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 [324…

```
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*
```

```
        y = self.L1*torch.sin(theta1) + self.L2*torch.sin(theta1+theta2) + self.L3*
        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",
    plt.scatter(preds[:,0], preds[:,1], s=25, c="red", marker="o", label="Predictio
    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="light
    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 [325…

```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, Float

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, des
slider2 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, des
slider3 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, des

interactive_plot = interactive(plot_unit_arm, theta1 = slider1, theta2 = slider2, t
output = interactive_plot.children[-1]
output.layout.height = '600px'

interactive_plot
```

Out[325…

```
interactive(children=(FloatSlider(value=0.0, description='theta1', layout=Layout(w
idth='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 [326…

```
import torch.nn.functional as F
class InverseArm(nn.Module):
    def __init__(self, hidden_layer_sizes=[24,24], max_angle=None):
        super().__init__()
        # YOUR CODE GOES HERE
        self.layers = nn.ModuleList()
```

```
        input_size = 2
        for hidden_size in hidden_layer_sizes:
            self.layers.append(nn.Linear(input_size, hidden_size))
            input_size = hidden_size
        self.layers.append(nn.Linear(hidden_layer_sizes[-1], 3))
        self.act1 = nn.ReLU()

        self.max_angle = max_angle


    def forward(self, xy):
        # YOUR CODE GOES HERE
        for layer in self.layers[:-1]:
            xy = F.relu(layer(xy))
        xy = self.layers[-1](xy)

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

        return xy
```

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 [327…

```
# YOUR CODE GOES HERE
from sklearn.model_selection import train_test_split
x = np.linspace(0, 2, 100)
y = np.linspace(0, 2, 100)
xgrid, ygrid = np.meshgrid(x, y)
data = np.column_stack([xgrid.ravel(), ygrid.ravel()])
X_train, X_val = train_test_split(data, test_size=0.2, random_state=42)

X_train = torch.tensor(X_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
```

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 [328…

```
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 train(model, X_train, X_val, lr = 0.01, epochs = 1000, gamma = 1, create_plot =
    # YOUR CODE GOES HERE
    fwd = ForwardArm()

    opt = optim.Adam(model.parameters(),lr=gamma*lr)
    lossfun = nn.MSELoss()
    train_hist = []
    val_hist = []

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

        model.eval()
        loss_val = lossfun(X_val, fwd(model(X_val)))
        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)

    if create_plot == True:
        plot_loss(train_hist,val_hist)
    return train_hist, val_hist
```

Training a model

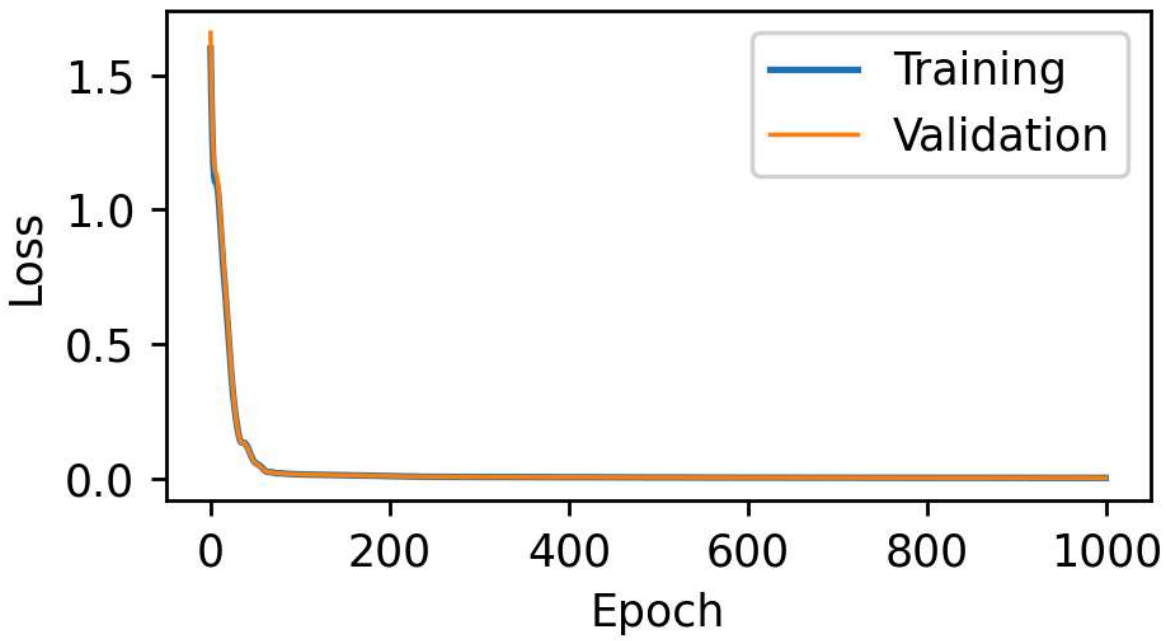
Create 3 models of different complexities (with `max_angle=None`):

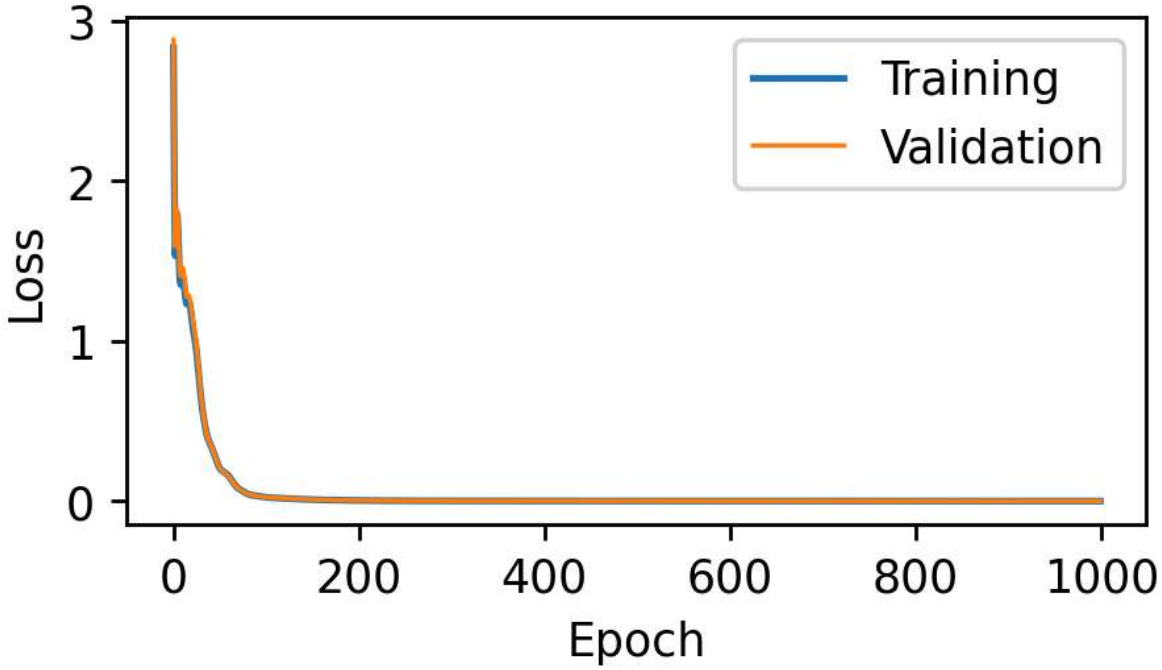
- `hidden_layer_sizes=[12]`
- `hidden_layer_sizes=[24,24]`
- `hidden_layer_sizes=[48,48,48]`

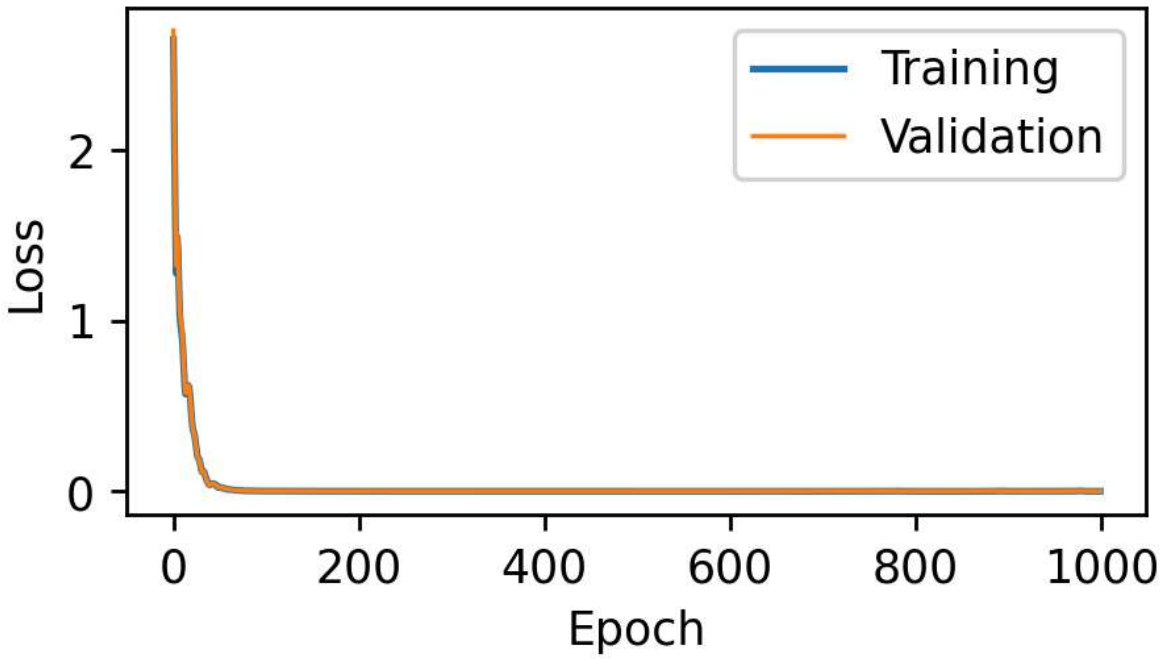
Train each model for 1000 epochs, learning rate 0.01, and gamma 0.995. Show the plot for each.

In [329…

```
# YOUR CODE GOES HERE
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()

hidden_layers_tag=["(12)","(24, 24)","(48, 48, 48)"]
hidden_layers=[[12],[24,24],[48,48,48]]

model1 = InverseArm(hidden_layer_sizes=hidden_layers[0])
model2 = InverseArm(hidden_layer_sizes=hidden_layers[1])
model3 = InverseArm(hidden_layer_sizes=hidden_layers[2])

for model in [model1, model2, model3]:
    loss_train, loss_val = train(model,X_train,X_val,gamma=0.995)
```





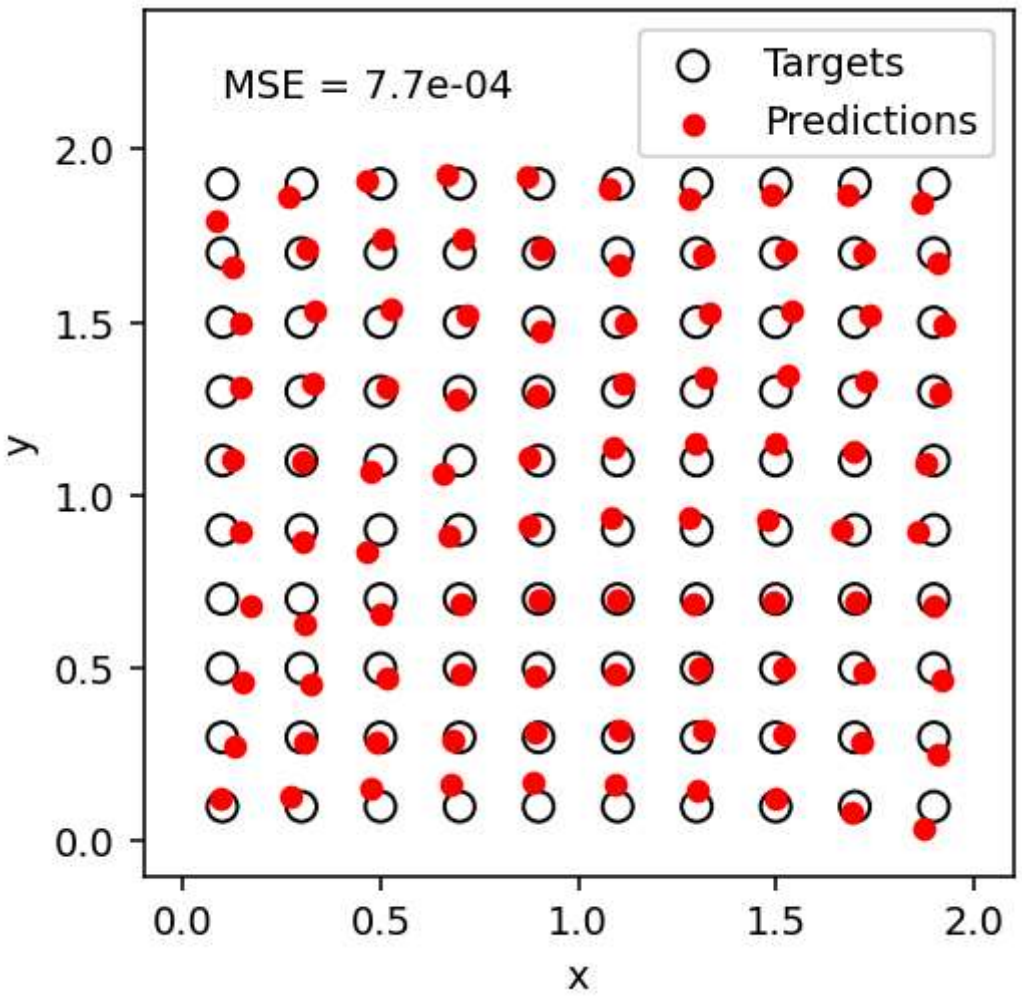


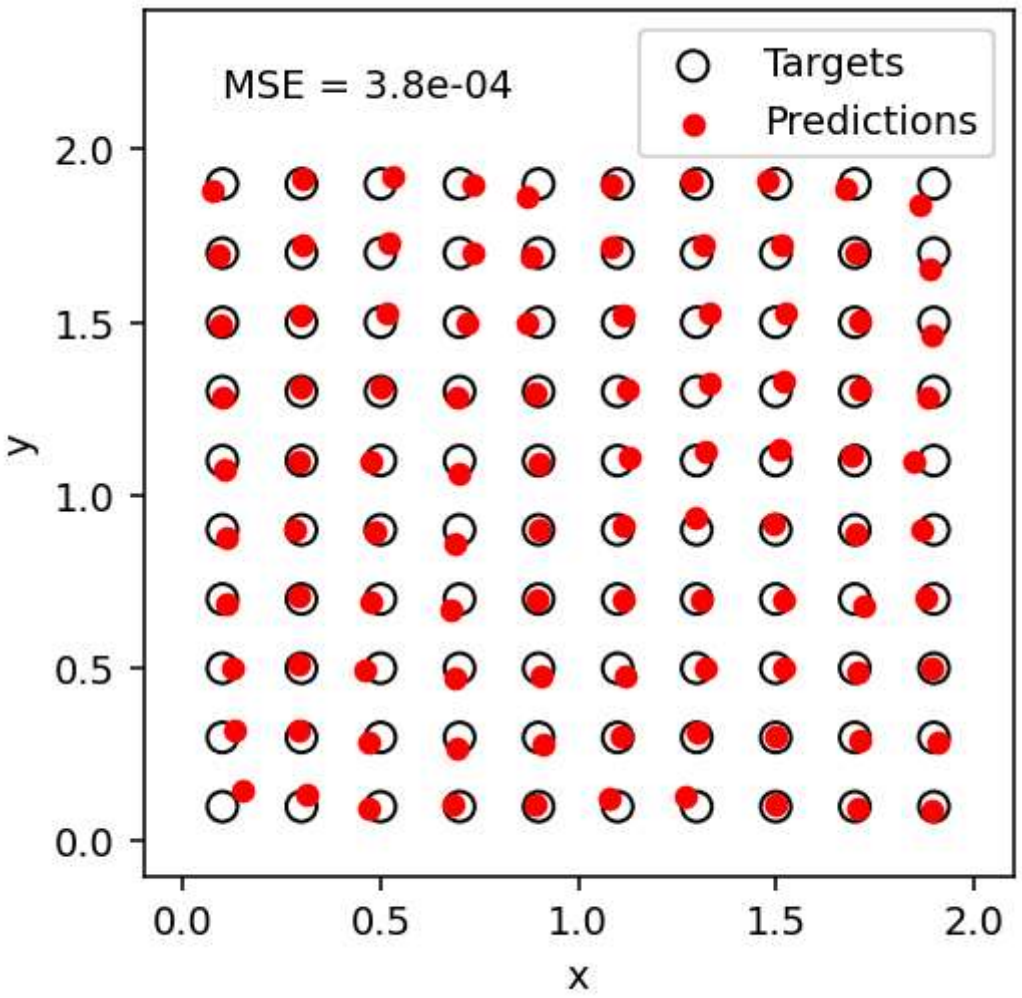
Visualizations

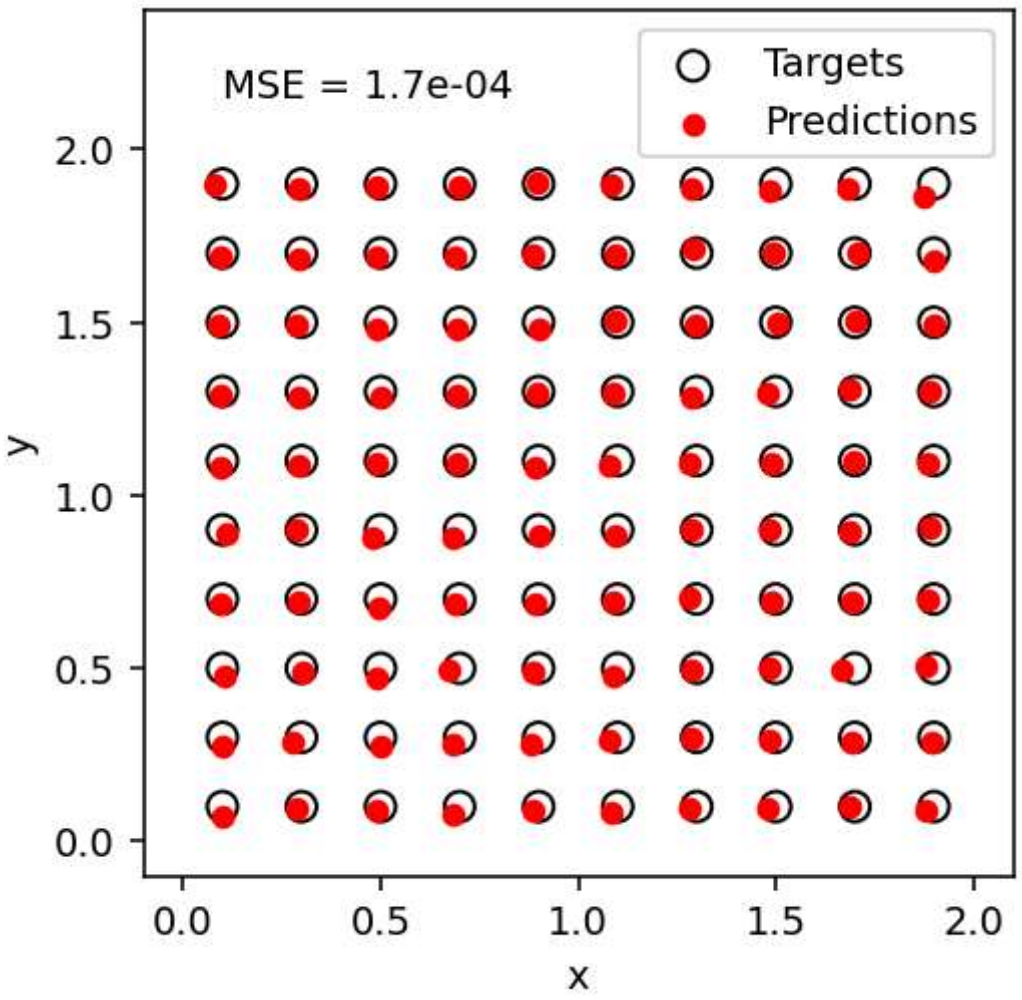
For each of your models, use the function `plot_predictions` to visualize model predictions on the domain. You should observe improvements with increasing network size.

In [330…

```
# YOUR CODE GOES HERE
plot_predictions(model1)
plot_predictions(model2)
plot_predictions(model3)
```


MSE = 7.7e-04
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x


MSE = 3.8e-04
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 [331…

```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, Float

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', disab
slider2 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100, description='y', disab

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

```
interactive_plot
```

Out[331…
```
interactive(children=(FloatSlider(value=1.0, description='x', layout=Layout(width
='550px'), max=2.5, min=-0.5,…
```

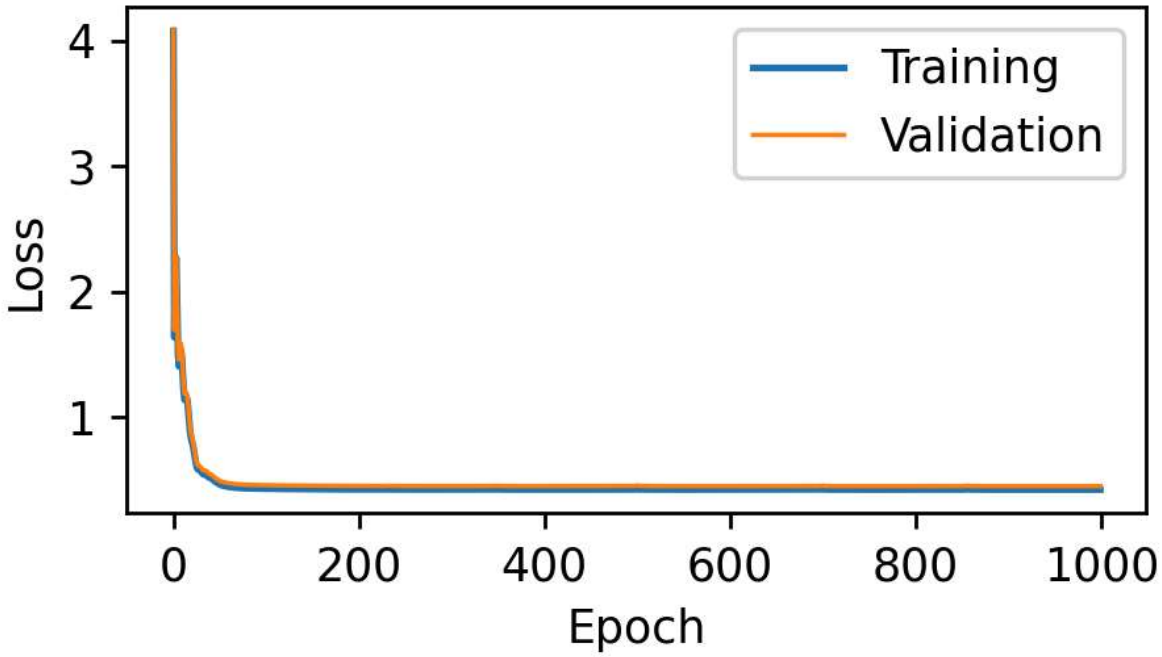
Training more neural networks

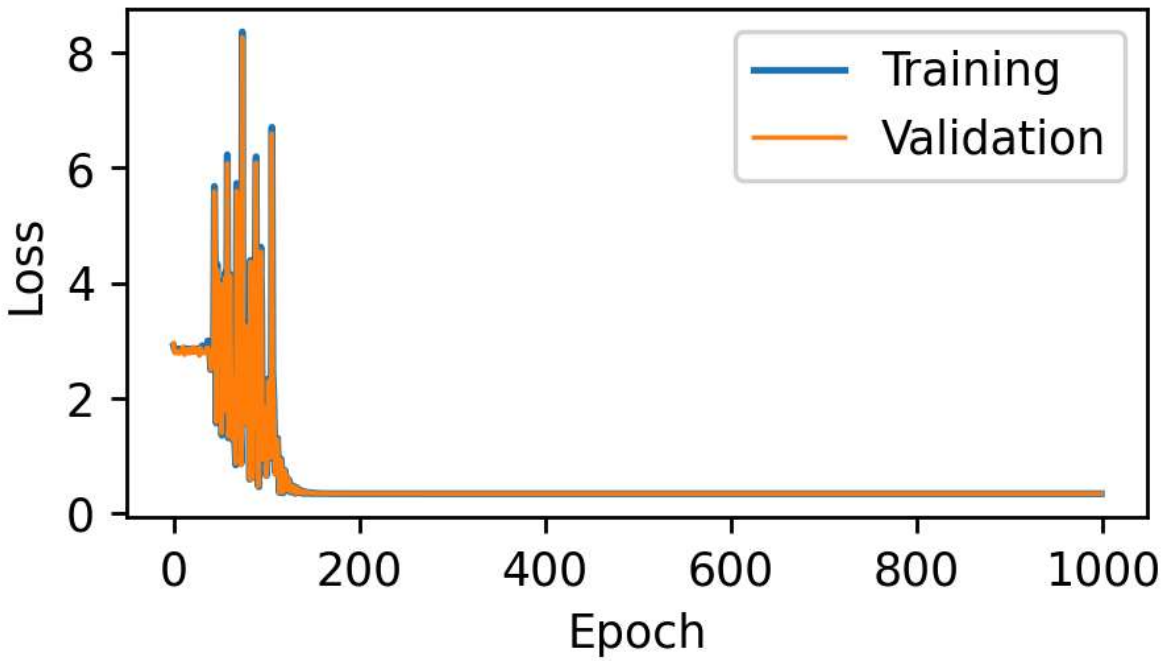
Now train more networks with the following details:

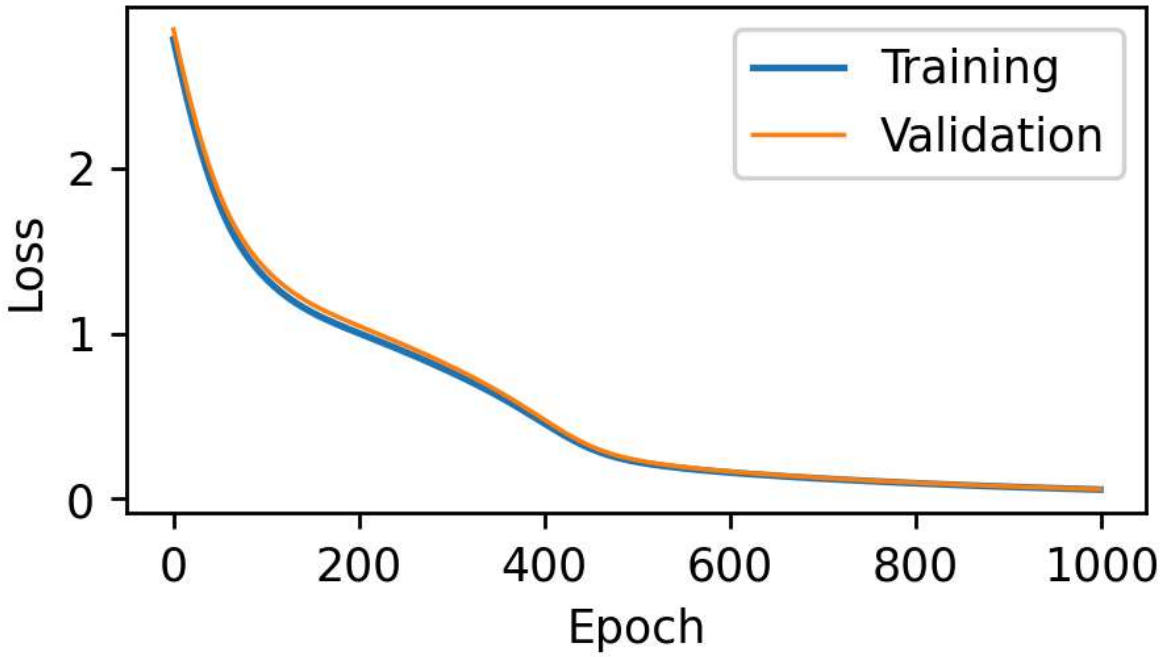
1. `hidden_layer_sizes=[48,48], max_angle=torch.pi/2` , train with `lr=0.01, epochs=1000, gamma=.995`
2. `hidden_layer_sizes=[48,48], max_angle=None` , train with `lr=1, epochs=1000, gamma=1`
3. `hidden_layer_sizes=[48,48], max_angle=2` , train with `lr=0.0001, epochs=300, gamma=1`

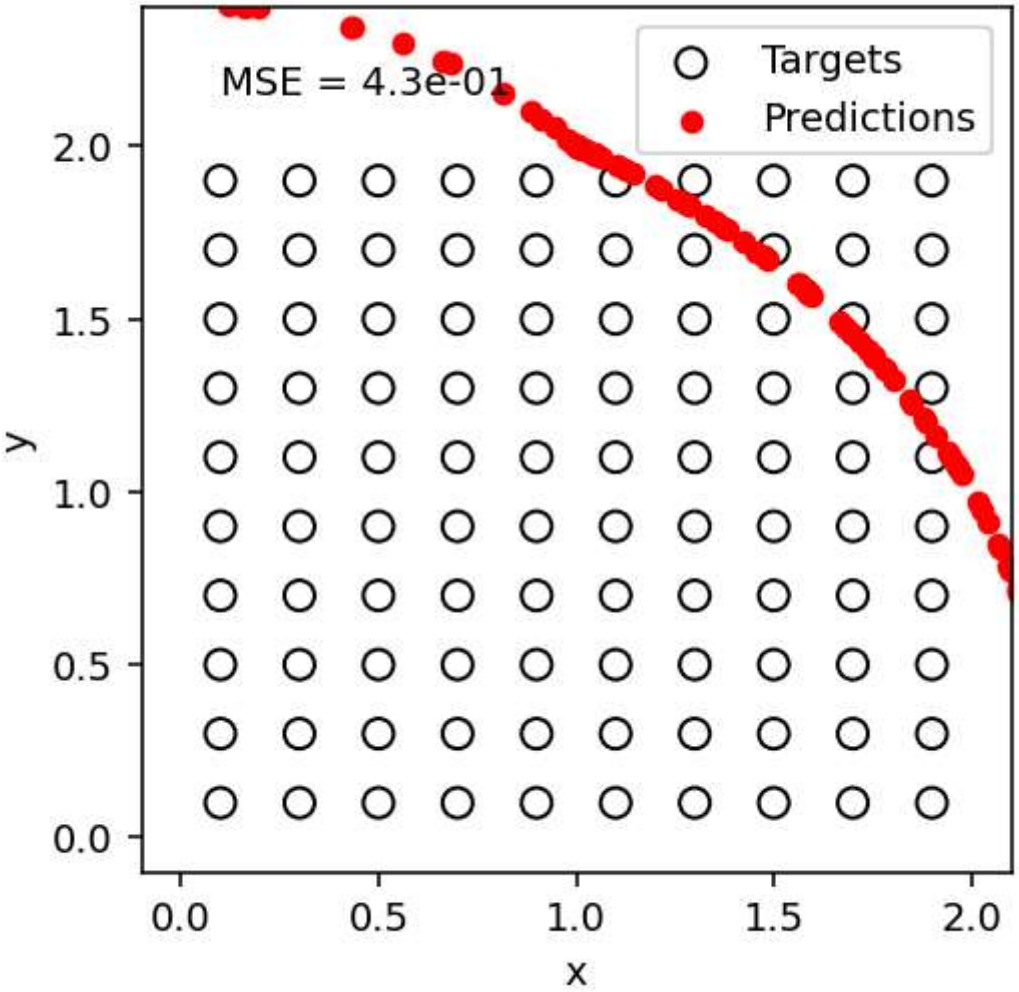
For each network, show a loss curve plot and a `plot_predictions` plot.

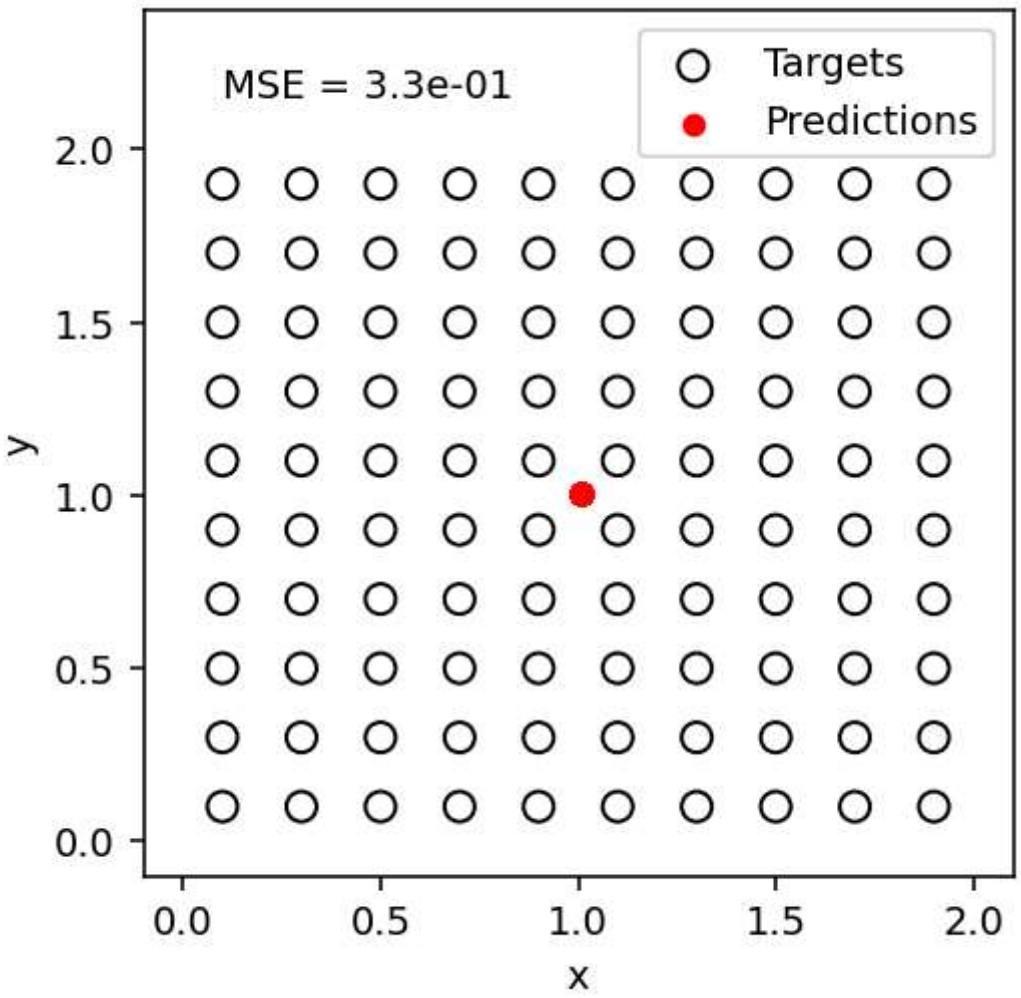
In [332…
```
# YOUR CODE GOES HERE
hidden_layers_tag=["(12)","(24, 24)","(48, 48, 48)"]
hidden_layers=[[48,48],[48,48],[48,48]]

model4 = InverseArm(hidden_layer_sizes=hidden_layers[0],max_angle=torch.pi/2)
model5 = InverseArm(hidden_layer_sizes=hidden_layers[1],max_angle=None)
model6 = InverseArm(hidden_layer_sizes=hidden_layers[2],max_angle=2)

loss_train1, loss_val1 = train(model4,X_train,X_val,gamma=0.995,lr=0.01,epochs=1000
loss_train2, loss_val2 = train(model5,X_train,X_val,gamma=1,lr=1,epochs=1000)
loss_train3, loss_val3 = train(model6,X_train,X_val,gamma=1,lr=0.0001,epochs=1000)

plot_predictions(model4)
plot_predictions(model5)
plot_predictions(model6)
```

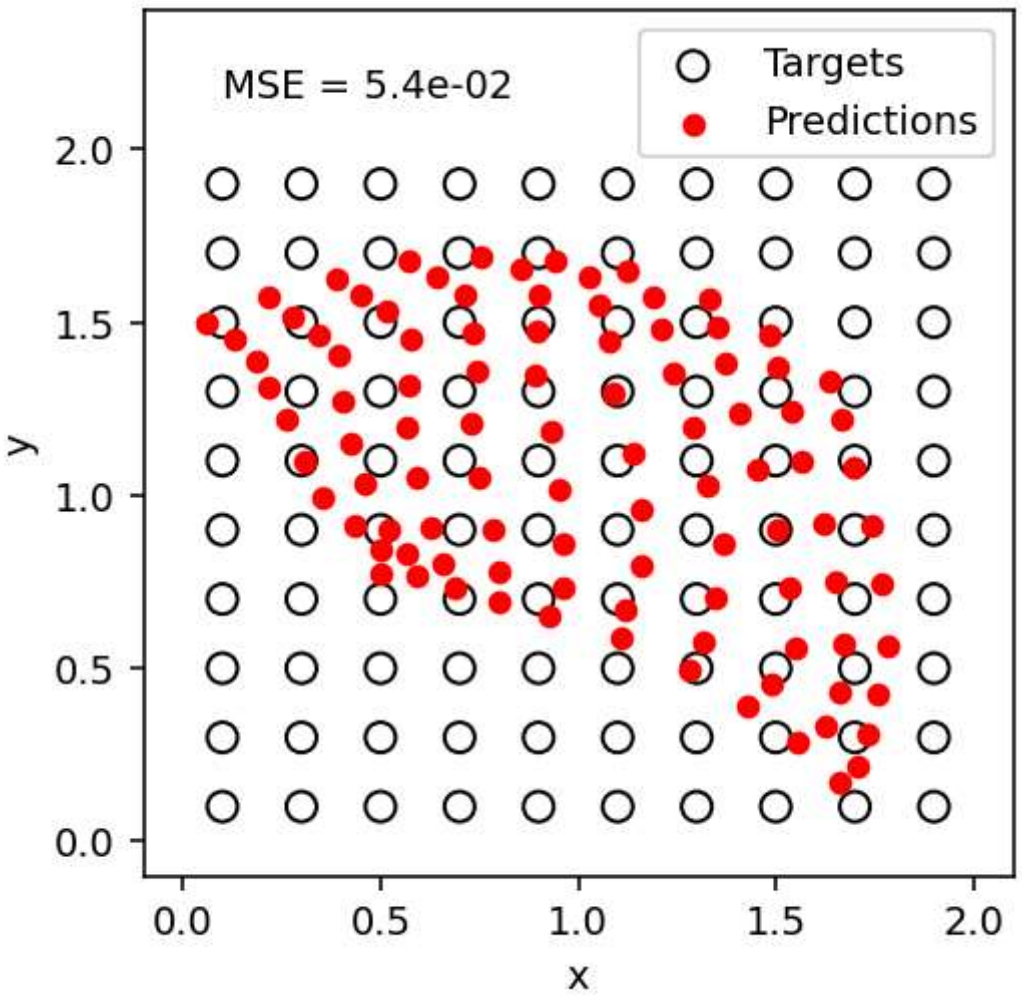

Training
Validation
Loss
Epoch
0
200
400
600
800
1000
1
2
3
4


Training
Validation
Loss
Epoch
0
200
400
600
800
1000
0
2
4
6
8


Training
Validation
Loss
Epoch
0
1
2
0
200
400
600
800
1000


MSE = 4.3e-01
Targets
Predictions
y
x
0.0
0.5
1.0
1.5
2.0


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

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

model1 has a max angle condition so the position is limited to part of the region
model2 has a very large learning rate which cause the curve to oscillate and hard to converge.
model3 has a very small learning rate which would take a much longer time to converge.