Assignment 4 - Neural Networks

BARBARA FLORES RIOS

Netid: bpf17

Names of students you worked with on this assignment: LIST HERE IF APPLICABLE (delete if not)

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

Instructions for all assignments can be found here.

Total points in the assignment add up to 90; an additional 10 points are allocated to presentation quality.

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.

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) as well as a random search of the hyperparameter space which as been shown to be an efficient way to achieve good hyperparameter values.

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 and the random search using a training/validation split, then retrain on the combined training and validation data before finally evaluating our generalization performance for both our final models 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 with
#  what's causing the warning and simply don't want to see it, we can use the
#  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'

from sklearn.neural_network import MLPClassifier
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.model_selection import RandomizedSearchCV
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from scipy.stats import loguniform

from sklearn.datasets import make_moons
```

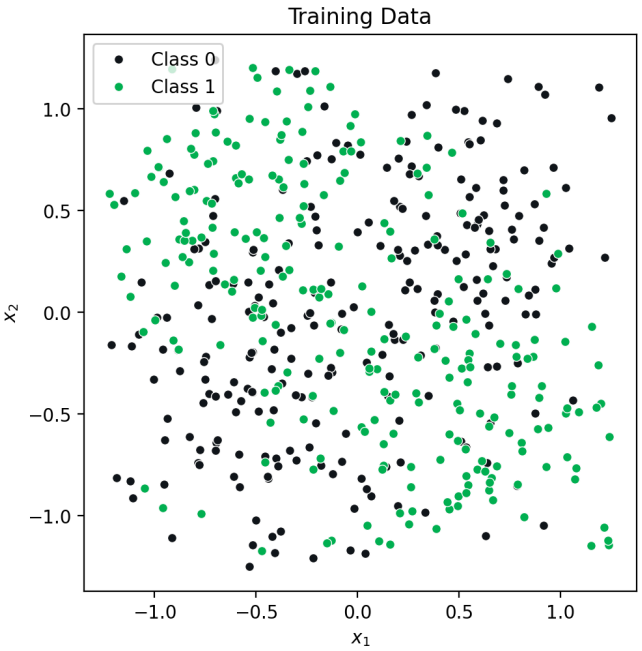
```
from sklearn.model_selection import train_test_split,  GridSearchCV
from sklearn.metrics import accuracy_score
```

In [ ]:
```
import numpy as np
from sklearn.model_selection import PredefinedSplit

# ------------------------------------------------------------------------
# Create the data
# ------------------------------------------------------------------------
# Data generation function to create a checkerboard-patterned dataset


def make_data_normal_checkerboard(n, noise=0):
    n_samples = int(n / 4)
    shift = 0.5
    c1a = np.random.randn(n_samples, 2) * noise + [-shift, shift]
    c1b = np.random.randn(n_samples, 2) * noise + [shift, -shift]
    c0a = np.random.randn(n_samples, 2) * noise + [shift, shift]
    c0b = np.random.randn(n_samples, 2) * noise + [-shift, -shift]
    X = np.concatenate((c1a, c1b, c0a, c0b), axis=0)
    y = np.concatenate((np.ones(2 * n_samples), np.zeros(2 * n_samples)))

    # Set a cutoff to the data and fill in with random uniform data:
    cutoff = 1.25
    indices_to_replace = np.abs(X) > cutoff
    for index, value in enumerate(indices_to_replace.ravel()):
        if value:
            X.flat[index] = np.random.rand() * 2.5 - 1.25
    return (X, y)


# Training datasets
np.random.seed(42)
noise = 0.45
X_train, y_train = make_data_normal_checkerboard(500, noise=noise)

# Validation and test data
X_val, y_val = make_data_normal_checkerboard(500, noise=noise)
X_test, y_test = make_data_normal_checkerboard(500, noise=noise)

# For RandomSeachCV, we will need to combine training and validation sets then
#  specify which portion is training and which is validation
# Also, for the final performance evaluation, train on all of the training AND validation data
X_train_plus_val = np.concatenate((X_train, X_val), axis=0)
y_train_plus_val = np.concatenate((y_train, y_val), axis=0)

# Create a predefined train/test split for RandomSearchCV (to be used later)
validation_fold = np.concatenate(
    (-1 * np.ones(len(y_train)), np.zeros(len(y_val))))
train_val_split = PredefinedSplit(validation_fold)
```

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

In [ ]:
```
import matplotlib.pyplot as plt

# Code to plot the sample data


def plot_data(ax, X, y, title, limits):
    # Select the colors to use in the plots
    color0 = "#121619"  # Dark grey
    color1 = "#00B050"  # Green
    color_boundary = "#858585"

    # Separate samples by class
    samples0 = X[y == 0]
    samples1 = X[y == 1]

    ax.plot(
        samples0[:, 0],
        samples0[:, 1],
        marker="o",
        markersize=5,
        linestyle="None",
        color=color0,
        markeredgecolor="w",
        markeredgewidth=0.5,
        label="Class 0",
    )
    ax.plot(
        samples1[:, 0],
        samples1[:, 1],
        marker="o",
        markersize=5,
        linestyle="None",
```

```
        color=color1,
        markeredgecolor="w",
        markeredgewidth=0.5,
        label="Class 1",
    )
    ax.set_title(title)
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")
    ax.legend(loc="upper left")
    ax.set_aspect("equal")


fig, ax = plt.subplots(constrained_layout=True, figsize=(5, 5))
limits = [-1.25, 1.25, -1.25, 1.25]
plot_data(ax, X_train, y_train, "Training Data", limits)
```



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) regularization coefficient, as well as the (d) 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 `MLPClassifier` as our neural network model):

- `learning_rate_init` = 0.03
- `hidden_layer_sizes` = (30,30) (two hidden layers, each with 30 nodes)
- `alpha` = 0 (regularization penalty)
- `solver` = 'sgd' (stochastic gradient descent 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` = 50 (size of the minibatch for stochastic gradient descent)
- `max_iter` = 500 (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.

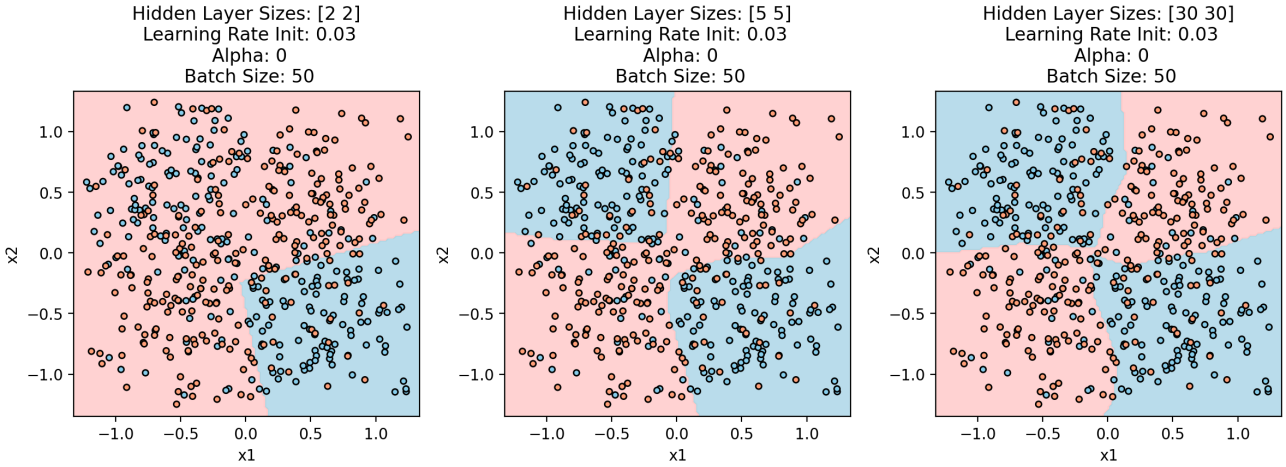
(a) Visualize the impact of different hyperparameter choices on classifier decision boundaries. Visualize the impact of different hyperparameter settings. Starting with the default settings above make the following changes (only change one hyperparameter at a time). For each hyperparameter value, plot the decision boundary on the training data (you will need to train the model once for each parameter value):

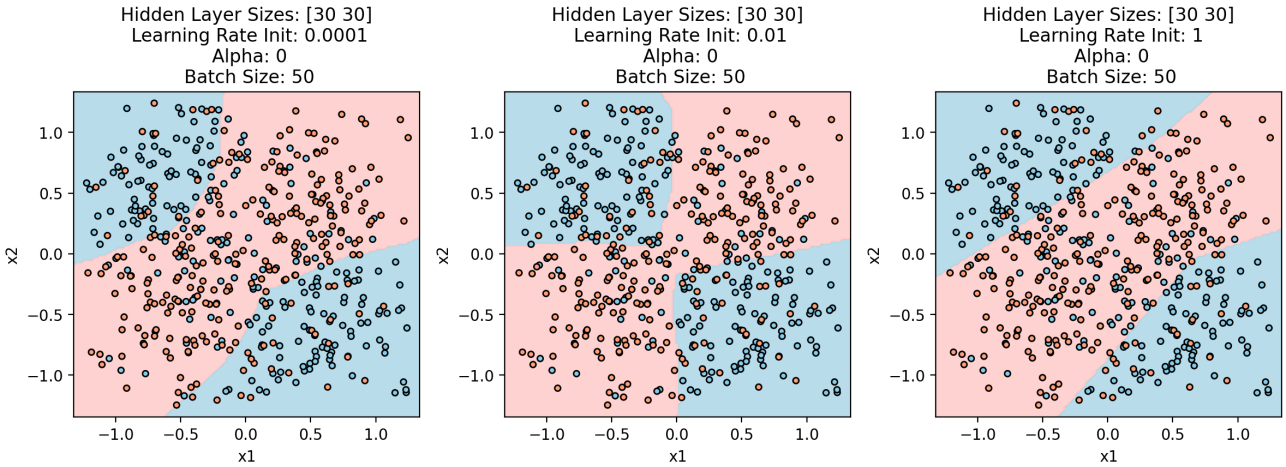
1. Vary the architecture (`hidden_layer_sizes`) by changing the number of nodes per layer while keeping the number of layers constant at 2: (2,2), (5,5), (30,30). Here (X,X) means a 2-layer network with X nodes in each layer.
2. Vary the learning rate: 0.0001, 0.01, 1
3. Vary the regularization: 0, 1, 10
4. Vary the batch size: 5, 50, 500

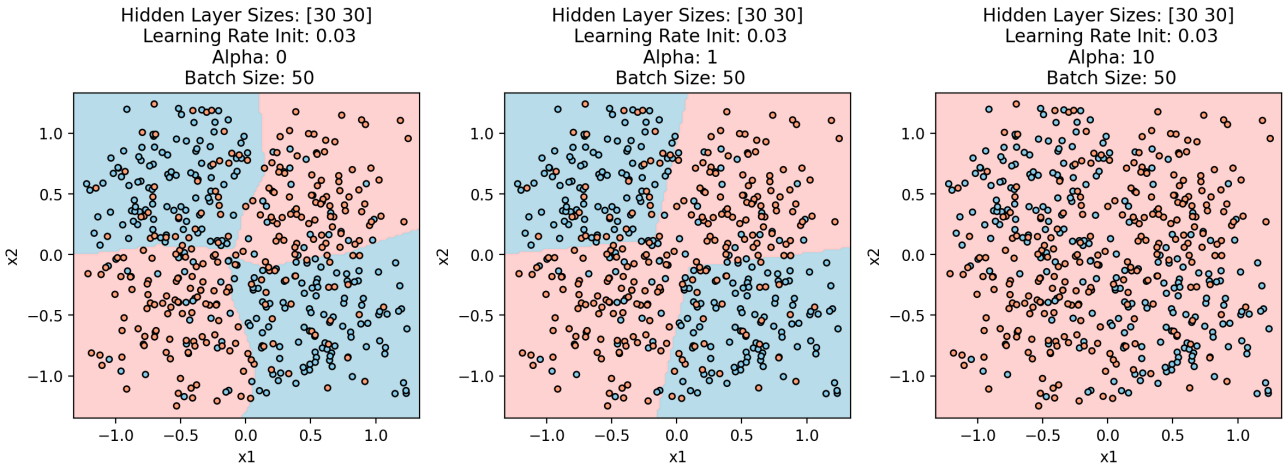
This should produce 12 plots, altogether. For easier comparison, please plot nodes & layers combinations, learning rates, regularization strengths, and batch sizes in four separate rows (with three columns each representing a different value for each of those hyperparameters).

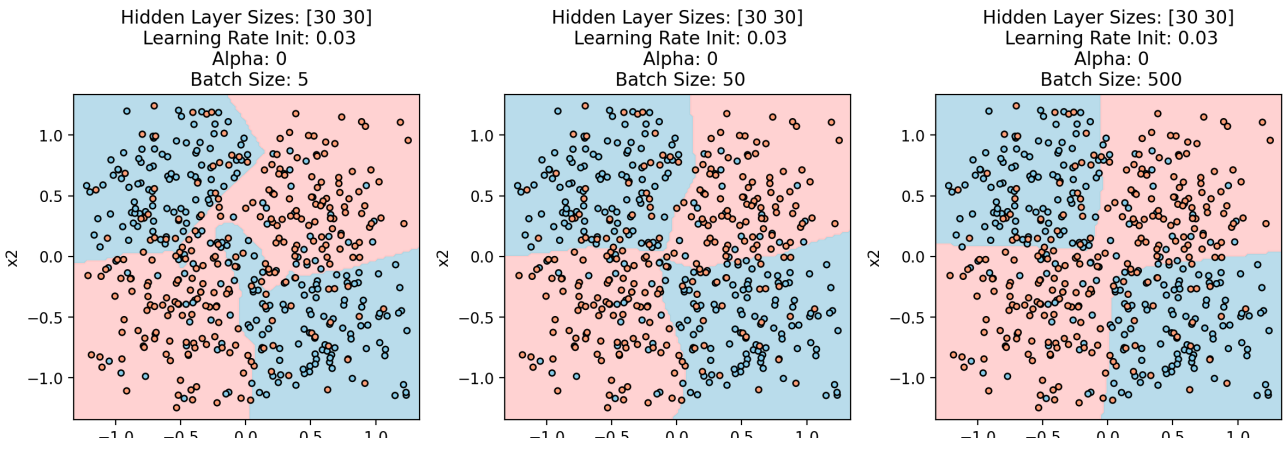
As you're exploring these settings, visit this website, the Neural Network Playground, which will give you the chance to interactively explore the impact of each of these parameters on a similar dataset to the one we use in this exercise. The tool also allows you to adjust the learning rate, batch size, 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.

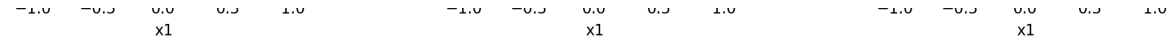
In [ ]:
```
boundary_colors = ListedColormap(["#FFCCCB", "#FFCCCB", "#ADD8E6"])
scatter_colors = ListedColormap(["#FFA07A", "#98FB98", "#87CEEB"])

hidden_layer_sizes = np.array([(2, 2), (5, 5), (30, 30)] + [(30, 30)] * 9)
learning_rates = [0.03] * 3 + [0.0001, 0.01, 1] + [0.03] * 6
alphas = [0] * 6 + [0, 1, 10] + [0] * 3
batch_sizes = [50] * 9 + [5, 50, 500]

fig, axs = plt.subplots(4, 3, figsize=(12, 18))

for i in range(4):
    for j in range(3):
        index = i * 3 + j
        hidden_layer_size = hidden_layer_sizes[index]
        learning_rate = learning_rates[index]
        alpha = alphas[index]
        batch_size = batch_sizes[index]

        clf = MLPClassifier(
            random_state=1,
            learning_rate_init=learning_rate,
            hidden_layer_sizes=hidden_layer_size,
            alpha=alpha,
            solver="sgd",
            tol=1e-5,
            early_stopping=False,
            activation="relu",
            n_iter_no_change=1000,
            batch_size=batch_size,
            max_iter=500,
        )

        clf.fit(X_train, y_train)

        x_min, x_max = X_train[:, 0].min() - 0.1, X_train[:, 0].max() + 0.1
        y_min, y_max = X_train[:, 1].min() - 0.1, X_train[:, 1].max() + 0.1

        xx, yy = np.meshgrid(
            np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02)
        )

        y_train_hat = clf.predict(
            np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

        axs[i, j].contourf(xx, yy, y_train_hat, alpha=0.8,
                           cmap=boundary_colors)
        axs[i, j].scatter(
            X_train[:, 0],
            X_train[:, 1],
            c=y_train,
            edgecolors="k",
            cmap=scatter_colors,
            s=15,
        )

        axs[i, j].set_title(
            f"\nHidden Layer Sizes: {hidden_layer_size}\n Learning Rate Init: {learning_rate}\n Alpha: {alpha}\n Batch Size:
        )
        axs[i, j].set_xlabel("x1")
        axs[i, j].set_ylabel("x2")

plt.tight_layout()
plt.show()
```


Hidden Layer Sizes: [2 2]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 50
Hidden Layer Sizes: [5 5]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 50
Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 50
x1
x2


Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.0001
Alpha: 0
Batch Size: 50
Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.01
Alpha: 0
Batch Size: 50
Hidden Layer Sizes: [30 30]
Learning Rate Init: 1
Alpha: 0
Batch Size: 50
x1
x2


Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 50
Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 1
Batch Size: 50
Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 10
Batch Size: 50
x1
x2


Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 5
Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 50
Hidden Layer Sizes: [30 30]
Learning Rate Init: 0.03
Alpha: 0
Batch Size: 500
x1
x2



> Looking at the preceding graphs, we can observe how different hyperparameter choices impact the classifier's decision boundaries. Additionally, it is important to consider that neural networks can be sensitive to initialization values, meaning that these results also depend on the seed used (in this case, random_state=1).
>
> This illustrates the sensitivity of neural networks to the selection of hyperparameters, presenting a challenge for hyperparameter selection that intensifies when attempting to find an optimal combination of various hyperparameters simultaneously. Hyperparameter optimization is a fundamental challenge in the design and training of neural network models, as it can significantly influence their performance and generalization ability.
>
> Systematically exploring different combinations of hyperparameters is crucial for finding the most suitable configuration for a specific problem. Furthermore, a deep understanding of how each hyperparameter affects the model's performance is essential for making informed decisions during the fine-tuning process.

(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. Here, overall accuracy is a reasonable performance metric since the classes are balanced and we don't weight one type of error as more important than the other; therefore, use the `score` method of the `MLPClassifier` for this. Create plots of accuracy vs each parameter you vary (this will result in three plots).

1. Vary learning rate logarithmically from 10^{-5} to 10^{0} with 20 steps
2. Vary the regularization parameter logarithmically from 10^{-8} to 10^{2} with 20 steps
3. Vary the batch size over the following values: $[1, 3, 5, 10, 20, 50, 100, 250, 500]$

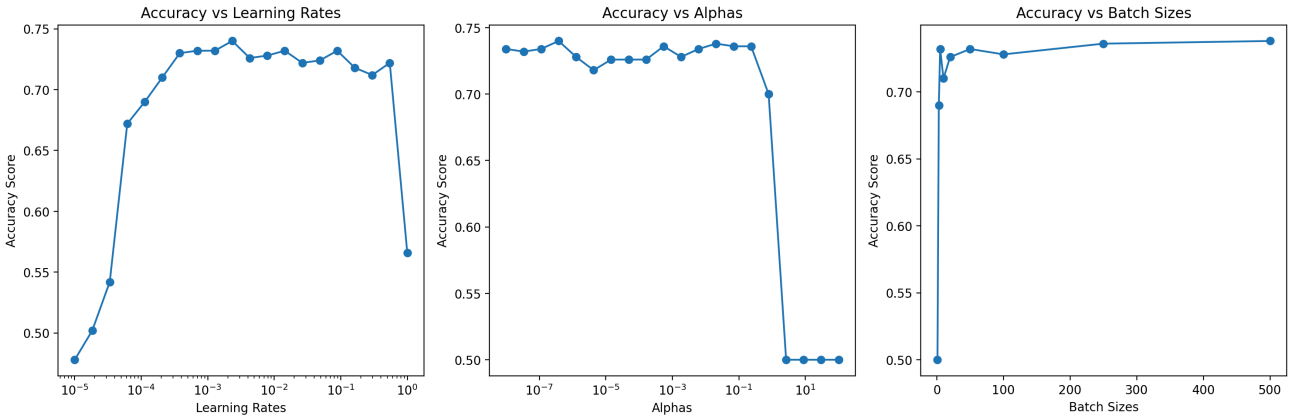
For each of these cases:

- 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 (without fixing a random seed) 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).

In [ ]:
```python
# learning_rates
learning_rates = np.logspace(np.log10(1e-5), np.log10(1e0), 20)
accuracy_scores_lr = []

for learning_rate in learning_rates:
    clf = MLPClassifier(
        learning_rate_init=learning_rate,
        hidden_layer_sizes=(30, 30),
        alpha=0,
        solver="sgd",
        tol=1e-5,
        early_stopping=False,
        activation="relu",
        n_iter_no_change=1000,
        batch_size=50,
        max_iter=500,
    )
    clf.fit(X_train, y_train)
    accuracy_score = clf.score(X_val, y_val)
    accuracy_scores_lr.append(accuracy_score)


# Alpha
alphas = np.logspace(np.log10(1e-8), np.log10(1e2), 20)
accuracy_scores_alpha = []

for alpha in alphas:
    clf = MLPClassifier(
        learning_rate_init=0.03,
        hidden_layer_sizes=(30, 30),
        alpha=alpha,
        solver="sgd",
        tol=1e-5,
        early_stopping=False,
        activation="relu",
        n_iter_no_change=1000,
        batch_size=50,
        max_iter=500,
    )
    clf.fit(X_train, y_train)
```

```
    accuracy_score = clf.score(X_val, y_val)
    accuracy_scores_alpha.append(accuracy_score)


# Batch Size
batch_sizes = [1, 3, 5, 10, 20, 50, 100, 250, 500]
accuracy_scores_bs = []

for batch_size in batch_sizes:
    clf = MLPClassifier(
        learning_rate_init=0.03,
        hidden_layer_sizes=(30, 30),
        alpha=0,
        solver="sgd",
        tol=1e-5,
        early_stopping=False,
        activation="relu",
        n_iter_no_change=1000,
        batch_size=batch_size,
        max_iter=500,
    )
    clf.fit(X_train, y_train)
    accuracy_score = clf.score(X_val, y_val)
    accuracy_scores_bs.append(accuracy_score)
```

In [ ]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# learning_rates
axs[0].plot(learning_rates, accuracy_scores_lr, marker="o")
axs[0].set_xscale("log")
axs[0].set_xlabel("Learning Rates")
axs[0].set_ylabel("Accuracy Score")
axs[0].set_title("Accuracy vs Learning Rates")

# alphas
axs[1].plot(alphas, accuracy_scores_alpha, marker="o")
axs[1].set_xscale("log")
axs[1].set_xlabel("Alphas")
axs[1].set_ylabel("Accuracy Score")
axs[1].set_title("Accuracy vs Alphas")

# batch_sizes
axs[2].plot(batch_sizes, accuracy_scores_bs, marker="o")
axs[2].set_xlabel("Batch Sizes")
axs[2].set_ylabel("Accuracy Score")
axs[2].set_title("Accuracy vs Batch Sizes")

plt.tight_layout()
plt.show()
```



> To observe how accuracy varies depending on the selection of hyperparameters including learning rate, regularization parameter, and batch size, we plot their variation while keeping everything else constant in our base model, as seen in the previous graph.
>
> However, since neural networks can be sensitive to initialization values, these graphs may exhibit some level of noise. Therefore, to select the optimal values of the hyperparameters, we run the fitting and scoring procedure 10 times and calculate the average, as shown in the following graph, which presents a smoother curve and provides a clearer view of the trend.

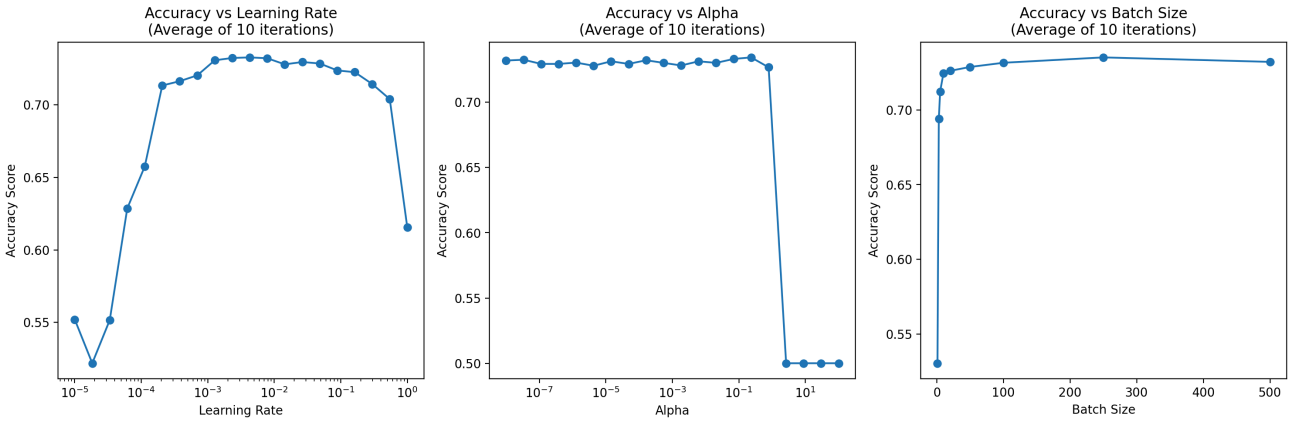
In [ ]:
```
# - Since neural networks can be sensitive to initialization values, you may notice these plots may be a bit noisy. Consider
# Rerunning the algorithm will change the initialization and therefore the output (assuming you do not set a random seed for

num_trials = 10

# learning_rates
```

```
learning_rates = np.logspace(np.log10(1e-5), np.log10(1e0), 20)
accuracy_scores_lr_avg = []

for learning_rate in learning_rates:
    accuracy_scores_lr = []

    for _ in range(num_trials):

        clf = MLPClassifier(
            learning_rate_init=learning_rate,
            hidden_layer_sizes=(30, 30),
            alpha=0,
            solver="sgd",
            tol=1e-5,
            early_stopping=False,
            activation="relu",
            n_iter_no_change=1000,
            batch_size=50,
            max_iter=500,
        )
        clf.fit(X_train, y_train)
        accuracy_score = clf.score(X_val, y_val)
        accuracy_scores_lr.append(accuracy_score)

    average_score = np.mean(accuracy_scores_lr)
    accuracy_scores_lr_avg.append(average_score)


# Alpha
alphas = np.logspace(np.log10(1e-8), np.log10(1e2), 20)
accuracy_scores_alpha_avg = []

for alpha in alphas:
    accuracy_scores_alpha = []
    for _ in range(num_trials):
        clf = MLPClassifier(
            learning_rate_init=0.03,
            hidden_layer_sizes=(30, 30),
            alpha=alpha,
            solver="sgd",
            tol=1e-5,
            early_stopping=False,
            activation="relu",
            n_iter_no_change=1000,
            batch_size=50,
            max_iter=500,
        )
        clf.fit(X_train, y_train)
        accuracy_score = clf.score(X_val, y_val)
        accuracy_scores_alpha.append(accuracy_score)
    average_score = np.mean(accuracy_scores_alpha)
    accuracy_scores_alpha_avg.append(average_score)


# Batch Size
batch_sizes = [1, 3, 5, 10, 20, 50, 100, 250, 500]
accuracy_scores_bs_avg = []

for batch_size in batch_sizes:
    accuracy_scores_bs = []
    for _ in range(num_trials):
        clf = MLPClassifier(
            learning_rate_init=0.03,
            hidden_layer_sizes=(30, 30),
            alpha=0,
            solver="sgd",
            tol=1e-5,
            early_stopping=False,
            activation="relu",
            n_iter_no_change=1000,
            batch_size=batch_size,
            max_iter=500,
        )
        clf.fit(X_train, y_train)
        accuracy_score = clf.score(X_val, y_val)
        accuracy_scores_bs.append(accuracy_score)
    average_score = np.mean(accuracy_scores_bs)
    accuracy_scores_bs_avg.append(average_score)
```

In [ ]:

```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# learning_rates
axs[0].plot(learning_rates, accuracy_scores_lr_avg, marker="o")
axs[0].set_xscale("log")
axs[0].set_xlabel("Learning Rate")
axs[0].set_ylabel("Accuracy Score")
```

```
axs[0].set_title(f"Accuracy vs Learning Rate\n(Average of {num_trials} iterations)")

# alphas
axs[1].plot(alphas, accuracy_scores_alpha_avg, marker="o")
axs[1].set_xscale("log")
axs[1].set_xlabel("Alpha")
axs[1].set_ylabel("Accuracy Score")
axs[1].set_title(f"Accuracy vs Alpha\n(Average of {num_trials} iterations)")

# batch_sizes
axs[2].plot(batch_sizes, accuracy_scores_bs_avg, marker="o")
axs[2].set_xlabel("Batch Size")
axs[2].set_ylabel("Accuracy Score")
axs[2].set_title(f"Accuracy vs Batch Size\n(Average of {num_trials} iterations)")

plt.tight_layout()
plt.show()
```



> From our 10 iterations, we obtained the following values of learning rate, regularization parameter, and batch size that maximized our accuracy on average.

In [ ]:
```
i_max_lr = np.argmax(accuracy_scores_lr_avg)
i_max_alpha = np.argmax(accuracy_scores_alpha_avg)
i_max_batch_size = np.argmax(accuracy_scores_bs_avg)

lr_max = learning_rates[i_max_lr]
alpha_max = alphas[i_max_alpha]
batch_size_max = batch_sizes[i_max_batch_size]

# save results
best_params_greedy = {
    "learning_rate_init": lr_max,
    "hidden_layer_sizes": None,
    "alpha": alpha_max,
    "batch_size": batch_size_max,
}

print(
    f"Maximum accuracy after {num_trials} iterations for learning rate: {lr_max:.4f} is {accuracy_scores_lr_avg[i_max_lr]:.3f}
)
print(
    f"Maximum accuracy after {num_trials} iterations for regularization parameter: {alpha_max:.3f} is {accuracy_scores_alpha_
)
print(
    f"Maximum accuracy after {num_trials} iterations for batch size: {batch_size_max} is {accuracy_scores_bs_avg[i_max_batch_s
)
```

```
Maximum accuracy after 10 iterations for learning rate: 0.0043 is 0.733
Maximum accuracy after 10 iterations for regularization parameter: 0.234 is 0.734
Maximum accuracy after 10 iterations for batch size: 250 is 0.735
```

> We will use the hyperparameter values of lr_max, alpha_max, and batch_size_max as new default settings for sections (c) and (d)

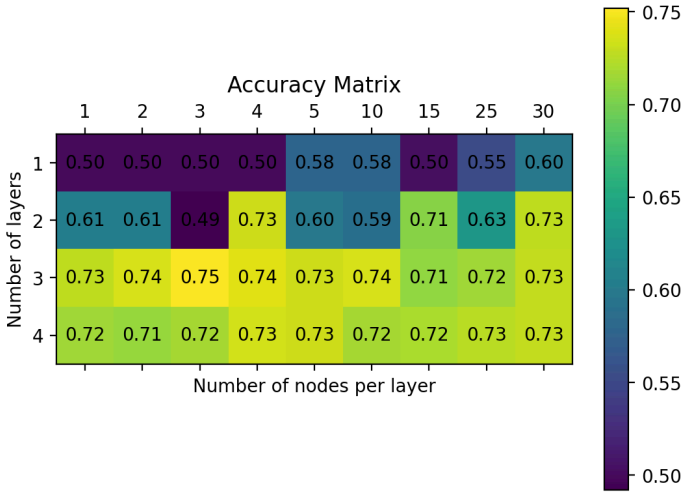
(c) Manual (greedy) hyperparameter tuning II: manually optimize hyperparameters that impact the model architecture. Next, we want to explore the impact of the model architecture on performance and optimize its selection. This means varying two parameters at a time instead of one as above. To do this, evaluate the validation accuracy 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 (25,25) 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: $[1, 2, 3, 4, 5, 10, 15, 25, 30]$

- Number of layers = $[1, 2, 3, 4]$ Report the accuracy 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.html). Be sure to include the numerical values of accuracy 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** (due to some randomness in the training process, any further precision is not typically meaningful).
- 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?

In [ ]:
```
n_nodes_list = [1, 2, 3, 4, 5, 10, 15, 25, 30]
n_layer_list = [1, 2, 3, 4]

accuracy_scores_layers = []

for n_ndes in n_nodes_list:
    for n_layer in n_layer_list:
        hidden_layer_sizes = tuple([n_ndes] * (n_layer))
        clf = MLPClassifier(
            random_state=1,
            learning_rate_init=best_params_greedy["learning_rate_init"],
            hidden_layer_sizes=hidden_layer_sizes,
            alpha=best_params_greedy["alpha"],
            solver="sgd",
            tol=1e-5,
            early_stopping=False,
            activation="relu",
            n_iter_no_change=1000,
            batch_size=best_params_greedy["batch_size"],
            max_iter=500,
        )
        clf.fit(X_train, y_train)
        accuracy_score = clf.score(X_val, y_val)
        accuracy_scores_layers.append(accuracy_score)

accuracy_matrix = np.array(accuracy_scores_layers).reshape(
    len(n_layer_list), len(n_nodes_list)
)
```

In [ ]:
```
fig, ax = plt.subplots()

cax = ax.matshow(accuracy_matrix, cmap="viridis")
ax.set_xticklabels([""] + n_nodes_list)
ax.set_yticklabels([""] + n_layer_list)

fig.colorbar(cax)

for i in range(len(n_layer_list)):
    for j in range(len(n_nodes_list)):
        plt.text(
            j,
            i,
            f"{accuracy_matrix[i, j]:.2f}",
            ha="center",
            va="center",
            color="black",
        )

plt.xlabel("Number of nodes per layer")
plt.ylabel("Number of layers")
plt.title("Accuracy Matrix")
plt.show()
```



> After manually exploring the optimization of hyperparameters that impact the model's architecture, by varying the number of nodes per layer between 1 and 30, and the number of layers between 1 and 4, using the hyperparameters learning_rate_init, batch_size, and alpha that showed the best individual accuracy performance in the previous question, we obtain the accuracy matrix previously plotted.
>
> Upon analyzing the results, we observed that `the highest accuracy, reaching 0.75, is achieved with 3 nodes per layer and 3 layers`. This set of hyperparameters was chosen due to its superior accuracy among the tested configurations. Additionally, another advantage of this architecture is its relative simplicity, resulting in a more computationally efficient model and a lower probability of overfitting.
>
> It is important to note that the choice of hyperparameters may be sensitive to initialization values. Additionally, while the hyperparameters learning rate, batch size, and alpha were individually optimized, the interaction between them and the model's architecture was not fully explored together. Conducting a thorough exploration could provide valuable insights into their interactions and influence on the model's performance.

In [ ]:
```
# save results
best_params_greedy["hidden_layer_sizes"] = (3, 3, 3)
```

(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 (this is provided as `X_train_plus_val` and `y_train_plus_val`).

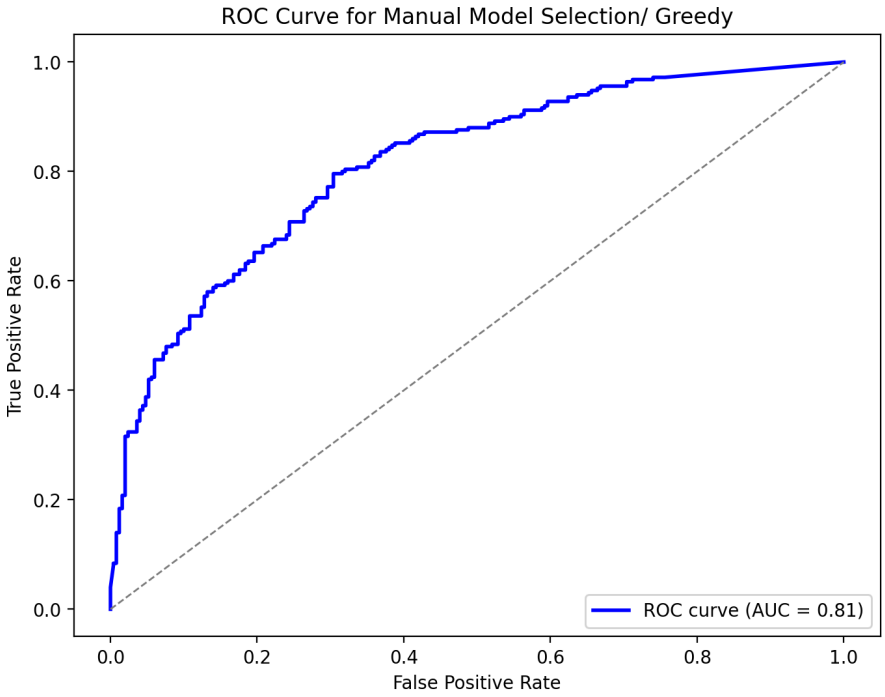
- Apply the trained model to the test data and report the accuracy of your final model on the test data.
- Plot an ROC curve of your performance (plot this with the curve in part (e) on the same set of axes you use for that question).

> d) Finally, after manually optimizing our hyperparameters and selecting the most suitable model based on our previous explorations, we proceed to evaluate the performance of our final manual model on the test data. We trained our model with the optimized hyperparameters using both the training and validation data combined, allowing us to maximize the model's generalization. Below, we present the accuracy results and the AUC score obtained on the test data.

In [ ]:
```
greedy = MLPClassifier(
    learning_rate_init=best_params_greedy["learning_rate_init"],
    hidden_layer_sizes=best_params_greedy["hidden_layer_sizes"],
    alpha=best_params_greedy["alpha"],
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=best_params_greedy["batch_size"],
    max_iter=500,
)

greedy.fit(X_train_plus_val, y_train_plus_val)
accuracy_score_test_greedy = greedy.score(X_test, y_test)

y_test_pred_proba_greedy = greedy.predict_proba(X_test)[:, 1]

fpr_greedy, tpr_greedy, thresholds_greedy = roc_curve(y_test, y_test_pred_proba_greedy)
auc_score_greedy = roc_auc_score(y_test, y_test_pred_proba_greedy)
```

In [ ]:
```
plt.figure(figsize=(8, 6))
plt.plot(
    fpr_greedy,
```

```
    tpr_greedy,
    color="blue",
    lw=2,
    label="ROC curve (AUC = %0.2f)" % auc_score_greedy,
)
plt.plot([0, 1], [0, 1], color="gray", lw=1, linestyle="--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for Manual Model Selection/ Greedy")
plt.legend(loc="lower right")
plt.show()
```



In [ ]:
```
print("Final manual model selection results on test data:")
print(f"Accuracy: {accuracy_score_test_greedy:.3f}")
print(f"AUC score: {auc_score_greedy:.3f}")
```

```
Final manual model selection results on test data:
Accuracy: 0.726
AUC score: 0.813
```

(e) Automated hyperparameter search through random search. The manual (greedy) approach (setting one or two parameters at a time holding the rest constant), provides good insights into how the neural network hyperparameters impacts model fitting for this particular training process. However, it is limited in one very problematic way: it depends heavily on a good "default" setting of the hyperparameters. Those were provided for you in this exercise, but are not generally know. Our manual optimization was somewhat greedy because we picked the hyperparameters one at a time rather than looking at different combinations of hyperparameters. Adopting such a pseudo-greedy approach to that manual optimization also limits our ability to more deeply search the hyperparameter space since we don't look at simultaneous changes to multiple parameters. Now we'll use a popular hyperparameter optimization tool to accomplish that: random search.

Random search is an excellent example of a hyperparameter optimization search strategy that has been shown to be more efficient (requiring fewer training runs) than another common approach: grid search. Grid search evaluates all possible combinations of hyperparameters from lists of possible hyperparameter settings - a very computationally expensive process. Yet another attractive alternative is Bayesian Optimization, which is an excellent hyperparameter optimization strategy but we will leave that to the interested reader.

Our particular random search tool will be Scikit-Learn's `RandomizedSearchCV`. This performs random search employing cross validation for performance evaluation (we will adjust this to ve a train/validation split).

Using `RandomizedSearchCV`, train on the training data while validating on the validation data (see instructions below on how to setup the train/validation split automatically). This tool will randomly pick combinations of parameter values and test them out, returning the best combination it finds as measured by performance on the validation set. You can use this example as a template for how to do this.

- To make this comparable to the training/validation setup used for the greedy optimization, we need to setup a training and validation split rather than use cross validation. To do this for `RandomSearchCV` we input the COMBINED training and validation dataset (`X_train_plus_val`, and `y_train_plus_val`) and we set the `cv` parameter to be the `train_val_split` variable we provided along with the dataset. This will setup the algorithm to make its assessments training just on the training data and evaluation on the validation data. Once `RandomSearchCV` completes its search, it will fit the model one more time to the combined training and validation data using the optimized parameters as we would want it to. *Note: The object returned by running fit (the random search) is NOT the best estimator. You can access the best estimator through the attribute* `.best_estimator_`, *assuming that you did not pass* `refit=False`.

- Set the number of iterations to at least 200 (you'll look at 200 random pairings of possible hyperparameters). You can go as high as you want, but it will take longer the larger the value.
- If you run this on Colab or any system with multiple cores, set the parameter `n_jobs` to -1 to use all available cores for more efficient training through parallelization
- You'll need to set the range or distribution of the parameters you want to sample from. Search over the same ranges as in previous problems. To tell the algorithm the ranges to search, use lists of values for candidate batch_size, since those need to be integers rather than a range; the `loguniform` `scipy` function for setting the range of the learning rate and regularization parameter, and a list of tuples for the `hidden_layer_sizes` parameter, as you used in the greedy optimization.
- Once the model is fit, use the `best_params_` property of the fit classifier attribute to extract the optimized values of the hyperparameters and report those and compare them to what was selected through the manual, greedy optimization.

For the final generalization performance assessment:

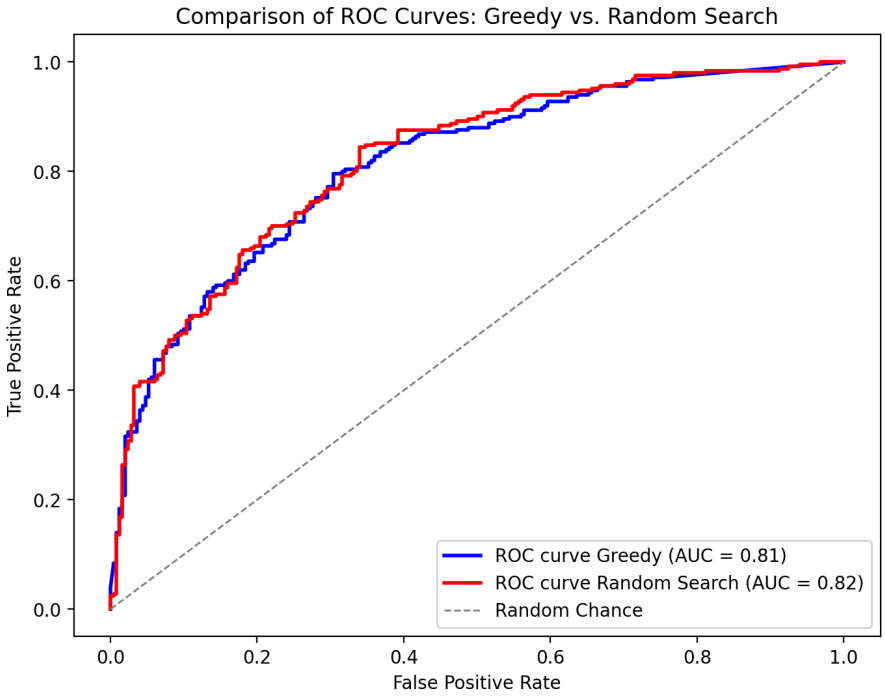
- State the accuracy of the optimized models on the test dataset
- Plot the ROC curve corresponding to your best model on the test dataset through greedy hyperparameter section vs the model identified through random search (these curves should be on the same set of axes for comparison). In the legend of the plot, report the AUC for each curve. This should be one single graph with 3 curves (one for greedy search, one for random search, and one representing random chance). Please also provide AUC score for greedy research and random search.
- Plot the finalalterfor the greedy and random search-based classifiers along with the test dataset to demonstrate the shape of the final boundary
- How did the generalization performance compare between the hyperparameters selected through the manual (greedy) search and the random search?

In [ ]:
```
# lists of possible hyperparameters

hidden_layer_sizes_list = []
for n_ndes in n_nodes_list:
    for n_layer in n_layer_list:
        hidden_layer_sizes = tuple([n_ndes] * (n_layer))
        hidden_layer_sizes_list.append(hidden_layer_sizes)

batch_sizes = [1, 3, 5, 10, 20, 50, 100, 250, 500]
learning_rates_loguniform = loguniform(1e-5, 1e0).rvs(20)
alphas_loguniform = loguniform(1e-8, 1e2).rvs(20)
```

In [ ]:
```
param_dist = {
    "hidden_layer_sizes": hidden_layer_sizes_list,
    "batch_size": batch_sizes,
    "learning_rate_init": learning_rates,
    "alpha": alphas,
}

clf = MLPClassifier(
    random_state=12,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    max_iter=500,
)

random_search = RandomizedSearchCV(
    estimator=clf,
    param_distributions=param_dist,
    n_iter=200,
    scoring="accuracy",
    cv=train_val_split,
    random_state=1,
    n_jobs=-1,
)

random_search.fit(X_train_plus_val, y_train_plus_val)
best_params_random_search = random_search.best_params_
accuracy_score_test_random_search = random_search.score(X_test, y_test)

y_test_pred_proba_random_search = random_search.predict_proba(X_test)[:, 1]

fpr_random_search, tpr_random_search, thresholds_random_search = roc_curve(
    y_test, y_test_pred_proba_random_search
)

auc_score_random_search = roc_auc_score(y_test, y_test_pred_proba_random_search)
```

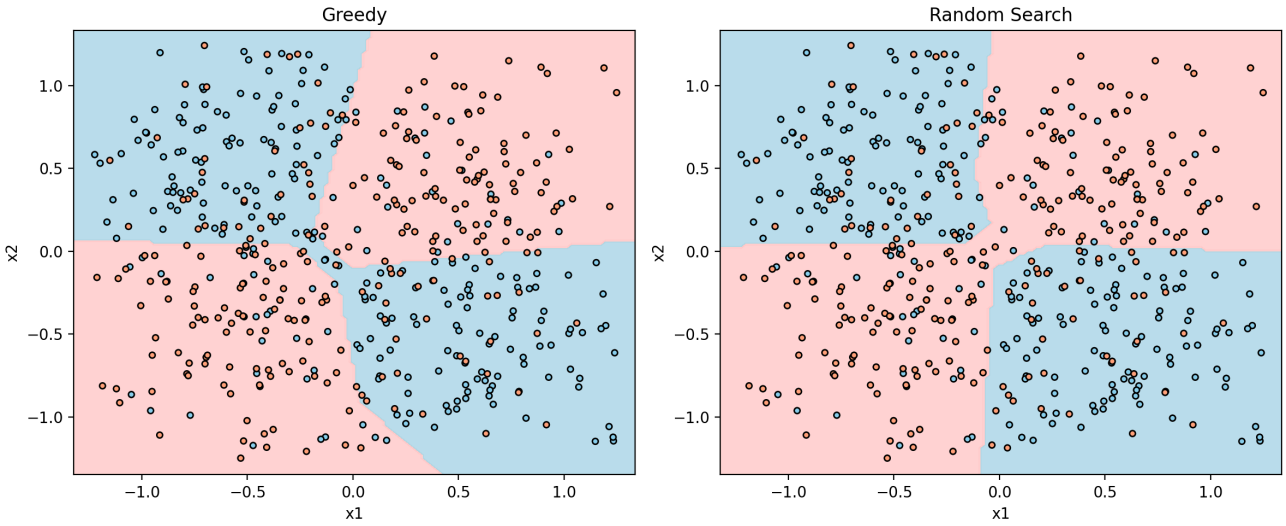
In [ ]:
```
random_chance_x = [0, 1]
random_chance_y = [0, 1]

plt.figure(figsize=(8, 6))
plt.plot(
    fpr_greedy,
    tpr_greedy,
```

```
    color="blue",
    lw=2,
    label="ROC curve Greedy (AUC = %0.2f)" % auc_score_greedy,
)
plt.plot(
    fpr_random_search,
    tpr_random_search,
    color="red",
    lw=2,
    label="ROC curve Random Search (AUC = %0.2f)" % auc_score_random_search,
)
plt.plot(
    random_chance_x,
    random_chance_y,
    color="gray",
    lw=1,
    linestyle="--",
    label="Random Chance",
)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Comparison of ROC Curves: Greedy vs. Random Search")
plt.legend(loc="lower right")
plt.show()
```



In [ ]:
```
boundary_colors = ListedColormap(["#FFCCCB", "#FFCCCB", "#ADD8E6"])
scatter_colors = ListedColormap(["#FFA07A", "#98FB98", "#87CEEB"])

x_min, x_max = X_train[:, 0].min() - 0.1, X_train[:, 0].max() + 0.1
y_min, y_max = X_train[:, 1].min() - 0.1, X_train[:, 1].max() + 0.1

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ax = axes[0]

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))

y_train_hat_greedy = greedy.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

ax.contourf(xx, yy, y_train_hat_greedy, alpha=0.8, cmap=boundary_colors)
ax.scatter(
    X_train[:, 0],
    X_train[:, 1],
    c=y_train,
    edgecolors="k",
    cmap=scatter_colors,
    s=15,
)

ax.set_title("Greedy")
ax.set_xlabel("x1")
ax.set_ylabel("x2")
```

```
ax = axes[1]

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))

y_train_hat_random_search = random_search.predict(
    np.c_[xx.ravel(), yy.ravel()]
).reshape(xx.shape)

ax.contourf(xx, yy, y_train_hat_random_search, alpha=0.8, cmap=boundary_colors)
ax.scatter(
    X_train[:, 0],
    X_train[:, 1],
    c=y_train,
    edgecolors="k",
    cmap=scatter_colors,
    s=15,
)

ax.set_title("Random Search")
ax.set_xlabel("x1")
ax.set_ylabel("x2")

plt.tight_layout()
plt.show()
```



In [ ]:
```
print("Greedy Search model\n" + "-" * 20)
print("Best Parameters:")
print(f"  - Learning Rate Init: {best_params_greedy['learning_rate_init']:.4f}")
print(f"  - Hidden Layer Sizes: {best_params_greedy['hidden_layer_sizes']}")
print(f"  - Alpha: {best_params_greedy['alpha']:.4f}")
print(f"  - Batch Size: {best_params_greedy['batch_size']}")
print(f"Accuracy: {accuracy_score_test_greedy:.3f}")
print(f"AUC score: {auc_score_greedy:.3f}")
print("\n")

print("Random Search model\n" + "-" * 20)
print("Best Parameters:")
print(f"  - Learning Rate Init: {best_params_random_search['learning_rate_init']:.4f}")
print(f"  - Hidden Layer Sizes: {best_params_random_search['hidden_layer_sizes']}")
print(f"  - Alpha: {best_params_random_search['alpha']:.4f}")
print(f"  - Batch Size: {best_params_random_search['batch_size']}")
print(f"Accuracy: {accuracy_score_test_random_search:.3f}")
print(f"AUC score: {auc_score_random_search:.3f}")
```

```
Greedy Search model
--------------------
Best Parameters:
  - Learning Rate Init: 0.0043
  - Hidden Layer Sizes: (3, 3, 3)
  - Alpha: 0.2336
  - Batch Size: 250
Accuracy: 0.726
AUC score: 0.813


Random Search model
--------------------
Best Parameters:
  - Learning Rate Init: 0.0004
  - Hidden Layer Sizes: (5,)
  - Alpha: 0.0000
  - Batch Size: 3
Accuracy: 0.736
AUC score: 0.820
```

> The comparison between models derived from the greedy hyperparameter section and those identified through random search is evident in the ROC curve graphs and preceding decision boundaries. The results, including the printing of the best chosen parameters, accuracy, and the AUC of the ROC curve for both models, are presented above.
>
> We can observe that both accuracy and AUC turned out to be better for the Random Search model compared to the Greedy Search model, improving from an accuracy for the test data of 0.726 to 0.736 and an AUC score from 0.813 to 0.820. Therefore, `the generalization capacity proved to be better in Random Search than in manual (greedy) hyperparameter selection`.
>
> Furthermore, the Random Search model resulted in having a simpler architecture, which is an advantage because it allows for easier training and is less susceptible to overfitting.
>
> In general, Random Search is a good option considering that neural networks heavily rely on a proper "default" setting of the hyperparameters. While in this exercise, we were able to start with a default parameter setting, usually, we won't have that possibility in real life.

2

[30 points] Build and test your own Neural Network for classification

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) [10 points of the 30] Create a neural network class that follows the `scikit-learn` classifier convention by implementing `fit`, `predict`, and `predict_proba` methods. Your `fit` method should run backpropagation on your training data using stochastic gradient descent. Assume the activation function is a sigmoid. Choose your model architecture to have two input nodes, two hidden layers with five 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 bug. 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).

NOTE: Due to the depth this question requires, some students may choose not to complete this section (in lieu of receiving the 10 points from this question). If you choose not to build your own neural network, or if your neural network is not functional prior to submission, then use the `scikit-learn` implementation instead in the questions below; where it asks to compare to `scikit-learn`, compare against a random forest classifier instead.

In [ ]:
```
# a)
```

```
class myNeuralNetwork(object):

    def __init__(self, n_in, n_layer1, n_layer2, n_out, learning_rate=0.01):
        """__init__
        Class constructor: Initialize the parameters of the network including
        the learning rate, layer sizes, and each of the parameters
        of the model (weights, placeholders for activations, inputs,
        deltas for gradients, and weight gradients). This method
        should also initialize the weights of your model randomly
            Input:
                n_in:          number of inputs
                n_layer1:      number of nodes in layer 1
                n_layer2:      number of nodes in layer 2
                n_out:         number of output nodes
                learning_rate: learning rate for gradient descent
            Output:
                none
        """
        self.learning_rate = learning_rate
        self.W = [
            np.random.uniform(-0.5, 0.5, size=(n_in, n_layer1)),
            np.random.uniform(-0.5, 0.5, size=(n_layer1, n_layer2)),
            np.random.uniform(-0.5, 0.5, size=(n_layer2, n_out)),
        ]

        self.dE_dW = None
        self.a = None

    def forward_propagation(self, x):
        """forward_propagation
        Takes a vector of your input data (one sample) and feeds
        it forward through the neural network, calculating activations and
        layer node values along the way.
            Input:
                x: a vector of data representing 1 sample [n_in x 1]
            Output:
                y_hat: a vector (or scaler of predictions) [n_out x 1]
                (typically n_out will be 1 for binary classification)
        """
        a1 = x @ self.W[0]  # hidden layer 1
        z1 = self.sigmoid(a1)
        a2 = z1 @ self.W[1]  # hidden layer 2
        z2 = self.sigmoid(a2)
        a3 = z2 @ self.W[2]
        self.a = [a1, a2, a3]
        y_hat = self.sigmoid(a3)

        return y_hat

    def compute_loss(self, X, y):
        """compute_loss
        Computes the current loss/cost function of the neural network
        based on the weights and the data input into this function.
        To do so, it runs the X data through the network to generate
        predictions, then compares it to the target variable y using
        the cost/loss function
            Input:
                X: A matrix of N samples of data [N x n_in]
                y: Target variable [N x 1]
            Output:
                loss: a scalar measure of loss/cost
        """
        y_hat = self.predict_proba(X)
        y = y.reshape(-1, 1)
        N = X.shape[0]
        loss = -(1 / N) * np.sum(y * np.log(y_hat) +
                                 (1 - y) * np.log(1 - y_hat))

        return loss

    def backpropagate(self, x, y):
        """backpropagate
        Backpropagate the error from one sample determining the gradients
        with respect to each of the weights in the network. The steps for
        this algorithm are:
            1. Run a forward pass of the model to get the activations
               Corresponding to x and get the loss functionof the model
               predictions compared to the target variable y
            2. Compute the deltas (see lecture notes) and values of the
               gradient with respect to each weight in each layer moving
               backwards through the network

            Input:
                x: A vector of 1 samples of data [n_in x 1]
                y: Target variable [scalar]
            Output:
```

```
            loss: a scalar measure of th loss/cost associated with x,y
                  and the current model weights
        """
        y_hat = self.forward_propagation(x)
        loss = self.compute_loss(x, y)

        dC_dy_hat = -y / y_hat + (1 - y) / (1 - y_hat)

        z2 = self.sigmoid(self.a[1])
        dC_dw3 = z2.T @ dC_dy_hat
        dC_da2 = dC_dy_hat @ self.W[2].T * self.sigmoid_derivative(self.a[1])

        z1 = self.sigmoid(self.a[0])
        dC_dw2 = z1.T @ dC_da2
        dC_da1 = dC_da2 @ self.W[1].T * self.sigmoid_derivative(self.a[0])

        dC_dw1 = x.T @ dC_da1

        self.dE_dW = [dC_dw1, dC_dw2, dC_dw3]

        # update weights
        self.stochastic_gradient_descent_step()

        return loss

    def stochastic_gradient_descent_step(self):
        """stochastic_gradient_descent_step [OPTIONAL - you may also do this
        directly in backpropagate]
        Using the gradient values computed by backpropagate, update each
        weight value of the model according to the familiar stochastic
        gradient descent update equation.

        Input: none
        Output: none
        """
        for i in range(len(self.W)):
            self.W[i] -= self.learning_rate * self.dE_dW[i]

    def fit(
        self,
        X,
        y,
        max_epochs=50,
        learning_rate=0.01,
        get_validation_loss=False,
        X_val=None,
        y_val=None,
    ):
        """fit
        Input:
            X: A matrix of N samples of data [N x n_in]
            y: Target variable [N x 1]
        Output:
            training_loss:   Vector of training loss values at the end of each epoch
            validation_loss: Vector of validation loss values at the end of each epoch
                             [optional output if get_validation_loss==True]
        """
        training_loss = []
        validation_loss = []
        N = X.shape[0]
        np.random.seed(321)

        for epoch in range(max_epochs):

            shuffled_indices = np.random.permutation(X.shape[0])
            X = X[shuffled_indices]
            y = y[shuffled_indices]

            for i in range(N):

                x_i = X[i: i + 1, :]
                y_i = np.array([y[i]])

                self.backpropagate(x_i, y_i)

            epoch_training_loss = self.compute_loss(X, y)
            training_loss.append(epoch_training_loss)

            if get_validation_loss:
                epoch_validation_loss = self.compute_loss(X_val, y_val)
                validation_loss.append(epoch_validation_loss)

        if get_validation_loss:
            return training_loss, validation_loss
        else:
            return training_loss
```

```python
    def predict_proba(self, X):
        """predict_proba
        Compute the output of the neural network for each sample in X, with the last layer's
        sigmoid activation providing an estimate of the target output between 0 and 1
            Input:
                X: A matrix of N samples of data [N x n_in]
            Output:
                y_hat: A vector of class predictions between 0 and 1 [N x 1]
        """

        y_hat = np.zeros((X.shape[0], 1))

        for i in range(X.shape[0]):
            # Forward propagation for each sample
            a1 = X[i] @ self.W[0]  # hidden layer 1
            z1 = self.sigmoid(a1)
            a2 = z1 @ self.W[1]  # hidden layer 2
            z2 = self.sigmoid(a2)
            a3 = z2 @ self.W[2]

            y_hat[i] = self.sigmoid(a3)

        return y_hat

    def predict(self, X, decision_thresh=0.5):
        """predict
        Compute the output of the neural network prediction for
        each sample in X, with the last layer's sigmoid activation
        providing an estimate of the target output between 0 and 1,
        then thresholding that prediction based on decision_thresh
        to produce a binary class prediction
            Input:
                X: A matrix of N samples of data [N x n_in]
                decision_threshold: threshold for the class confidence score
                                    of predict_proba for binarizing the output
            Output:
                y_hat: A vector of class predictions of either 0 or 1 [N x 1]
        """
        pred = (self.predict_proba(X) > decision_thresh).astype(int)
        return pred.flatten()

    def sigmoid(self, X):
        """sigmoid
        Compute the sigmoid function for each value in matrix X
            Input:
                X: A matrix of any size [m x n]
            Output:
                X_sigmoid: A matrix [m x n] where each entry corresponds to the
                           entry of X after applying the sigmoid function
        """
        return 1 / (1 + np.exp(-X))

    def sigmoid_derivative(self, X):
        """sigmoid_derivative
        Compute the sigmoid derivative function for each value in matrix X
            Input:
                X: A matrix of any size [m x n]
            Output:
                X_sigmoid: A matrix [m x n] where each entry corresponds to the
                           entry of X after applying the sigmoid derivative function
        """
        return self.sigmoid(X) * (1 - self.sigmoid(X))
```

(b) Apply your neural network.

- Create training, validation, and test datasets using `sklearn.datasets.make_moons(N, noise=0.20)` data, where $N_{train} = 500$ and $N_{test} = 100$. The validation dataset should be a portion of your training dataset that you hold out for hyperparameter tuning.
- **Cost function plots**. Train and validate your model on this dataset plotting your training and validation cost learning curves on the same set of axes. This is the 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. You're free to use any methods you deem fit to tune your hyperparameters like grid search, random search, Bayesian optimization etc.
- **Decision boundary plots**. In two subplots, plot the training data on one subplot and the validation data on the other subplot. On each plot, also plot the decision boundary from your neural network trained on the training data.
- **ROC Curve plots**. Report your performance on the test data with an ROC curve and the corresponding AUC score. Compare against the `scikit-learn` `MLPClassifier` trained with the same parameters on the same set of axes and include the chance diagonal. *Note: if you chose not to build your own neural network in part (a) above, or if your neural network is not functional prior to submission, then use the `scikit-learn` `MLPClassifier` class instead for the neural network and compare it against a random forest classifier instead. Be sure to set the hidden layer sizes, epochs, and learning rate for that model, if so.*

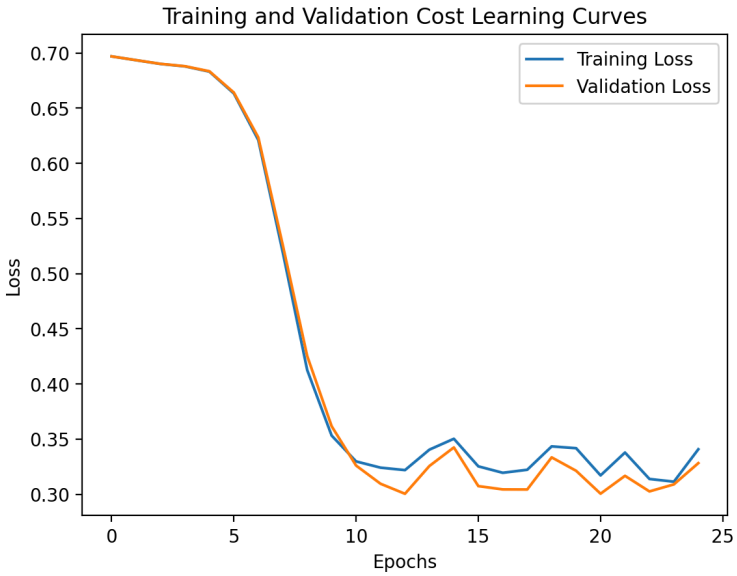
- **Remember to retrain your model.** After selecting your hyperparameters using the validation data set, when evaluating the final performance on the ROC curve, it's good practice to retrain your model with the selected hyperparameters on the train + validation dataset, before evaluating on the test data.

Note if you opted not to build your own neural network: in this case, for hyperparameter tuning, we recommend using the `partial_fit` method to train your model for every epoch. Partial fit allows you to incrementally fit on one sample at a time.

In [ ]:
```
# (b)
np.random.seed(1234)

N_train = 500
N_test = 100
validation_ratio = 0.2

# Training datasets
X_train, y_train = make_moons(n_samples=int(500 * 0.8), noise=0.20, random_state=123)

# Validation and test data
X_val, y_val = make_moons(n_samples=int(500 * 0.2), noise=0.20, random_state=456)
X_test, y_test = make_moons(n_samples=100, noise=0.20, random_state=789)


X_train_plus_val = np.concatenate((X_train, X_val), axis=0)
y_train_plus_val = np.concatenate((y_train, y_val), axis=0)
```

In [ ]:
```
NN = myNeuralNetwork(n_in=2, n_layer1=5, n_layer2=5, n_out=1)

training_loss, validation_loss = NN.fit(
    X_train,
    y_train,
    max_epochs=25,
    learning_rate=0.001,
    get_validation_loss=True,
    X_val=X_val,
    y_val=y_val,
)
```

In [ ]:
```
plt.plot(training_loss, label="Training Loss")
plt.plot(validation_loss, label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Validation Cost Learning Curves")
plt.legend()
plt.show()
```



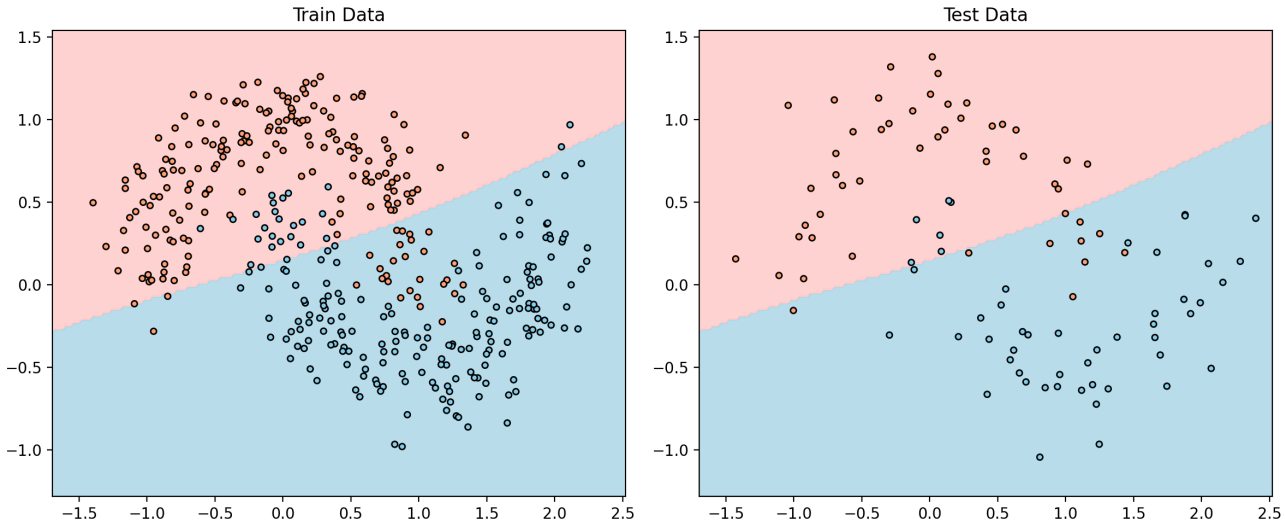
In [ ]:
```
max_epochs_values = [10, 15, 25, 30, 35]
learning_rate_values = [0.001, 0.01, 0.1, 1.0]

best_accuracy = -1
best_max_epochs = None
best_learning_rate = None

for max_epochs in max_epochs_values:
    for learning_rate in learning_rate_values:
        NN = myNeuralNetwork(n_in=2, n_layer1=5, n_layer2=5, n_out=1)

        NN.fit(X_train, y_train, max_epochs=max_epochs, learning_rate=learning_rate)
```

```
        y_pred = NN.predict(X_val)

        accuracy = accuracy_score(y_val, y_pred)

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_max_epochs = max_epochs
            best_learning_rate = learning_rate

print("Best accuracy found:", best_accuracy)
print("Best hyperparameters:")
print("max_epochs:", best_max_epochs)
print("learning_rate:", best_learning_rate)
```

```
Best accuracy found: 0.86
Best hyperparameters:
max_epochs: 35
learning_rate: 0.001
```

In [ ]:
```
np.random.seed(123)
max_epochs_final = 35
learning_rate_final = 0.001

NN_final = myNeuralNetwork(n_in=2, n_layer1=5, n_layer2=5, n_out=1)

loss = NN_final.fit(
    X_train,
    y_train,
    max_epochs=max_epochs_final,
    learning_rate=learning_rate_final,
)
```

In [ ]:
```
boundary_colors = ListedColormap(["#FFCCCB", "#FFCCCB", "#ADD8E6"])
scatter_colors = ListedColormap(["#FFA07A", "#98FB98", "#87CEEB"])

x_min, x_max = X_train[:, 0].min() - 0.3, X_train[:, 0].max() + 0.3
y_min, y_max = X_train[:, 1].min() - 0.3, X_train[:, 1].max() + 0.3

fig, axes = plt.subplots(1, 2, figsize=(12, 5))


ax = axes[0]

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))

y_train_hat = NN_final.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

ax.contourf(xx, yy, y_train_hat, alpha=0.8, cmap=boundary_colors)
ax.scatter(
    X_train[:, 0],
    X_train[:, 1],
    c=y_train,
    edgecolors="k",
    cmap=scatter_colors,
    s=15,
)
ax.set_title("Train Data")


ax = axes[1]

y_test_hat = NN_final.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

ax.contourf(xx, yy, y_test_hat, alpha=0.8, cmap=boundary_colors)
ax.scatter(
    X_test[:, 0],
    X_test[:, 1],
    c=y_test,
    edgecolors="k",
    cmap=scatter_colors,
    s=15,
)
ax.set_title("Test Data")

plt.tight_layout()
plt.show()
```

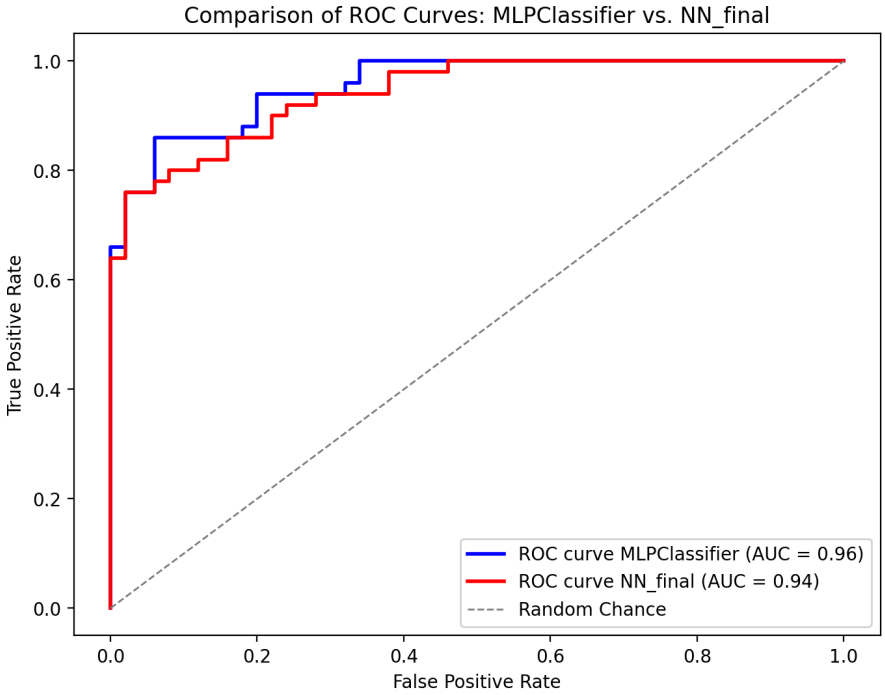


In [ ]:
```
MLPClassifier = MLPClassifier(
    random_state=1,
    learning_rate_init=learning_rate_final,
    hidden_layer_sizes=(5, 5),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="logistic",
    n_iter_no_change=1000,
    batch_size=1,
    max_iter=max_epochs_final,
)


MLPClassifier.fit(X_train_plus_val, y_train_plus_val)

y_test_pred_proba_MLPClassifier = MLPClassifier.predict_proba(X_test)[:, 1]

fpr_MLPClassifier, tpr_MLPClassifier, thresholds_MLPClassifier = roc_curve(
    y_test, y_test_pred_proba_MLPClassifier
)
auc_score_MLPClassifier = roc_auc_score(
    y_test, y_test_pred_proba_MLPClassifier)
```

In [ ]:
```
# my model

NN_final.fit(X_train_plus_val, y_train_plus_val)

y_test_pred_proba_NN_final = NN_final.predict_proba(X_test)

fpr_NN_final, tpr_NN_final, thresholds_NN_final = roc_curve(
    y_test, y_test_pred_proba_NN_final
)

auc_score_NN_final = roc_auc_score(y_test, y_test_pred_proba_NN_final)
```

In [ ]:
```
plt.figure(figsize=(8, 6))

# Plot MLPClassifier ROC curve
plt.plot(
    fpr_MLPClassifier,
    tpr_MLPClassifier,
    color="blue",
    lw=2,
    label="ROC curve MLPClassifier (AUC = %0.2f)" % auc_score_MLPClassifier,
)

# Plot NN_final ROC curve
plt.plot(
    fpr_NN_final,
    tpr_NN_final,
    color="red",
    lw=2,
    label="ROC curve NN_final (AUC = %0.2f)" % auc_score_NN_final,
)

# Plot the random chance line
plt.plot(
    random_chance_x,
```

```
    random_chance_y,
    color="gray",
    lw=1,
    linestyle="--",
    label="Random Chance",
)

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Comparison of ROC Curves: MLPClassifier vs. NN_final")
plt.legend(loc="lower right")
plt.show()
```



(c) Suggest two ways in which you neural network implementation could be improved: are there any options we discussed in class that were not included in your implementation that could improve performance?

> My implementation of the neural network from scratch has several limitations as it considers various hyperparameters and architecture choices as fixed. Although the implementation achieves some basic functionality, there are significant areas for improvement in terms of its performance and predictive capacity.
>
> One of the main limitations of my implementation is the `minibatch size`, which is currently set to 1 (Stochastic gradient descent). While this configuration may be suitable in some cases, it could limit the training efficiency and stability of the model.
>
> Another limitation is the rigidity in the `network architecture`. Currently, the implementation fixes the use of two hidden layers, with a specific number of nodes in each layer. However, exploring more flexible architectures that allow variations in the number of layers and nodes could enhance the network's ability to capture complex relationships in the data and improve its predictive capacity.
>
> Additionally, the choice of `activation function` is another aspect that could be improved. Although the sigmoid activation function has been used in the current implementation, alternatives such as ReLU could enhance the network's ability to model nonlinear relationships in the data and improve its performance.
>
> `Weight initialization` is another aspect to consider. Currently, the implementation initializes the weights with values uniformly distributed between -0.5 and 0.5. However, other initialization methods, such as Xavier or He initialization, could be more suitable.
>
> Finally, the current implementation does not include `regularization` techniques, which could lead to overfitting issues in complex datasets. The addition of regularization techniques could improve the model's generalization ability and its capacity to generalize to unseen data.