Assignment 2 - Fitting Neural Networks (v2)

Summary of major changes for version 2

1. In Problem 1, "Step 1" of creating data, I have changed the code to generate 1500 data samples. The underlying function is also somewhat simpler
2. Corrected problem 1(b) I have updated all the default parameters of the neural network to be more appropriate to the problem.
3. In problem 1(c,d) I have updated the parameter sets that you should explore in the optimization of your model to make them more interesting.
4. In problem 2 I have corrected a few typos.
5. In problem 2(b), I no longer ask you to compare your model direclty against a scikit learn model on the same plots.

Aspen Morgan

Netid: 790907699

Note: this assignment falls under collaboration Mode 2: Individual Assignment – Collaboration Permitted. Please refer to the syllabus for additional information.

Total points in the assignment add up to 80

Learning objectives

Through completing this assignment you will be able to...

1. Identify key hyperparameters in neural networks and how they can impact model training and fit
2. Build, tune the parameters of, and apply feed-forward neural networks to data
3. Implement and explain each and every part of a standard fully-connected neural network and its operation including feed-forward propagation, backpropagation, and gradient descent.
4. Apply a standard neural network implementation and search the hyperparameter space to select optimized values.
5. Develop a detailed understanding of the math and practical implementation considerations of neural networks, one of the most widely used machine learning tools, so that it can be leveraged for learning about other neural networks of different model architectures.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor
```

1

[60 points] Exploring and optimizing neural network hyperparameters

Neural networks have become ubiquitous in the machine learning community, demonstrating exceptional performance over a wide range of supervised learning tasks. The benefits of these techniques come at a price of increased computational complexity and model designs with increased numbers of hyperparameters that need to be correctly set to make these techniques work. It is common that poor hyperparameter choices in neural networks result in significant decreases in model generalization performance. The goal of this exercise is to better understand some of the key hyperparameters you will encounter in practice using neural networks so that you can be better prepared to tune your model for a given application. Through this exercise, you will explore two common approaches to hyperparameter tuning a manual approach where we greedily select the best individual hyperparameter (often people will pick potentially sensible options, try them, and hope it works).

To explore this, we'll be using the example data created below throughout this exercise and the various training, validation, test splits. We will select each set of hyperparameters for our greedy/manual approach, then retrain on the combined training and validation data before finally evaluating our generalization performance for our final model on the test data.

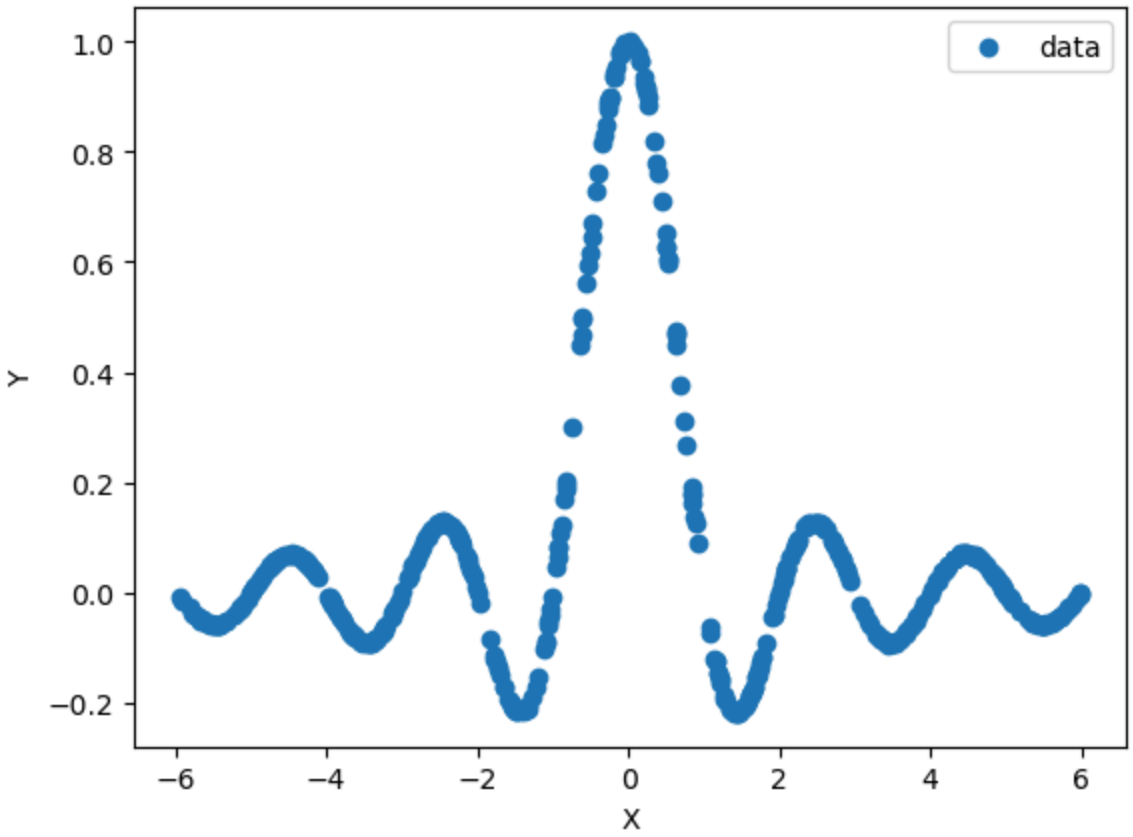
In [ ]:
```
# Optional for clear plotting on Macs
# %config InlineBackend.figure_format='retina'

# Some of the network training leads to warnings. When we know and are OK wi
#  what's causing the warning and simply don't want to see it, we can use th
#  following code. Run this block
#  to disable warnings
import sys
import os
import warnings

if not sys.warnoptions:
    warnings.simplefilter("ignore")
    os.environ["PYTHONWARNINGS"] = 'ignore'
```

Step 1: Create the data

In [ ]:
```
def makeSincData(n,noise=0.03):
  # Define the grid of points
  x = np.random.uniform(-6,6,n)

  # Calculate the sinc function
  y = np.sinc(np.sqrt((x)**2))+np.random.normal(0, noise, n)

  return (x,y)

# Number of samples
n = 500
noise = 0.001

# Validation and test data
x_train,y_train = makeSincData(n, noise)
x_val,y_val = makeSincData(n, noise)
x_test,y_test = makeSincData(n, noise)
```

To help get you started we should always begin by visualizing our training data, here's some code that does that:

In [ ]:
```
# Visualize the data
plt.scatter(x_train,y_train)
plt.xlabel('X')
plt.ylabel('Y')
plt.legend(['data'])
plt.show()
```



The hyperparameters we want to explore control the architecture of our model and how our model is fit to our data. These hyperparameters include the (a) learning rate, (b) batch size, and the (c) model architecture hyperparameters (the number of layers and the number of nodes per layer). We'll explore each of these and determine an optimized configuration of the network for this problem through this exercise. For all of the settings we'll explore and just, we'll assume the following default hyperparameters for the model (we'll use scikit learn's `MLPRegressor` as our neural network model):

- `learning_rate` ='constant', (we don't change learning rate during training)
- `learning_rate_init` = 0.01
- `hidden_layer_sizes` = (20,20) (two hidden layers, each with 20 nodes)
- `alpha` = 0 (regularization penalty - we have not discussed this in class yet)
- `solver` = 'adam' (the adam optimizer)
- `tol` = 1e-5 (this sets the convergence tolerance)
- `early_stopping` = False (this prevents early stopping)
- `activation` = 'relu' (rectified linear unit)
- `n_iter_no_change` = 1000 (this prevents early stopping)
- `batch_size` = 100 (size of the minibatch for adam)
- `max_iter` = 2000 (maximum number of epochs, which is how many times each data point will be used, not the number of gradient steps)

This default setting is our initial guess of what good values may be. Notice there are many model hyperparameters in this list: any of these could potentially be options to

search over. We constrain the search to those hyperparameters that are known to have a significant impact on model performance.

(a) Visualize the impact of different hyperparameter choices on classifier decision boundaries. Visualize the impact of different hyperparameter settings. Starting with the default settings above make the following changes (only change one hyperparameter at a time). For each hyperparameter value, plot the predictions made by your trained neural network; use 100 uniformly spaced x-values and pass them through your neural network to get y predictions. Use these 100 (x, y) pairs to create a line plot, and then on the same axes scatter plot the training data (note: you will need to train the model once for each parameter value, resulting in a separate plot for each trained model):

1. Vary the architecture (`hidden_layer_sizes`) by changing the number of nodes per layer while keeping the number of layers constant at 2: (5,5), (20,20), (50,50). Here (X,X) means a 2-layer network with X nodes in each layer.
2. Vary the learning rate: 0.001, 0.01, 0.1
3. Vary the batch size: 20, 100, 500

As you're exploring these settings, visit this website, the Neural Network Playground, which will give you the chance to interactively explore the impact of each of these parameters on a similar dataset to the one we use in this exercise. The tool also allows you to adjust the learning rate, batch size, regularization coefficient, and the architecture and to see the resulting decision boundary and learning curves. You can also visualize the model's hidden node output and its weights, and it allows you to add in transformed features as well. Experiment by adding or removing hidden layers and neurons per layer and vary the hyperparameters.

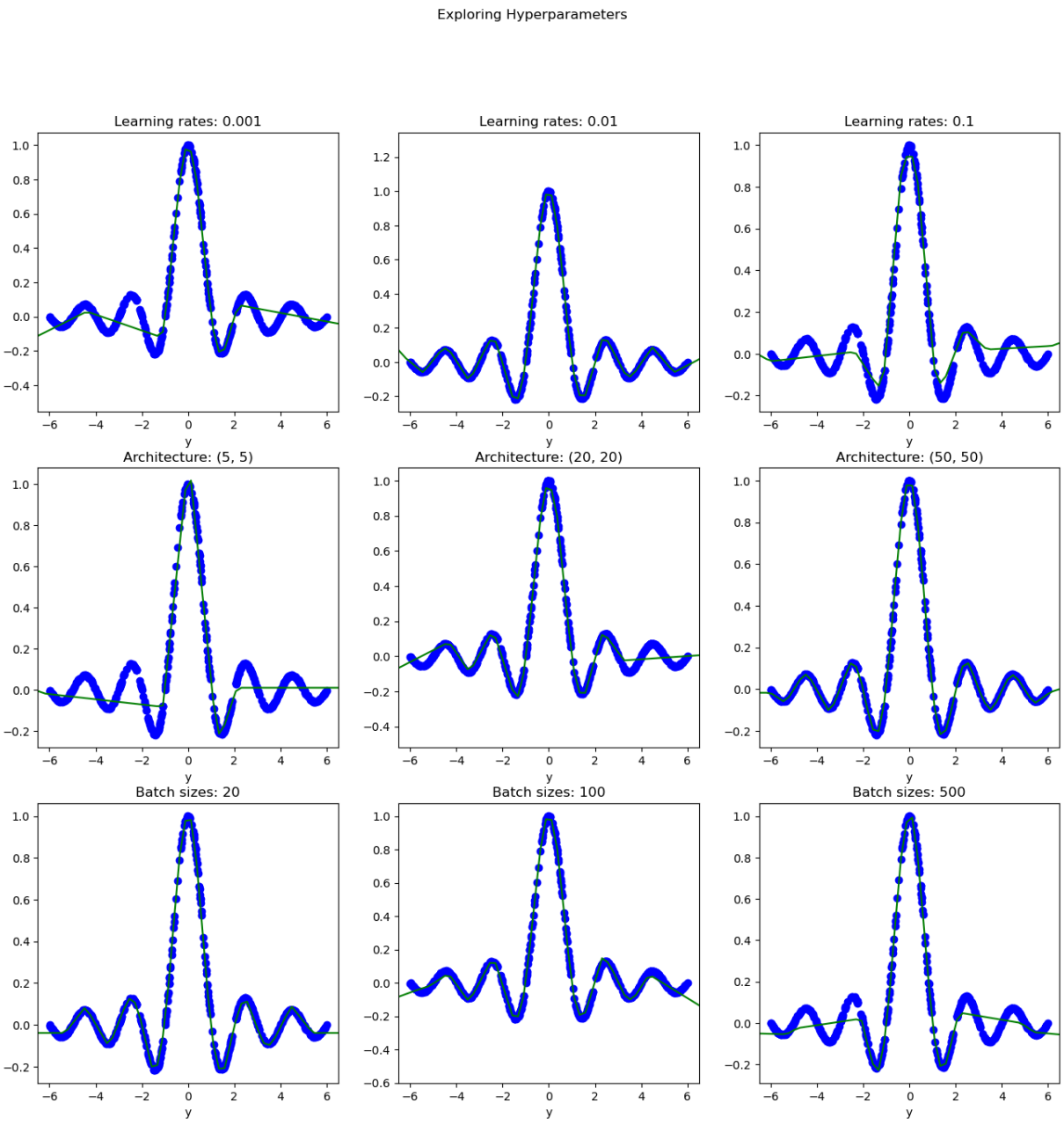
ANSWER

In [ ]:
```
# add empty bias term to x_train and to generated samples
x_train_df = pd.DataFrame({0: np.ones(len(x_train)), 1: x_train})
x_val_df = pd.DataFrame({0: np.ones(len(x_val)), 1: x_val})
x_samples = np.linspace(-6, 6, 100)
x_samples_df = pd.DataFrame({0: np.ones(len(x_samples)), 1: x_samples})
x_train_val_df = pd.concat([x_train_df, x_val_df])
x_test_df = pd.DataFrame({0: np.ones(len(x_test)), 1: x_test})
```

In [ ]:
```
learning_rates = [0.001, 0.01, 0.1]
architectures = [(5,5), (20,20), (50,50)]
batch_sizes = 20, 100, 500

fig, ax = plt.subplots(3, 3, figsize=(16,15))
for x in range(len(learning_rates)):
    nn = MLPRegressor(learning_rate_init=learning_rates[x],
                      hidden_layer_sizes=(20,20),
                      alpha=0,
                      solver='adam',
                      tol=1e-5,
```

```
                        early_stopping=False,
                        activation='relu',
                        n_iter_no_change=1000,
                        batch_size=100,
                        max_iter=2000)
    nn.fit(x_train_df, y_train.ravel())
    y_samples_pred = np.array((nn.predict(x_samples_df))).ravel()
    ax[0][x].plot(x_samples, y_samples_pred, label='Predicted', color='greer
    ax[0][x].set_xlim(-6.5, 6.5)
    ax[0][x].scatter(x_train, y_train, label='Training data', color='blue')
    ax[0][x].set_title('Learning rates: ' + str(learning_rates[x]))
    ax[0][x].set_xlabel('x')
    ax[0][x].set_xlabel('y')

for x in range(len(architectures)):
    nn = MLPRegressor(learning_rate_init=0.01,
                        hidden_layer_sizes=architectures[x],
                        alpha=0,
                        solver='adam',
                        tol=1e-5,
                        early_stopping=False,
                        activation='relu',
                        n_iter_no_change=1000,
                        batch_size=100,
                        max_iter=2000)
    nn.fit(x_train_df, y_train.ravel())
    y_samples_pred = np.array((nn.predict(x_samples_df))).ravel()
    ax[1][x].plot(x_samples, y_samples_pred, label='Predicted', color='greer
    ax[1][x].set_xlim(-6.5, 6.5)
    ax[1][x].scatter(x_train, y_train, label='Training data', color='blue')
    ax[1][x].set_title('Architecture: ' + str(architectures[x]))
    ax[1][x].set_xlabel('x')
    ax[1][x].set_xlabel('y')


for x in range(len(batch_sizes)):
    nn = MLPRegressor(learning_rate_init=0.01,
                        hidden_layer_sizes=(20, 20),
                        alpha=0,
                        solver='adam',
                        tol=1e-5,
                        early_stopping=False,
                        activation='relu',
                        n_iter_no_change=1000,
                        batch_size=batch_sizes[x],
                        max_iter=2000)
    nn.fit(x_train_df, y_train.ravel())
    y_samples_pred = np.array((nn.predict(x_samples_df))).ravel()
    ax[2][x].plot(x_samples, y_samples_pred, label='Predicted', color='greer
    ax[2][x].set_xlim(-6.5, 6.5)
    ax[2][x].scatter(x_train, y_train, label='Training data', color='blue')
    ax[2][x].set_title('Batch sizes: ' + str(batch_sizes[x]))
    ax[2][x].set_xlabel('x')
    ax[2][x].set_xlabel('y')
```

```
fig.suptitle('Exploring Hyperparameters')
fig.show()
```



(b) Manual (greedy) hyperparameter tuning I: manually optimize hyperparameters that govern the learning process, one hyperparameter at a time. Now with some insight into which settings may work better than others, let's more fully explore the performance of these different settings in the context of our validation dataset through a manual optimization process. Holding all else constant (with the default settings mentioned above), vary each of the following parameters as specified below. Train your algorithm on the training data, and evaluate the performance of your trained algorithm on the validation dataset using mean-squared-error (MSE). Create plots of MSE vs each parameter you vary (this will result in two plots). For full credit, make sure you label the axes, and use a logarithmic y-axis for your plots so it will be easierto see the small differences in MSE between the models.

1. Vary learning rate logarithmically from 10^{-4} to 10^{0} with 5 uniformly-spaced values
2. Vary the batch size over the following values: $[20, 50, 100, 200, 500]$

For each of these cases:

- Based on the results, report your optimal choices for each of these hyperparameters and why you selected them.
- Since neural networks can be sensitive to initialization values, you may notice these plots may be a bit noisy. Consider this when selecting the optimal values of the hyperparameters. If the noise seems significant, run the fit and score procedure multiple times and report the average. Rerunning the algorithm will change the initialization and therefore the output (assuming you do not set a random seed for that algorithm).
- Use the chosen hyperparameter values as the new default settings for section (c) and (d).

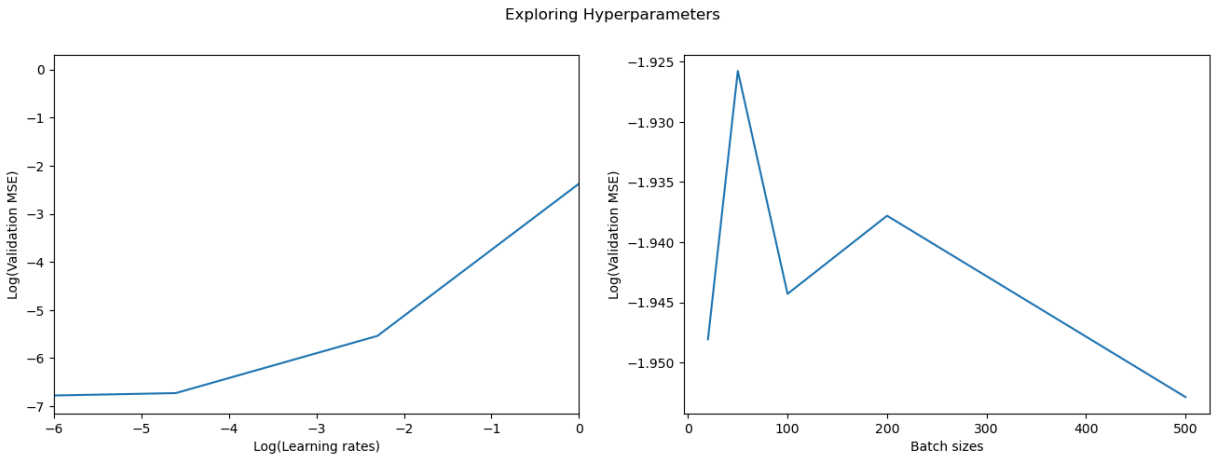
ANSWER

In [ ]:
```
# I extended the learning rates a tad to see edge behavior
learning_rates = np.logspace(-6, 0, 7)
batch_sizes = [20,50,100,200,500]

mse_lr, mse_bs = [], []

for x in range(len(learning_rates)):
    nn = MLPRegressor(learning_rate_init=learning_rates[x],
                      hidden_layer_sizes=(20,20),
                      alpha=0,
                      solver='adam',
                      tol=1e-5,
                      early_stopping=False,
                      activation='relu',
                      n_iter_no_change=1000,
                      batch_size=100,
                      max_iter=2000)
    nn.fit(x_train_df, y_train.ravel())
    y_val_pred = np.array((nn.predict(x_val_df))).ravel()
    mse_lr.append(mean_squared_error(y_val, y_val_pred))

for x in range(len(batch_sizes)):
    nn = MLPRegressor(learning_rate_init=0.01,
                      hidden_layer_sizes=(20,20),
                      alpha=0,
                      solver='adam',
                      tol=1e-5,
                      early_stopping=False,
                      activation='relu',
                      n_iter_no_change=1000,
                      batch_size=batch_sizes[x],
                      max_iter=2000)
    nn.fit(x_train_df, y_train.ravel())
```

```
    y_val_pred = np.array((nn.predict(x_test_df))).ravel()
    mse_bs.append(mean_squared_error(y_val, y_val_pred))
```

In [ ]:
```
# plot results
fig, ax = plt.subplots(1, 2, figsize=(16,5))
fig.suptitle('Exploring Hyperparameters')
ax[0].plot(np.log(learning_rates), np.log(mse_lr))
ax[0].set_xlim(-6, 0)
ax[0].set_xlabel('Log(Learning rates)')
ax[0].set_ylabel('Log(Validation MSE)')
ax[1].plot(batch_sizes, np.log(mse_bs))
ax[1].set_xlabel('Batch sizes')
ax[1].set_ylabel('Log(Validation MSE)')
```

Out[ ]:
```
Text(0, 0.5, 'Log(Validation MSE)')
```



- Based on these graphs, the best learning rate in 1e-4 to 1 is 1e-4 and the best batch size is 100. The learning rate is best in a greedy sense because it minimizes the validation MSE when all else is held equal. The batch size of 100 is the best without making the batch sizes unusually large.

(c) Manual (greedy) hyperparameter tuning II: manually optimize hyperparameters that impact the model architecture. Next, we want to explore the impact of the model architecture on performance and optimize its selection. This means varying two parameters at a time instead of one as above. To do this, evaluate the MSE resulting from training the model using each pair of possible numbers of nodes per layer and number of layers from the lists below. We will assume that for any given configuration the number of nodes in each layer is the same (e.g. (2,2,2), which would be a 3-layer network with 2 hidden node in each layer and (10,10) are valid, but (2,5,3) is not because the number of hidden nodes varies in each layer). Use the manually optimized values for learning rate, regularization, and batch size selected from section (b).

- Number of nodes per layer: $[5, 10, 20, 50]$
- Number of hidden layers = $[1, 2, 3]$

Report the MSE of your model on the validation data. For plotting these results, use heatmaps to plot the data in two dimensions. To make the heatmaps, you can use [this code for creating heatmaps] https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap Be sure to include the numerical values of the logarithm of the MSE in each grid square, as shown in the linked example and label your x, y, and color axes as always. For these numerical values, round them to **2 decimal places**.

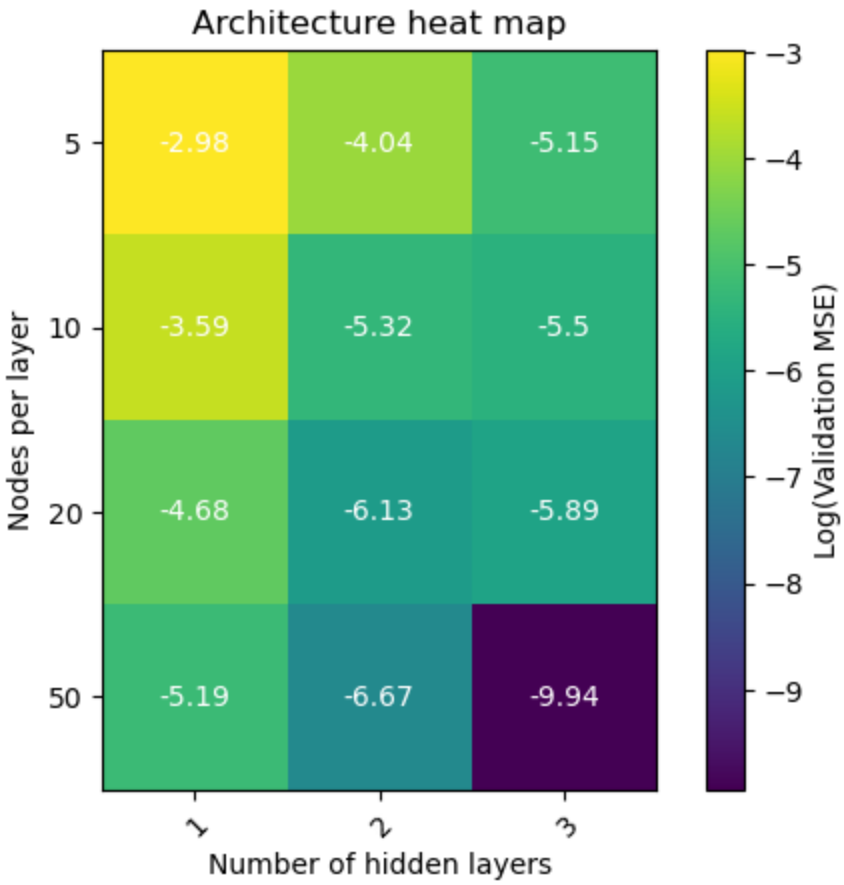
- When you select your optimized parameters, be sure to keep in mind that these values may be sensitive to the data and may offer the potential to have high variance for larger models. Therefore, select the model with the highest accuracy but lowest number of total model weights (all else equal, the simpler model is preferred).
- What do the results show? Which parameters did you select and why?

ANSWER

In [ ]:
```
mses, nodes, layers = [], [], []
nodes_per_layer = [5, 10, 20, 50]
for x in nodes_per_layer:
    hidden_layers = [(x), (x, x), (x, x, x)]
    for y in range(len(hidden_layers)):
        nn = MLPRegressor(learning_rate_init=1e-4,
                          hidden_layer_sizes=hidden_layers[y],
                          alpha=0,
                          solver='adam',
                          tol=1e-5,
                          early_stopping=False,
                          activation='relu',
                          n_iter_no_change=1000,
                          batch_size=100,
                          max_iter=2000)
        nn.fit(x_train_df, y_train.ravel())
        y_val_pred = np.array((nn.predict(x_val_df))).ravel()
        mses.append(mean_squared_error(y_val, y_val_pred))
        nodes.append(x)
        layers.append(y+1)
```

In [ ]:
```
# heat map
layers = [1, 2, 3]
mse_grid = np.array(np.round(np.log(mses), 2)).reshape(4, 3)
fig, ax = plt.subplots()
im = ax.imshow(mse_grid)

# Show all ticks and label them with the respective list entries
ax.set_yticks(np.arange(len(nodes_per_layer)), labels=nodes_per_layer)
ax.set_xticks(np.arange(len(layers)), labels=layers)

# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")
```

```
# Loop over data dimensions and create text annotations.
for i in range(len(nodes_per_layer)):
    for j in range(len(layers)):
        text = ax.text(j, i, mse_grid[i, j],
                       ha="center", va="center", color="w")

cbar = ax.figure.colorbar(im, ax=ax, label='Log(Validation MSE)')
ax.set_ylabel('Nodes per layer')
ax.set_xlabel('Number of hidden layers')

ax.set_title("Architecture heat map")
plt.show()
```



- These results show that 3 hidden layers with 50 nodes per layer minimizes the error.
- Since the reduction in the log error between 3 hidden layers with 50 nodes each and the other options, I will use that architecture.

(d) Manual (greedy) model selection and retraining. Based the optimal choice of hyperparameters, train your model with your optimized hyperparameters on all the training data AND the validation data.

- Apply the trained model to the test data and report the accuracy of your final model on the test data, in terms of MSE
- Scatter plot the test data and make a line plot using 100 sampled predictions from your model over the x-domain of your test data

ANSWER

In [ ]:
```
# train and predict on all data
nn = MLPRegressor(learning_rate_init=1e-4,
                  hidden_layer_sizes=[50, 50, 50],
                  alpha=0,
                  solver='adam',
                  tol=1e-5,
                  early_stopping=False,
                  activation='relu',
                  n_iter_no_change=1000,
                  batch_size=100,
                  max_iter=2000)
nn.fit(x_train_val_df, np.concatenate([y_train, y_val]))
```

Out[ ]:
```
MLPRegressor(alpha=0, batch_size=100, hidden_layer_sizes=[50, 50, 50],
             learning_rate_init=0.0001, max_iter=2000, n_iter_no_change=1000,
             tol=1e-05)
```

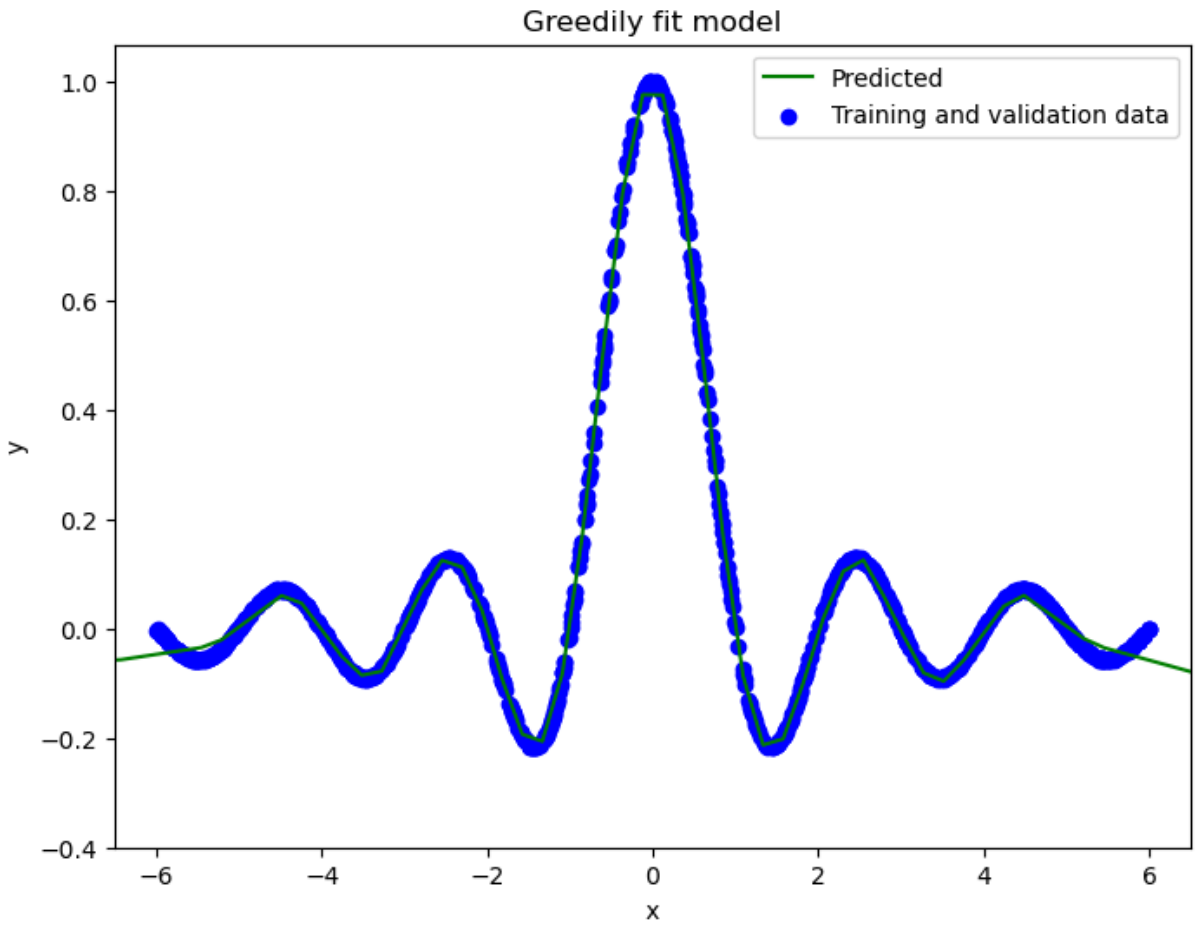
In [ ]:
```
y_samples_pred = np.array((nn.predict(x_samples_df))).ravel()
y_test_pred = np.array((nn.predict(x_test_df))).ravel()
```

In [ ]:
```
# print mse
print('Test MSE', mean_squared_error(y_test, y_test_pred))
```

```
Test MSE 0.00012711170980627738
```

In [ ]:
```
# plot
fig, ax = plt.subplots(1, 1, figsize=(8,6))
y_samples_pred = np.array((nn.predict(x_samples_df))).ravel()
ax.plot(x_samples, y_samples_pred, label='Predicted', color='green')
ax.set_xlim(-6.5, 6.5)
ax.scatter(np.concatenate([x_train, x_val]), np.concatenate([y_train, y_val]
ax.set_title('Greedily fit model')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
```

Out[ ]: <matplotlib.legend.Legend at 0x126ed0f50>



2

[20 points] Build and test your own Neural Network for regression

There is no better way to understand how one of the core techniques of modern machine learning works than to build a simple version of it yourself. In this exercise you will construct and apply your own neural network classifier. You may use numpy if you wish but no other libraries.

(a) Create a neural network class that follows the `scikit-learn` classifier convention by implementing `fit` and `predict`. Your `fit` method should run backpropagation on your training data using stochastic gradient descent (SGD, not ADAM!). Assume the activation function is a ReLU. Choose your model architecture to have one input node, two hidden layers with 10 nodes each, and one output node.

To guide you in the right direction with this problem, please find a skeleton of a neural network class below. You absolutely MAY use additional methods beyond those

suggested in this template, but the methods listed below are the minimum required to implement the model cleanly.

Strategies for debugging. One of the greatest challenges of this implementations is that there are many parts and a bug could be present in any of them. Here are some recommended tips:

- *Development environment*. Consider using an Integrated Development Environment (IDE). I strongly recommend the use of VS Code and the Python debugging tools in that development environment.
- *Unit tests*. You are strongly encouraged to create unit tests for most modules. Without doing this will make your code extremely difficult to debug. You can create simple examples to feed through the network to validate it is correctly computing activations and node values. Also, if you manually set the weights of the model, you can even calculate backpropagation by hand for some simple examples (admittedly, that unit test would be challenging and is optional, but a unit test is possible).
- *Compare against a similar architecture*. You can also verify the performance of your overall neural network by comparing it against the `scikit-learn` implementation and using the same architecture and parameters as your model (your model outputs will certainly not be identical, but they should be somewhat similar for similar parameter settings).

(b) Apply your neural network.

- Utilize the training and validation datasets from problem 1
- Train and test your model on this dataset plotting your learning curves (training and validation error for each epoch of stochastic gradient descent, where an epoch represents having trained on each of the training samples one time).
- Tune the learning rate and number of training epochs for your model to improve performance as needed.
- In two subplots, plot the training data on one subplot, and the validation data on the other subplot. Use the strategy from problem 1 for plotting a trained model (e.g., create 100 uniformly spaced x values and pass them through the models).
- Report the MSE of your trained MLP. How does the performance of your MLP compare to similar scikit models you trained in problem 1? Are they the same, very similar, or very different? Why?

ANSWER

In [ ]:
```
class myNeuralNetwork(object):
    def __init__(self, n_in=1, n_out=1, n_layer1=10, n_layer2=10, learning_r
                 max_epochs=200, batch_sizes=100):
        self.n_in = n_in # input dimensions
        self.n_out = n_out # output dimensions
        self.n_layer1 = n_layer1
        self.n_layer2 = n_layer2
```

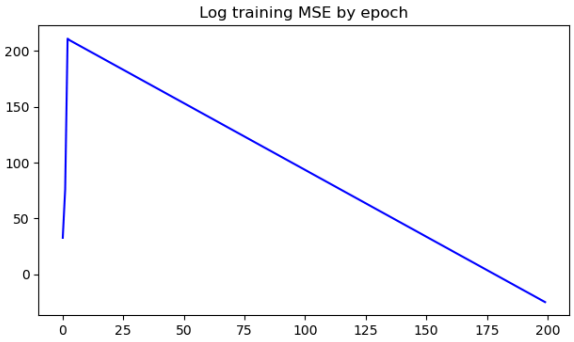
```
        self.learning_rate = learning_rate # learning rate
        self.max_epochs = max_epochs # maximum epochs
        self.batch_sizes = batch_sizes # batch sizes

        # w[i][j][k] := layer i, to node j, from node k where k_0 is bias we
        self.w = [np.random.randn(n_layer1, n_in + 1),
                  np.random.randn(n_layer2, n_layer1 + 1),
                  np.random.randn(n_out, n_layer2 + 1)]

    def forward_propagation(self, x):
        # process input layer
        f = [[]]
        h = [[]]
        h[0].append(x)
        for out_node in range(len(self.w[0])):
            f[0].append((np.dot(self.w[0][out_node][1:], x) + self.w[0][out_

        # process hidden layers
        for layer in range(1, 3):
            h.append([])
            f.append([])

            # h depends on previous layer size
            for in_node in range(len(self.w[layer-1])):
                h[layer].append(max(0.0, f[layer-1][in_node]))

            # f depends on current layer size
            for out_node in range(len(self.w[layer])):
                f[layer].append(np.dot(self.w[layer][out_node][1:], h[layer]

        y_hat = f[-1]
        self.f = f
        self.h = h

        return y_hat # return the estimate

    def compute_loss(self, X, y):
        # mean squared error:
        y_pred = []
        for x in X:
            y_pred.append(self.forward_propagation(x))
        return (y - y_pred).sum()**2/len(y)

    def backpropagate(self, x, y):
        # run forward pass
        y_hat = self.forward_propagation(x)

        # deltas (dl/df_i) need length f
        delta = [0] * len(self.f)
        last = len(self.f) - 1

        # last delta is derivative of error between predicted (f_last) and a
        delta[last] = y_hat - y

        l_sizes = [self.n_layer1, self.n_layer2, self.n_out]
        # compute deltas (dl/df) backwards
```

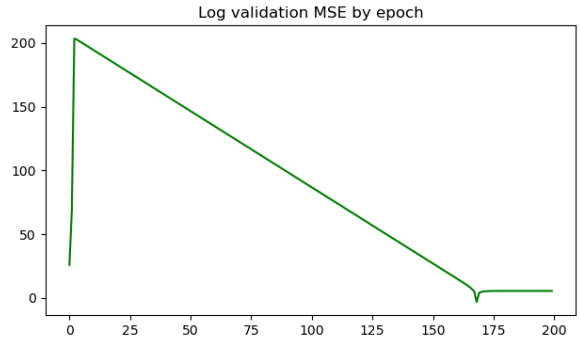
```
        for layer in np.flip(range(len(delta) - 1)):
            delta[layer] = []
            for left_node in range(l_sizes[layer]):
                if self.f[layer][left_node] <= 0: dhdf = 0
                else: dhdf = 1

                summation = 0.0
                for right_node in range(l_sizes[layer + 1]):
                    summation += delta[layer+1][right_node]*self.w[layer+1][

                delta[layer].append(summation * dhdf)

        # compute weight derivatives (dl/dw)
        l_sizes = [self.n_in, self.n_layer1, self.n_layer2, self.n_out]
        dldw = []
        for layer in range(3):
            dldw.append([])
            # dldw = delta from next layer * h from last layer
            for right_node in range(l_sizes[layer + 1]):
                    dldw[layer].append([])
                    dldw[layer][right_node].append(delta[layer][right_node])

                    # there are h's for each node besides biases
                    for left_node in range(l_sizes[layer]):
                        dldw[layer][right_node].append(delta[layer][right_nc

        # return updated gradients
        self.dldw = dldw

    def stochastic_gradient_descent_step(self, X_sub, y_sub):
        batch_grad = []
        for layer in range(len(self.w)):
           batch_grad.append(np.zeros_like(self.w[layer]))

        for index in range(len(X_sub)):
            self.backpropagate(X_sub[index], y_sub[index]) # updates dldws
            for layer in range(len(self.dldw)):
                batch_grad[layer] += self.dldw[layer]

        # update weights by layer using sgd
        for layer in range(len(self.w)):
            self.w[layer] = self.w[layer] - self.learning_rate*batch_grad[la

    def fit(self, X, y):
        # initialize empty arrays for error
        train_error = []
        val_error = []

        # remove some samples for validation
        sample = round(len(X)*0.1)
        select = np.random.choice(np.arange(len(X)), sample, replace=False)
        X_v = X[select] # sub array to run sgd on
        y_v = y[select]
        X = np.delete(X, select)
        y = np.delete(y, select)
```

```
        # continue fitting until reached max epochs
        epoch = 0
        while epoch < self.max_epochs:
            # elements that haven't been used in this epoch
            X_left = np.copy(X)
            y_left = np.copy(y)

            while len(X_left) > 0:
                # sample without replacement based on batch size or whats le
                sample = max(self.batch_sizes, len(X_left))
                select = np.random.choice(np.arange(len(X_left)), sample, re
                X_sub = X_left[select] # sub array to run sgd on
                y_sub = y_left[select]

                # run stochastic gradient descent
                self.stochastic_gradient_descent_step(X_sub, y_sub)

                # remove elements from list in this epoch
                X_left = np.delete(X_left, select)
                y_left = np.delete(y_left, select)

            # get epochs error
            train_error.append(self.compute_loss(X, y))
            val_error.append(self.compute_loss(X_v, y_v))

            epoch += 1 # increment epoch counter


        return train_error, val_error

    def predict(self, X):
        y_hat = []
        for x in X:
           y_hat.append(self.forward_propagation(x))

        return np.array(y_hat)
```

In [ ]:

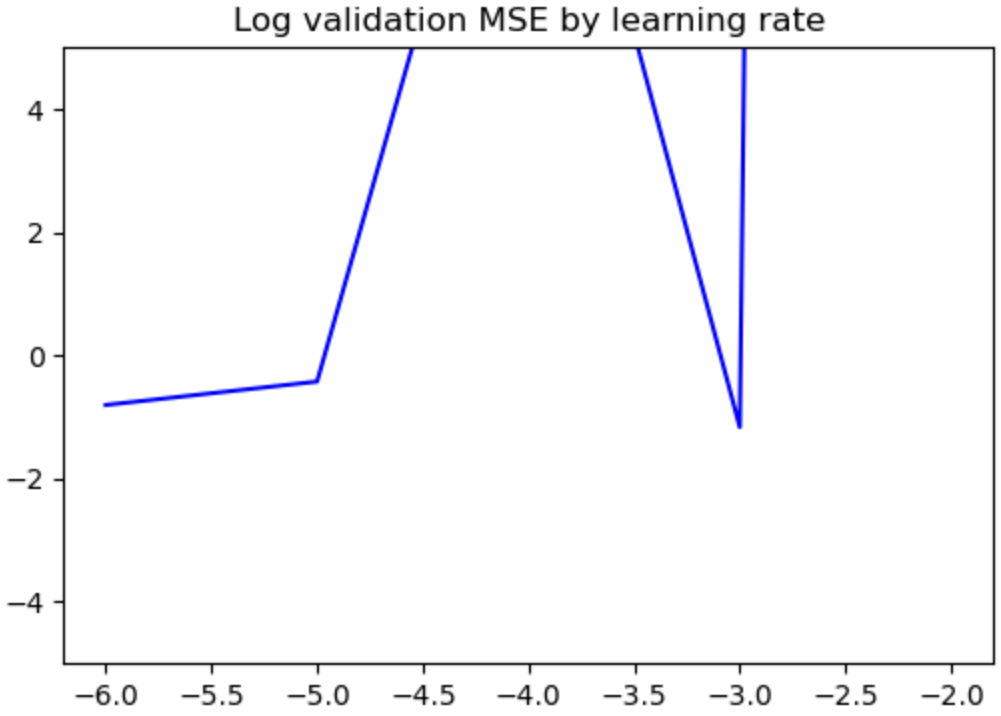
```
# plot training and validation error using defaults
nn = myNeuralNetwork()
train_error, val_error = nn.fit(x_train, y_train)

# plot learning curves (test max_epoch sizes)
fig, ax = plt.subplots(1, 2, figsize=(16, 4))
ax[0].plot(np.arange(200), np.log(train_error), color='blue')
ax[0].set_title('Log training MSE by epoch')
ax[1].plot(np.arange(200), np.log(val_error), color='green')
ax[1].set_title('Log validation MSE by epoch');
```





In [ ]:
```
# greedily fit learning rate
lrs = [0.000001, 0.00001, 0.0001, 0.001, 0.01]
mse_lr = []
for lr in lrs:
    nn = myNeuralNetwork(max_epochs=200, learning_rate=lr, batch_sizes=100)
    nn.fit(x_train, y_train)
    y_val_pred = nn.predict(x_val)
    mse_lr.append(mean_squared_error(y_val, y_val_pred))
```

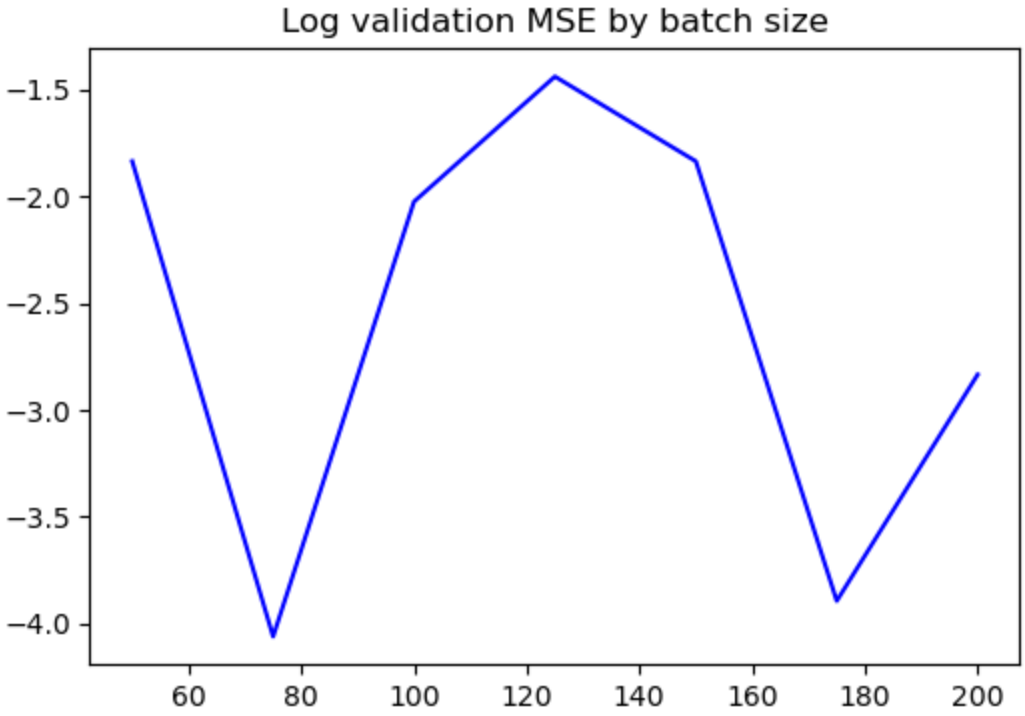
In [ ]:
```
# plot
fig, ax = plt.subplots(1, 1, figsize=(6, 4))
ax.plot(np.log10(lrs), np.log10(mse_lr), color='blue')
ax.set_ylim(-5, 5)
ax.set_title('Log validation MSE by learning rate');
```



In [ ]:
```
# greedily find batch size
bs_mses = []
for bs in [50, 75, 100, 125, 150, 175, 200]:
    nn = myNeuralNetwork(max_epochs=200, learning_rate=0.00001, batch_sizes=
    nn.fit(x_train, y_train)
```

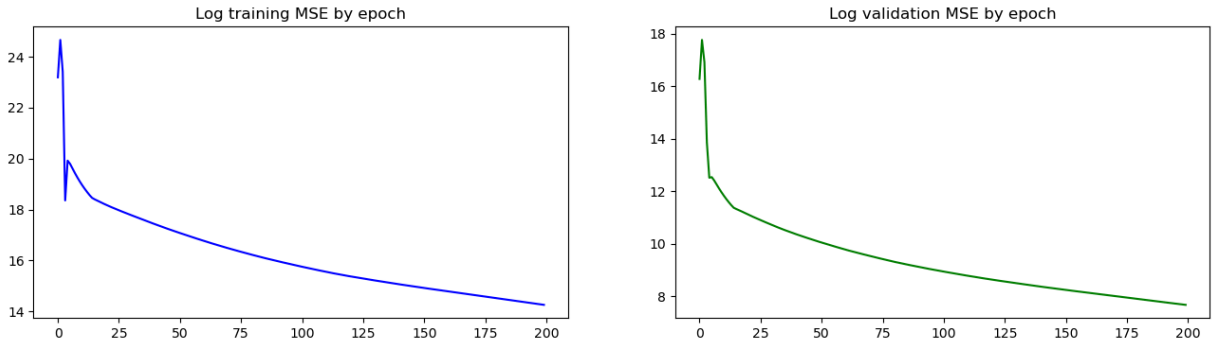
```
    y_val_pred = nn.predict(x_val)
    bs_mses.append(mean_squared_error(y_val, y_val_pred))
```

In [ ]:
```
# plot
fig, ax = plt.subplots(1, 1, figsize=(6, 4))
ax.plot([50, 75, 100, 125, 150, 175, 200], np.log(bs_mses), color='blue')
ax.set_title('Log validation MSE by batch size');
```



In [ ]:
```
# fit on training data and fit on validation data
nn = myNeuralNetwork(max_epochs=200, learning_rate=0.00001, batch_sizes=75)
train_error, val_error = nn.fit(x_train, y_train)
```

In [ ]:
```
# plot learning curves (test max_epoch sizes)
fig, ax = plt.subplots(1, 2, figsize=(16, 4))
ax[0].plot(np.arange(200), np.log(train_error), color='blue')
ax[0].set_title('Log training MSE by epoch')
ax[1].plot(np.arange(200), np.log(val_error), color='green')
ax[1].set_title('Log validation MSE by epoch');
```



In [ ]:
```
# updated parameters
me = 200
lr = 0.00001
```

```
bs = 75
nn = myNeuralNetwork(max_epochs=me, learning_rate=lr, batch_sizes=bs)

# fit on training data
nn.fit(x_train, y_train)
```

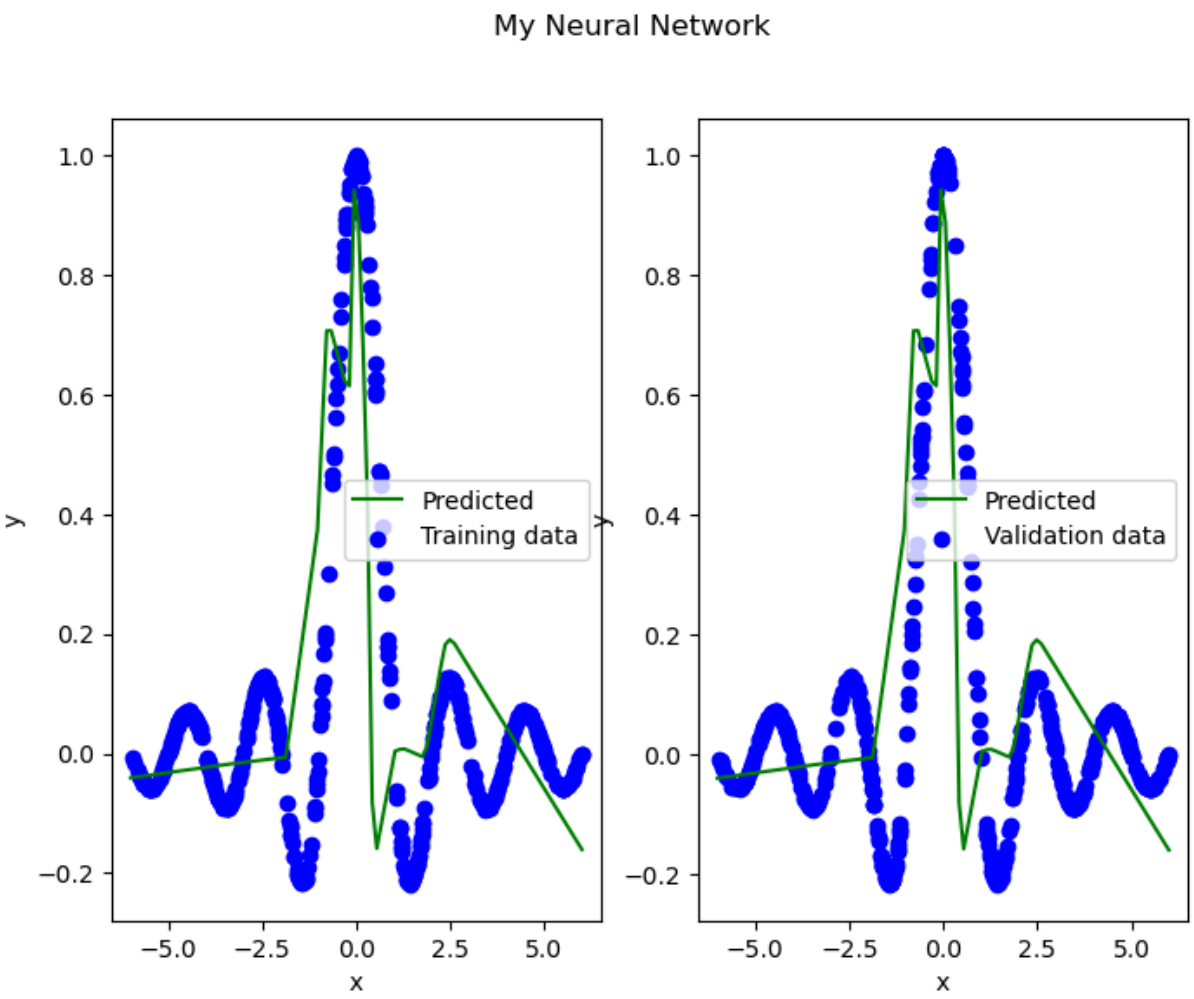
In [ ]:
```
# predict
y_test_pred = nn.predict(x_test)
```

In [ ]:
```
print('Test mean squared error: %2.3f' % mean_squared_error(y_test, y_test_p
print('Test root mean squared error: %2.3f' % np.sqrt(mean_squared_error(y_v
```

```
Test mean squared error: 0.044
Test root mean squared error: 0.363
```

In [ ]:
```
# predict on 100 samples
x_samples = np.linspace(-6, 6, 100)
y_samples_pred = nn.predict(x_samples)

# plot against training data
fig, ax = plt.subplots(1, 2, figsize=(8,6))
ax[0].plot(x_samples, y_samples_pred, label='Predicted', color='green')
ax[0].set_xlim(-6.5, 6.5)
ax[0].scatter(x_train, y_train, label='Training data', color='blue')
ax[0].set_xlabel('x')
ax[0].set_ylabel('y')
ax[0].legend()

ax[1].plot(x_samples, y_samples_pred, label='Predicted', color='green')
ax[1].set_xlim(-6.5, 6.5)
ax[1].scatter(x_val, y_val, label='Validation data', color='blue')
ax[1].set_xlabel('x')
ax[1].set_ylabel('y')
ax[1].legend()

fig.suptitle('My Neural Network')
```

Out[ ]:
```
Text(0.5, 0.98, 'My Neural Network')
```



Comparison of my model and sklearn:

My neural network is a lot worse. Some of this has to do with hidden layer sizes. In the first part, the best model had an architechture of [50, 50]. Since my model was limited to [10, 10], it makes sense it would have worse performance.

But ultimately, I think some of the decreased performance is some bug in my code. This was a nightmare to code using scalar formulas. In the future, I would try using matrix forms.