Problem 1:

$$a_1 = x_0 w_1 = 2(-1) = -2$$
$$a_2 = x_1 w_2 = 3(-2) = -6$$
$$a_3 = x_2 w_3 = 7(-6) = -42$$

$$e = t - a_3 = -40 - (-42) = 2$$

$$\delta_3 = -ef'(a_3) = -2 * 1 = -2$$
$$\delta_2 = \delta_3 w_3 f'(a_2) = -2(7)(1) = -14$$
$$\delta_1 = \delta_2 w_2 f'(a_1) = -14(3)(1) = -42$$

$$\frac{\delta L}{\delta w_3} = \delta_3 x_2 = -2(-6) = 12$$
$$\frac{\delta L}{\delta w_2} = \delta_2 x_1 = (-14)(-2) = 28$$
$$\frac{\delta L}{\delta w_1} = \delta_1 x_0 = -42(2) = -84$$

$$\boldsymbol{\frac{\delta L}{\delta w_3} = 12}, \qquad \boldsymbol{\frac{\delta L}{\delta w_2} = 28}, \qquad \boldsymbol{\frac{\delta L}{\delta w_1} = -84}$$

M8-HW1

November 4, 2023

1 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

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()
```

1.1 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
 ↪= 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
 ↪= 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
 ↪= 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
```

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

1.2 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__()
        self.seq = nn.Sequential(
            nn.Linear(2,hidden_layer_sizes[0]),
        )
        for i in range(0,len(hidden_layer_sizes)-1):
            self.seq.append(nn.ReLU())
            self.seq.append(nn.Linear(hidden_layer_sizes[i-1],␣
↪hidden_layer_sizes[i]))
        self.seq.append(nn.ReLU())
        self.seq.append(nn.Linear(hidden_layer_sizes[-1], 3))
        self.max_angle = max_angle
        if max_angle is not None:
            self.seq.append(nn.Tanh())

    def forward(self, xy):
        if self.max_angle is not None:
            return self.seq(xy) * self.max_angle
        return self.seq(xy)
```

1.3 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)

[ ]:
```
x,y = np.meshgrid(np.linspace(0,2,100),np.linspace(0,2,100))
x = x.reshape(-1,1)
y = y.reshape(-1,1)
N = x.shape[0]
idx = np.random.permutation(N)
X = np.concatenate([x,y], axis=1)
```

```
X_train = torch.Tensor(X[idx[:int(N*0.8)]])
X_val = torch.Tensor(X[idx[int(N*0.8):]])
print(f"X_train Size: {X_train.shape[0]}\t X_val Size: {X_val.shape[0]}")
```

```
X_train Size: 8000       X_val Size: 2000
```

1.4 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))`.

[ ]:
```
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 = True):

    train_hist = []
    val_hist = []

    fwd_arm = ForwardArm()

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

    for epoch in range(epochs):
        model.train()
        fwd = fwd_arm.forward(model(X_train))
        loss_train = ((fwd[:,0] - X_train[:,0])**2).sum() + ((fwd[:,1] -␣
 ↪X_train[:,1])**2).sum()
        train_hist.append(loss_train.item())
```

```
        model.eval()
        fwd = fwd_arm.forward(model(X_val))
        loss_val = ((fwd[:,0] - X_val[:,0])**2).sum() + ((fwd[:,1] - X_val[:
↪,1])**2).sum()
        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}")
    if (create_plot):
        plot_loss(train_hist, val_hist)
    return
```

1.5 Training a model

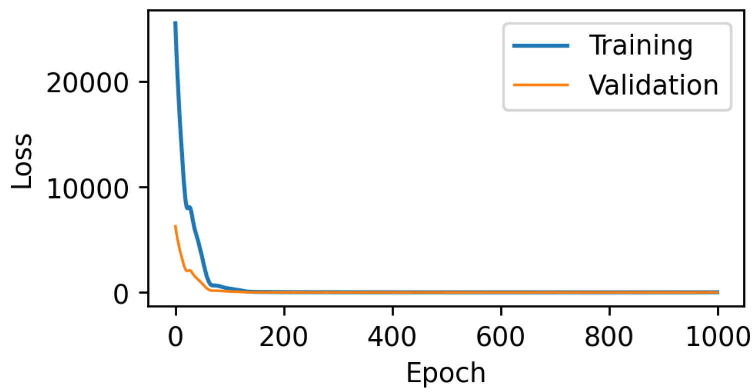
Create 3 models of different complexities (with `max_angle=None`): - `hidden_layer_sizes=[12]` - `hidden_layer_sizes=[24,24]` - `hidden_layer_sizes=[48,48,48]`

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

```
[ ]: model1 = InverseArm(hidden_layer_sizes=[12], max_angle=None)
     model2 = InverseArm(hidden_layer_sizes=[24,24], max_angle=None)
     model3 = InverseArm(hidden_layer_sizes=[48,48,48], max_angle=None)

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

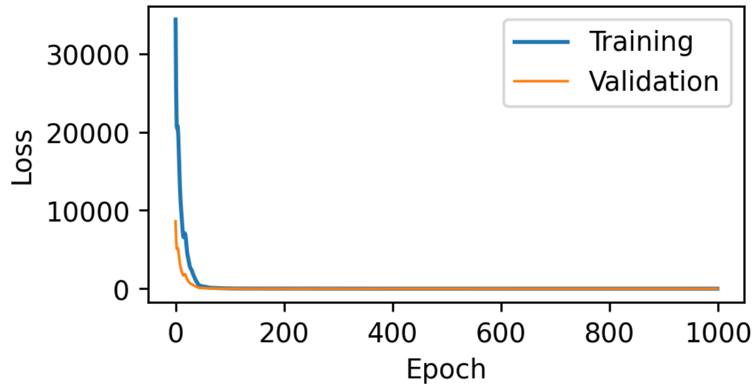
```
Epoch    0 of 1000:   Train Loss = 25546.5645   Validation Loss = 6280.7920
Epoch   40 of 1000:   Train Loss = 5233.4805   Validation Loss = 1312.8132
Epoch   80 of 1000:   Train Loss = 603.5000   Validation Loss = 152.6647
Epoch  120 of 1000:   Train Loss = 193.5890   Validation Loss = 47.6573
Epoch  160 of 1000:   Train Loss = 41.4407   Validation Loss = 10.0890
Epoch  200 of 1000:   Train Loss = 28.9926   Validation Loss = 7.2588
Epoch  240 of 1000:   Train Loss = 24.0554   Validation Loss = 5.9660
Epoch  280 of 1000:   Train Loss = 20.0644   Validation Loss = 4.9348
Epoch  320 of 1000:   Train Loss = 17.6081   Validation Loss = 4.3392
Epoch  360 of 1000:   Train Loss = 16.4072   Validation Loss = 4.0608
Epoch  400 of 1000:   Train Loss = 15.1366   Validation Loss = 3.7793
Epoch  440 of 1000:   Train Loss = 14.1027   Validation Loss = 3.5733
Epoch  480 of 1000:   Train Loss = 13.3510   Validation Loss = 3.3931
```

```
Epoch  520 of 1000:   Train Loss = 12.7572   Validation Loss = 3.2550
Epoch  560 of 1000:   Train Loss = 12.4656   Validation Loss = 3.1904
Epoch  600 of 1000:   Train Loss = 12.0076   Validation Loss = 3.0836
Epoch  640 of 1000:   Train Loss = 11.6399   Validation Loss = 2.9945
Epoch  680 of 1000:   Train Loss = 11.3400   Validation Loss = 2.9210
Epoch  720 of 1000:   Train Loss = 11.0204   Validation Loss = 2.8486
Epoch  760 of 1000:   Train Loss = 11.6010   Validation Loss = 3.0059
Epoch  800 of 1000:   Train Loss = 10.3242   Validation Loss = 2.6837
Epoch  840 of 1000:   Train Loss = 9.9205   Validation Loss = 2.5803
Epoch  880 of 1000:   Train Loss = 9.5713   Validation Loss = 2.4851
Epoch  920 of 1000:   Train Loss = 9.2069   Validation Loss = 2.3885
Epoch  960 of 1000:   Train Loss = 8.8864   Validation Loss = 2.3010
```

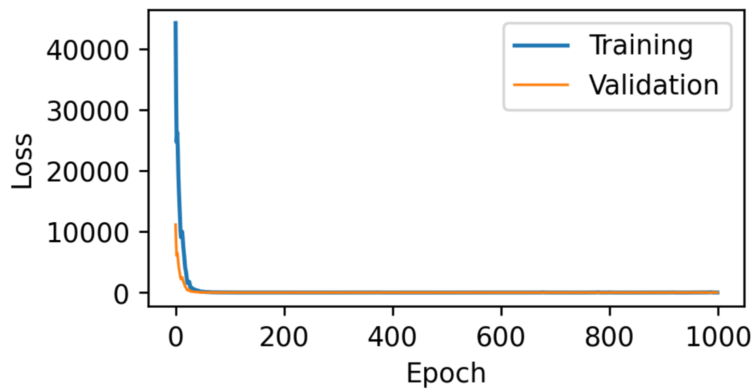



```
Epoch    0 of 1000:   Train Loss = 34373.7578   Validation Loss = 8594.2656
Epoch   40 of 1000:   Train Loss = 755.1193   Validation Loss = 194.0777
Epoch   80 of 1000:   Train Loss = 67.2362   Validation Loss = 18.2116
Epoch  120 of 1000:   Train Loss = 18.7699   Validation Loss = 4.7104
Epoch  160 of 1000:   Train Loss = 10.9724   Validation Loss = 2.8248
Epoch  200 of 1000:   Train Loss = 8.8017   Validation Loss = 2.2716
Epoch  240 of 1000:   Train Loss = 5.7757   Validation Loss = 1.4593
Epoch  280 of 1000:   Train Loss = 5.0449   Validation Loss = 1.3011
Epoch  320 of 1000:   Train Loss = 4.6540   Validation Loss = 1.2005
Epoch  360 of 1000:   Train Loss = 3.9765   Validation Loss = 1.0749
Epoch  400 of 1000:   Train Loss = 3.4201   Validation Loss = 0.9314
Epoch  440 of 1000:   Train Loss = 3.2506   Validation Loss = 0.8940
Epoch  480 of 1000:   Train Loss = 3.0020   Validation Loss = 0.8251
Epoch  520 of 1000:   Train Loss = 2.8602   Validation Loss = 0.7898
Epoch  560 of 1000:   Train Loss = 2.6529   Validation Loss = 0.7316
Epoch  600 of 1000:   Train Loss = 2.5693   Validation Loss = 0.7120
Epoch  640 of 1000:   Train Loss = 2.6078   Validation Loss = 0.7260
Epoch  680 of 1000:   Train Loss = 2.3562   Validation Loss = 0.6567
Epoch  720 of 1000:   Train Loss = 4.3666   Validation Loss = 1.1521
Epoch  760 of 1000:   Train Loss = 2.2730   Validation Loss = 0.6410
```

```
Epoch  800 of 1000:   Train Loss = 2.1342   Validation Loss = 0.6008
Epoch  840 of 1000:   Train Loss = 2.2143   Validation Loss = 0.6260
Epoch  880 of 1000:   Train Loss = 2.3154   Validation Loss = 0.6524
Epoch  920 of 1000:   Train Loss = 1.9529   Validation Loss = 0.5500
Epoch  960 of 1000:   Train Loss = 2.5349   Validation Loss = 0.6832
```



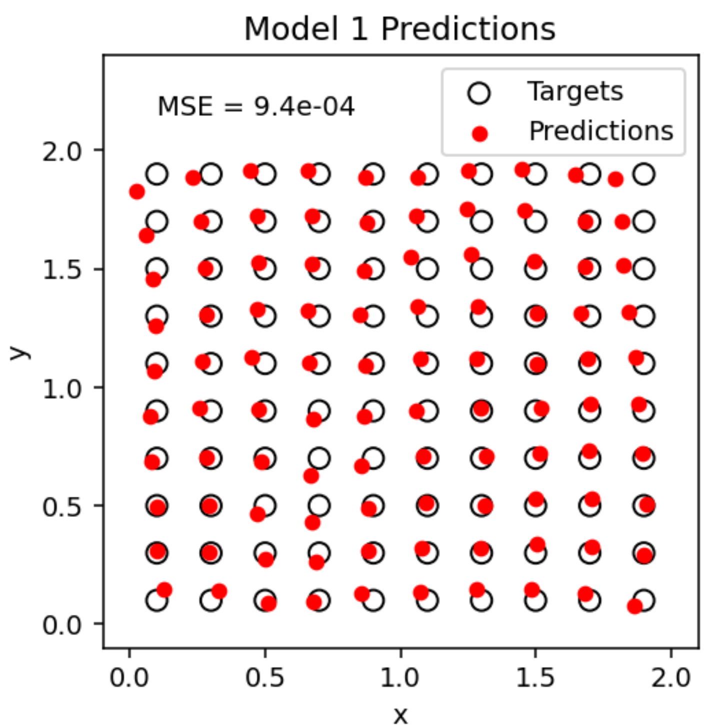

```
Epoch    0 of 1000:   Train Loss = 44264.2422   Validation Loss = 11154.2012
Epoch   40 of 1000:   Train Loss = 366.9883   Validation Loss = 90.7924
Epoch   80 of 1000:   Train Loss = 27.2176   Validation Loss = 7.3276
Epoch  120 of 1000:   Train Loss = 9.8337   Validation Loss = 2.5741
Epoch  160 of 1000:   Train Loss = 6.0183   Validation Loss = 1.6150
Epoch  200 of 1000:   Train Loss = 4.6248   Validation Loss = 1.2447
Epoch  240 of 1000:   Train Loss = 3.8746   Validation Loss = 1.0435
Epoch  280 of 1000:   Train Loss = 3.3499   Validation Loss = 0.9101
Epoch  320 of 1000:   Train Loss = 2.8528   Validation Loss = 0.7708
Epoch  360 of 1000:   Train Loss = 3.8809   Validation Loss = 0.9941
Epoch  400 of 1000:   Train Loss = 3.0453   Validation Loss = 0.7888
Epoch  440 of 1000:   Train Loss = 2.1278   Validation Loss = 0.5720
Epoch  480 of 1000:   Train Loss = 1.9743   Validation Loss = 0.5302
Epoch  520 of 1000:   Train Loss = 2.1048   Validation Loss = 0.5699
Epoch  560 of 1000:   Train Loss = 2.7764   Validation Loss = 0.7138
Epoch  600 of 1000:   Train Loss = 13.8121   Validation Loss = 3.5463
Epoch  640 of 1000:   Train Loss = 1.8781   Validation Loss = 0.5065
Epoch  680 of 1000:   Train Loss = 12.9397   Validation Loss = 3.2489
Epoch  720 of 1000:   Train Loss = 2.0195   Validation Loss = 0.5442
Epoch  760 of 1000:   Train Loss = 1.6425   Validation Loss = 0.4371
Epoch  800 of 1000:   Train Loss = 30.8539   Validation Loss = 7.8174
Epoch  840 of 1000:   Train Loss = 1.9294   Validation Loss = 0.5225
Epoch  880 of 1000:   Train Loss = 1.3876   Validation Loss = 0.3758
Epoch  920 of 1000:   Train Loss = 5.9344   Validation Loss = 1.4942
Epoch  960 of 1000:   Train Loss = 1.7825   Validation Loss = 0.4666
```

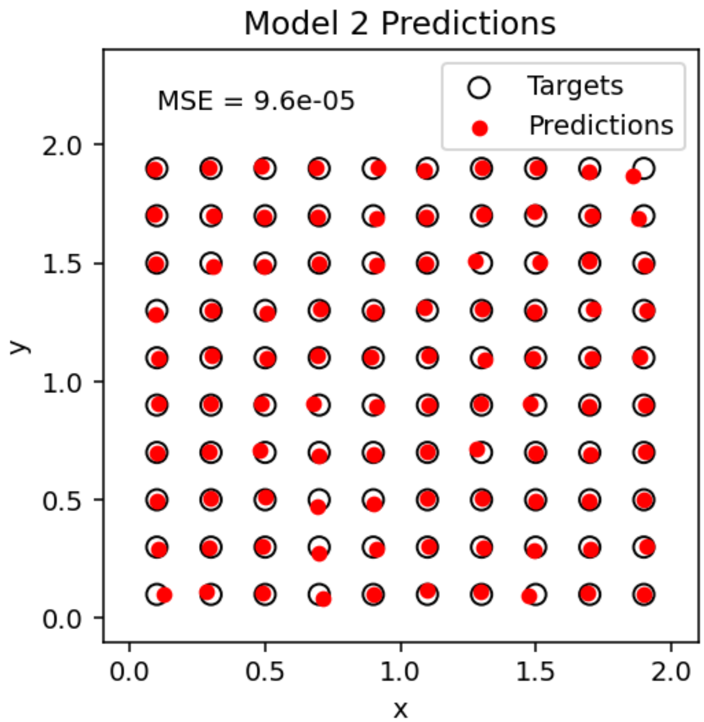


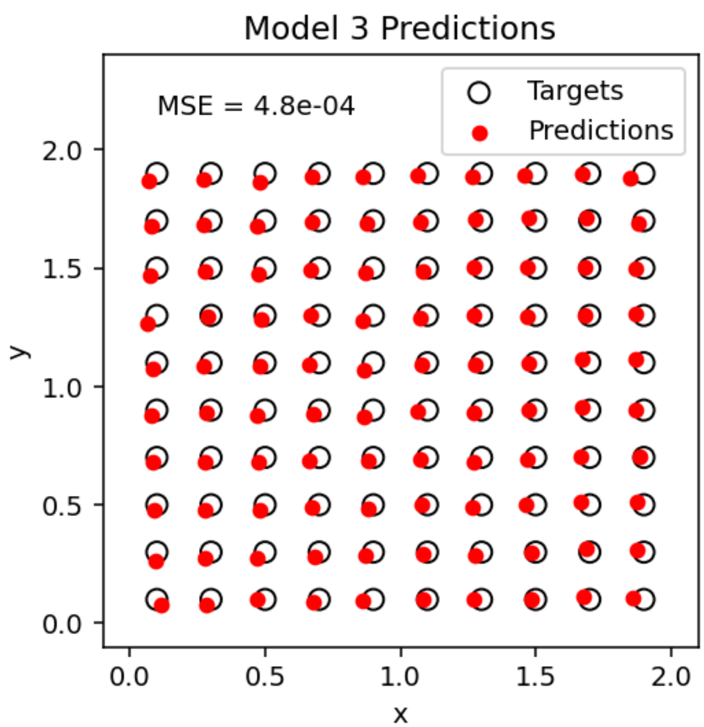
1.6 Visualizations

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.

[ ]:
```
plot_predictions(model1, "Model 1 Predictions")
plot_predictions(model2, "Model 2 Predictions")
plot_predictions(model3, "Model 3 Predictions")
```




Model 2 Predictions
MSE = 9.6e-05
Targets
Predictions
y
x
0.0
0.5
1.0
1.5
2.0


Model 3 Predictions
MSE = 4.8e-04
Targets
Predictions
y
x
0.0
0.5
1.0
1.5
2.0

1.7 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
model = model1
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
```

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

1.8 Training more neural networks

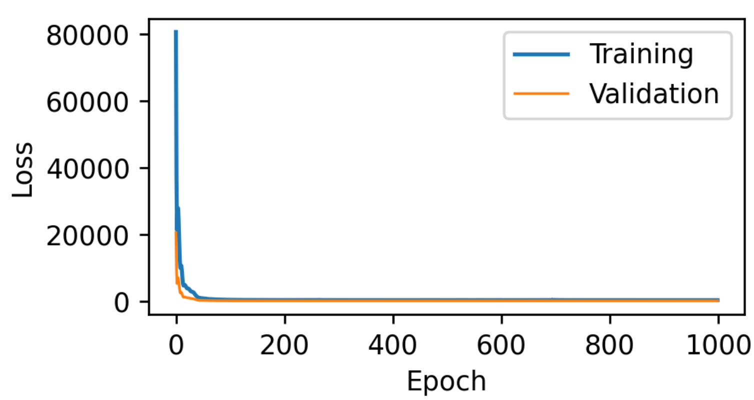
Now train more networks with the following details: 1. `hidden_layer_sizes=[48,48]`, `max_angle=torch.pi/2`, train with `lr=0.01`, `epochs=1000`, `gamma=.995` 2. `hidden_layer_sizes=[48,48]`, `max_angle=None`, train with `lr=1`, `epochs=1000`, `gamma=1` 3. `hidden_layer_sizes=[48,48]`, `max_angle=2`, train with `lr=0.0001`, `epochs=300`, `gamma=1`

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

```
[ ]: model1 = InverseArm(hidden_layer_sizes=[48,48], max_angle=torch.pi/2)
model2 = InverseArm(hidden_layer_sizes=[48,48], max_angle=None)
model3 = InverseArm(hidden_layer_sizes=[48,48], max_angle=2)

train(model1, X_train, X_val, lr=0.01, epochs=1000, gamma=0.995,␣
 ↪create_plot=True)
train(model2, X_train, X_val, lr=1, epochs=1000, gamma=1, create_plot=True)
```

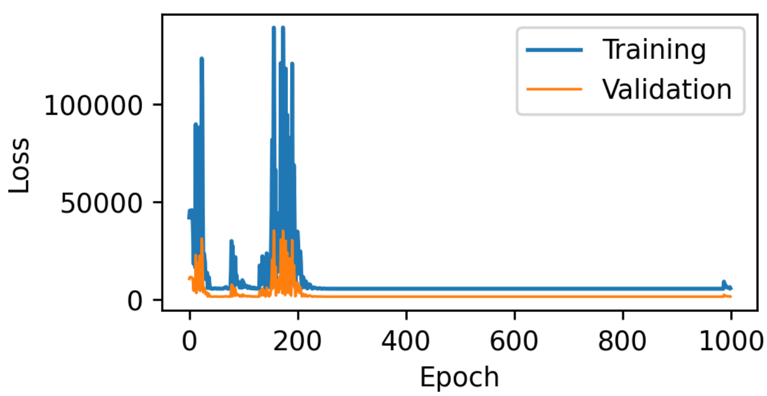
```
train(model3, X_train, X_val, lr=0.0001, epochs=300, gamma=1, create_plot=True)
```

```
Epoch    0 of 1000:   Train Loss = 80679.2109   Validation Loss = 20523.1016
Epoch   40 of 1000:   Train Loss = 1183.4551   Validation Loss = 298.9445
Epoch   80 of 1000:   Train Loss = 431.3502   Validation Loss = 109.6607
Epoch  120 of 1000:   Train Loss = 329.2130   Validation Loss = 85.4530
Epoch  160 of 1000:   Train Loss = 304.8269   Validation Loss = 79.3076
Epoch  200 of 1000:   Train Loss = 296.8060   Validation Loss = 77.4190
Epoch  240 of 1000:   Train Loss = 291.0080   Validation Loss = 76.0475
Epoch  280 of 1000:   Train Loss = 288.3641   Validation Loss = 75.4505
Epoch  320 of 1000:   Train Loss = 285.7700   Validation Loss = 74.8172
Epoch  360 of 1000:   Train Loss = 283.8395   Validation Loss = 74.3492
Epoch  400 of 1000:   Train Loss = 282.3068   Validation Loss = 73.9939
Epoch  440 of 1000:   Train Loss = 282.1997   Validation Loss = 73.9915
Epoch  480 of 1000:   Train Loss = 284.5198   Validation Loss = 74.5711
Epoch  520 of 1000:   Train Loss = 287.7758   Validation Loss = 75.4995
Epoch  560 of 1000:   Train Loss = 279.0705   Validation Loss = 73.1947
Epoch  600 of 1000:   Train Loss = 295.5007   Validation Loss = 77.5301
Epoch  640 of 1000:   Train Loss = 278.5186   Validation Loss = 73.0631
Epoch  680 of 1000:   Train Loss = 277.7566   Validation Loss = 72.8770
Epoch  720 of 1000:   Train Loss = 280.5244   Validation Loss = 73.6237
Epoch  760 of 1000:   Train Loss = 277.4835   Validation Loss = 72.8100
Epoch  800 of 1000:   Train Loss = 276.8701   Validation Loss = 72.6479
Epoch  840 of 1000:   Train Loss = 276.5109   Validation Loss = 72.5532
Epoch  880 of 1000:   Train Loss = 276.1619   Validation Loss = 72.4620
Epoch  920 of 1000:   Train Loss = 275.8745   Validation Loss = 72.3878
Epoch  960 of 1000:   Train Loss = 275.6368   Validation Loss = 72.3284
```



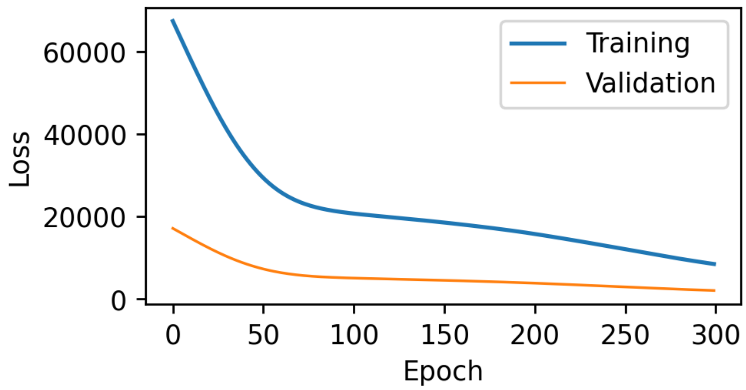

```
Epoch    0 of 1000:   Train Loss = 41916.7227   Validation Loss = 10490.3389
Epoch   40 of 1000:   Train Loss = 5580.8760   Validation Loss = 1397.8684
Epoch   80 of 1000:   Train Loss = 27271.2773   Validation Loss = 7010.5996
Epoch  120 of 1000:   Train Loss = 5842.3164   Validation Loss = 1504.8444
Epoch  160 of 1000:   Train Loss = 7810.2197   Validation Loss = 1995.0405
```

```
Epoch  200 of 1000:   Train Loss = 34657.6406   Validation Loss = 8775.2568
Epoch  240 of 1000:   Train Loss = 5697.0640   Validation Loss = 1463.8599
Epoch  280 of 1000:   Train Loss = 5428.1333   Validation Loss = 1376.9443
Epoch  320 of 1000:   Train Loss = 5425.9106   Validation Loss = 1375.6538
Epoch  360 of 1000:   Train Loss = 5425.9038   Validation Loss = 1375.7328
Epoch  400 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7371
Epoch  440 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7346
Epoch  480 of 1000:   Train Loss = 5425.9048   Validation Loss = 1375.7485
Epoch  520 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7361
Epoch  560 of 1000:   Train Loss = 5426.2710   Validation Loss = 1375.6338
Epoch  600 of 1000:   Train Loss = 5425.9082   Validation Loss = 1375.7102
Epoch  640 of 1000:   Train Loss = 5425.9038   Validation Loss = 1375.7336
Epoch  680 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7365
Epoch  720 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7362
Epoch  760 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7362
Epoch  800 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7366
Epoch  840 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7371
Epoch  880 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7371
Epoch  920 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7375
Epoch  960 of 1000:   Train Loss = 5425.9043   Validation Loss = 1375.7375
```




```
Epoch    0 of 300:   Train Loss = 67431.0859   Validation Loss = 17155.5176
Epoch   12 of 300:   Train Loss = 56133.7344   Validation Loss = 14233.1855
Epoch   24 of 300:   Train Loss = 45643.7852   Validation Loss = 11515.0762
Epoch   36 of 300:   Train Loss = 36858.4531   Validation Loss = 9237.4844
Epoch   48 of 300:   Train Loss = 30263.4727   Validation Loss = 7528.7607
Epoch   60 of 300:   Train Loss = 25845.8203   Validation Loss = 6386.8262
Epoch   72 of 300:   Train Loss = 23225.0059   Validation Loss = 5712.7666
Epoch   84 of 300:   Train Loss = 21776.6562   Validation Loss = 5343.5337
Epoch   96 of 300:   Train Loss = 20943.3418   Validation Loss = 5135.2290
Epoch  108 of 300:   Train Loss = 20370.2148   Validation Loss = 4994.2476
Epoch  120 of 300:   Train Loss = 19871.3066   Validation Loss = 4871.4351
Epoch  132 of 300:   Train Loss = 19369.9844   Validation Loss = 4747.1338
```

```
Epoch  144 of 300:   Train Loss = 18837.3984   Validation Loss = 4615.2271
Epoch  156 of 300:   Train Loss = 18273.6602   Validation Loss = 4475.2837
Epoch  168 of 300:   Train Loss = 17668.4824   Validation Loss = 4324.5068
Epoch  180 of 300:   Train Loss = 17021.2168   Validation Loss = 4163.7529
Epoch  192 of 300:   Train Loss = 16311.2080   Validation Loss = 3988.0532
Epoch  204 of 300:   Train Loss = 15515.2637   Validation Loss = 3790.8984
Epoch  216 of 300:   Train Loss = 14664.0664   Validation Loss = 3580.2590
Epoch  228 of 300:   Train Loss = 13777.4844   Validation Loss = 3361.4927
Epoch  240 of 300:   Train Loss = 12865.3184   Validation Loss = 3136.9336
Epoch  252 of 300:   Train Loss = 11938.1113   Validation Loss = 2909.2393
Epoch  264 of 300:   Train Loss = 11001.1943   Validation Loss = 2679.4839
Epoch  276 of 300:   Train Loss = 10086.4434   Validation Loss = 2455.8218
Epoch  288 of 300:   Train Loss = 9227.4385   Validation Loss = 2246.8289
```



1.9 Prompts

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

In the first case, the model wasn't allowed to predict across the whole range of joint angles needed to reach each configuration.

In the second case, the model had too large of a learning rate leading to the noisy training and validation loss.

In the third case, the learning rate was too low with not enough epochs thus the model didn't have enough time to fully train to the data.

M8-L1-P1

November 4, 2023

1 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]])
```

1.1 Define activation function and its derivative

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

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

def del_sigmoid(x):
    s = sigmoid(x)
    return s*(1-s)
```

2 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

[ ]:
```
a1 = W1 @ x0
x1 = sigmoid(a1)
a2 = W2 @ x1
y = a2
```

```
print(y)
```

```
[[-1.31123207]]
```

2.1 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 $\frac{\partial L}{\partial W_1}$.

[ ]:
```
e = t - y
L = 0.5*(e.T @ e)

delta_2 = -e
dLdw2 = delta_2 * x1
delta_1 = delta_2 @ W2 @ del_sigmoid(a1)
dLdw1 = delta_1 * x0

print(dLdw2)
print(dLdw1)
```

```
[[8.21031503e-02]
 [2.43316128e-24]
 [1.04899215e-08]]
[[1.59095662]
 [4.77286987]]
```

M8-L2-P1

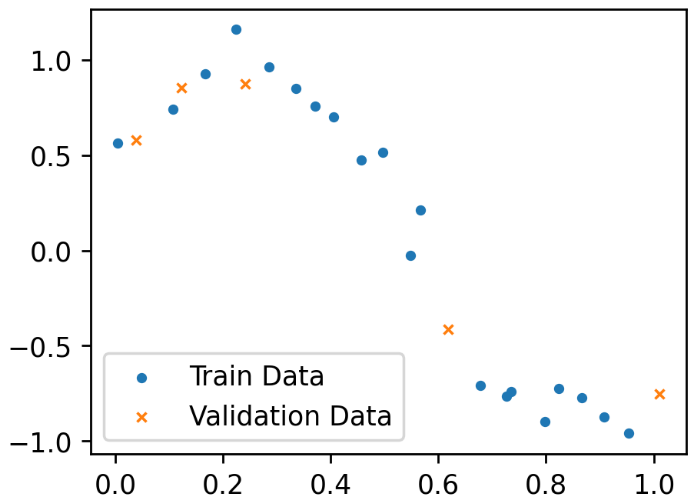
November 4, 2023

1 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="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()
```



1.1 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)

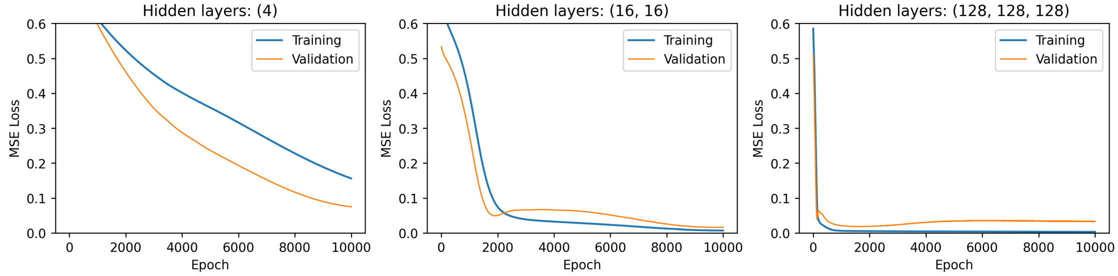
[ ]:
```
class Model1(nn.Module):
    def __init__(self, N_hidden=6, N_in=2, N_out=3):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(1, 4),
            nn.ReLU(),
            nn.Linear(4, 1)
        )
    def forward(self,x):
        return self.seq(x)

class Model2(nn.Module):
    def __init__(self, N_hidden=6, N_in=2, N_out=3):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(1, 16),
```

```
            nn.ReLU(),
            nn.Linear(16, 16),
            nn.ReLU(),
            nn.Linear(16, 1),
        )
    def forward(self,x):
        return self.seq(x)

class Model3(nn.Module):
    def __init__(self, N_hidden=6, N_in=2, N_out=3):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(1, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
        )
    def forward(self,x):
        return self.seq(x)

model1 = Model1()
model2 = Model2()
model3 = Model3()
```

1.2 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()
```



1.3 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).

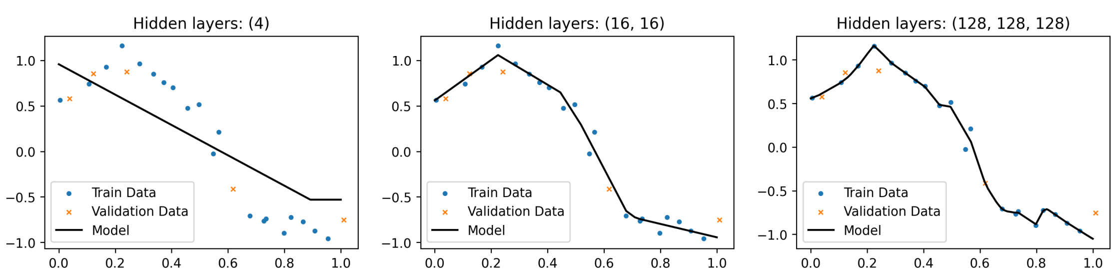
[ ]:
```
for i, model in enumerate([model1, model2, model3]):
    train_loss, val_loss = get_loss(model)
    print(f"Model {i+1}, hidden layers {hidden_layers[i]:>15}:   Train MSE: {train_loss:.4f}    Test MSE: {val_loss:.4f}")
```

```
Model 1, hidden layers             (4):   Train MSE: 0.1560    Test MSE: 0.0749
Model 2, hidden layers        (16, 16):   Train MSE: 0.0067    Test MSE: 0.0161
Model 3, hidden layers (128, 128, 128):   Train MSE: 0.0032    Test MSE: 0.0326
```

1.4 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()
```



1.5 Questions

1. For the model that overfits the most, describe what happens to the loss curves while training.

The training loss curve decreases rapidly at the start staying at a very low value for a long time during training. The overfitting is shown clearly by the validation curve reaching a global minimum in the middle of training and then increasing as training continues.

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

The slope of the loss curve never reaches a flat point leveling out over time. The slope is still similar to the start of training even at the end of training. The loss curve also only gets beneath the validation curve towards the very end of training.

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

The curve decreases rapidly at the start and then levels off towards the end of training approaching a horizontal asymptote staying underneath the validation curve.

M8-L2-P2

November 4, 2023

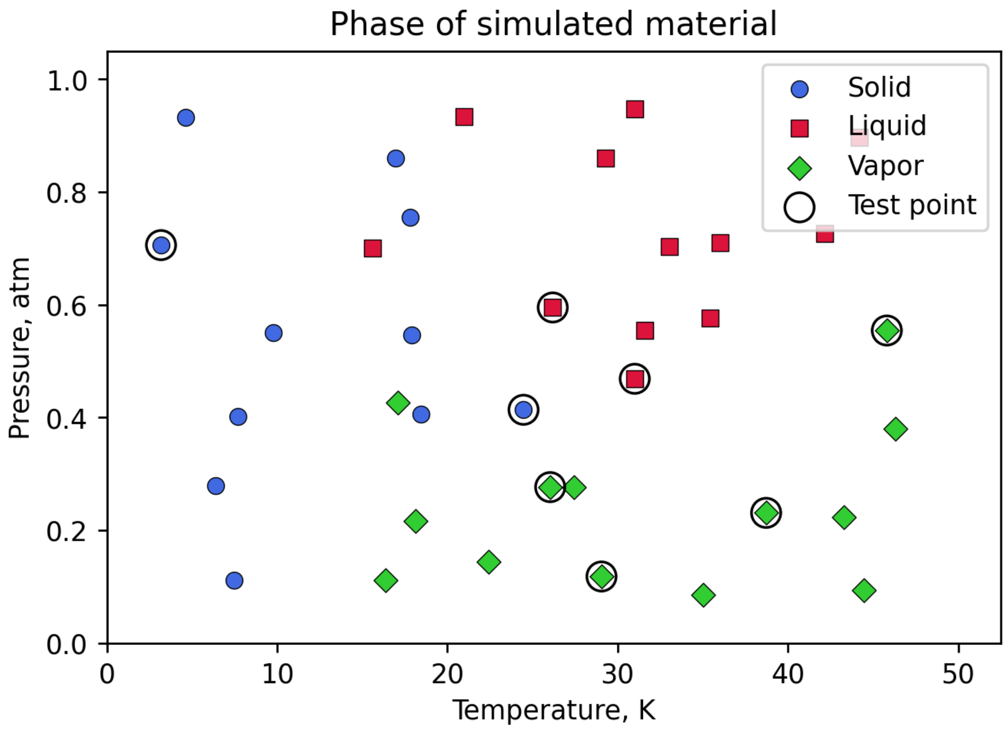
0.1 M8-L2 Problem 2

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

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
```

```python
x1 = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.04883182996897,35.03654799338904,44.45894113066656,6.375872112626925,18.117730007820796,26.036627605010292,27.434415188257777,38.71725038082664,43.28894919752904,7.680445610939323,18.45596638292661,17.110360581978867,24.47129299701541,31.002183974403255,46.32619845547938,9.781567509498505,17.90012148246819,26.186183422327638,31.59158564216724,35.41479362252932,45.805291762864556,3.182744258689332,15.599210213275237,17.833532874090462,33.04668917049584,36.018483217500716,42.146619399905234,4.64555612104627,16.942336894342166,20.961503322165484,29.284339488686488,30.98789800436355,44.17635497075877,])
x2 = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.11818202991034835,0.0859507900573786,0.09370319537993416,0.2797631195927265,0.216022547162927,0.27667667154456677,0.27706378696181594,0.2310382561073841,0.22289262976548535,0.40154283509241845,0.4063710770942623,0.427019677041788,0.41386015134623205,0.46883738380592266,0.38020448107480287,0.5508876756094834,0.5461309517884996,0.5953108325465398,0.5553291602539782,0.5766310772856306,0.5544425592001603,0.705896958364552,0.7010375141164304,0.7556329589465274,0.7038182951348614,0.7096582361680054,0.7268725170660963,0.9320993229847936,0.8597101275793062,0.9337944907498804,0.8596098407893963,0.9476459465013396,0.8968651201647702,])
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()
```



0.2 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__()
        self.seq = nn.Sequential(
            nn.Linear(2,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,20),
            nn.ReLU(),
            nn.Linear(20,20),
```

```
            nn.ReLU(),
            nn.Linear(20,3),
            nn.Softmax()
        )

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

    def forward(self,X):
        return self.seq(X)
```

0.3 Training

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

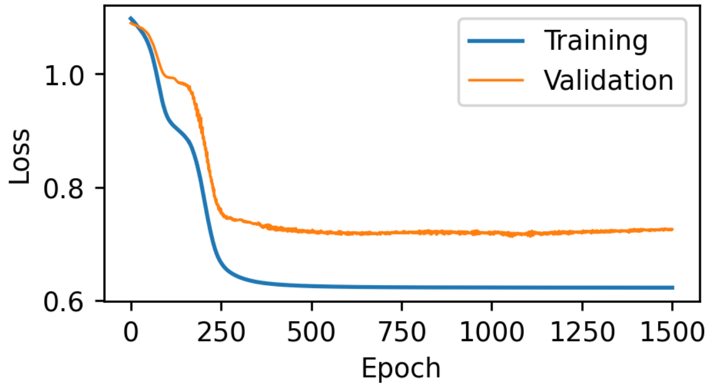
- Define a loss function (for multiclass classification)
- Define an optimizer and call it `opt`. You may choose which optimizer.

Make sure the training curves you get are reasonable.

[ ]:
```
model = PhaseNet()

lr = 0.001
epochs = 1500

lossfun = nn.CrossEntropyLoss()

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

train_hist = []
val_hist = []

def getArray(index):
    arr = np.zeros(3)
    arr[index] = 1
    return arr
train_y_new = np.array([getArray(i) for i in train_y])
train_y_new = torch.Tensor(train_y_new)
val_y_new = np.array([getArray(i) for i in val_y])
val_y_new = torch.Tensor(val_y_new)

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

    model.eval()
```

```
    loss_val = lossfun(model(val_x), val_y_new)
    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 1500:   Train Loss = 1.0978   Validation Loss = 1.0896
Epoch   60 of 1500:   Train Loss = 1.0309   Validation Loss = 1.0551
Epoch  120 of 1500:   Train Loss = 0.9077   Validation Loss = 0.9923
Epoch  180 of 1500:   Train Loss = 0.8485   Validation Loss = 0.9374
Epoch  240 of 1500:   Train Loss = 0.6795   Validation Loss = 0.7702
Epoch  300 of 1500:   Train Loss = 0.6417   Validation Loss = 0.7429
Epoch  360 of 1500:   Train Loss = 0.6319   Validation Loss = 0.7321
Epoch  420 of 1500:   Train Loss = 0.6281   Validation Loss = 0.7243
Epoch  480 of 1500:   Train Loss = 0.6261   Validation Loss = 0.7227
Epoch  540 of 1500:   Train Loss = 0.6251   Validation Loss = 0.7189
Epoch  600 of 1500:   Train Loss = 0.6244   Validation Loss = 0.7167
Epoch  660 of 1500:   Train Loss = 0.6240   Validation Loss = 0.7197
Epoch  720 of 1500:   Train Loss = 0.6237   Validation Loss = 0.7202
Epoch  780 of 1500:   Train Loss = 0.6236   Validation Loss = 0.7195
Epoch  840 of 1500:   Train Loss = 0.6234   Validation Loss = 0.7218
Epoch  900 of 1500:   Train Loss = 0.6234   Validation Loss = 0.7198
Epoch  960 of 1500:   Train Loss = 0.6233   Validation Loss = 0.7201
Epoch 1020 of 1500:   Train Loss = 0.6232   Validation Loss = 0.7201
Epoch 1080 of 1500:   Train Loss = 0.6232   Validation Loss = 0.7182
Epoch 1140 of 1500:   Train Loss = 0.6231   Validation Loss = 0.7170
Epoch 1200 of 1500:   Train Loss = 0.6231   Validation Loss = 0.7201
Epoch 1260 of 1500:   Train Loss = 0.6231   Validation Loss = 0.7219
Epoch 1320 of 1500:   Train Loss = 0.6231   Validation Loss = 0.7220
Epoch 1380 of 1500:   Train Loss = 0.6230   Validation Loss = 0.7204
Epoch 1440 of 1500:   Train Loss = 0.6230   Validation Loss = 0.7234
Epoch 1500 of 1500:   Train Loss = 0.6230   Validation Loss = 0.7260
```



0.4 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()
```