Experiment 6

Artificial Neural Networks

1. Introduction

Artificial neural networks (ANNs) are a type of machine learning model that is inspired by the human brain. They are made up of interconnected nodes, called neurons, that are arranged in layers. The neurons in each layer are connected to the neurons in the next layer. Together with the strength of these connections, the ANN model would represent a specific function that maps the inputs to the outputs. A learning algorithm is used to tune the strength of these connections so that they can approximate a specific function.

ANNs can learn complex relationships between the input and output data and are thus treated as powerful tools that can be used to solve a variety of problems such as Image and speech recognition, Natural language processing, Autonomous vehicles, Healthcare diagnostics, Financial prediction, etc. However, they can be computationally expensive to train.

ANNs exhibit various architecture types, and selecting the appropriate one depends on the nature of the problem at hand. These architectures encompass:

- **Feedforward Neural Networks (FNNs):** Data flows from input to output without loops. They're used for various tasks, like classification and regression.
- **Convolutional Neural Networks (CNNs):** Specialized for image and video analysis, CNNs automatically learn hierarchical features from input data.
- **Recurrent Neural Networks (RNNs):** Suitable for sequence data (e.g., time series or text), RNNs have feedback loops for processing sequences.
- **Long Short-Term Memory (LSTM) Networks:** A type of RNN, LSTMs excel at capturing long-range dependencies in sequences.

ANNs learn by adjusting the weights and biases of their connections based on observed data. The learning process involves forward propagation (calculating outputs for a given input) and backward propagation (adjusting weights using gradient descent and the backpropagation algorithm). This iterative process aims to minimize a loss function that quantifies the difference between predicted and actual outcomes.

2. Architecture of Feedforward Neural Networks

FNNs, also known as a Multi-Layer Perceptron (MLPs), are a type of artificial neural network where information flows in one direction, from the input layer through one or more hidden layers to the output layer. The architecture of an FNN consists of different components, each with a specific purpose. The general architecture of MLP is shown in Figure 1.
Here's an overview of the main components of FNNs:

- **Input Layer:** The input layer consists of neurons that receive input features. The number of neurons in this layer corresponds to the number of input features. Each neuron in the input layer represents a specific feature of the input data.
- **Hidden Layers:** Hidden layers are intermediary layers between the input and output layers. Each hidden layer consists of multiple neurons that process the information from the previous layer and pass it on to the next layer. The number of hidden layers and the number of neurons in each layer are hyperparameters that you can adjust based on the complexity of the problem and the dataset. Hidden layers allow FNNs to learn complex hierarchical features and patterns in the data.
- **Neurons (Nodes):** Each neuron in a hidden layer or the output layer receives inputs from the previous layer's neurons, applies weights to these inputs, and passes the result through an activation function. Neurons in the hidden layers often use non-linear activation functions (e.g., ReLU, sigmoid, and tanh) to introduce non-linearity to the model, enabling it to capture complex relationships in the data.

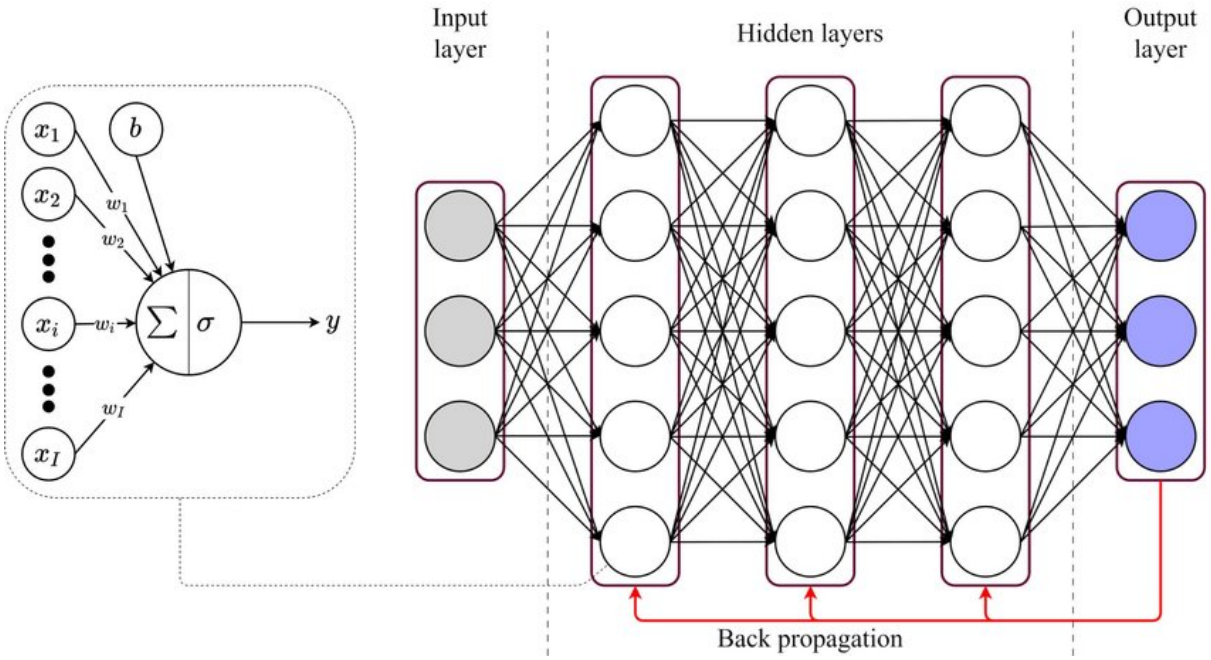

Figure 1: General Architecture of Feedforward Neural Networks

- **Weights and biases:** Each connection between neurons has an associated weight that determines the strength of the connection. These weights are learned during training. Each neuron also has a bias term that influences its output. Biases are also learned during training.
- **Output Layer:** The output layer produces the final predictions or classifications based on the information processed in the hidden layers. The number of neurons in the output layer depends on the type of task you're solving. For binary classification, you may have a single neuron with output ranging from 0 to 1. For multi-class classification, you'd have a neuron for each class, outputting the probability of that class.
- **Activation Functions:** Activation functions introduce non-linearity to the network, enabling it to model complex relationships in the data. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, tanh, and softmax (for multi-class classification).

FNNs can be categorized as perceptrons and Multi-Layer perceptrons (MLPs). A perceptron is a simple FNN that can be used to solve linearly separable problems. It is made up of a single layer of neurons, each of which computes a weighted sum of its inputs and applies a non-linear activation function to the result. The perceptron can be trained to classify data points by adjusting the weights and biases of its neurons. Changing the weights/threshold makes the decision boundary move. The general structure of the perceptron is shown in Figure 2:

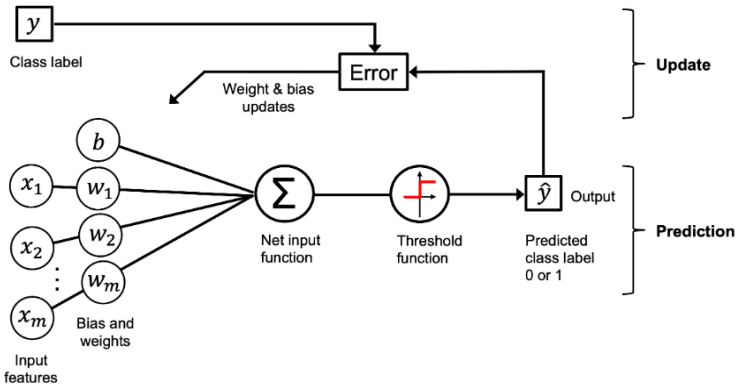

Figure 2: Perceptron Structure

An MLP is a more complex ANN that can be used to solve both linearly separable and non-linearly separable problems. The neurons in the hidden layers of an MLP can learn non-linear relationships between the inputs and the outputs of the network, which allows it to solve problems that a perceptron cannot. Choosing the right architecture for a FNN is a critical step in the machine learning process. The architecture of an FNN mainly refers to the number of layers, the number of neurons in each layer, and the activation function, where the number of layers is treated as the most important parameter. The main factors to consider when choosing an FNN architecture are:

- **The complexity of the task:** the more complex the task, the more layers and hidden neurons will be needed. For example: if the problem is linearly separable, meaning the classes can be separated by a straight line (in 2D) or a hyperplane (in higher dimensions), a perceptron can be effective. On the other hand, if the problem is not linearly separable or the data has high-dimensional or complex features, using perceptrons is not effective, and you must use MLPs. It's worth mentioning that choosing the number of layers in an FNN must strike a balance between ensuring the network has the capacity to learn complex patterns and avoiding unnecessary complexity that could lead to overfitting.
- **The size of the training data:** The larger the training data, the more layers and neurons the network will need. This is because the network needs to learn the patterns in the data, and a larger data set will have more patterns.
- **The available computational resources:** The number of layers and neurons in an FNN will also depend on the computational resources that are available. A network with a large number of layers and neurons will require more computing power to train.

3. Training MLPs

Gradient descent and backpropagation are the most widely used methods for training MLPs and other types of neural networks.

Gradient Descent (GD): GD is a foundational optimization algorithm extensively used in ANNs. The training process involves finding optimal weights and biases for these neurons that minimize the loss function, indicating the discrepancy between predicted and actual outputs. To achieve this goal, it performs two steps iteratively.

1. Compute the gradient of the loss with respect to weights and biases: during each iteration of GD, the algorithm calculates the gradient of the loss function with respect to the weights and biases of the network. This gradient indicates how much each weight and bias should be adjusted to reduce the loss.
2. Update weights and biases: with the computed gradients, GD updates the weights and biases of the network in the direction that decreases the loss. The magnitude of the update is determined by the learning rate. Smaller learning rates result in cautious steps, while larger rates can lead to overshooting or divergence. The balance between convergence speed and stability is crucial.

Backpropagation: Backpropagation is a crucial component of training neural networks. It's a method for calculating the gradients of the loss function with respect to the model's parameters, layer by layer. The gradients are propagated backward through the network, starting from the output layer and moving towards the input layer. Backpropagation utilizes the chain rule of calculus to compute the gradients efficiently.

The combination of gradient descent and backpropagation allows the network's parameters (weights and biases) to be updated in the direction that minimizes the loss function. This process iteratively fine-tunes the parameters to improve the network's performance on the training data. Training MLP using gradient descent and backpropagation involves implementing the forward pass to compute predictions and the backward pass to compute gradients for the network's parameters. Here's a step-by-step guide:

1. **Preparing Network Architecture:** The architecture includes the number of layers, the number of neurons in each layer, the activation functions, and the loss function.
2. **Initializations:** the initialization includes the following:

    a. Weights and biases: common techniques include random initialization and using smart techniques like Xavier/Glorot initialization.
    b. Hyperparameters: like learning rate, batch size, and number of epochs
3. **Choose Optimization Algorithm:** select optimization algorithm to update the network's parameters. Gradient Descent as an example
4. **Gradient Descent Training Loop:**
    a. Present a training example: A training example is presented to the perceptron. The training example consists of a set of inputs and a desired output.
    b. Forward pass: During the forward pass, the input data is fed through the network layer by layer, and the activations are calculated at each layer.
        i. Input Layer: Initialize the input activations with the training data.
        ii. Hidden Layers: For each hidden layer, calculate the weighted sum of the input activations and the layer's weights:
        iii. Apply the activation function to the weighted sum to compute the output activations:
        iv. Output Layer: For the output layer, calculate the weighted sum and apply an appropriate activation function (e.g., sigmoid, softmax).
    c. Backward Pass (Backpropagation): During the backward pass, gradients of the loss with respect to each parameter are calculated and propagated backward through the layers.
        i. Calculate the error at the output layer. This is the difference between the desired output and the predicted output.
        ii. Compute Output Layer Gradient (delta_output): Calculate the gradient of the loss function with respect to the output layer weights ($dloss/dW_{output}$). This is done using the chain rule.
        iii. Use the gradient to update the output layer weights.
        $$\mathbf{W_{new} = W_{old} - (larning\_rate * dLoss/dW)}$$
        iv. Propagate the error to the hidden layers. This is done by multiplying the error at the output layer by the weights connecting the output layer to the hidden layer.
        v. Calculate the gradient of the loss function with respect to the hidden layer weights. This is done using the chain rule.
        vi. Use the gradient to update the hidden layer weights.
        vii. Repeat steps ii to vi until updating parameters in the input layer.
    d. Repeat steps a to c for a predefined number of iterations (epochs) or error is minimized.

For Perceptron, the forward pass and the backward pass will be as follows; we assume using the sigmoid activation function and the Mean Squared Error (MSE) loss function.

$$Output_{predicted} = Sigmoid(S)$$
$$Loss = ½ (Output_{desired} - Output_{predicted})^2$$

Where:

$$s = \sum w_i * x_i + bias$$
$$Sigmoid = 1/(1 + e^{-x})$$

By chain rule,

$$dLoss/dW_i = [(dE/dpredicted)*(dpredicted/ds)*(ds/dW)])$$
$$dE/dpredicted = Output_{predicted} - Output_{desired}$$
$$dpredicted/ds = [(1/(1 + e^{-s}))(1-(1/ (1 + e^{-s}))]]$$
$$ds/dW_i = X_i$$
$$dLoss/dW_i = (Output_{predicted} - Output_{desired})*[(1/(1 + e^{-s}))(1-(1/ (1 + e^{-s}))]*X_i$$
$$Wi_{new} = Wi_{old} - (larning\_rate * dLoss/dW)*Xi$$
$$Bias_{new} = Biasi_{old} - (larning\_rate * dLoss/dW)$$

5. Implementing Perceptron in Python

In this section, we will implement Perceptron in Python as a binary classifier and also as an approximation to linear functions. Example 1 shows a simple Python code for a perceptron as a binary classifier with a step activation function and Stochastic Gradient Descent (SGD) training algorithm. The code defines a Perceptron class with methods for initialization, step activation, prediction, training, and testing. It demonstrates the use of the perceptron to solve the logical AND operation.

In this section, we will implement Perceptron in Python as a binary classifier and as an approximation to linear functions. Example 1 shows a simple Python code for a perceptron as a binary classifier with a step activation function and SGD training algorithm. The code defines a Perceptron class with methods for initialization, step activation, prediction, training, and testing. It demonstrates the use of the perceptron to solve the logical AND operation.

Example 1: Python code for a binary classifier perceptron with a step activation function and SGD training method. Note that, in SGD, the model parameters are updated after each selected sample.

```
import numpy as np


class Perceptron:
    def __init__(self, input_size, learning_rate, epochs):
        # Initialize weights and bias with random values
        self.weights = np.random.rand(input_size)
        self.bias = np.random.rand()

        # Set learning rate and number of epochs
        self.learning_rate = learning_rate
        self.epochs = epochs


    def step_activation(self, x):
        # Step activation function
        return 1 if x >= 0 else 0

    def predict(self, x):
        # Compute the weighted sum of inputs and bias
        net_input = np.dot(x, self.weights) + self.bias

        # Apply step activation function to the net input
        return self.step_activation(net_input)

    def trainSGD(self, X, y):
        # Training loop
        for epoch in range(self.epochs):
            # Initialize total error for the epoch
            total_error = 0

            # Shuffle the training examples for this epoch
            indices = np.arange(len(X))
            np.random.shuffle(indices)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            # Iterate over each shuffled training example
            for i in range(len(X_shuffled)):
                # Make a prediction for the current input
                prediction = self.predict(X_shuffled[i])

                # Compute the error (desired - predicted)
                error = y_shuffled[i] - prediction
```

```
                # Update weights and bias using stochastic gradient descent
                self.weights += self.learning_rate * error * X_shuffled[i]
                self.bias += self.learning_rate * error

                # Accumulate the absolute error for the epoch
                total_error += abs(error)

            # Print total errors for each epoch
            print(f"Epoch {epoch + 1}, Total Absolute Error: {total_error}")



# Training samples
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

# Create a perceptron instance with 2 input neurons
perceptron = Perceptron(input_size=2, learning_rate=0.01, epochs=10)

# Train the perceptron on the dataset
perceptron.trainSGD(X, y)

# Test the trained perceptron on the testing samples
test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for data in test_data:
    prediction = perceptron.predict(data)
    print(f"Input: {data}, Prediction: {prediction}")
```

Task 6.1: Run the code in Example 1 with more epochs [10, 20, 100, and 200] and compute the accuracy of the tested samples for each case. Did we need to run with more epochs? Justify your answer.
Task 6.2: Run the same code in Example 1 with a higher learning rate [0.05, 0.1, 0.3]. Draw on your observations.

Perceptrons can also be used to approximate linear functions. Let's consider an example of using a perceptron to approximate a simple linear function with one input variable. Example 2 shows a simple Python code for a perceptron as an approximation to the **f(x) = 3x + 2** linear function with a linear activation function and SGD training algorithm. The code defines a Perceptron class with methods for linear activation, linear derivative, prediction, training using SGD, and calculating the Mean Squared Error (MSE) loss. In addition, the example contains the code to generate synthetic data in order to train and test the Perceptron.

Example 2: Python code for a perceptron with a linear activation function, MSE loss, and SGD training method

```
import numpy as np
import pandas as pd

class Perceptron:
    def __init__(self, input_size, learning_rate, epochs):
        self.weights = np.random.randn(input_size)
        self.bias = np.random.randn()
        self.learning_rate = learning_rate
        self.epochs = epochs

    def linear(self, x):
        return x

    def linear_derivative(self, x):
```

```
        return 1

    def predict(self, x):
        net_input = np.dot(x, self.weights) + self.bias
        return self.linear(net_input)

    def trainSGD(self, X, y):
        for epoch in range(self.epochs):
            # Initialize Mean Squared Error for this epoch
            total_mse = 0.0

            # Shuffle the training examples for this epoch
            indices = np.arange(len(X))
            np.random.shuffle(indices)
            X_shuffled = X[indices]
            Y_shuffled = y[indices]

            # Iterate on each sample
            for i in range(len(X)):
                prediction = self.predict(X_shuffled[i])
                error = Y_shuffled[i] - prediction

                # Compute gradients using linear derivative
                delta = error * self.linear_derivative(prediction)

                # Update weights and bias using stochastic gradient descent
                self.weights += self.learning_rate * delta * X_shuffled[i]
                self.bias += self.learning_rate * delta

                # Accumulate the squared error for this example
                total_mse += error ** 2

            # Calculate the mean squared error for this epoch
            mean_mse = total_mse / len(X)
                print(f"Epoch {epoch + 1}/{self.epochs}, Mean Squared Error:
{mean_mse:.4f}")

# Generate synthetic data
np.random.seed(42)  # For reproducibility
num_samples = 100
x = np.random.uniform(low=0, high=10, size=num_samples)
z = 3 * x + 2

# Create a DataFrame to store the data
data = pd.DataFrame({'x': x, 'z': z})

# Extract the values of all columns (variables x and y) except the last one from
DataFrame
F = data.iloc[:, :-1].values
# Extract the values of the last column (output z) from DataFrame
O = data.iloc[:, -1].values

# Create a perceptron instance with appropriate input size
input_size = 1
perceptron = Perceptron(input_size=input_size, learning_rate=0.001, epochs=10)

# Train the perceptron on the dataset
perceptron.trainSGD(F, O)

# Test the trained model
for data in x:
  prediction = perceptron.predict(data)
  print(f"Input: {data}, Actual: {3*data+2}, Prediction: {prediction}")
```

Task 6.3: Run the code in Example 2 and notice the difference between the actual output and the predicted output.
Task 6.4: Run the code in Example 2 with more epochs [100]. Compare the results in terms of MSE and prediction error with the results in Task 6.3. Justify your answer.
Task 6.5: with epochs = 100, plot:

- **a.** MSE with respect to the epoch number;
- **b.** The actual function and the approximated function

Task 6.6: Modify the code in Example 2 to approximate ***f(x,y) = 2x + 3y.***

7. Performance of Perceptron on Nonlinear problems

When a perceptron is used to solve a nonlinear problem, it might suffer from underfitting because it cannot capture the complex patterns and relationships in the data. Nonlinear problems require more complex decision boundaries that cannot be represented by a single layer of perceptrons. The XOR problem is a classic example that highlights the limitations of a single-layer perceptron in solving nonlinearly separable problems. Since the XOR problem requires a nonlinear boundary, as shown in Figure 3, a single-layer perceptron cannot solve it effectively and won't achieve a low error rate. Let's discuss how a perceptron performs on the XOR problem through the following tasks.

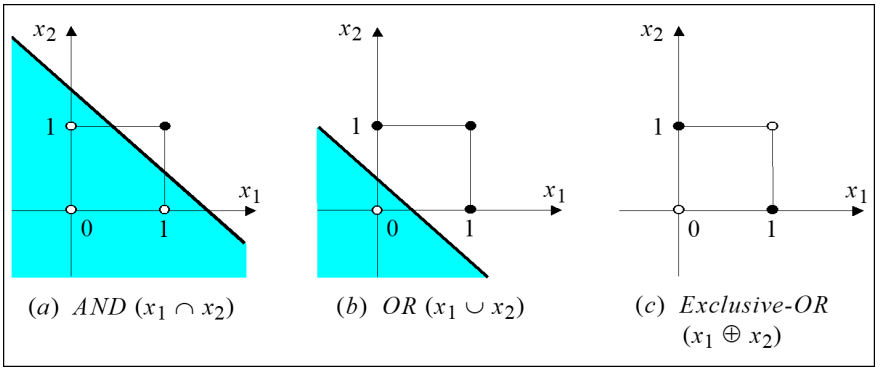

Figure 3: A perceptron can learn the operations *AND* and *OR*, but not *Exclusive-OR*.

Task 6.7: Run the code in Example 1 to solve the logical XOR operation with 1000 epochs and compute the accuracy of the tested samples. [note: you need to change the training and testing data to reflect XOR logical operation.]

Task 6.8: Run the code in Example 1 to solve the logical XOR operation with 10000 epochs and compute the accuracy of the tested samples. Does the perceptron perform well on approximating XOR with more epochs? Justify your answer.

8. Building and experimenting MLPs with Scikit Learn

The neural_network submodule in scikit-learn provides tools and classes related to neural network-based machine learning algorithms. It's designed to offer basic neural network capabilities for tasks like classification and regression. Libraries like TensorFlow, Keras, or PyTorch provide a wider range of features and customization options for building and training neural networks. Sklearn provides two classes for building neural network models, MLPClassifier and MLPRegressor. The MLPClassifier is a Multi-Layer perceptron specifically designed for classification tasks. The MLPClassifier allows you to configure the number of hidden layers, the number of neurons in each hidden layer, and other hyperparameters. MLPClassifier can be used for both binary classification and multiclass classification tasks. The number of classes in your problem will determine whether you are performing binary or multiclass classification. Example 3 shows a Python code for building and training an MLP using MLPClassifier on the iris data set The code also plots MSE loss vs. epoch number.

Example 3: Python code for building and training an MLP using MLPClassifier

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and validation sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an MLPClassifier with one hidden layer of 10 neurons
mlp = MLPClassifier(hidden_layer_sizes=(10,), max_iter=100)

# Train the MLPClassifier
mlp.fit(X_train, y_train)

# make prediction on the testing part
y_pred = mlp.predict(X_test)

# Calculate and print the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")

# Plot the loss curve
plt.plot(mlp.loss_curve_, marker='o', label='Train Loss')
plt.title('Loss Curve during Training')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Task 6.9: Run the code in Example 3 with the following customization of the MLPClassifier. For each case, save the results and compare the accuracy of the testing samples and the loss plot for each case.

1. hidden_layer_sizes=(10,), max_iter=1000
2. hidden_layer_sizes=(50,), activation=relu, , max_iter=1000, learning_rate=0.01
3. hidden_layer_sizes=(35, 15), activation=tanh, max_iter=1000, learning_rate=0.01

Visualizing the decision boundaries of an MLPClassifier is crucial for understanding how the model separates classes in the input space. It aids in model evaluation, identifying issues like overfitting, assessing feature importance, and guiding hyperparameter tuning. Decision boundary plots offer an intuitive way to communicate complex model behavior and guide further analysis. Example 4 shows a Python code for building and training two MLPClassifier on the iris data set with different parameters. In addition, the code plots MSE loss and the decision boundary of each trained model. For plotting purposes, the model was trained on the two most important features using the information gain selection method.

Example 4: Python code for training two MLPClassifiers and plotting the MSE loss and the decision boundary of each trained model.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
```

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mlxtend.plotting import plot_decision_regions
from sklearn.feature_selection import SelectKBest, mutual_info_classif


# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Select the two most important features based on information gain
k_best = SelectKBest(score_func=mutual_info_classif, k=2)
X_selected = k_best.fit_transform(X, y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2,
random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create two MLPClassifiers with different configurations
mlp1 = MLPClassifier(hidden_layer_sizes=(50,), max_iter=1000, random_state=42)
mlp2 = MLPClassifier(hidden_layer_sizes=(35,15), max_iter=1000, random_state=42)

# Train the MLPClassifiers
mlp1.fit(X_train, y_train)
mlp2.fit(X_train, y_train)

# Create subplots for decision regions and loss curves
fig, axes = plt.subplots(2, 2, figsize=(10, 6))

# Plot loss curve for mlp1
axes[0, 0].plot(mlp1.loss_curve_, marker='o')
axes[0, 0].set_title('MLP1 Loss Curve')
axes[0, 0].set_xlabel('Iteration')
axes[0, 0].set_ylabel('Loss')

# Plot loss curve for mlp2
axes[0, 1].plot(mlp2.loss_curve_, marker='o')
axes[0, 1].set_title('MLP2 Loss Curve')
axes[0, 1].set_xlabel('Iteration')
axes[0, 1].set_ylabel('Loss')

# Plot decision regions for mlp1
plot_decision_regions(X_train, y_train, clf=mlp1, legend=2, ax=axes[1, 0])
axes[1, 0].set_title('Decision Regions - MLP1')
axes[1, 0].set_xlabel('Feature 1')
axes[1, 0].set_ylabel('Feature 2')

# Plot decision regions for mlp2
plot_decision_regions(X_train, y_train, clf=mlp2, legend=2, ax=axes[1, 1])
axes[1, 1].set_title('Decision Regions - MLP2')
axes[1, 1].set_xlabel('Feature 1')
axes[1, 1].set_ylabel('Feature 2')

# Adjust layout for better spacing
plt.tight_layout()

# Show the plot
plt.show()
```

Task 6.10: Run the code in Example 4. According to the decision boundary of each model, which one is better? Justify your answer.

Task 6.11: Modify Example 4 by using PCA to choose the best two features instead of using information gain. According to the decision boundary of each model, which one is better? Justify your answer.

9. Choosing Network Structure

Choosing the right architecture for MLPs is a critical step in the machine learning process. The architecture of an MLP mainly refers to the number of layers, the number of neurons in each layer, and the activation function, where the number of layers is treated as the most important parameter. The number of hidden neurons and the number of layers in MLP have a significant impact on the network's ability to handle different levels of problem complexity, as well as its susceptibility to overfitting and underfitting. In complex problems, determining whether to increase the number of hidden neurons or the number of layers in a neural network depends on various factors. There is no one-size-fits-all answer, and often a combination of both approaches may yield the best results. However, let's consider some guidelines for making this decision:

- **Depth vs. Width:** Increasing the number of hidden neurons in a layer allows the network to capture or learn more complex representations or patterns in the data. Learning more complex representations means capturing non-linear relationships, fine-grained patterns, and subtle variations in the input data. On the other hand, increasing the number of layers enables the network to capture hierarchical features and abstractions. This means that the network's ability to learn and represent information at multiple levels of abstraction In many real-world problems, data can be organized in a hierarchical manner, where high-level features are built upon lower-level features.
- **Overfitting and Regularization:** If overfitting is a concern, it might be better to focus on increasing the number of layers rather than adding many hidden neurons to each layer. Deep architectures with proper regularization techniques can help mitigate overfitting.
- **Vanishing and Exploding Gradients:** Very deep networks might suffer from vanishing gradients, making training difficult. Techniques like batch normalization can help alleviate this issue.
- **Dataset Consideration:** With a small dataset, a simpler architecture with fewer hidden neurons and layers might be more appropriate. Deep networks might overfit due to a lack of sufficient training examples. In addition, a simpler architecture might be more suitable for noisy or outlier-rich datasets to prevent overfitting.
- **Computational Complexity:** Deeper networks generally require more computational resources and longer training times.

The purpose of the activation function is to introduce non-linearity into the output of a neuron. The activation function enables the MLP to capture more complex patterns and makes it capable of learning and approximating a wide variety of functions. When selecting an activation function, it's important to consider the specific characteristics of your problem, the network architecture, and potential challenges like vanishing gradients.

Sigmoid, tanh, and ReLU are the most popular activation functions. Sigmoid and tanh suffer from the vanishing gradient problem due to their gradual saturation, which can slow down training in deep networks. In addition, they are sensitive to outliers. Thus, they are less commonly used in hidden layers of deep networks today. On the other hand, ReLU has become the default choice for most hidden layers in deep networks due to its faster convergence and mitigation of the vanishing gradient problem for positive inputs. ReLU is also more robust to noise, outliers, and computational efficiency. However, ReLU requires careful initialization and regularization techniques. Let's discuss the impact of the activation functions on training FNN through the following tasks.

Example 5 shows a Python code for building and training three MLPClassifiers on the spiral data set with one hidden layer of a different number of neurons. In addition, the code plots MSE loss and the decision boundary of each trained model.

Example 5: Python code for building and training three MLPClassifiers on the spiral data set

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mlxtend.plotting import plot_decision_regions
from sklearn.feature_selection import SelectKBest, mutual_info_classif


file_path = r'C:\Users\Aziz\PycharmProjects\pythonProject\spiral.csv'
data = pd.read_csv(file_path)

# Assuming the last column is the target variable and the rest are features
X = data.iloc[:, :-1]
y = data.iloc[:, -1]
y = y.values
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create three MLPClassifiers with different configurations
mlp1 = MLPClassifier(hidden_layer_sizes=(10), max_iter=1000, random_state=42,
activation='tanh',learning_rate_init=0.001)
mlp2 = MLPClassifier(hidden_layer_sizes=(40), max_iter=1000, random_state=42,
activation='tanh', learning_rate_init=0.001)
mlp3 = MLPClassifier(hidden_layer_sizes=(100), max_iter=1000, random_state=42,
activation='tanh', learning_rate_init=0.001)

# Train the MLPClassifiers
mlp1.fit(X_train, y_train)
mlp2.fit(X_train, y_train)
mlp3.fit(X_train, y_train)

# Create subplots for decision regions and loss curves
fig, axes = plt.subplots(2, 3, figsize=(12, 8))


# Plot loss curve for mlp1
axes[0, 0].plot(mlp1.loss_curve_, marker='o')
axes[0, 0].set_title('MLP1 Loss Curve')
axes[0, 0].set_xlabel('Iteration')
axes[0, 0].set_ylabel('Loss')

# Plot loss curve for mlp2
axes[0, 1].plot(mlp2.loss_curve_, marker='o')
axes[0, 1].set_title('MLP2 Loss Curve')
axes[0, 1].set_xlabel('Iteration')
axes[0, 1].set_ylabel('Loss')

# Plot loss curve for mlp3
axes[0, 2].plot(mlp3.loss_curve_, marker='o')
axes[0, 2].set_title('MLP3 Loss Curve')
axes[0, 2].set_xlabel('Iteration')
axes[0, 2].set_ylabel('Loss')

# Plot decision regions for mlp1
```

```
plot_decision_regions(X_train, y_train, clf=mlp1, legend=2, ax=axes[1, 0])
axes[1, 0].set_title('Decision Regions - MLP1')
axes[1, 0].set_xlabel('Feature 1')
axes[1, 0].set_ylabel('Feature 2')

# Plot decision regions for mlp2
plot_decision_regions(X_train, y_train, clf=mlp2, legend=2, ax=axes[1, 1])
axes[1, 1].set_title('Decision Regions - MLP2')
axes[1, 1].set_xlabel('Feature 1')
axes[1, 1].set_ylabel('Feature 2')

# Plot decision regions for mlp3
plot_decision_regions(X_train, y_train, clf=mlp3, legend=2, ax=axes[1, 2])
axes[1, 2].set_title('Decision Regions - MLP3')
axes[1, 2].set_xlabel('Feature 1')
axes[1, 2].set_ylabel('Feature 2')

# Adjust layout for better spacing
plt.tight_layout()

# Show the plot
plt.show()
```

Task 6.12: Run the code in Example Five and save the resulted plots. 1. Which model performs better in the spiral data set? 2. If we increase the iteration to 10,000, does the first model (mlp1) fit the data? Justify your answer.

Task 6.13: Modify the models in Example 5 (please make a copy) to use the relu activation function. Run the modified code and save the resulting plots. Which model performs better in the spiral data set? 4. Compared to the models built in Example 5, does changing the activation function improve performance in these models? Justify your answer.

Task 6.14: 1. Modify the models in Example 5 (make a copy) to have the following hyperparameters. mlp1: hidden_layer_sizes=(7, 3), mlp2: hidden_layer_sizes=(27, 13), and mlp3 hidden_layer_sizes=(70, 30), all of them using the relu activation function. 2. Run the modified code and save the resulting plots. 3. Which model performs better in the spiral data set? 4. Compared to the models built in Example 5 and Task 13, which of these models performs better in the spiral data set? Justify your answer.

10. Impact of Optimization algorithm

Gradient descent is an optimization algorithm that is used to train deep learning models. There are a number of different variants of gradient descent, each with its own advantages and disadvantages. Some of the most common variants include batch gradient descent, stochastic gradient descent (SGD), mini-batch gradient descent, momentum, and adaptive learning rate algorithms. They all aim to minimize the loss function by iteratively adjusting model parameters (weights and biases). Training methods have distinct impacts on the training convergence speed, sensitivity to noise, and computational efficiency. The table below compares these methods.

Algorithm	Convergence Speed	Sensitivity to Noise/Outliers	Handling Local/Saddle Point	Computational Efficiency
Batch Gradient Descent	Slow	Less sensitive	Can get stuck in saddle points	Less efficient for large datasets

Stochastic Gradient Descent	Faster	More sensitive	Less likely to get stuck, may oscillate	Efficient for large datasets, high variance
Mini-Batch Gradient Descent	Balanced	Moderate sensitivity	Balances stability with efficiency	Efficient for parallel processing, balances
Momentum	Accelerates	Reduces sensitivity	Can help to escape local and saddle points	Adds minimal computational overhead and is efficient
Adaptive Learning Rate (e.g., Adagrad, RMSprop, Adam)	Accelerates	Reduces sensitivity	Can help to escape local and saddle points	Efficient with adaptive adjustments, some overhead

In practice, Adam is a good default choice. However, in many cases, SGD+Momentum can outperform Adam but may require more tuning. For the momentum algorithm, the commonly used momentum value is 0.9. For the Adam algorithm, the commonly used parameters are: β_1: A typical value is 0.9; $\beta 2$: A typical value is 0.999; and ε (epsilon) a typical value is 1e-7 or 1e-8.

Task 6.15: Update the code from example 5 to train three MLPClassifier models with hidden_layer_sizes=(27, 13) and max_iter=1000. Configure the first model with the SGD optimizer, the second with SGD+Momentum (momentum = 0.9), and the third with Adam using the specified parameters above. Evaluate the performance of these optimizers on the spiral dataset and provide reasoning for which optimizer performs better.

11. Handling Overfitting in ANNs

Overfitting occurs when a model learns to perform exceptionally well on the training data but struggles to generalize effectively to unseen or new data. It happens when the network captures not only the underlying patterns in the data but also the noise and random fluctuations present in the training samples. Overfitting happens due to several reasons, such as:

- The training data size is too small and does not contain enough data samples to accurately represent all possible input data values.
- The training data contains large amounts of irrelevant information, called noisy data.
- The model complexity is high (deep layers, huge number of hidden neurons, and too many parameters). In this case, the network may be able to learn the training data too well, including the noise and outliers.

Preventing overfitting in artificial neural networks (ANNs) is crucial for achieving models that generalize well to new data. Several methods have been introduced to prevent overfitting in ANNs, such as feature selection, data augmentation, reducing model complexity, Cross-Validation, regularization techniques, batch normalization, early stopping, and dropouts. In this section, we will demonstrate using regularization techniques to handle overfitting in ANNs. More techniques will be studied later.

Regularization involves adding a penalty term to the loss function during training. This penalty discourages the model from becoming too complex or having large parameter values, which helps control the model's ability to fit noise into the training data. L1 and L2 are the most common types of regularization. The choice of which regularization technique to use depends on the specific problem, dataset, and model architecture. Table 2 summarizes the difference between L1 and L2.

L1 Regularization	L2 Regularization
1. L1 penalizes sum of absolute values of weights.	1. L2 penalizes sum of square values of weights.
2. L1 generates model that is simple and interpretable.	2. L2 regularization is able to learn complex data patterns.
3. L1 is robust to outliers.	3. L2 is not robust to outliers.

In scikit-learn's MLPClassifier, regularization is controlled through the alpha parameter, which represents the L2 regularization term. L2 regularization adds a penalty term to the loss function based on the squared magnitudes of the weights. The optimal value of alpha often needs to be determined through hyperparameter tuning. Too much regularization (very large alpha) may lead to underfitting, while too little regularization (very small alpha) may lead to overfitting. Example 6 shows a Python code for building and training three MLP classifiers on the spiral data set with three hidden layers of a different number of neurons. In addition, the code plots MSE loss and the decision boundary of each trained model. The example illustrates how regularization controls overfitting when increasing the number of layers.

Example 6: Python code for training three MLP classifiers with different value of alpha

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mlxtend.plotting import plot_decision_regions
from sklearn.feature_selection import SelectKBest, mutual_info_classif
import numpy as np



file_path = r'C:\Users\Aziz\PycharmProjects\pythonProject\spiral.csv'
data = pd.read_csv(file_path)

# Add outliers to a random subset of the data
outlier_fraction = 0.2  # Adjust the fraction of outliers based on your
preference
outliers_mask = np.random.rand(data.shape[0]) < outlier_fraction
spiral_data_with_outliers = data.copy()
spiral_data_with_outliers.iloc[outliers_mask, :2] += np.random.uniform(-7, 7,
(np.sum(outliers_mask), 2))

# Assuming the last column is the target variable and the rest are features
X = spiral_data_with_outliers.iloc[:, :-1]
y = spiral_data_with_outliers.iloc[:, -1]
y = y.values
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

mlp1 = MLPClassifier(hidden_layer_sizes=(100, 50, 20), max_iter=1000,
random_state=42, solver='adam',learning_rate_init=0.001, beta_1=0.9,
beta_2=0.999, epsilon=1e-8)
```

```
mlp2    =    MLPClassifier(hidden_layer_sizes=(100,    50,    20),    max_iter=1000,
random_state=42,      solver='adam',learning_rate_init=0.001,           beta_1=0.9,
beta_2=0.999, epsilon=1e-8, alpha=0.001)
mlp3    =    MLPClassifier(hidden_layer_sizes=(100,    50,    20),    max_iter=1000,
random_state=42, solver='adam',
   learning_rate_init=0.001,  beta_1=0.9, beta_2=0.999, epsilon=1e-8, alpha=0.3)

# Train the MLPClassifiers
mlp1.fit(X_train, y_train)
mlp2.fit(X_train, y_train)
mlp3.fit(X_train, y_train)

# Create subplots for decision regions and loss curves
fig, axes = plt.subplots(2, 3, figsize=(12, 8))

# Plot loss curve for mlp1
axes[0, 0].plot(mlp1.loss_curve_, marker='o')
axes[0, 0].set_title('MLP1 Loss Curve')
axes[0, 0].set_xlabel('Iteration')
axes[0, 0].set_ylabel('Loss')

# Plot loss curve for mlp2
axes[0, 1].plot(mlp2.loss_curve_, marker='o')
axes[0, 1].set_title('MLP2 Loss Curve')
axes[0, 1].set_xlabel('Iteration')
axes[0, 1].set_ylabel('Loss')

# Plot loss curve for mlp3
axes[0, 2].plot(mlp3.loss_curve_, marker='o')
axes[0, 2].set_title('MLP3 Loss Curve')
axes[0, 2].set_xlabel('Iteration')
axes[0, 2].set_ylabel('Loss')

# Plot decision regions for mlp1
plot_decision_regions(X_train, y_train, clf=mlp1, legend=2, ax=axes[1, 0])
axes[1, 0].set_title('Decision Regions - MLP1')
axes[1, 0].set_xlabel('Feature 1')
axes[1, 0].set_ylabel('Feature 2')

# Plot decision regions for mlp2
plot_decision_regions(X_train, y_train, clf=mlp2, legend=2, ax=axes[1, 1])
axes[1, 1].set_title('Decision Regions - MLP2')
axes[1, 1].set_xlabel('Feature 1')
axes[1, 1].set_ylabel('Feature 2')

# Plot decision regions for mlp3
plot_decision_regions(X_train, y_train, clf=mlp3, legend=2, ax=axes[1, 2])
axes[1, 2].set_title('Decision Regions - MLP3')
axes[1, 2].set_xlabel('Feature 1')
axes[1, 2].set_ylabel('Feature 2')

# Adjust layout for better spacing
plt.tight_layout()

# Show the plot
plt.show()
```

Task 6.16: Run Example 6 and figure out which value of alpha does better. Justify your answer.

13. Automatic MLP hyperparameter optimization

Tuning the hyperparameters of MLP is crucial to achieving optimal performance and ensuring that your neural network model is well-suited to your specific task and dataset. Hyperparameter tuning involves adjusting various settings that impact the network architecture and training process. Hyperparameter tuning can:

- Lead to significant improvements in your model's performance,
- It helps you strike the right balance between model complexity and generalization (helps prevent overfitting or underfitting).
- Customize your model to the specific characteristics of your data, such as its size, complexity, and noise level.
- Determine the number of hidden layers, the number of neurons per layer, and other architectural choices that have a significant impact on the model's capacity to learn complex relationships in the data.

Manually tuning the hyperparameters of MLP can be a time-consuming and challenging process. To mitigate these drawbacks, it's recommended to use automated hyperparameter optimization techniques. There are a number of automated techniques that can be used to tune MLP hyperparameters. These techniques can be more efficient and less prone to overfitting than manual tuning. Here are some of the automated hyperparameter optimization techniques available in scikit-learn:

- **GridSearchCV:** GridSearchCV performs an exhaustive search over a specified parameter grid. It tries all possible combinations of hyperparameters you define and evaluates the model's performance using cross-validation. It is the most comprehensive hyperparameter tuning technique, but it can be computationally expensive, especially for large datasets or complex problems. GridSearchCV is suitable when you have a limited number of hyperparameters to tune.
- **RandomizedSearchCV:** Similar to GridSearchCV, but instead of exhaustively searching through all parameter combinations, it randomly samples a specified number of parameter combinations. . It is less computationally expensive than GridSearchCV, but it may not find the best hyperparameters for a particular problem. RandomizedSearchCV is useful when the search space is large and you want to explore a broader range of values.

Additionally, there are external libraries like **Optuna** and **Hyperopt** that provide more advanced and efficient automated hyperparameter optimization techniques. Optuna boasts flexibility and various optimization algorithms like TPE and CMA-ES, along with strong integration with scikit-learn. Hyperopt emphasizes sequential model-based optimization and supports parallel execution. These libraries can offer better performance in terms of finding optimal hyperparameters with fewer evaluations. Example 7 shows how to apply GridSearchCV to optimize some MLP hyperparameters on the spiral dataset.

Example 7: using GridSearchCV to optimize some MLP hyperparameters

```
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
import numpy as np

file_path = r'C:\Users\Aziz\PycharmProjects\pythonProject\spiral.csv'
data = pd.read_csv(file_path)

# Add outliers to a random subset of the data
outlier_fraction = 0.2  # Adjust the fraction of outliers based on your
preference
outliers_mask = np.random.rand(data.shape[0]) < outlier_fraction
spiral_data_with_outliers = data.copy()
spiral_data_with_outliers.iloc[outliers_mask, :2] += np.random.uniform(-7, 7,
(np.sum(outliers_mask), 2))
```

```
# Assuming the last column is the target variable and the rest are features
X = spiral_data_with_outliers.iloc[:, :-1]
y = spiral_data_with_outliers.iloc[:, -1]
y = y.values
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create the neural network model
mlp = MLPClassifier(max_iter=1000, random_state=42)

# Define hyperparameter grid
param_grid = {
    'hidden_layer_sizes': [(20), (60), (100), (13, 7), (40, 20)],
    'activation': ['relu', 'tanh'],
    }

# Instantiate GridSearchCV with the model and parameter grid
grid_search = GridSearchCV(mlp, param_grid, cv=5, scoring='accuracy')

# Perform the hyperparameter search on the training data
grid_search.fit(X_train, y_train)

# Get the best hyperparameters and their corresponding accuracy
best_params = grid_search.best_params_
best_accuracy = grid_search.best_score_

# Create the best model with the best parameters and train on the full training
set
best_model = MLPClassifier(**best_params, max_iter=1000, random_state=42)
best_model.fit(X_train, y_train)

# Evaluate the best model on the test set
test_accuracy = best_model.score(X_test, y_test)

# Print results
print("Best Hyperparameters:", best_params)
print("Best Accuracy:", best_accuracy)
print("Test Accuracy:", test_accuracy)
```

Task 6.17: Run the code in Example 6 and figure out what the optimal hyperparameters are.
Task 6.18: Modify and run the code in example 6 to include solver [sgd and adam], learning_rate_init [0.001, 0.01, 0.1], and alpha [0.001, 0.01, 0.1]. Figure out what the optimal hyperparameters are.