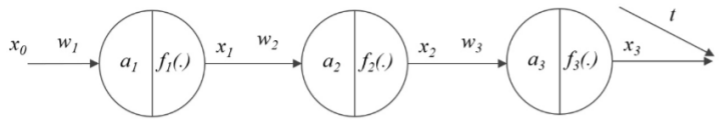
24-787: Machine Learning and Artificial Intelligence for Engineers
Ryan Wu
ID: weihuanw
Homework 8
Due: Mar 23, 2024

Concept Questions:

Problem 1

Given: $x_0=2$, $w_1=-1$, $w_2=3$, $w_3=7$, linear (identity) activation function, $L=\frac{1}{2}e^Te$, $t=-40$



Find: $\frac{\partial L}{\partial w_3}$, $\frac{\partial L}{\partial w_2}$, $\frac{\partial L}{\partial w_1}$

Equations: $\frac{\partial L}{\partial w_3}=\delta_3 x_2$, where $\delta_3=-(t-x_3)f_3'(a_3)$, $a_3=w_3x_2$

$\frac{\partial L}{\partial w_2}=\delta_2 x_1$, where $\delta_2=\delta_3 w_3 f_2'(a_2)$, $a_2=w_2x_1$

$\frac{\partial L}{\partial w_1}=\delta_1 x_0$, where $\delta_1=\delta_2 w_2 f_1'(a_1)$, $a_1=w_1x_0$

$x_3=x_0\cdot w_1\cdot w_2\cdot w_3$, $x_2=x_0\cdot w_1\cdot w_2$, $x_1=x_0\cdot w_1$

Solutions:

$x_1=x_0\cdot w_1 \rightarrow x_1=2(-1) \rightarrow x_1=-2$

$x_2=x_0\cdot w_1\cdot w_2 \rightarrow x_2=2(-1)(3) \rightarrow x_2=-6$

$x_3=x_0\cdot w_1\cdot w_2\cdot w_3 \rightarrow x_3=2(-1)(3)(7) \rightarrow x_3=-42$

$a_1=w_1x_0 \rightarrow a_1=(-1)(2) \rightarrow a_1=-2$

$a_2=w_2x_1 \rightarrow a_2=3(-2) \rightarrow a_2=-6$

$a_3=w_3x_2 \rightarrow a_3=7(-6) \rightarrow a_3=-42$

$(\frac{\partial L}{\partial w_3})$

$\frac{\partial L}{\partial w_3}=\delta_3 x_2 \rightarrow \frac{\partial L}{\partial w_3}=-(t-x_3)f_3'(a_3)x_2 \rightarrow \frac{\partial L}{\partial w_3}=-(-40-(-42))(1)(-6)=\frac{\partial L}{\partial w_3}=12$

$\frac{\partial L}{\partial w_2}=\delta_2 x_1 \rightarrow \frac{\partial L}{\partial w_2}=\delta_3 w_3 f_2'(a_2) x_1 \rightarrow \frac{\partial L}{\partial w_2}=(-2)(7)(1)(-2) \rightarrow \frac{\partial L}{\partial w_2}=28$

$\frac{\partial L}{\partial w_1}=\delta_1 x_0 \rightarrow \frac{\partial L}{\partial w_1}=\delta_2 w_2 f_1'(a_1) x_0 \rightarrow \frac{\partial L}{\partial w_1}=(-14)(3)(1)(2) \rightarrow \frac{\partial L}{\partial w_1}=-84$

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, $t = -2$

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

```
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:

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

# derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - 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

```
# YOUR CODE GOES HERE
# foward propagation

a1 = np.dot(W1, x0)
out1 = sigmoid(a1)
```

```
a2 = np.dot(W2, out1)
# out2 = sigmoid(a2)
y = a2

# print the output
print("The output of the network y: ", y)
```

```
The output of the network y:  [[-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 $\frac{\partial L}{\partial W_1}$.

```
# YOUR CODE GOES HERE
# backward propagation

# derivative of the loss function with respect to the output y
f_prime = 1

# compute the gradients (with respect to weights W1 and W2)
gamma_2 = -(t - y) * f_prime
d2 = np.dot(gamma_2, out1.T)
gamma_1 = np.dot(W2.T, gamma_2) * sigmoid_derivative(out1)
d1 = np.dot(gamma_1, x0.T)

# print the gradients
print("The gradient with respect to W2:\n ", d2)
print("The gradient with respect to W1:\n ", d1)
```

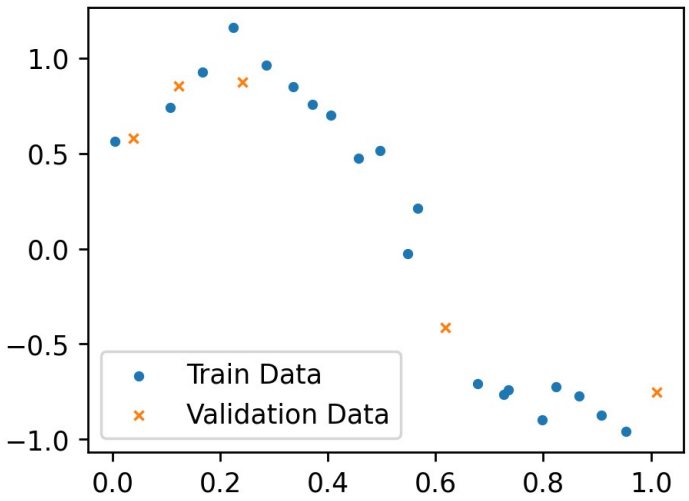
```
The gradient with respect to W2:
  [[8.21031503e-02 2.43316128e-24 1.04899215e-08]]
The gradient with respect to W1:
  [[ 1.59095673e+00  4.77287018e+00]
 [-9.73264513e-24 -2.91979354e-23]
 [-1.04899214e-07 -3.14697641e-07]]
```

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.

```
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="M
odel",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:

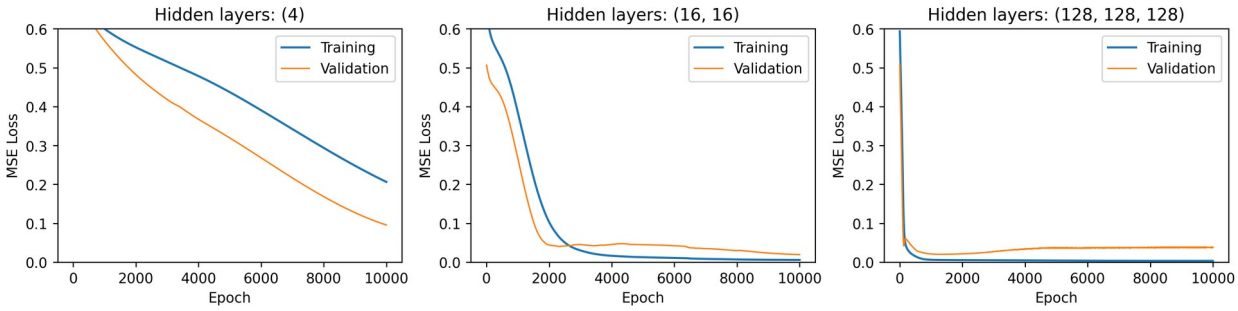
- `model1`: 1 hidden layer with 4 neurons. That is, the network should have a linear transformation from size 1 to size 4. Then a ReLU activation should be applied. Finally, a linear transformation from size 4 to size 1 gives the network output. (Note: Your regression network should not have an activation 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)

```
# YOUR CODE GOES HERE
# model1 (1 hidden layer with 4 neurons)
model1 = nn.Sequential(
    nn.Linear(1, 4),
    nn.ReLU(),
    nn.Linear(4, 1)
)

# model2 (2 hidden layers each with 16 neurons)
model2 = nn.Sequential(
    nn.Linear(1, 16),
    nn.ReLU(),
    nn.Linear(16, 16),
    nn.ReLU(),
    nn.Linear(16, 1)
)

# model3 (3 hidden layers each with 128 neurons)
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.

```
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).

```
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:.4f}")
```

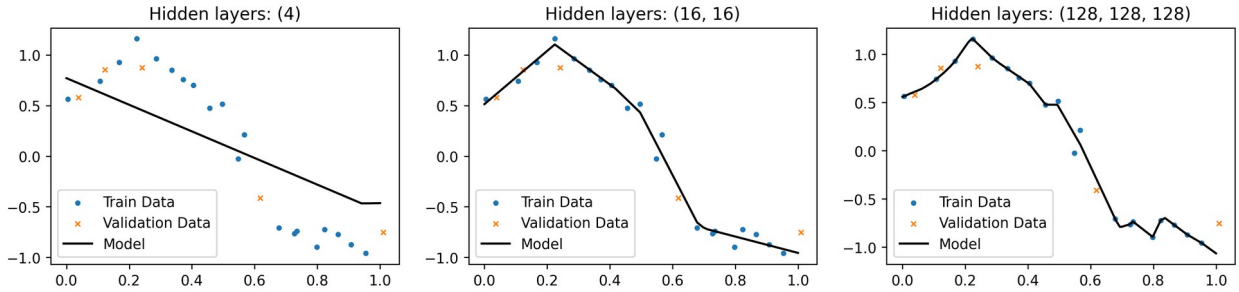
```
Model 1, hidden layers             (4):   Train MSE: 0.2061    Test
MSE: 0.0954
Model 2, hidden layers        (16, 16):   Train MSE: 0.0057    Test
MSE: 0.0192
Model 3, hidden layers (128, 128, 128):   Train MSE: 0.0031    Test
MSE: 0.0375
```

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.

```
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.

   Model 3 (3 hidden layers each with 128 neurons) is the model that overfits the most. We can observe that its loss curves converge the fastest during training. The loss curve for train data also converges to a lower MSE compared to the loss curve for validation data. The loss curve for the train data is also very close to zero.

2. For the model that underfits the most, describe what happens to the loss curves while training.

   Model 1 (1 hidden layer with 4 neurons) is the model that underfits the most. We can observe that its loss curves converge the slowest during training. The loss curve for train data never converges to a smaller MSE compared to the loss curve for validation data. Furthermore, both MSE loss for training and validation data are relatively large.

3. For the "best" model, what happens to the loss curves while training?

   Model 2 (2 hidden layers each with 16 neurons) is the "best" model in our use case. The loss curves converge around the 2000 epoch, which is between Model 1 and Model 3. The loss curve for train data also converges to a lower MSE compared to the loss curve for validation data.

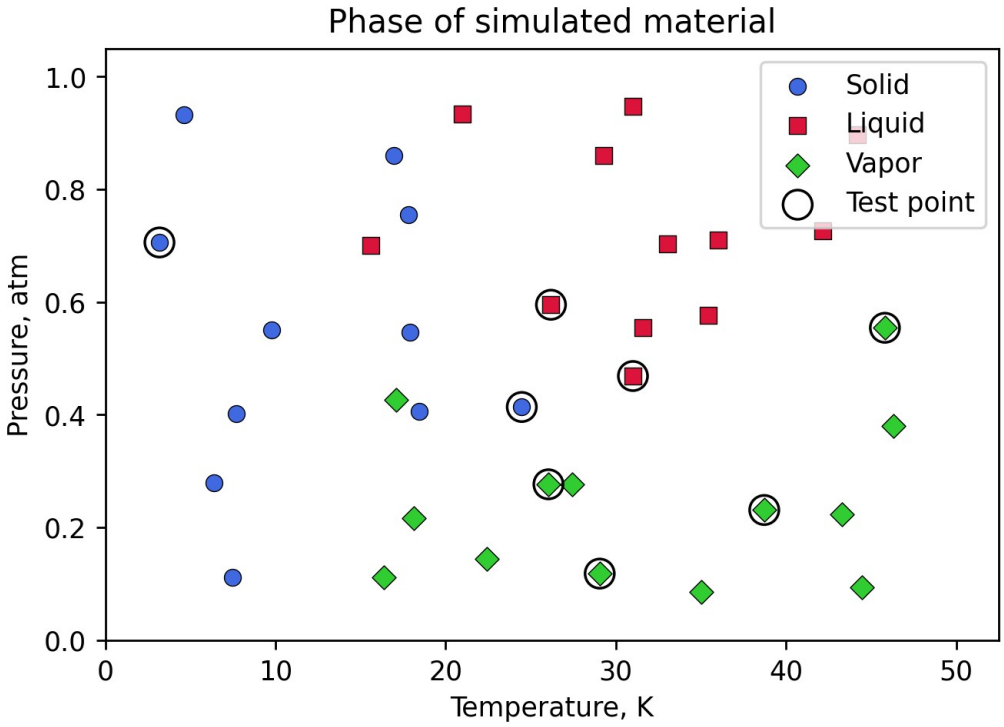
M8-L2 Problem 2

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.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import torch
from torch import nn, optim
import torch.nn.functional as F

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.
04883182996897,35.03654799338904,44.45894113066656,6.375872112626925,1
8.117730007820796,26.036627605010292,27.434415188257777,38.71725038082
664,43.28894919752904,7.680445610939323,18.45596638292661,17.110360581
978867,24.47129299701541,31.002183974403255,46.32619845547938,9.781567
509498505,17.90012148246819,26.186183422327638,31.59158564216724,35.41
479362252932,45.805291762864556,3.182744258689332,15.599210213275237,1
7.833532874090462,33.04668917049584,36.018483217500716,42.146619399905
234,4.64555612104627,16.942336894342166,20.961503322165484,29.28433948
8686488,30.98789800436355,44.17635497075877,])
x2 =
np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0
.11818202991034835,0.0859507900573786,0.09370319537993416,0.2797631195
927265,0.216022547162927,0.27667667154456677,0.27706378696181594,0.231
0382561073841,0.22289262976548535,0.40154283509241845,0.40637107709426
```

```
23,0.427019677041788,0.41386015134623205,0.46883738380592266,0.3802044
8107480287,0.5508876756094834,0.5461309517884996,0.5953108325465398,0.
5553291602539782,0.5766310772856306,0.5544425592001603,0.7058969583645
52,0.7010375141164304,0.7556329589465274,0.7038182951348614,0.70965823
61680054,0.7268725170660963,0.9320993229847936,0.8597101275793062,0.93
37944907498804,0.8596098407893963,0.9476459465013396,0.896865120164770
2,])
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="limegreen")]
    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()`.

```
class PhaseNet(nn.Module):
    def __init__(self):
        super().__init__()
        # YOUR CODE GOES HERE
        # 4 layers, 2 input, 3 output
        # 3 hidden layers with 10 neurons each
        self.lin1 = nn.Linear(2,10)
        self.lin2 = nn.Linear(10,10)
        self.lin3 = nn.Linear(10,10)
        self.lin4 = nn.Linear(10,3)
        self.act = nn.ReLU()

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

    def forward(self, X):
        # YOUR CODE GOES HERE
        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)
        x = self.act(x) # Activation at third hidden layer
        x = self.lin4(x)
        x = F.softmax(x, dim=1)  # Activation at fourth hidden layer
(softmax for classification problem)
        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.

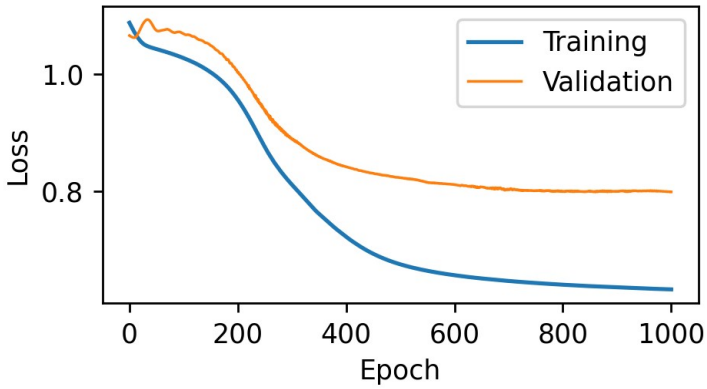
Make sure the training curves you get are reasonable.

```
model = PhaseNet()

lr = 0.001
epochs = 1000

# Define loss function
# YOUR CODE GOES HERE
lossfun = nn.CrossEntropyLoss() # Cross-entropy loss for mutliclass
classification

# Define an optimizer, `opt`
```

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


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.item():.4f}")


plot_loss(train_hist, val_hist)
```

```
Epoch    0 of 1000:   Train Loss = 1.0877   Validation Loss = 1.0659
Epoch   40 of 1000:   Train Loss = 1.0455   Validation Loss = 1.0879
Epoch   80 of 1000:   Train Loss = 1.0342   Validation Loss = 1.0716
Epoch  120 of 1000:   Train Loss = 1.0193   Validation Loss = 1.0638
Epoch  160 of 1000:   Train Loss = 0.9964   Validation Loss = 1.0414
Epoch  200 of 1000:   Train Loss = 0.9564   Validation Loss = 1.0027
Epoch  240 of 1000:   Train Loss = 0.8927   Validation Loss = 0.9515
Epoch  280 of 1000:   Train Loss = 0.8333   Validation Loss = 0.9036
Epoch  320 of 1000:   Train Loss = 0.7909   Validation Loss = 0.8768
Epoch  360 of 1000:   Train Loss = 0.7531   Validation Loss = 0.8552
Epoch  400 of 1000:   Train Loss = 0.7224   Validation Loss = 0.8420
Epoch  440 of 1000:   Train Loss = 0.6982   Validation Loss = 0.8326
Epoch  480 of 1000:   Train Loss = 0.6815   Validation Loss = 0.8260
Epoch  520 of 1000:   Train Loss = 0.6704   Validation Loss = 0.8208
Epoch  560 of 1000:   Train Loss = 0.6626   Validation Loss = 0.8143
Epoch  600 of 1000:   Train Loss = 0.6568   Validation Loss = 0.8114
Epoch  640 of 1000:   Train Loss = 0.6523   Validation Loss = 0.8073
Epoch  680 of 1000:   Train Loss = 0.6486   Validation Loss = 0.8040
Epoch  720 of 1000:   Train Loss = 0.6455   Validation Loss = 0.8023
Epoch  760 of 1000:   Train Loss = 0.6429   Validation Loss = 0.8005
Epoch  800 of 1000:   Train Loss = 0.6407   Validation Loss = 0.7998
Epoch  840 of 1000:   Train Loss = 0.6386   Validation Loss = 0.7991
Epoch  880 of 1000:   Train Loss = 0.6369   Validation Loss = 0.8007
Epoch  920 of 1000:   Train Loss = 0.6353   Validation Loss = 0.7993
```

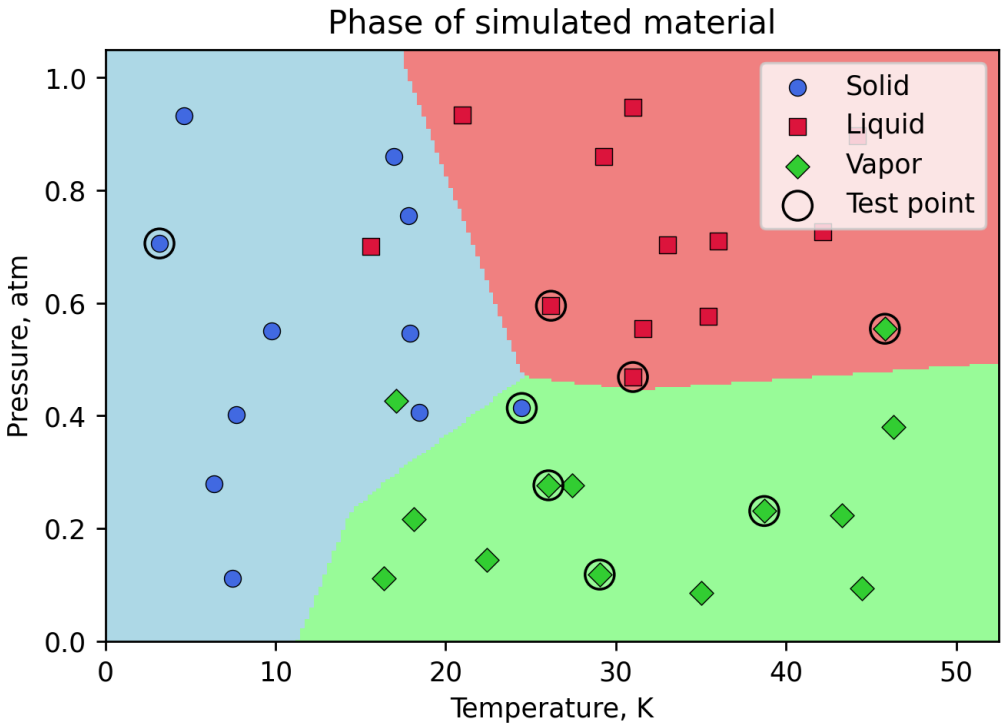
```
Epoch  960 of 1000:   Train Loss = 0.6339   Validation Loss = 0.8004
Epoch 1000 of 1000:   Train Loss = 0.6326   Validation Loss = 0.7991
```



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.

```
plot_data(newfig=True)
plot_model(model)
plt.show()
```



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

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

import torch
from torch import nn, optim
import torch.nn.functional as F

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+theta3)
        y = self.L1*torch.sin(theta1) +
self.L2*torch.sin(theta1+theta2) +
self.L3*torch.sin(theta1+theta2+theta3)
        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",markeredgecolor="black")
    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.

```
%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,continuous_update=True,orientation
='horizontal',readout=False,layout = Layout(width='550px'))
slider2 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75,
step=np.pi/100,
description='theta2',disabled=False,continuous_update=True,orientation
='horizontal',readout=False,layout = Layout(width='550px'))
slider3 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75,
step=np.pi/100,
description='theta3',disabled=False,continuous_update=True,orientation
='horizontal',readout=False,layout = Layout(width='550px'))

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

interactive_plot
```


```
{"model_id":"9cf59459b0784b408f348a890c4418b5","version_major":2,"vers
ion_minor":0}
```


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.

```
class InverseArm(nn.Module):
    def __init__(self, hidden_layer_sizes=[24,24], max_angle=None):
        super().__init__()
        # YOUR CODE GOES HERE
        # 2 inputs, 3 outputs
        # 2 hidden layers, each with 24 neurons
        # ReLU activation at end of each hidden layer
        # max_angle clamp values with tanh function
        self.hidden_layers = nn.ModuleList()
        self.max_angle = max_angle

        input_size = 2
        output_size = 3

        for size in hidden_layer_sizes:
            self.hidden_layers.append(nn.Linear(input_size, size))
            input_size = size
        self.output_layer = nn.Linear(input_size, output_size)
        # self.output_layer = nn.Linear(hidden_layer_sizes[-1], 3)


    def forward(self, xy):
        # YOUR CODE GOES HERE
        x = xy
        for layer in self.hidden_layers:
            x = F.relu(layer(x))
        x = self.output_layer(x)

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

        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)

```
# YOUR CODE GOES HERE
# generate dataset with x-y coordinates (100 by 100 meshgrid, from 0
to 2)
# randomly split 80% of points are in X_train, 20% in X_val

# set up 100 by 100 meshgrid
x = np.linspace(0, 2, 100)
y = np.linspace(0, 2, 100)
x, y = np.meshgrid(x, y)
# flatten the meshgrid
xy = np.vstack([x.flatten(), y.flatten()]).T

# randomly split the data into 80 and 20
np.random.shuffle(xy)
split_data = int(0.8 * len(xy))
X_train, X_val = xy[:split_data], xy[split_data:]

# convert to tensor
X_train = torch.tensor(X_train, dtype=torch.float)
X_val = torch.tensor(X_val, dtype=torch.float)

# print for testing
# print ("xy shape:", xy.shape)
# print("X_train shape:", X_train.shape)
# print("X_val shape:", X_val.shape)
```

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))`.

```
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
    opt = optim.Adam(model.parameters(), lr=lr)
    sch = ExponentialLR(opt, gamma)

    # loss_fn = F.mse_loss()

    train_loss = []
    val_loss = []

    # training loop
    for epoch in range(epochs):
        model.train()
        opt.zero_grad()
        train_output = model(X_train)
        fwd = ForwardArm()
        train_pred = fwd(train_output)
        # loss = loss_fn(train_output, train_pred)
        # loss = F.mse_loss(train_pred, X_train)
        loss = F.mse_loss(X_train, train_pred)
        train_loss.append(loss.item())
        loss.backward()
        opt.step()

        # validation loop
        model.eval()
        with torch.no_grad():
            val_output= model(X_val)
            val_pred = fwd(val_output)
            # val_loss.append(F.mse_loss(val_pred, X_val).item())
            val_loss.append(F.mse_loss( X_val, val_pred).item())

        sch.step()

        # print loss progress info 25 times during training
        # if epoch % int(epochs//25) == 0:
        #     print(f"Epoch {epoch} of {epochs}... \tAverage loss:
{loss.item()}")

        train_loss.append(loss.item())
```

```
    # plot the loss curves
    if create_plot:
        plt.figure(dpi=250)
        plt.plot(train_loss, label="Training Loss")
        plt.plot(val_loss, label="Validation Loss")
        plt.xlabel('Epoch')
        plt.ylabel('Loss (MSE)')
        plt.title(f"Loss Curves for Model with
{len(model.hidden_layers)} hidden layers")
        plt.legend()
        plt.show()

    return
```

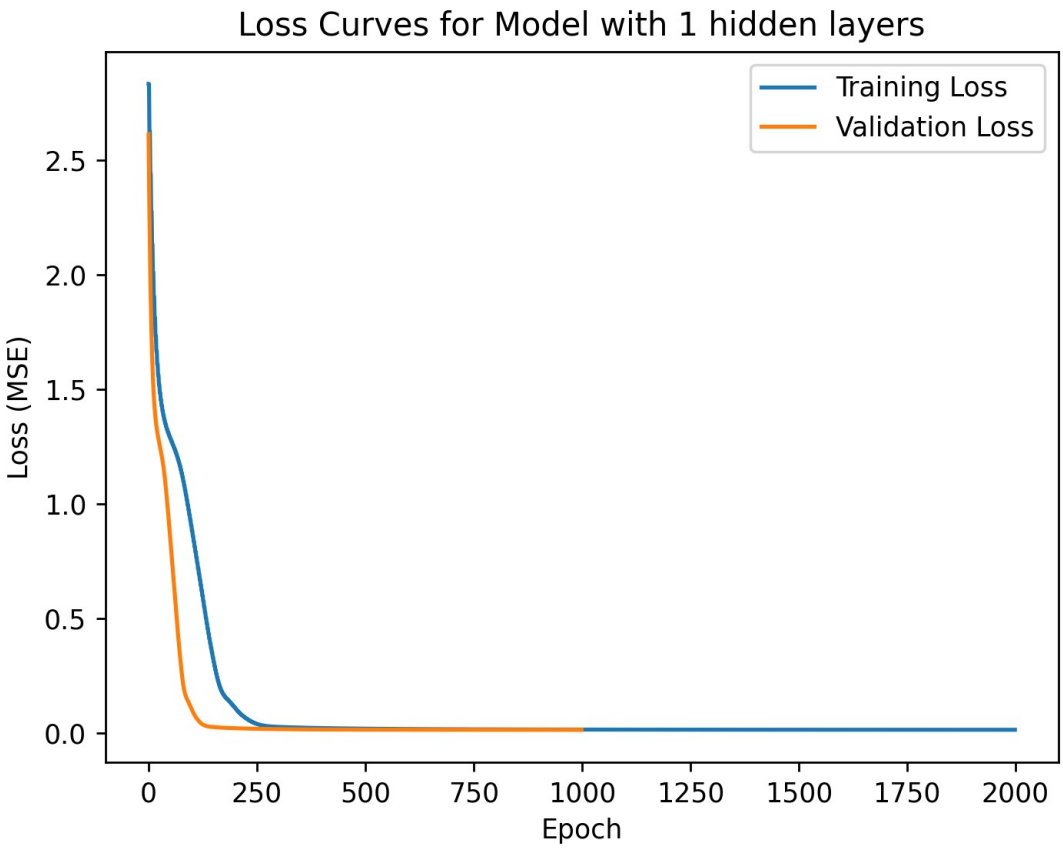
Training a model

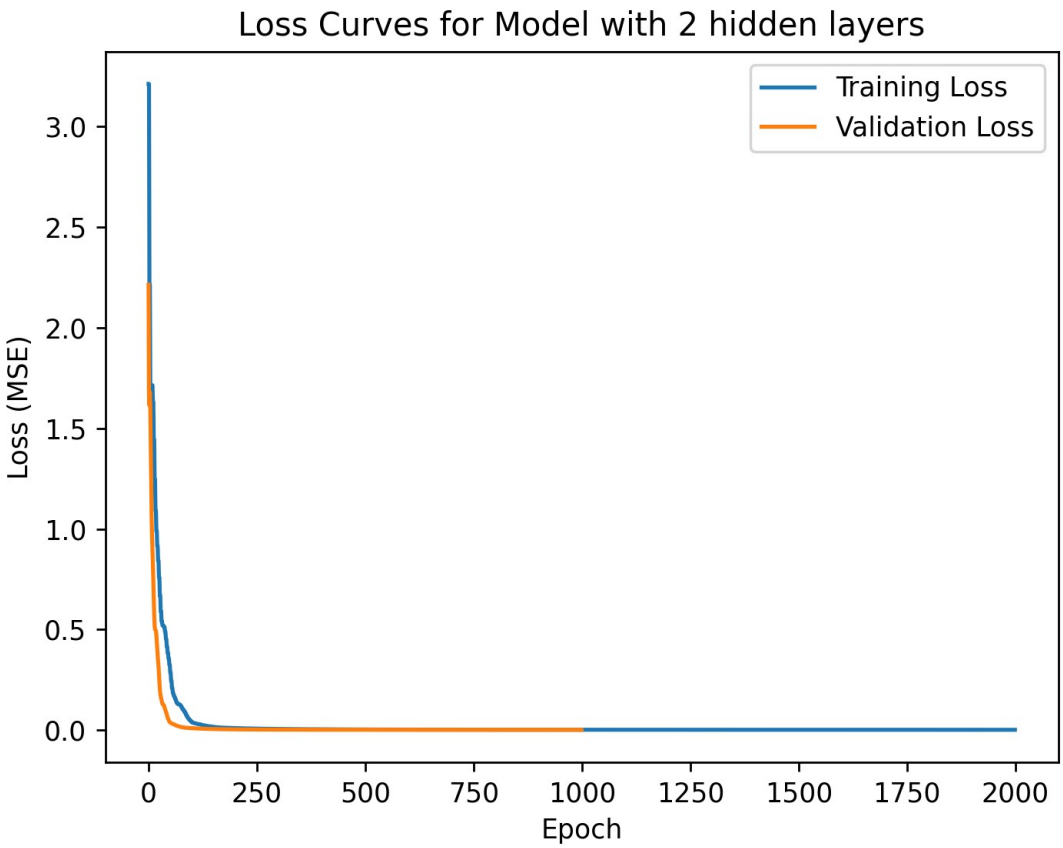
Create 3 models of different complexities (with `max_angle=None`):

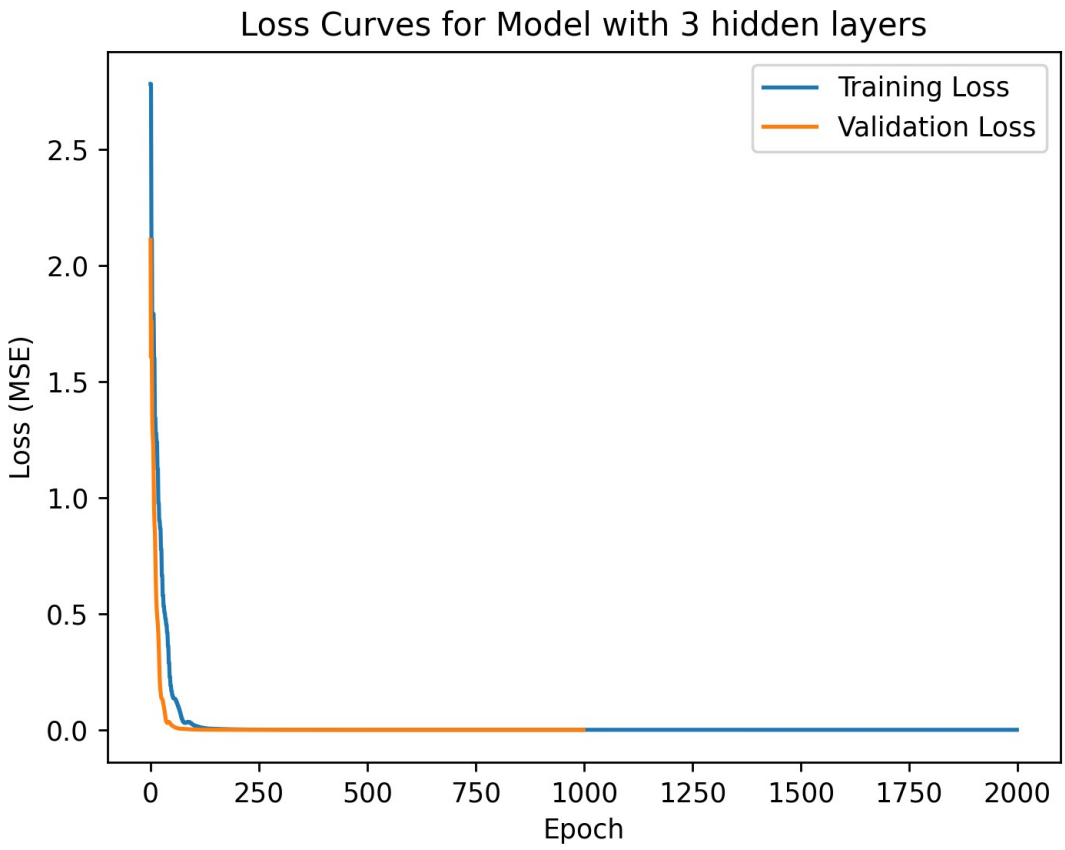
- `hidden_layer_sizes=[12]`
- `hidden_layer_sizes=[24,24]`
- `hidden_layer_sizes=[48,48,48]`

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

```
# YOUR CODE GOES HERE
# 1000 epochs, learning rate 0.01, gamma 0.995, and show plot

# model 1 (hidden_layer_sizes=[12], max_angle=None)
model1 = InverseArm(hidden_layer_sizes=[12], max_angle=None)
train(model1, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995,
create_plot=True)

# model 2 (hidden_layer_sizes=[24,24], max_angle=None)
model2 = InverseArm(hidden_layer_sizes=[24,24], max_angle=None)
train(model2, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995,
create_plot=True)

# model 3 (hidden_layer_sizes=[48, 48, 48], max_angle=None)
model3 = InverseArm(hidden_layer_sizes=[48,48,48], max_angle=None)
train(model3, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995,
create_plot=True)
```


Loss Curves for Model with 1 hidden layers
Training Loss
Validation Loss
Loss (MSE)
2.5
2.0
1.5
1.0
0.5
0.0
0
250
500
750
1000
1250
1500
1750
2000
Epoch

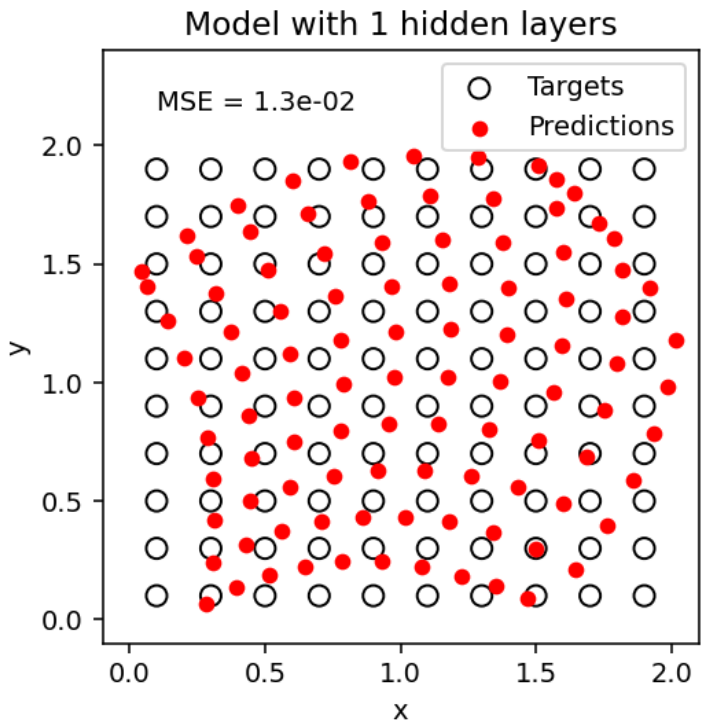

Loss Curves for Model with 2 hidden layers
Training Loss
Validation Loss
Loss (MSE)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
250
500
750
1000
1250
1500
1750
2000
Epoch

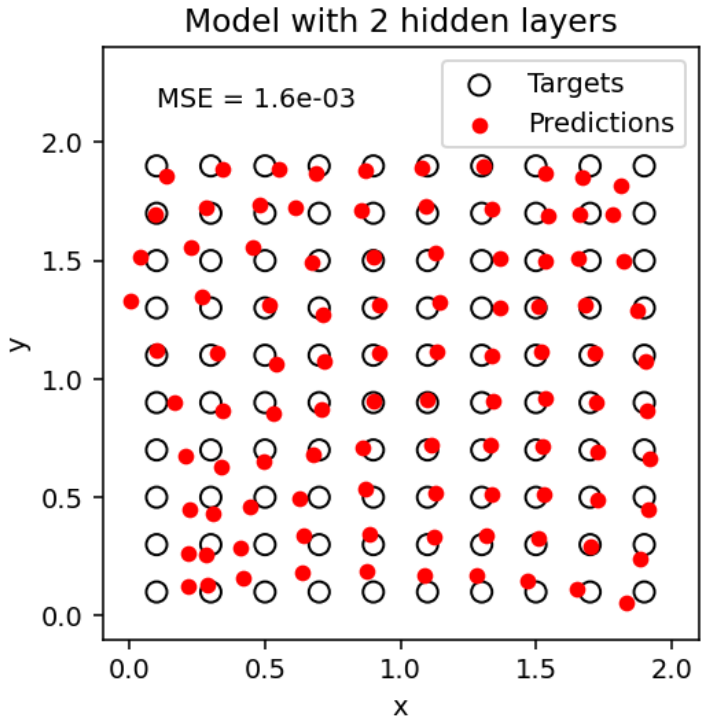


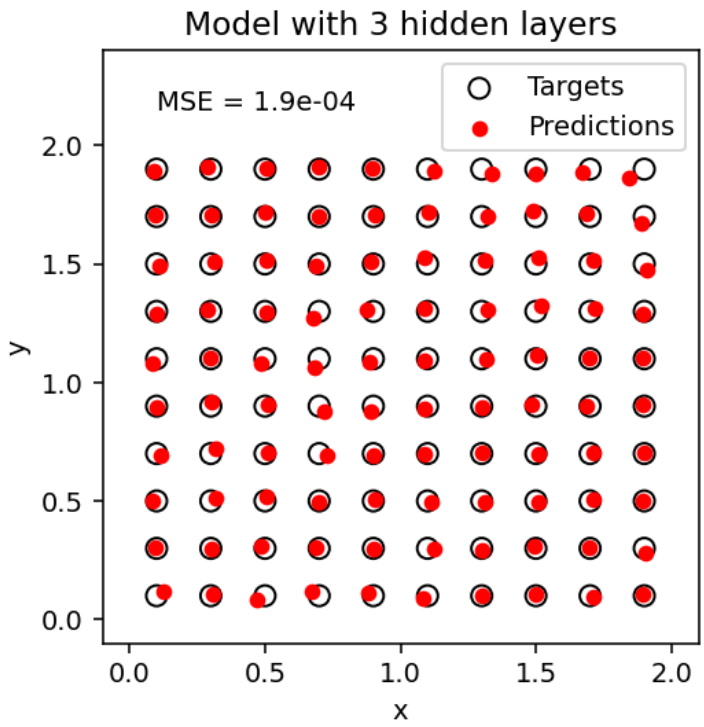
Visualizations

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.

```
# YOUR CODE GOES HERE
# plot the predictions for each model
for model in [model1, model2, model3]:
    plot_predictions(model, title=f"Model with
{len(model.hidden_layers)} hidden layers")
```


Model with 1 hidden layers
MSE = 1.3e-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


Model with 2 hidden layers
MSE = 1.6e-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`.)

```
%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 =
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,
orientation='horizontal', readout=False, layout =
Layout(width='550px'))
slider2 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100,
description='y', disabled=False, continuous_update=True,
orientation='horizontal', readout=False, layout =
Layout(width='550px'))
```

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

interactive_plot
```

{"model_id":"c6a7e1d9618c425f91d02a79158e618b","version_major":2,"version_minor":0}

Training more neural networks

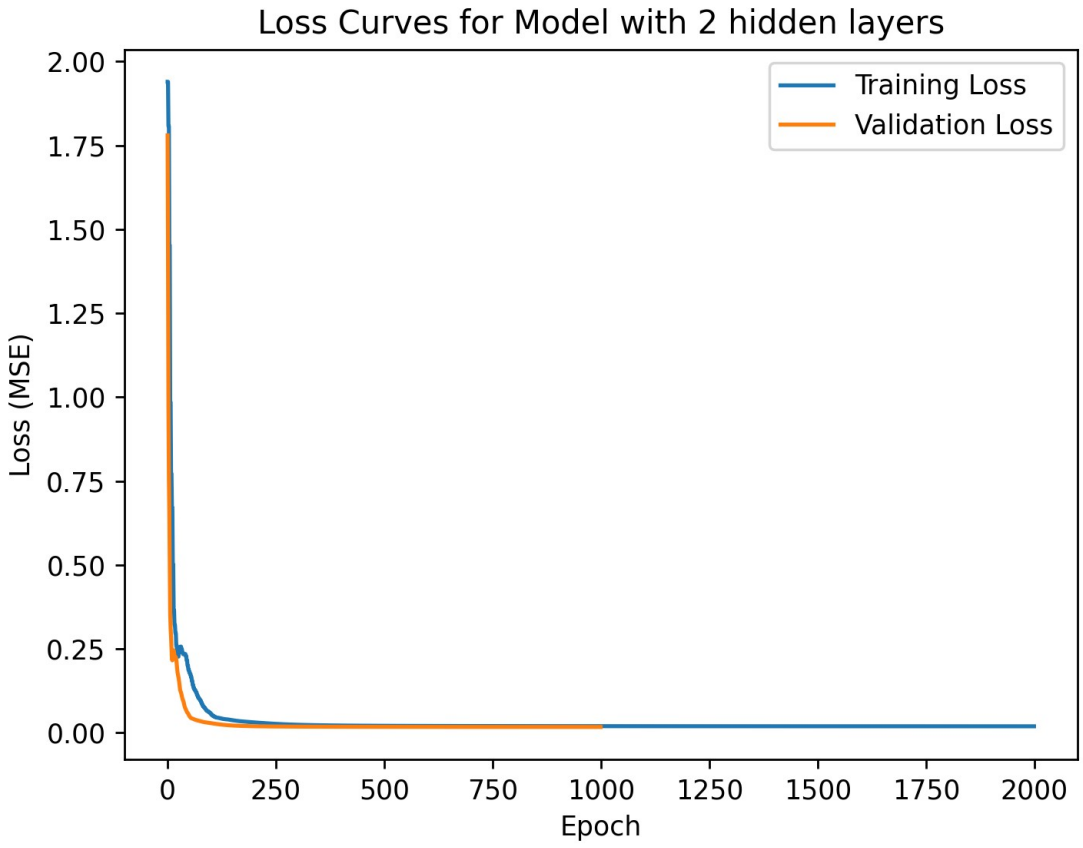
Now train more networks with the following details:

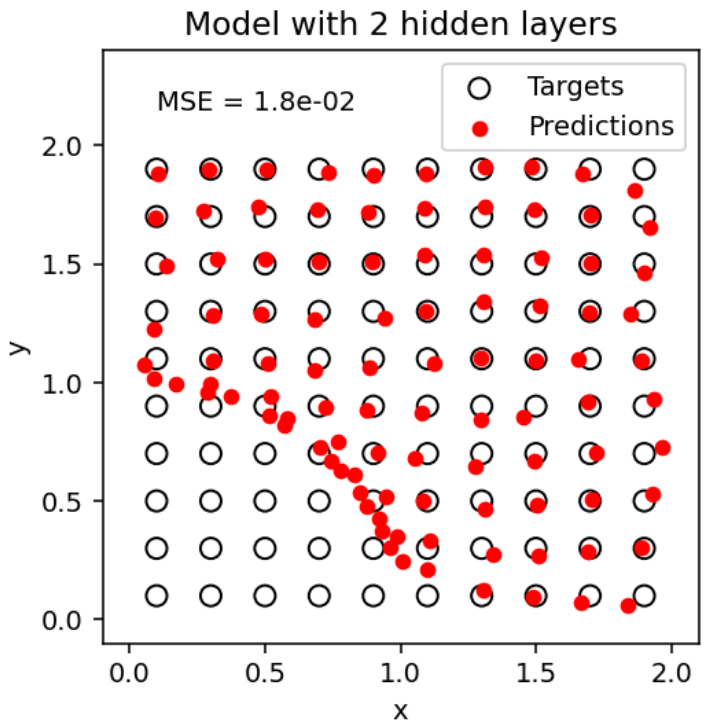
1. `hidden_layer_sizes=[48,48], max_angle=torch.pi/2`, train with `lr=0.01, epochs=1000, gamma=.995`
2. `hidden_layer_sizes=[48,48], max_angle=None`, train with `lr=1, epochs=1000, gamma=1`
3. `hidden_layer_sizes=[48,48], max_angle=2`, train with `lr=0.0001, epochs=300, gamma=1`

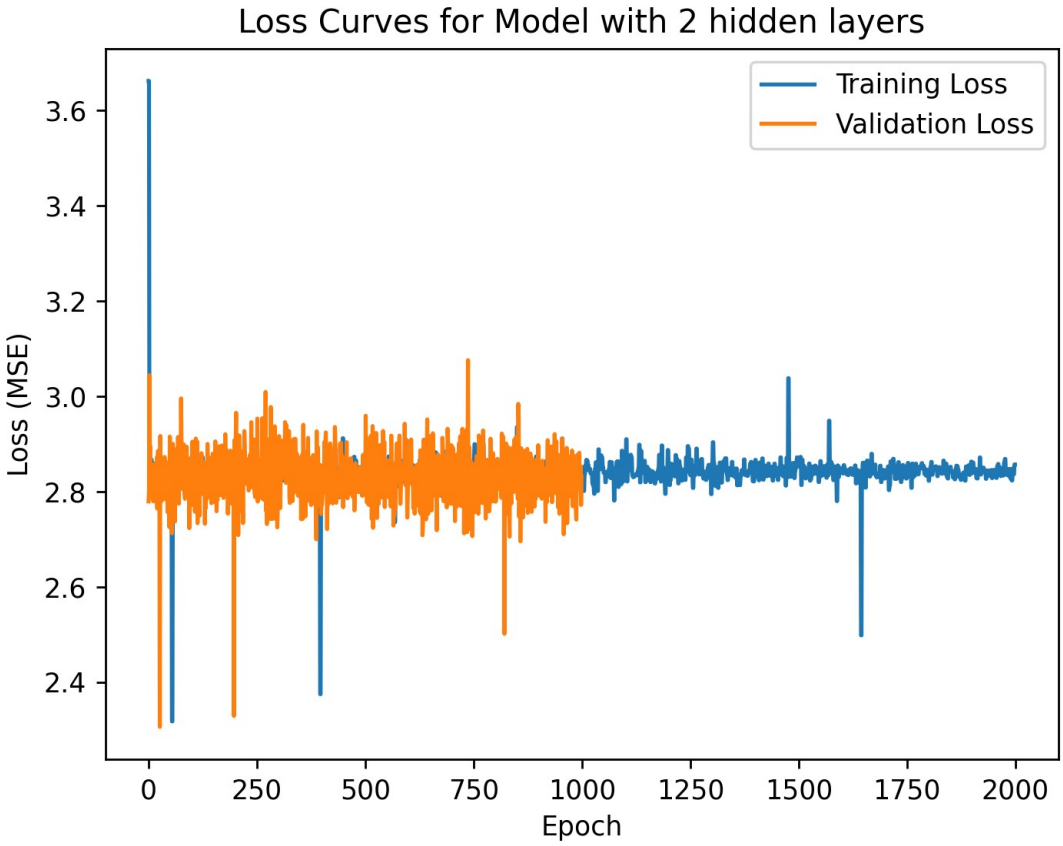
For each network, show a loss curve plot and a `plot_predictions` plot.

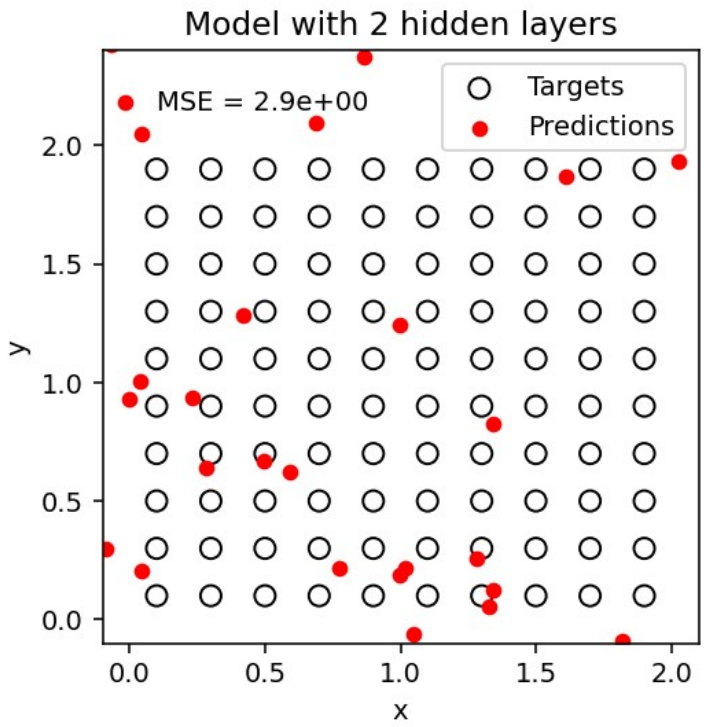
```
# YOUR CODE GOES HERE
# model 4 (hidden_layer_sizes=[48,48], max_angle=touch.pi/2, lr=0.01,
epochs=1000, gamma=0.995)
model4 = InverseArm(hidden_layer_sizes=[48,48],
max_angle=torch.tensor(np.pi/2))
train(model4, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995,
create_plot=True)
plot_predictions(model4, title=f"Model with
{len(model4.hidden_layers)} hidden layers")

# model 5 (hidden_layer_sizes=[48,48], max_angle=None, lr=1,
epochs=1000, gamma=1)
model5 = InverseArm(hidden_layer_sizes=[48,48], max_angle=None)
train(model5, X_train, X_val, lr=1, epochs=1000, gamma=1,
create_plot=True)
plot_predictions(model5, title=f"Model with
{len(model5.hidden_layers)} hidden layers")

# model 6 (hidden_layer_sizes=[48,48], max_angle=2, lr=0.0001,
epochs=300, gamma=1)
model6 = InverseArm(hidden_layer_sizes=[48,48], max_angle=2)
train(model6, X_train, X_val, lr=0.0001, epochs=300, gamma=1,
create_plot=True)
plot_predictions(model6, title=f"Model with
{len(model6.hidden_layers)} hidden layers")
```

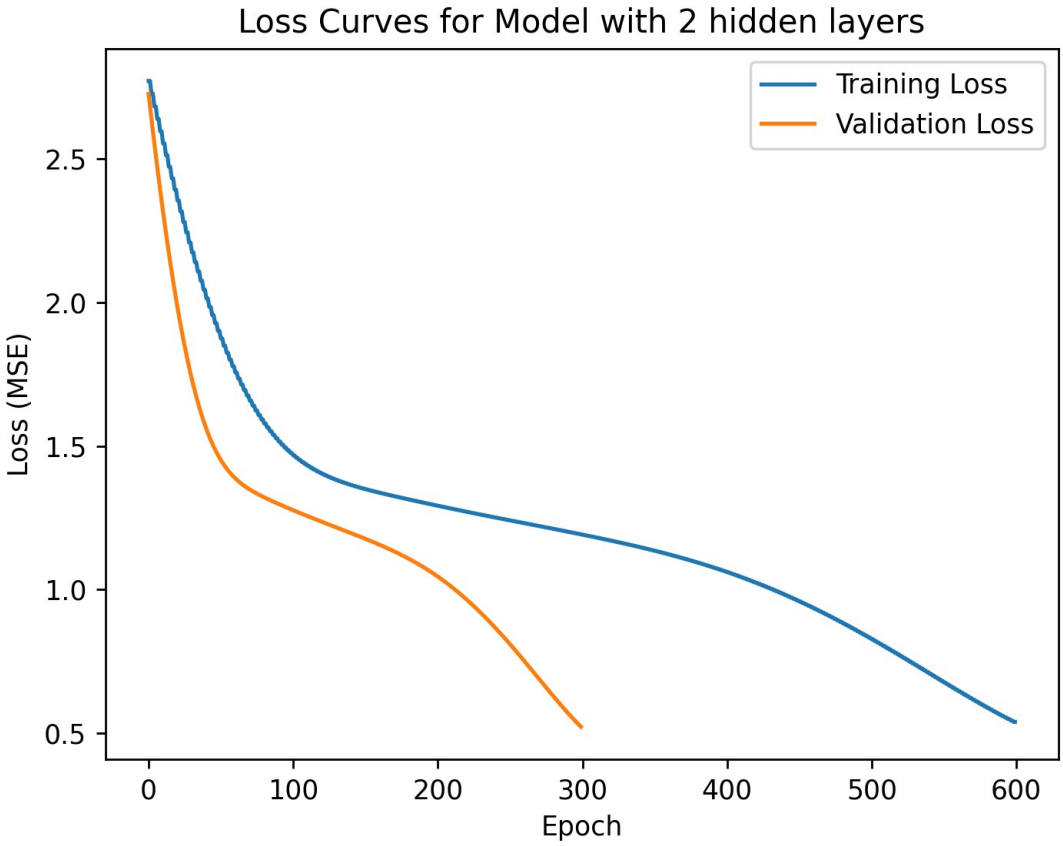
```
# # show the loss curves and predictions for each model
# for model in [model4, model5, model6]:
#     plot_predictions(model, title= f"Model with
{len(model.hidden_layers)} hidden layers")
```

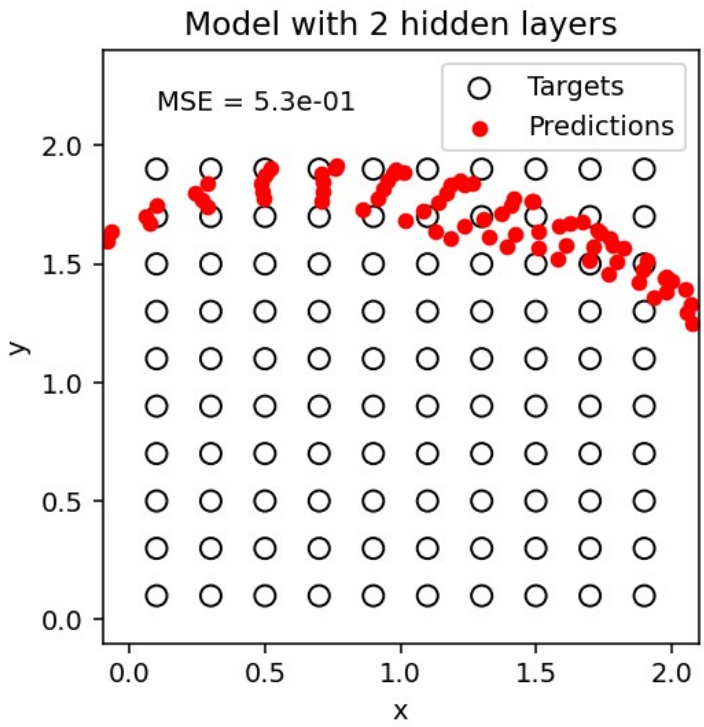



Model with 2 hidden layers
MSE = 1.8e-02
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x


Loss Curves for Model with 2 hidden layers
Training Loss
Validation Loss
3.6
3.4
3.2
3.0
2.8
2.6
2.4
Loss (MSE)
0
250
500
750
1000
1250
1500
1750
2000
Epoch


Model with 2 hidden layers
MSE = 2.9e+00
Targets
Predictions
y
x
0.0
0.5
1.0
1.5
2.0


Loss Curves for Model with 2 hidden layers
Training Loss
Validation Loss
Loss (MSE)
Epoch
0.5
1.0
1.5
2.0
2.5
0
100
200
300
400
500
600



Prompts

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

Model 4 (hidden_layer_sizes=[48,48], max_angle=touch.pi/2, lr=0.01, epochs=1000, gamma=0.995)

```
The max_angle limits the output range of my training network to [-π/2, π/2]. This restriction in the range can result in incorrect prediction.
```

Model 5 (hidden_layer_sizes=[48,48], max_angle=None, lr=1, epochs=1000, gamma=1)

```
The learning rate in this case is too large. This can cause the optimizer to overshoot and instability during training. The above can all lead to incorrect predictions.
```

Model 6 (hidden_layer_sizes=[48,48], max_angle=2, lr=0.0001, epochs=300, gamma=1)

```
The number of epochs in this model is too small for proper training output. The small epochs can lead to incomplete optimization and underfitting of the data. The training results can be suboptimal.
```