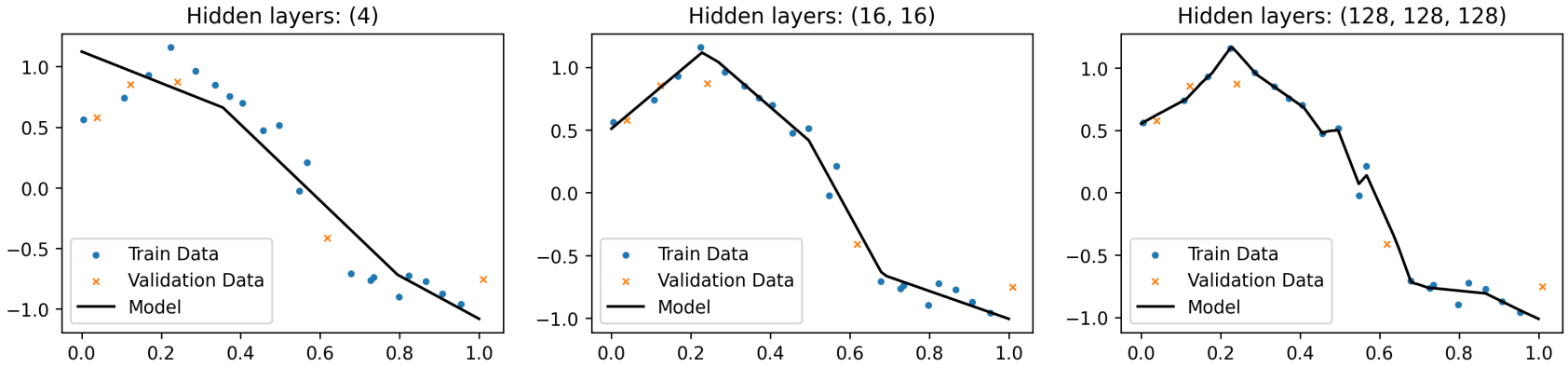
In [25]:
```
for i, model in enumerate([model1, model2, model3]):
    train_loss, val_loss = get_loss(model)
    print(f"Model {i+1}, hidden layers {hidden_layers[i]:>15}:   Train MSE: {train_loss:.4f}    Test MSE: {val_loss:
```

```
Model 1, hidden layers             (4):   Train MSE: 0.0520    Test MSE: 0.0887
Model 2, hidden layers         (16, 16):   Train MSE: 0.0061    Test MSE: 0.0272
Model 3, hidden layers (128, 128, 128):   Train MSE: 0.0017    Test MSE: 0.0364
```

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 [26]:
```
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?

In [28]:
```
print("1) Model 3 with hidden layers (128, 128, 128) has Train MSE: 0.0017 and Test MSE: 0.0364. This model exhibits
print("2) Model 1 with just 4 hidden layers doesn't perform as well as the other models. It's the simplest model and
print("3) The best model is typically one that neither overfits nor underfits and has a good balance between training
```

```
1) Model 3 with hidden layers (128, 128, 128) has Train MSE: 0.0017 and Test MSE: 0.0364. This model exhibits the low
est training MSE, yet its test MSE is notably higher compared to its training MSE. This difference is indicative of o
verfitting. Throughout the training process, the loss curve for the training set would consistently diminish, whereas
the validation loss might either increase or plateau. Such behavior demonstrates a discrepancy between training and v
alidation performance.

2) Model 1 with just 4 hidden layers doesn't perform as well as the other models. It's the simplest model and misses
out on capturing the data's patterns. When trained, both its training and validation errors drop slowly and might set
tle at higher errors compared to other models.

3) The best model is typically one that neither overfits nor underfits and has a good balance between training and va
lidation performance. From the provided results, Model 2 with hidden layers (16, 16) seems to be the best model as it
has a relatively low training MSE and its test MSE is also quite low. For this model, during training, both the train
ing and validation loss curves would decrease, with the validation loss closely following the training loss without a
significant gap or increase. This indicates a good balance between fitting the training data and generalizing to new
data.
```

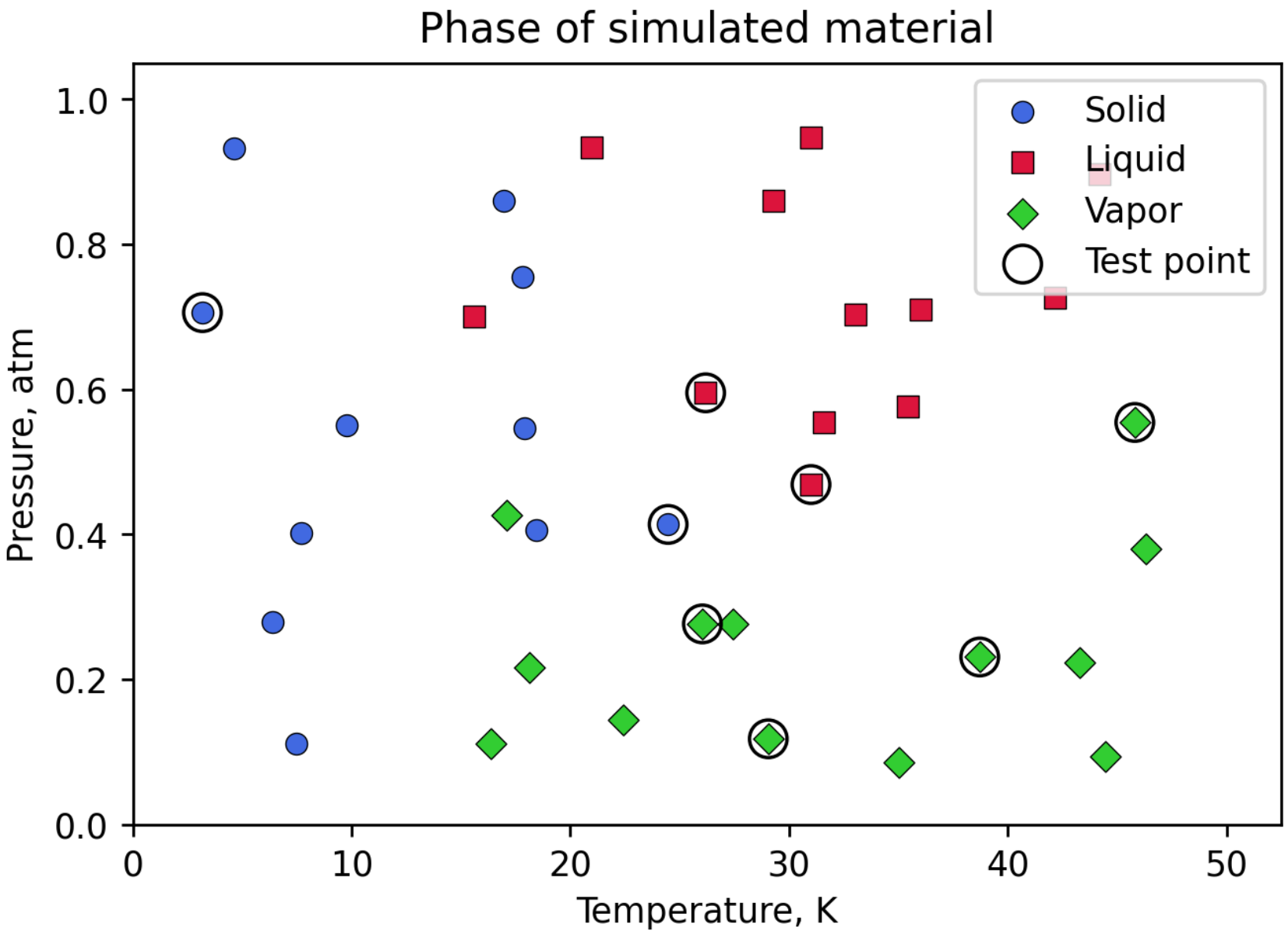
M8-L2 Problem 2

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 [26]:
```
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([7.4881350392732475,16.351893663724194,22.427633760716436,29.04883182996897,35.03654799338904,44.458941
x2 = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.11818202991034835,0.0859507900573786,0.09
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="limegr
    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[i])

    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 [29]:
```
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 = torch.nn.functional.relu(self.fc1(X))
        X = torch.nn.functional.relu(self.fc2(X))
        X = torch.nn.functional.softmax(self.fc3(X),dim=1)
        return X
```

Training

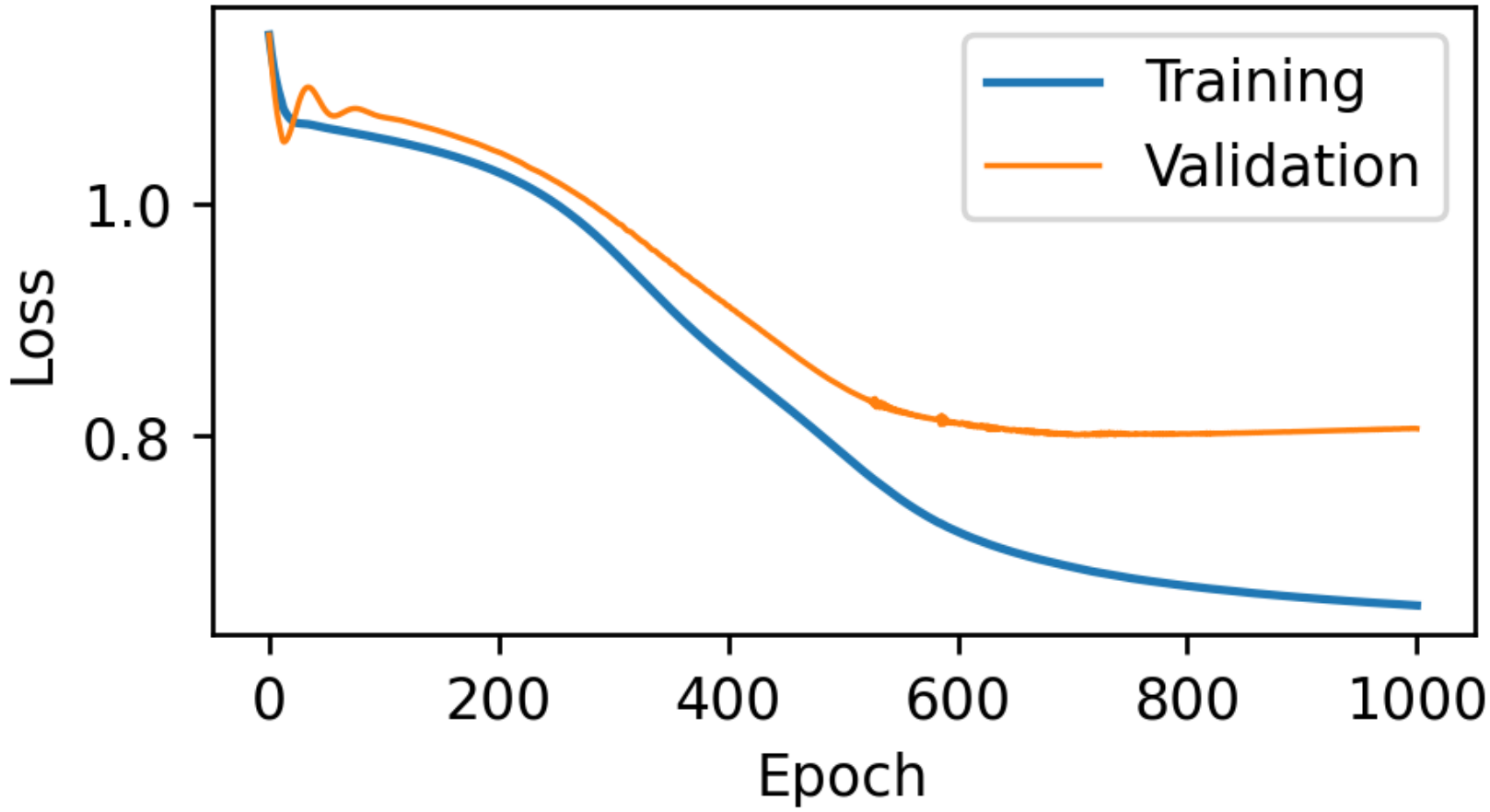
Most of the training code has been provided below. Please add the following where indicated:

- 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 [31]:
```
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 = torch.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_val.it


plot_loss(train_hist, val_hist)
```

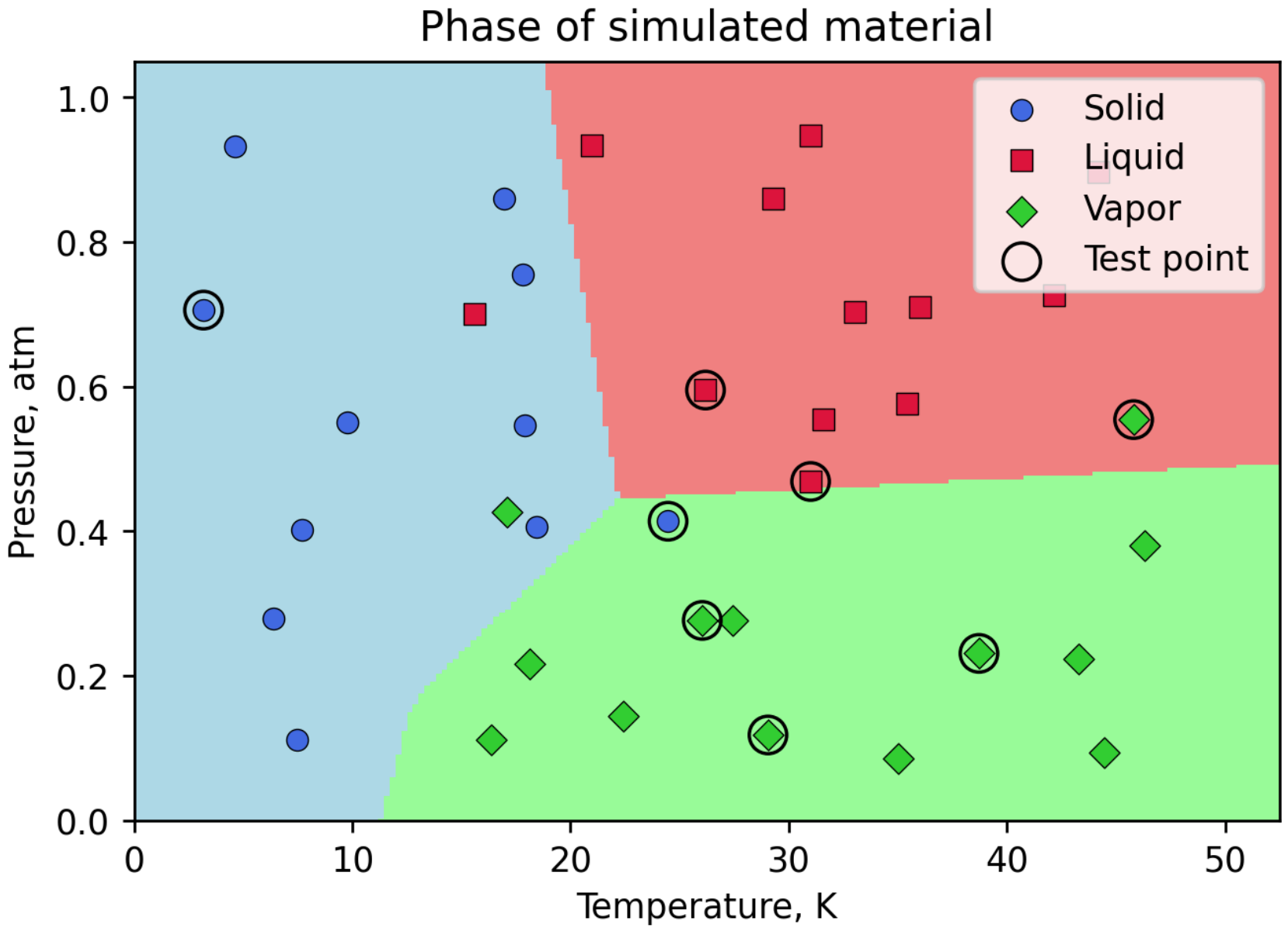
```
Epoch    0 of 1000:   Train Loss = 1.1457   Validation Loss = 1.1455
Epoch   40 of 1000:   Train Loss = 1.0676   Validation Loss = 1.0955
Epoch   80 of 1000:   Train Loss = 1.0600   Validation Loss = 1.0815
Epoch  120 of 1000:   Train Loss = 1.0518   Validation Loss = 1.0707
Epoch  160 of 1000:   Train Loss = 1.0414   Validation Loss = 1.0591
Epoch  200 of 1000:   Train Loss = 1.0272   Validation Loss = 1.0448
Epoch  240 of 1000:   Train Loss = 1.0066   Validation Loss = 1.0251
Epoch  280 of 1000:   Train Loss = 0.9770   Validation Loss = 1.0007
Epoch  320 of 1000:   Train Loss = 0.9392   Validation Loss = 0.9718
Epoch  360 of 1000:   Train Loss = 0.8999   Validation Loss = 0.9411
Epoch  400 of 1000:   Train Loss = 0.8653   Validation Loss = 0.9123
Epoch  440 of 1000:   Train Loss = 0.8338   Validation Loss = 0.8829
Epoch  480 of 1000:   Train Loss = 0.8015   Validation Loss = 0.8543
Epoch  520 of 1000:   Train Loss = 0.7681   Validation Loss = 0.8322
Epoch  560 of 1000:   Train Loss = 0.7391   Validation Loss = 0.8182
Epoch  600 of 1000:   Train Loss = 0.7177   Validation Loss = 0.8109
Epoch  640 of 1000:   Train Loss = 0.7027   Validation Loss = 0.8062
Epoch  680 of 1000:   Train Loss = 0.6918   Validation Loss = 0.8029
Epoch  720 of 1000:   Train Loss = 0.6830   Validation Loss = 0.8015
Epoch  760 of 1000:   Train Loss = 0.6763   Validation Loss = 0.8017
Epoch  800 of 1000:   Train Loss = 0.6710   Validation Loss = 0.8022
Epoch  840 of 1000:   Train Loss = 0.6665   Validation Loss = 0.8026
Epoch  880 of 1000:   Train Loss = 0.6628   Validation Loss = 0.8035
Epoch  920 of 1000:   Train Loss = 0.6595   Validation Loss = 0.8045
Epoch  960 of 1000:   Train Loss = 0.6567   Validation Loss = 0.8056
Epoch 1000 of 1000:   Train Loss = 0.6542   Validation Loss = 0.8065
```



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 [32]:
```
plot_data(newfig=True)
plot_model(model)
plt.show()
```


Phase of simulated material
Solid
Liquid
Vapor
Test point
Pressure, atm
Temperature, K
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

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 [55]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
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
        y = self.L1 * torch.sin(theta1) + self.L2 * torch.sin(theta1 + theta2) + self.L3 * torch.sin(theta1 + theta2
        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)
    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", markeredgecolo
    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 [33]:
```
%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=False,
slider2 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta2',disabled=False,
slider3 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta3',disabled=False,

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

interactive_plot
```

Out[33]:
```
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 [56]:
```
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
        layers = []
        layers.append(nn.Linear(2, hidden_layer_sizes[0]))
        layers.append(nn.ReLU())
        for i in range(1, len(hidden_layer_sizes)):
            layers.append(nn.Linear(hidden_layer_sizes[i-1], hidden_layer_sizes[i]))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(hidden_layer_sizes[-1], 3))
        if self.max_angle is not None:
            layers.append(nn.Tanh())
            layers.append(nn.Hardtanh(min_val=-self.max_angle, max_val=self.max_angle))
        self.fc = nn.Sequential(*layers)
```

```
    def forward(self, xy):
        # YOUR CODE GOES HERE
        x = self.fc(xy)
        return x
```

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 [57]:
```
# YOUR CODE GOES HERE
x, y = np.meshgrid(np.linspace(0, 2, 100), np.linspace(0, 2, 100))
X = np.vstack([x.flatten(), y.flatten()]).T
X_train, X_val = train_test_split(X, test_size=0.2)
```

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 [71]:
```
def train(model, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995, create_plot=True):
    model.train()
    cr = nn.MSELoss()
    opt = optim.Adam(model.parameters(), lr=lr)
    shd = optim.lr_scheduler.ExponentialLR(opt, gamma=gamma)
    training_loss = []
    validation_loss = []
    fwd= ForwardArm()

    for i in range(epochs):
        X_tr_ten = torch.from_numpy(X_train).float()
        X_val_ten = torch.from_numpy(X_val).float()
        ang = model(X_tr_ten)
        op = fwd(ang)
        loss = cr(op, X_tr_ten)
        opt.zero_grad()
        loss.backward()
        opt.step()
        training_loss.append(loss.item())
        model.eval()
        with torch.no_grad():
            val_ang = model(X_val_ten)
            val_op = fwd(val_ang)
            val_loss = cr(val_op, X_val_ten)
            validation_loss.append(val_loss.item())

        if i % 100 == 0:
            print(f'Epoch {i} train loss = {loss.item()} val loss = {val_loss.item()}')
        shd.step()
        model.train()

    if create_plot:
        plt.figure(figsize=(10, 5))
        plt.plot(training_loss, label='Train Loss')
        plt.plot(validation_loss, label='Validation Loss')
        plt.xlabel('Epoch')
        plt.ylabel('MSE Loss')
```

```
        plt.title('Training and Validation Losses')
        plt.legend()
        plt.show()
    return model
```

Training a model

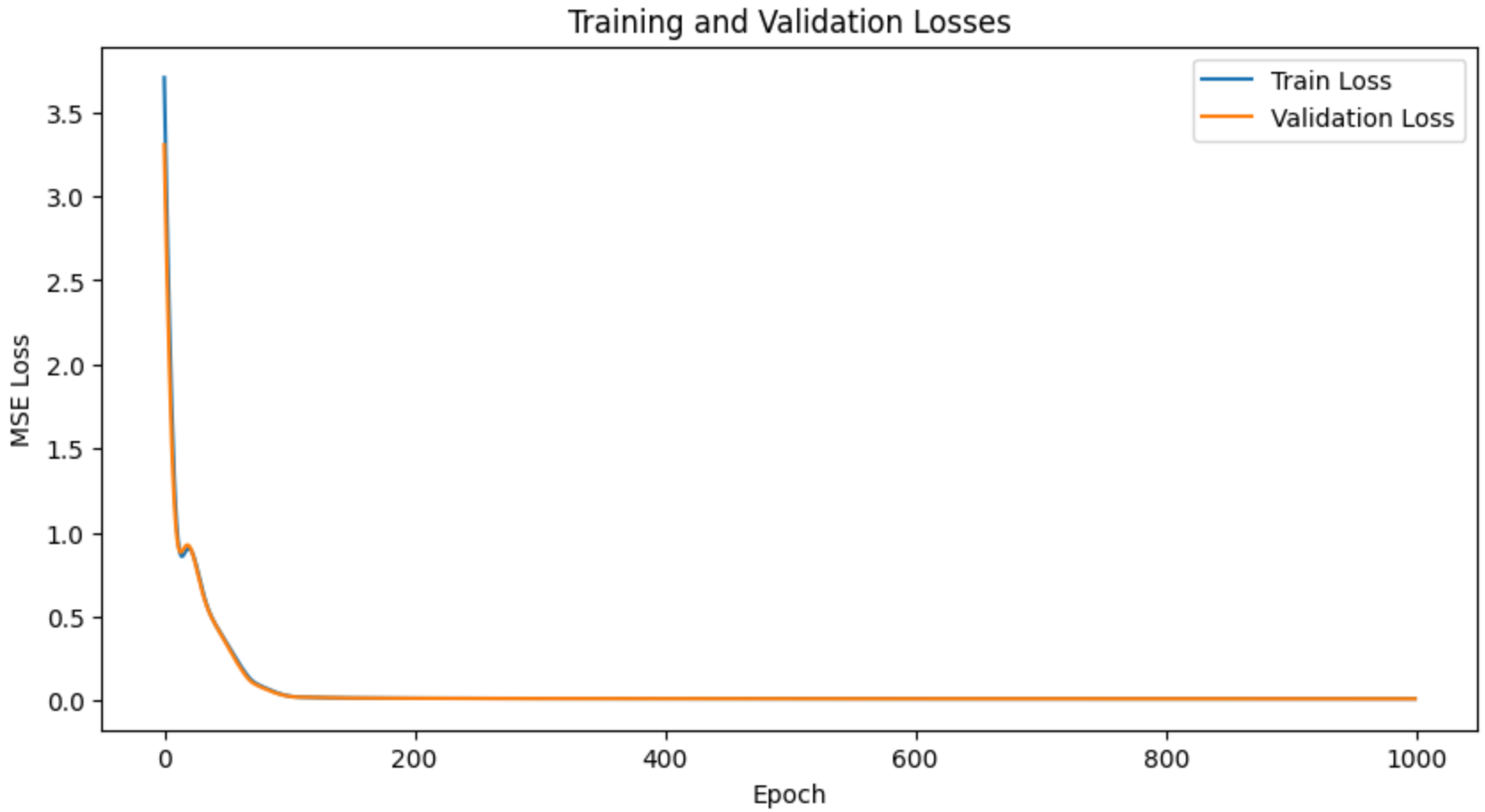
Create 3 models of different complexities (with `max_angle=None`):

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

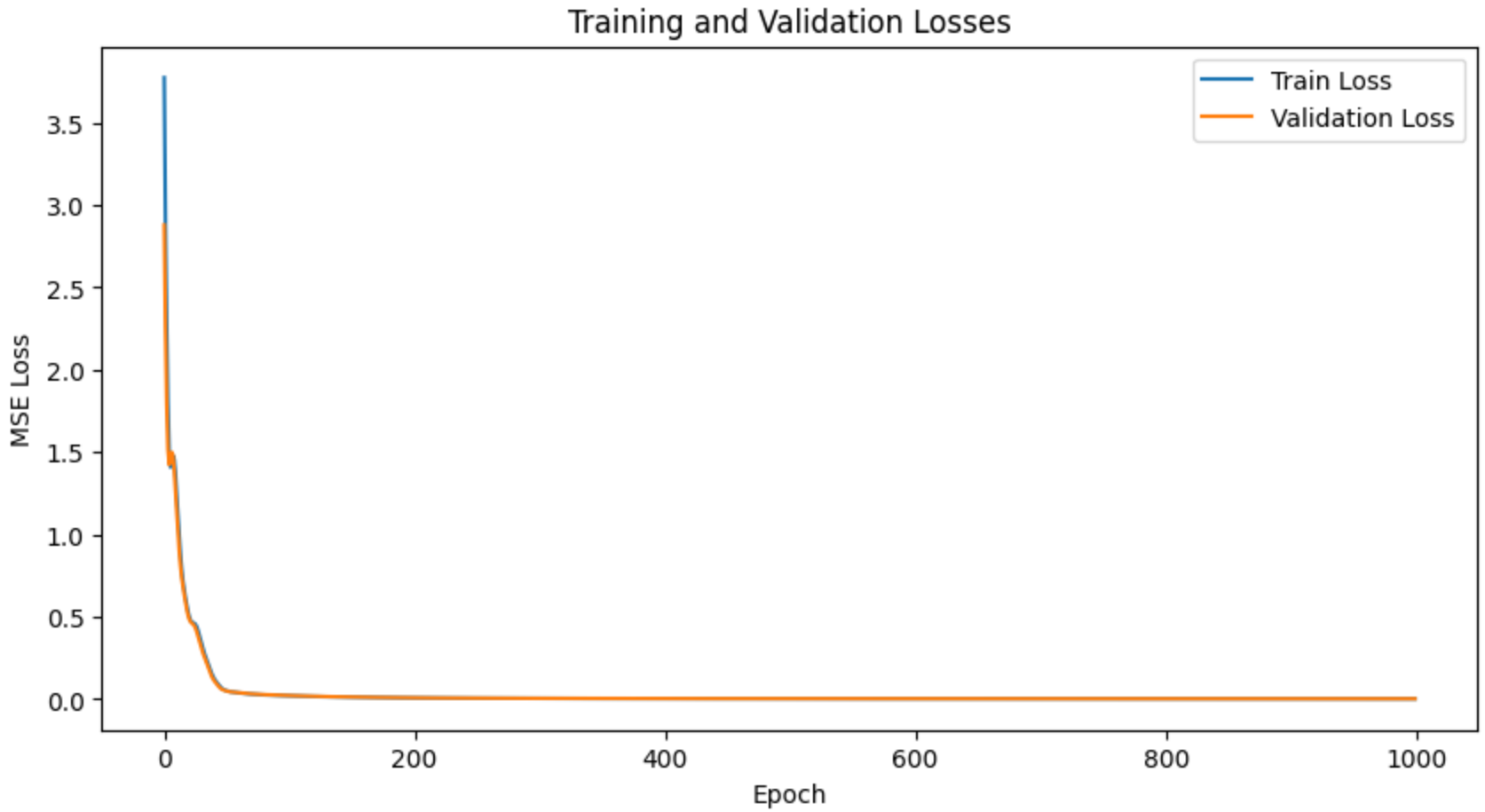
Train each model for 1000 epochs, learning rate 0.01, and gamma 0.995. Show the plot for each.

In [72]:
```
# YOUR CODE GOES HERE
model_1 = InverseArm([12])
model_2 = InverseArm([24, 24])
model_3 = InverseArm([48, 48, 48])

train(model_1, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995, create_plot=True)
train(model_2, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995, create_plot=True)
train(model_3, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995, create_plot=True)
```

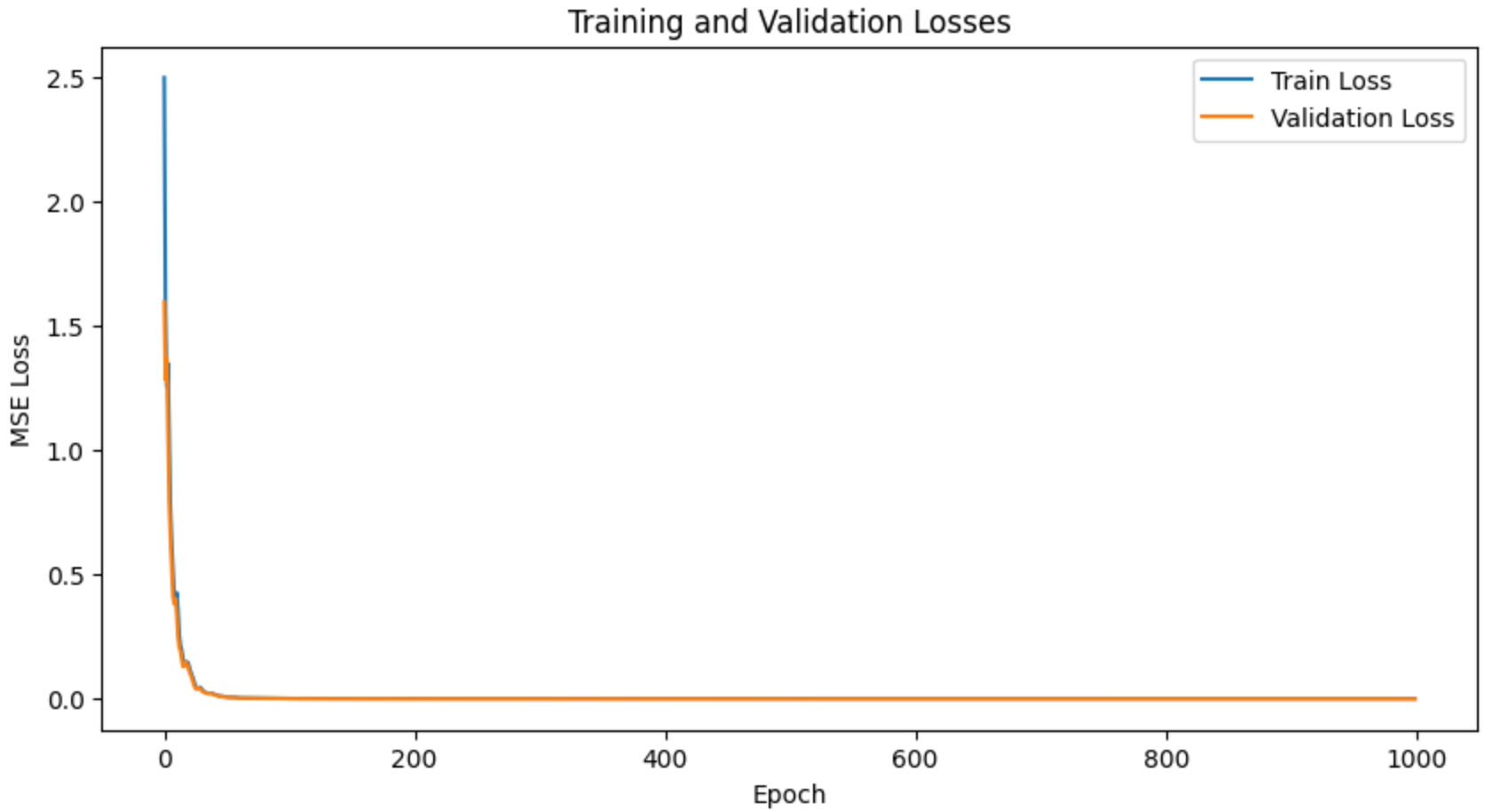
```
Epoch 0 train loss = 3.7049760818481445 val loss = 3.305222988128662
Epoch 100 train loss = 0.028093980625271797 val loss = 0.026456376537680626
Epoch 200 train loss = 0.015191786922514439 val loss = 0.01497284322977066
Epoch 300 train loss = 0.013723562471568584 val loss = 0.013505052775144577
Epoch 400 train loss = 0.013121205382049084 val loss = 0.012884859926998615
Epoch 500 train loss = 0.012802879326045513 val loss = 0.012556017376482487
Epoch 600 train loss = 0.012615183368325233 val loss = 0.01236350741237402
Epoch 700 train loss = 0.012499595992267132 val loss = 0.012244309298694134
Epoch 800 train loss = 0.012426290661096573 val loss = 0.012168945744633675
Epoch 900 train loss = 0.012379203923046589 val loss = 0.01212054118514061
```




```
Epoch 0 train loss = 3.774707555770874 val loss = 2.879422664642334
Epoch 100 train loss = 0.020421328023076057 val loss = 0.02175823599100113
Epoch 200 train loss = 0.007848592475056648 val loss = 0.008435597643256187
Epoch 300 train loss = 0.004591870121657848 val loss = 0.0049354094080626965
Epoch 400 train loss = 0.0033836194779723883 val loss = 0.003612231696024537
Epoch 500 train loss = 0.002826656913384795 val loss = 0.0030060133431106806
Epoch 600 train loss = 0.0025382433086633682 val loss = 0.0026966820005327463
Epoch 700 train loss = 0.0023760891053825617 val loss = 0.0025234369095414877
Epoch 800 train loss = 0.0022795062977820635 val loss = 0.00242129759863019
Epoch 900 train loss = 0.0022202744148671627 val loss = 0.0023583732545375824
```




```
Epoch 0 train loss = 2.498358964920044 val loss = 1.596420168876648
Epoch 100 train loss = 0.0013076066970825195 val loss = 0.0012916051782667637
Epoch 200 train loss = 0.0005248989327810705 val loss = 0.0005554915405809879
Epoch 300 train loss = 0.0003445088223088327 val loss = 0.0003773739153984934
Epoch 400 train loss = 0.0002886111033149063 6 val loss = 0.0003203086380381137
Epoch 500 train loss = 0.0002633528492879122 5 val loss = 0.0002939938276540488
Epoch 600 train loss = 0.0002496572560630739 val loss = 0.0002795915061142295 6
Epoch 700 train loss = 0.0002415531344013288 6 val loss = 0.0002709104155655950 3
Epoch 800 train loss = 0.0002365310647292062 6 val loss = 0.0002655406715348363
Epoch 900 train loss = 0.0002333780576009303 3 val loss = 0.0002621368330437690
```

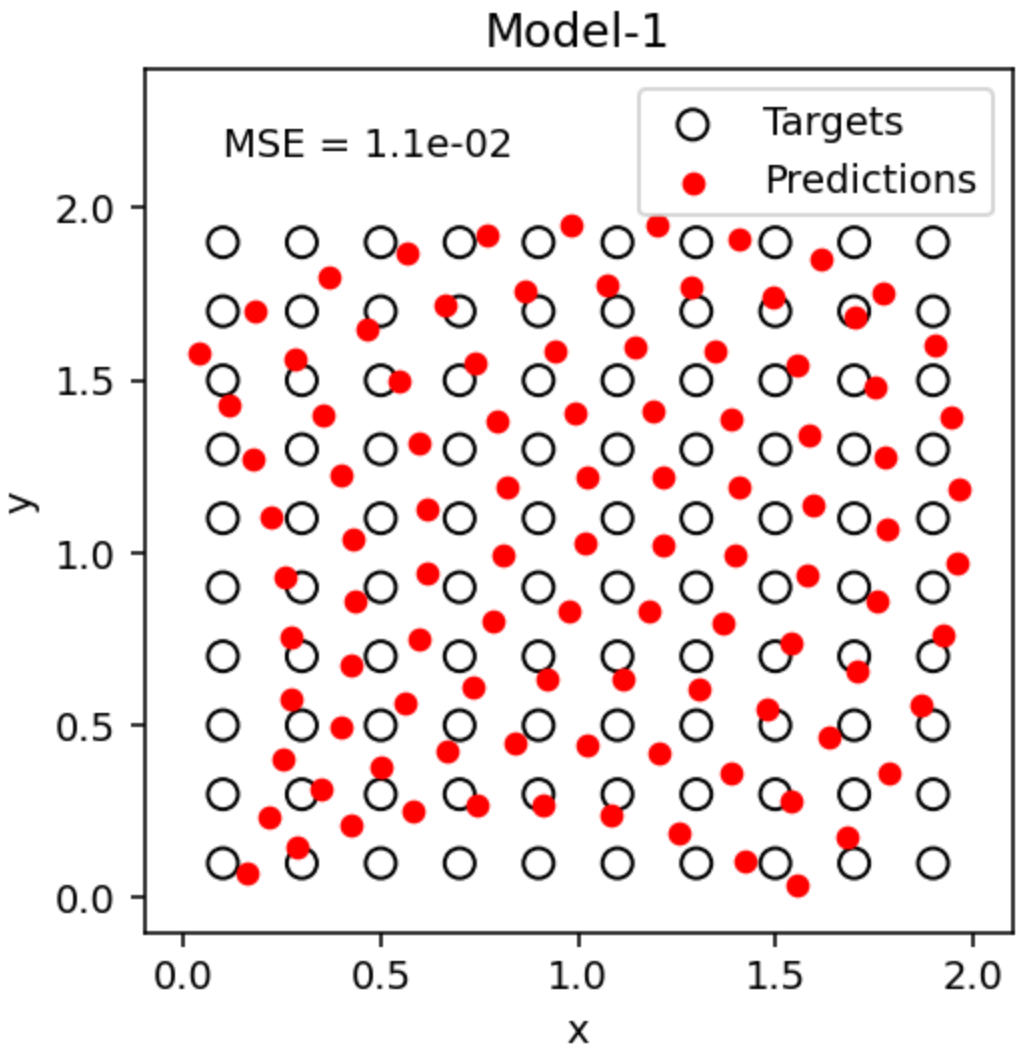


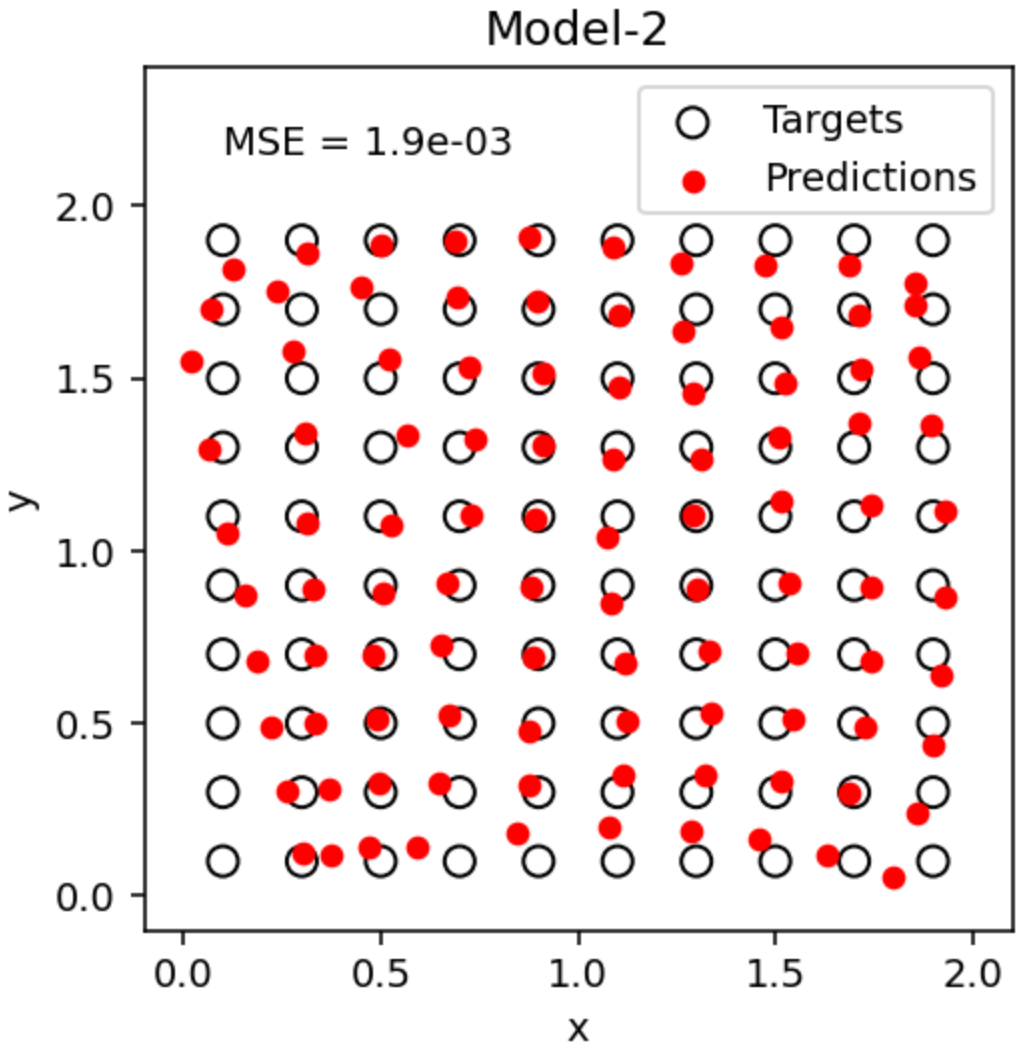
Out[72]:
```
InverseArm(
  (fc): Sequential(
    (0): Linear(in_features=2, out_features=48, bias=True)
    (1): ReLU()
    (2): Linear(in_features=48, out_features=48, bias=True)
    (3): ReLU()
    (4): Linear(in_features=48, out_features=48, bias=True)
    (5): ReLU()
    (6): Linear(in_features=48, out_features=3, bias=True)
  )
)
```

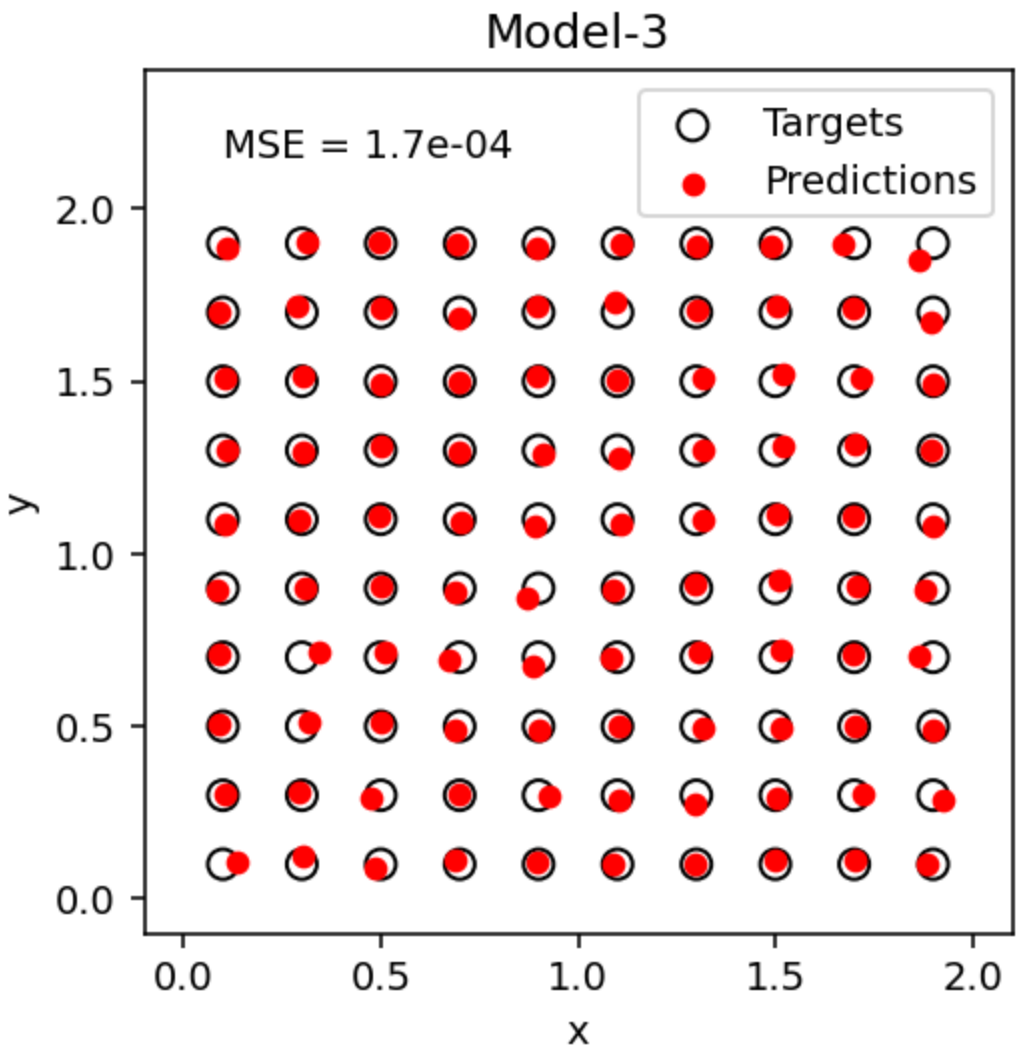
Visualizations

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 [74]:
```
# YOUR CODE GOES HERE
plot_predictions(model_1, title="Model-1")
plot_predictions(model_2, title="Model-2")
plot_predictions(model_3, title="Model-3")
```




Model-2
MSE = 1.9e-03
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x



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 [81]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, FloatSlider, Dropdown
model = model_3
```

```
def plot_inverse(x, y):
    xy = torch.Tensor([[x,y]])
    theta1, theta2, theta3 = model(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=True,
slider2 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100, description='y', disabled=False, continuous_update=True,

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

interactive_plot
```

Out[81]:
```
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:

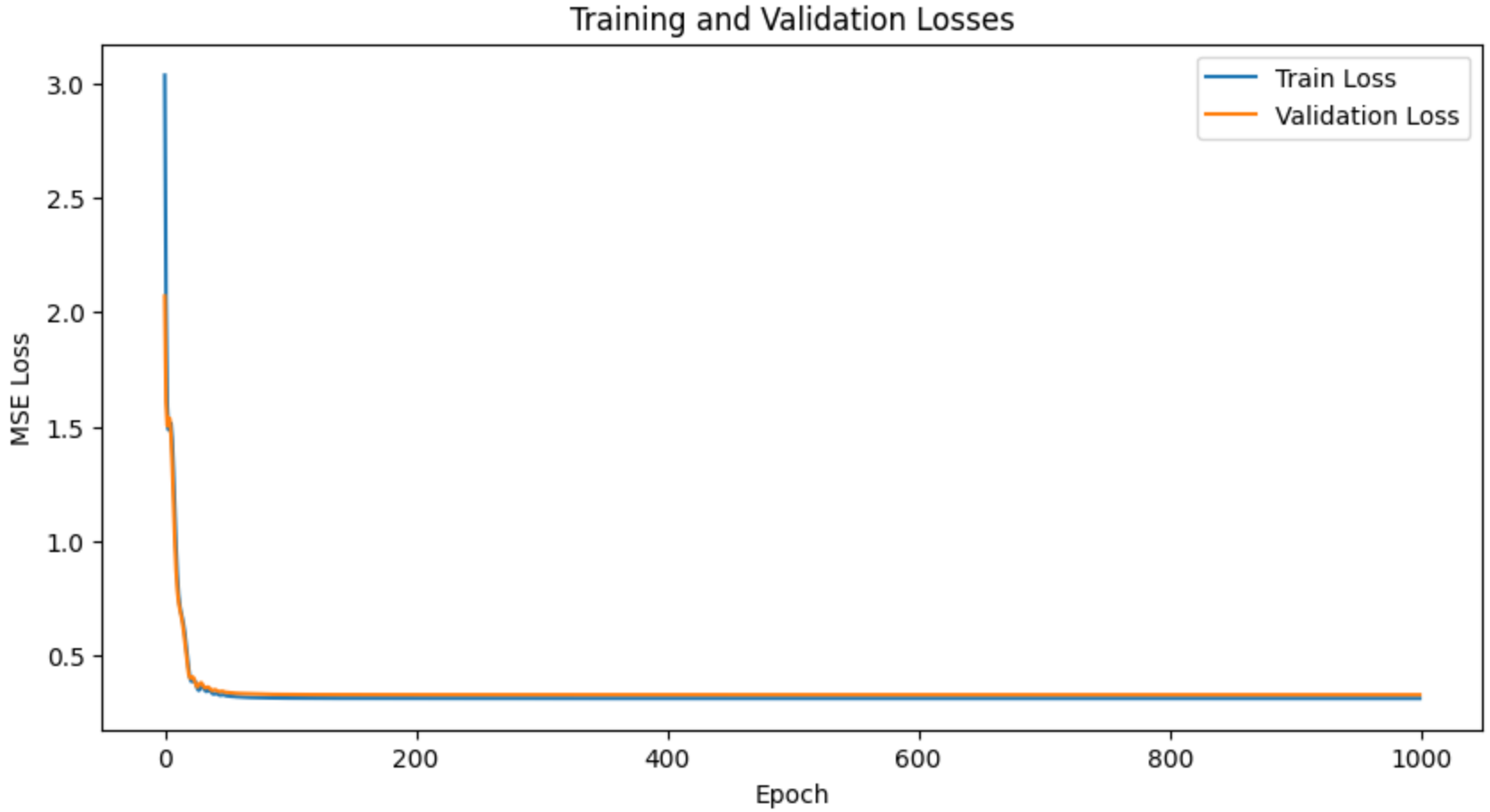
1. `hidden_layer_sizes=[48,48], max_angle=torch.pi/2` , train with `lr=0.01, epochs=1000, gamma=.995`
2. `hidden_layer_sizes=[48,48], max_angle=None` , train with `lr=1, epochs=1000, gamma=1`
3. `hidden_layer_sizes=[48,48], max_angle=2` , train with `lr=0.0001, epochs=300, gamma=1`

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

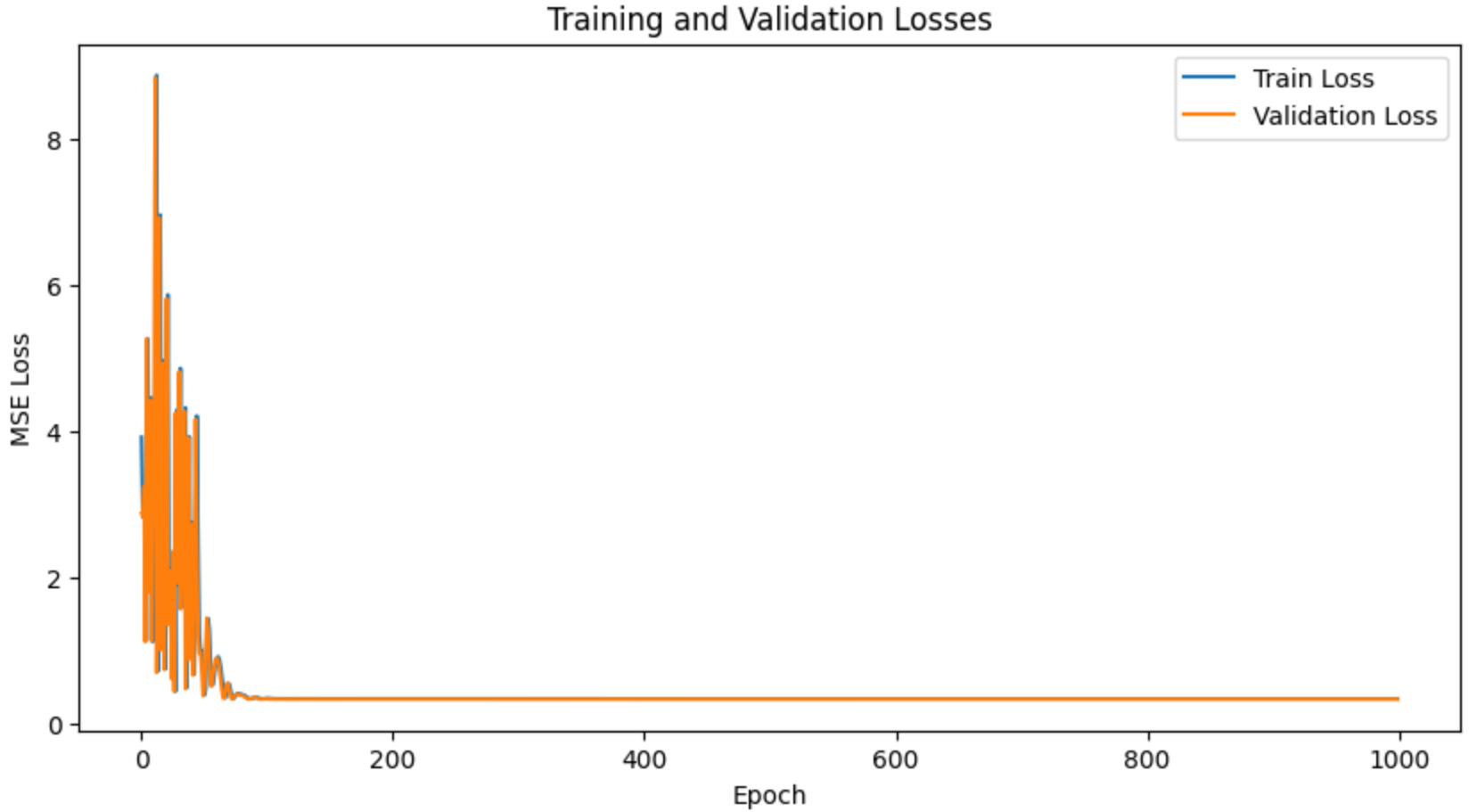
In [89]:
```
# YOUR CODE GOES HERE
model_1 = InverseArm(hidden_layer_sizes=[48, 48], max_angle=torch.pi / 2)
model_2 = InverseArm(hidden_layer_sizes=[48, 48], max_angle=None)
model_3 = InverseArm(hidden_layer_sizes=[48, 48], max_angle=2)

train(model_1, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995, create_plot=True)
train(model_2, X_train, X_val, lr=1, epochs=1000, gamma=1, create_plot=True)
train(model_3, X_train, X_val, lr=0.0001, epochs=300, gamma=1, create_plot=True)

plot_predictions(model_1, title="Model-1")
```

```
plot_predictions(model_2, title="Model-2")
plot_predictions(model_3, title="Model-3")
```

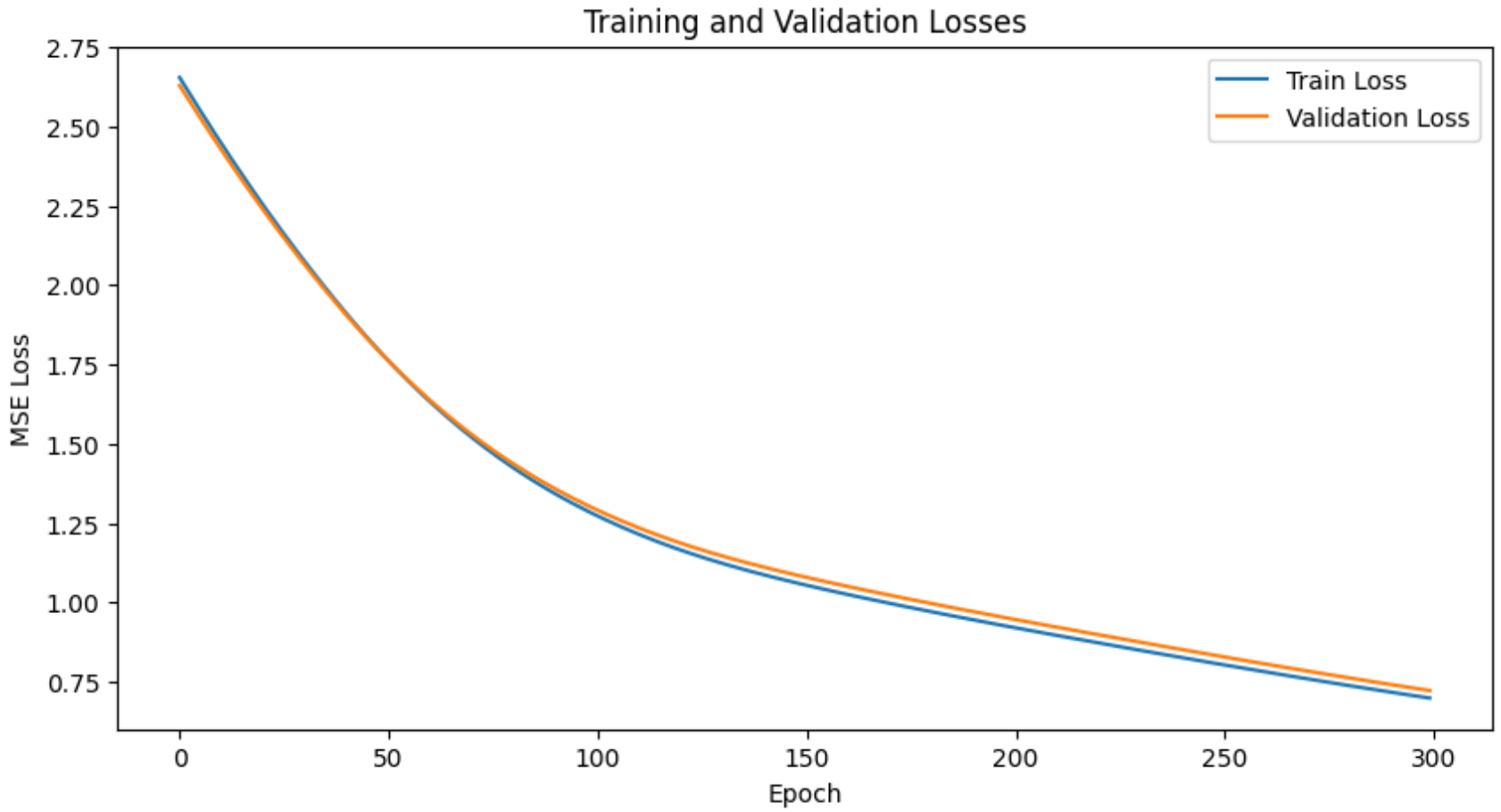
```
Epoch 0 train loss = 3.0359625816345215 val loss = 2.0704996585845947
Epoch 100 train loss = 0.3137739598751068 val loss = 0.3297020196914673
Epoch 200 train loss = 0.31281664967536926 val loss = 0.3288631737232208
Epoch 300 train loss = 0.3126669228076935 val loss = 0.3287266492843628
Epoch 400 train loss = 0.312605619430542 val loss = 0.32867249846458435
Epoch 500 train loss = 0.312574177980423 val loss = 0.3286433517932892
Epoch 600 train loss = 0.3125563859939575 val loss = 0.3286263048648834
Epoch 700 train loss = 0.31254565715789795 val loss = 0.3286157250404358
Epoch 800 train loss = 0.31253907084465027 val loss = 0.3286091685295105
Epoch 900 train loss = 0.31253471970558167 val loss = 0.32860496640205383
```

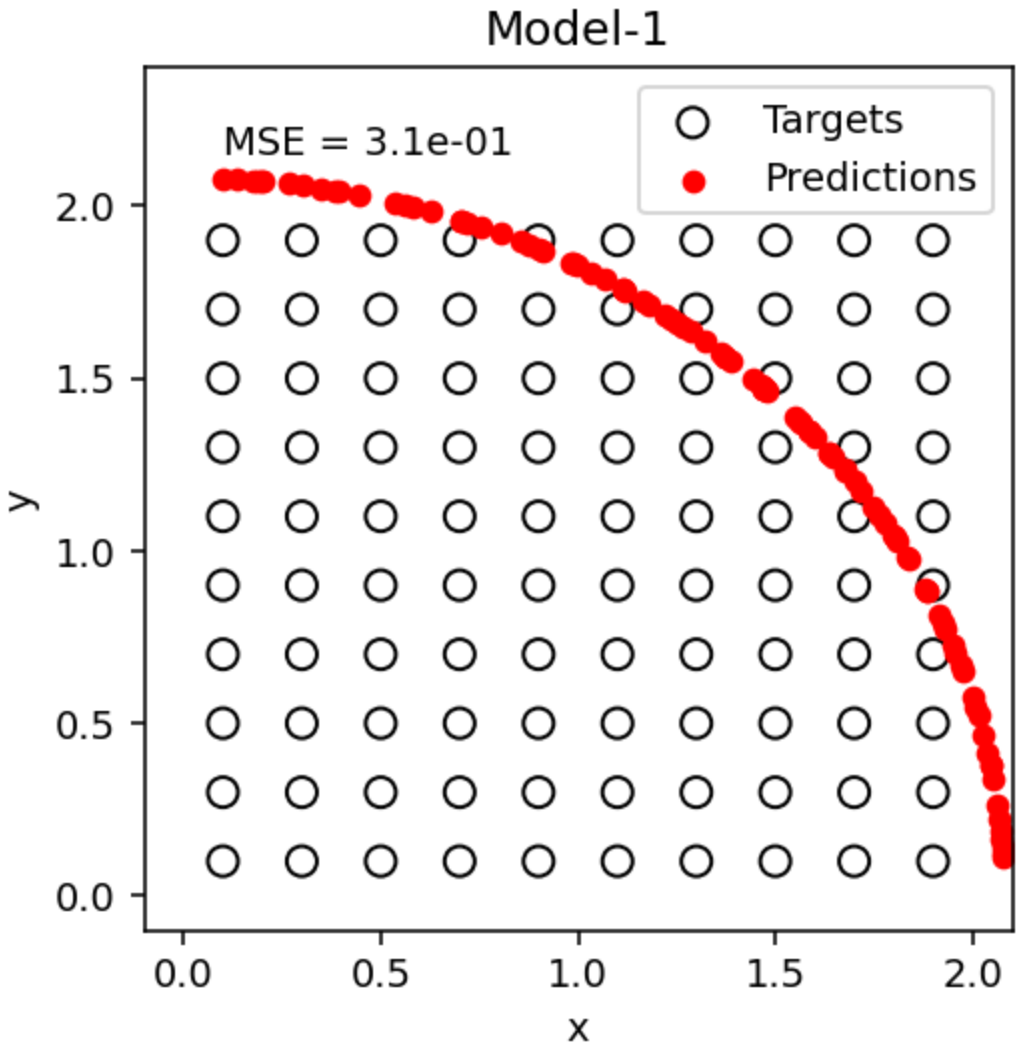


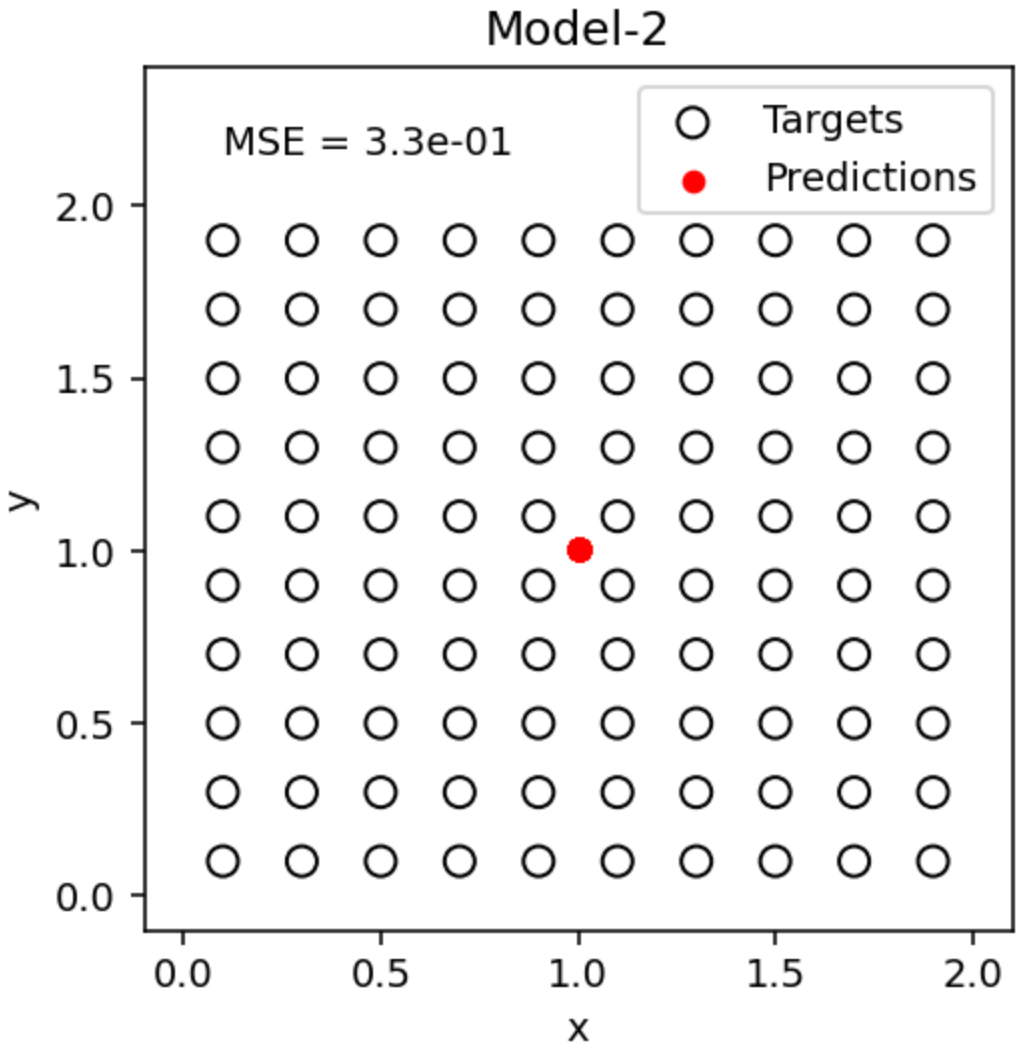
```
Epoch 0 train loss = 3.9232420921325684 val loss = 2.88222336769104
Epoch 100 train loss = 0.3487170338630676 val loss = 0.342442125082016
Epoch 200 train loss = 0.3410458564758301 val loss = 0.3361835181713104
Epoch 300 train loss = 0.341045618057251 val loss = 0.3361751437187195
Epoch 400 train loss = 0.34104564785957336 val loss = 0.33617517352104187
Epoch 500 train loss = 0.3410455882549286 val loss = 0.33617517352104187
Epoch 600 train loss = 0.341045618057251 val loss = 0.3361751437187195
Epoch 700 train loss = 0.341045618057251 val loss = 0.3361751437187195
Epoch 800 train loss = 0.341045618057251 val loss = 0.3361751437187195
Epoch 900 train loss = 0.3410455882549286 val loss = 0.3361751437187195
```

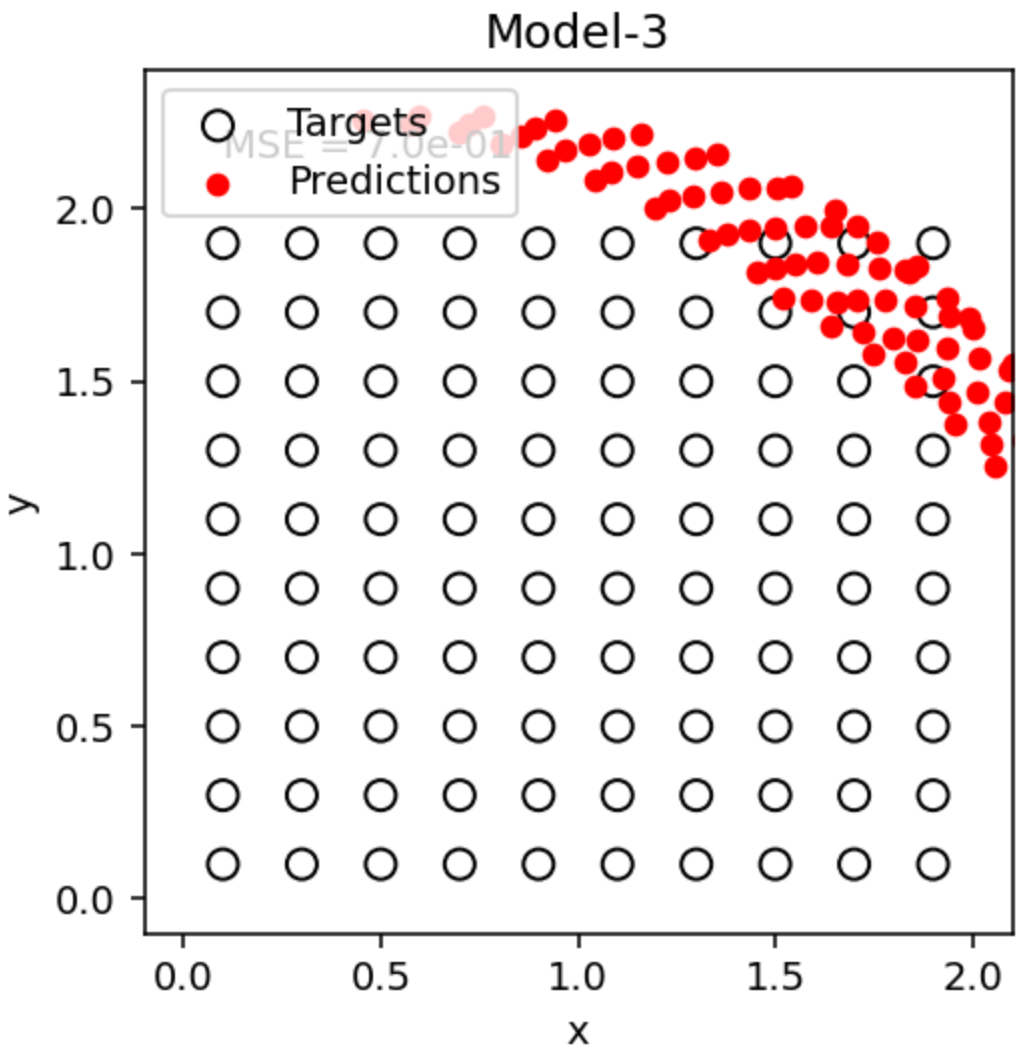



```
Epoch 0 train loss = 2.6542818546295166 val loss = 2.628788709640503
Epoch 100 train loss = 1.2724281549453735 val loss = 1.2905681133270264
Epoch 200 train loss = 0.9199756979942322 val loss = 0.9459047913551331
```


Training and Validation Losses
Train Loss
Validation Loss
MSE Loss
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0
50
100
150
200
250
300
Epoch


Model-1
MSE = 3.1e-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


Model-2
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



Prompts

Neither of these models should have great performance. Describe what went wrong in each case.

In [78]:
```
print("""Model 1 is configured with a high maximum angle, limiting the model's predictive range and potentially restr

Model 2's learning rate of 1 is quite high, which can lead to unstable training behavior, difficulties in convergence
```

```
Model 3's learning rate of 0.0001 is very low. While this can be useful for gradual learning, the limitation of 300 t
```

Model 1 is configured with a high maximum angle, limiting the model's predictive range and potentially restricting it s effectiveness across the entire range of possible outcomes.

Model 2's learning rate of 1 is quite high, which can lead to unstable training behavior, difficulties in convergenc e, and overshooting the optimal solutions in the parameter space.

Model 3's learning rate of 0.0001 is very low. While this can be useful for gradual learning, the limitation of 300 t raining epochs might result in too slow a convergence, preventing the model from learning effectively within the give n number of iterations.