1. Implementation of K Nearest Neighbour Algorithm

K-Nearest Neighbours is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining, and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data). We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:

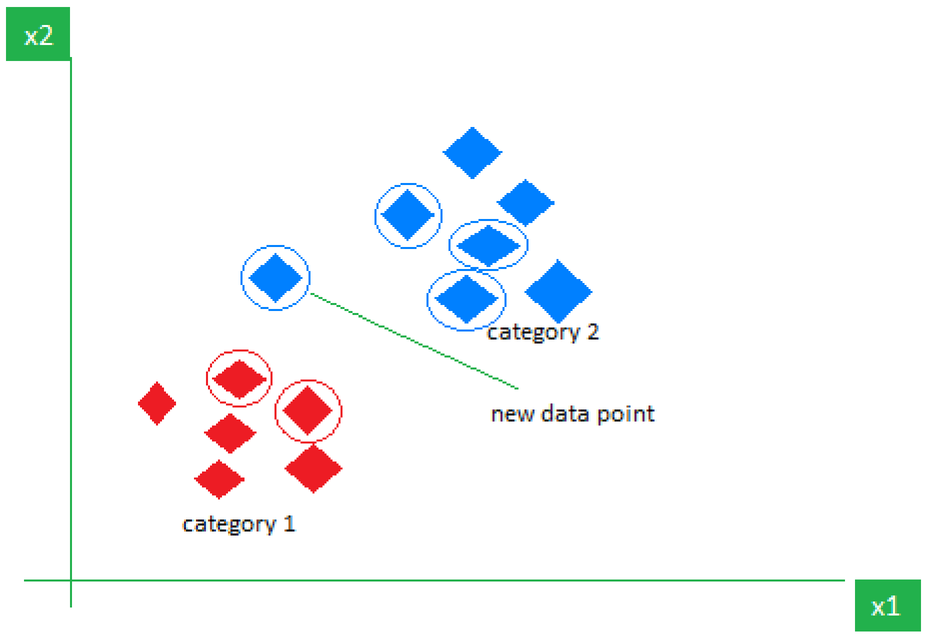
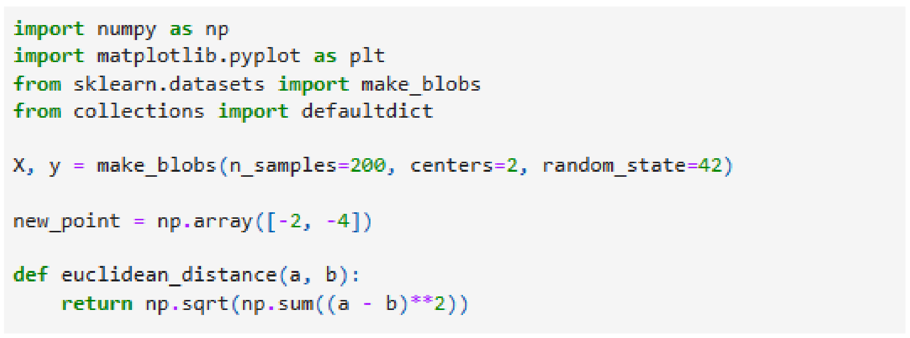


Fig. 1.1 KNN Algorithm working visualization

Now, given another set of data points (also called testing data), allocate these points to a group by analyzing the training set.

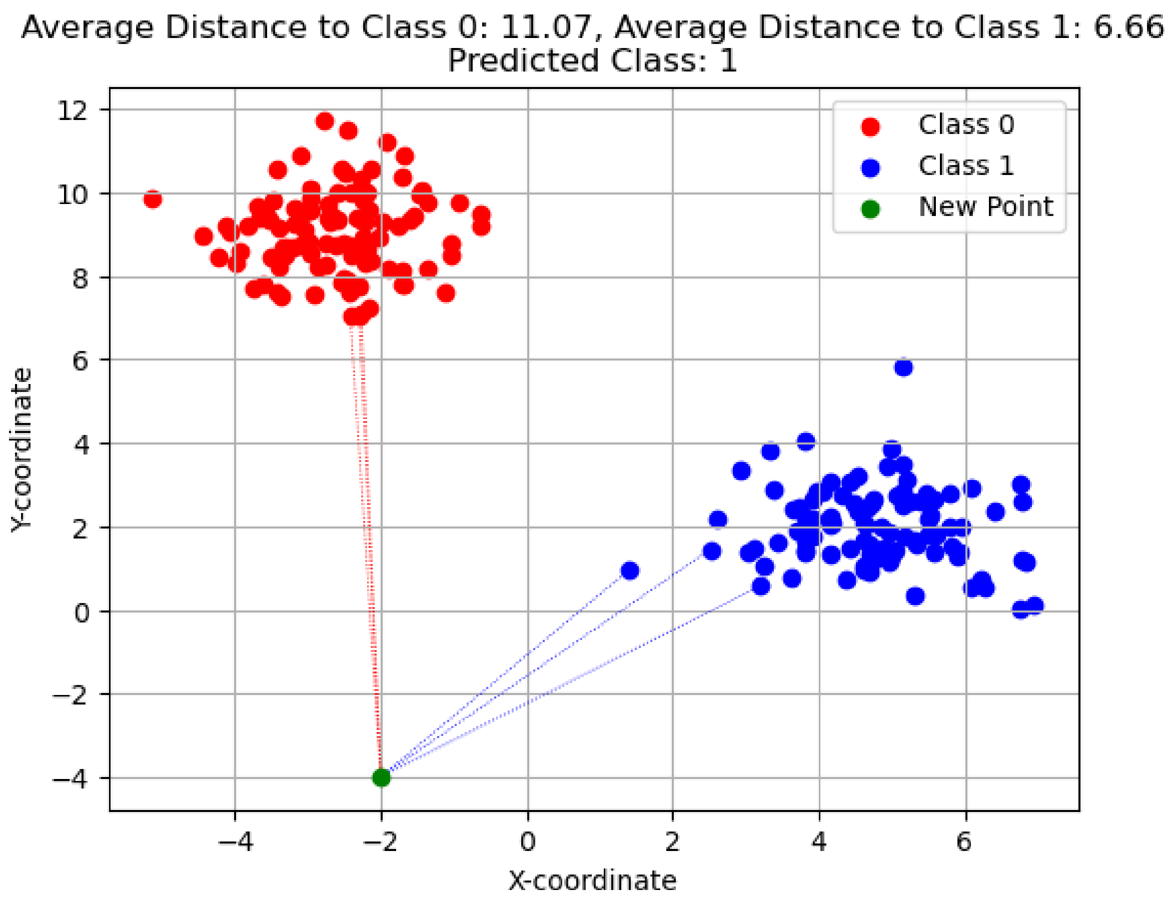
Code:



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from collections import defaultdict

X, y = make_blobs(n_samples=200, centers=2, random_state=42)

new_point = np.array([-2, -4])

def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b)**2))
```

```python
k = 3
class_distances = defaultdict(list)

distances = [(euclidean_distance(new_point, x), class_label, x) for x, class_label in zip(X, y)]
distances.sort()

for distance, class_label, x in distances:
    if len(class_distances[class_label]) < k:
        class_distances[class_label].append((distance, x))

avg_distances = {class_label: np.mean([d[0] for d in dist]) for class_label, dist in class_distances.items()}

predicted_class = min(avg_distances, key=avg_distances.get)

plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='r', label='Class 0')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='b', label='Class 1')
plt.scatter(new_point[0], new_point[1], color='g', label='New Point', zorder=5)

for class_label, dist in class_distances.items():
    color = 'r' if class_label == 0 else 'b'
    for distance, x in dist:
        plt.plot([new_point[0], x[0]], [new_point[1], x[1]], f'{color}:', linewidth=0.6)

plt.legend()
plt.title(f'Average Distance to Class 0: {avg_distances[0]:.2f}, Average Distance to Class 1: {avg_distances[1]:.2f}\nPredicted Class: {predicted_class}')
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.grid(True)
plt.show()
```

Output:



2. Implementation of OR, AND and XOR Gate by Single Layer Perceptron

Single Layer Perceptron is one of the oldest and first introduced neural networks. It was proposed by Frank Rosenblatt in 1958. Perceptron is also known as an artificial neural network. Perceptron is mainly used to compute the logical gate like AND, OR, and XOR which has binary input and binary output.
The main functionality of the perceptron is:-

- Takes input from the input layer
- Weight them up and sum it up.
- Pass the sum to the nonlinear function to produce the output.

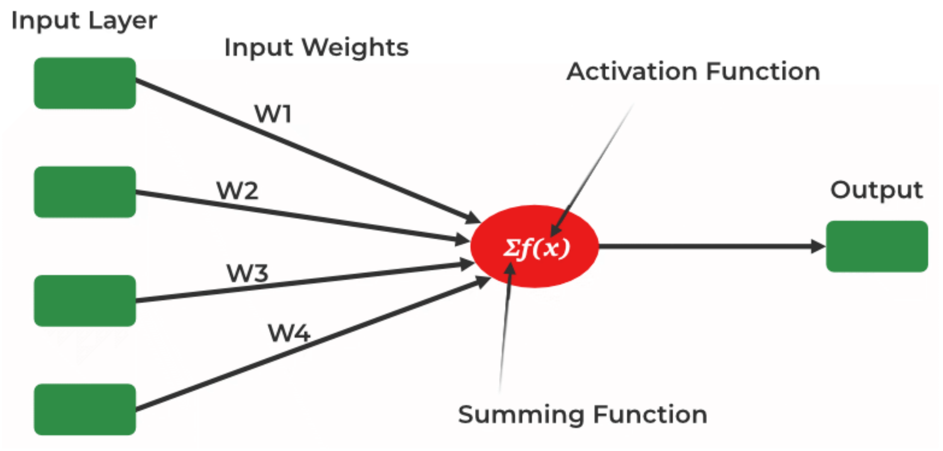


Fig. 2.1 Single-layer neural network

Here activation functions can be anything like sigmoid, tanh, relu Based on the requirement we will be choosing the most appropriate nonlinear activation function to produce the better result. Now let us implement a single-layer perceptron.

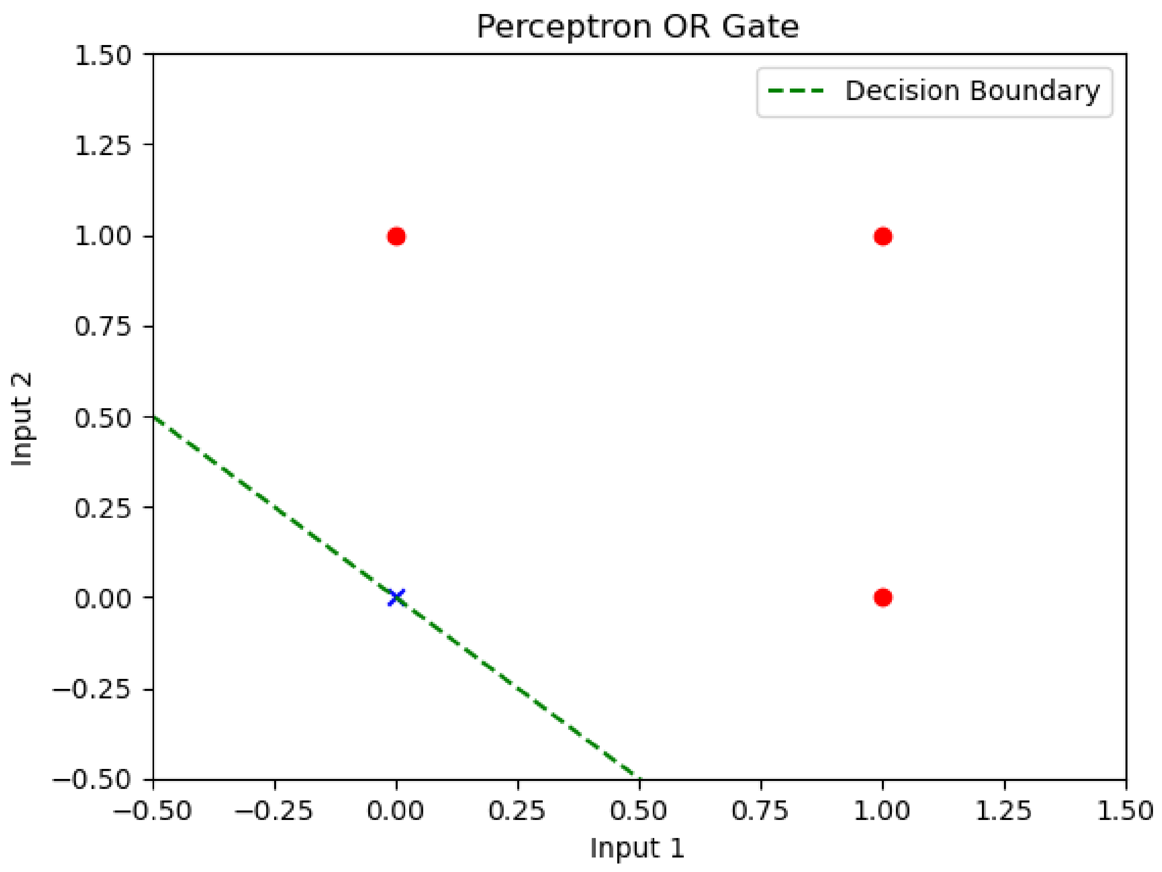
2.1 OR Gate:

```python
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, learning_rate=0.1):
        self.weights = np.zeros(3)
        self.learning_rate = learning_rate

    def predict(self, inputs):
             inputs = np.append(inputs, 1)
        summation = np.dot(inputs, self.weights)
        return self.activation_function(summation)

    def activation_function(self, x):
        return 1 if x > 0 else 0

    def train(self, training_inputs, labels, epochs):
        for epoch in range(epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                error = label - prediction

                inputs = np.append(inputs, 1)
                self.weights += self.learning_rate * error * inputs
```

```
training_inputs = np.array([[0, 0],
                            [0, 1],
                            [1, 0],
                            [1, 1]])
labels = np.array([0, 1, 1, 1])

perceptron = Perceptron(learning_rate=0.1)
perceptron.train(training_inputs, labels, epochs=50)

predictions = []
for inputs in training_inputs:
    prediction = perceptron.predict(inputs)
    predictions.append(prediction)
    print(f"Input: {inputs}, Prediction: {prediction}")

plt.figure()
for point, label in zip(training_inputs, labels):
    plt.scatter(point[0], point[1], marker='o' if label == 1 else 'x', c='r' if label == 1 else 'b')

x_values = np.linspace(-0.5, 1.5, 100)
y_values = -(perceptron.weights[0]/perceptron.weights[1])*x_values - (perceptron.weights[2]/perceptron.weights[1])
plt.plot(x_values, y_values, 'g--', label='Decision Boundary')

plt.xlim(-0.5, 1.5)
plt.ylim(-0.5, 1.5)
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('Perceptron OR Gate')
plt.legend()
plt.show()
```

Output:

```
Input: [0 0], Prediction: 0
Input: [0 1], Prediction: 1
Input: [1 0], Prediction: 1
Input: [1 1], Prediction: 1
```



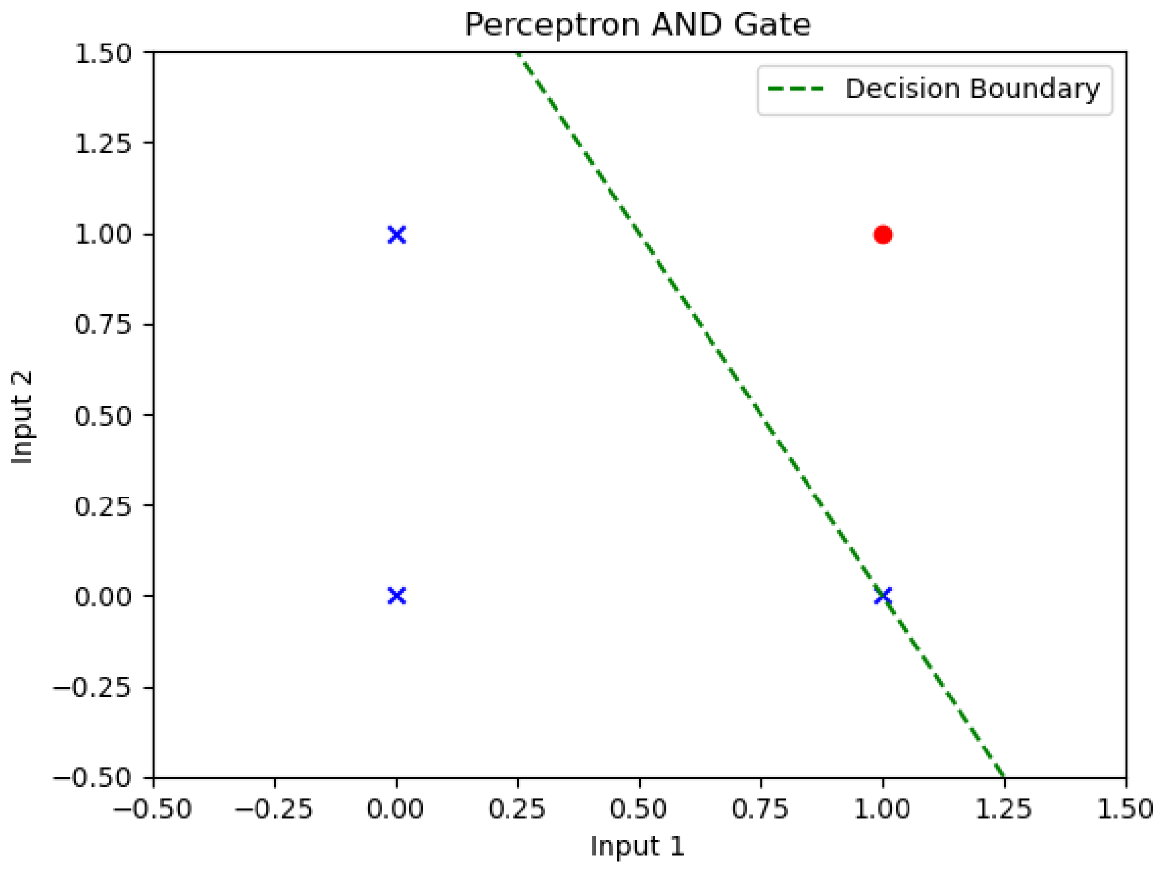
2.2 AND Gate:

```python
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, learning_rate=0.1):
        self.weights = np.zeros(3)
        self.learning_rate = learning_rate

    def predict(self, inputs):

        inputs = np.append(inputs, 1)
        summation = np.dot(inputs, self.weights)
        return self.activation_function(summation)

    def activation_function(self, x):
        return 1 if x > 0 else 0

    def train(self, training_inputs, labels, epochs):
        for epoch in range(epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                error = label - prediction

                inputs = np.append(inputs, 1)
                self.weights += self.learning_rate * error * inputs
```

```python
training_inputs = np.array([[0, 0],
                            [0, 1],
                            [1, 0],
                            [1, 1]])
labels = np.array([0, 0, 0, 1])

perceptron = Perceptron(learning_rate=0.1)
perceptron.train(training_inputs, labels, epochs=50)

predictions = []
for inputs in training_inputs:
    prediction = perceptron.predict(inputs)
    predictions.append(prediction)
    print(f"Input: {inputs}, Prediction: {prediction}")

plt.figure()
for point, label in zip(training_inputs, labels):
    plt.scatter(point[0], point[1], marker='o' if label == 1 else 'x', c='r' if label == 1 else 'b')

x_values = np.linspace(-0.5, 1.5, 100)
y_values = -(perceptron.weights[0]/perceptron.weights[1])*x_values - (perceptron.weights[2]/perceptron.weights[1])
plt.plot(x_values, y_values, 'g--', label='Decision Boundary')

plt.xlim(-0.5, 1.5)
plt.ylim(-0.5, 1.5)
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('Perceptron AND Gate')
plt.legend()
plt.show()
```

Output:

```
Input: [0 0], Prediction: 0
Input: [0 1], Prediction: 0
Input: [1 0], Prediction: 0
Input: [1 1], Prediction: 1
```

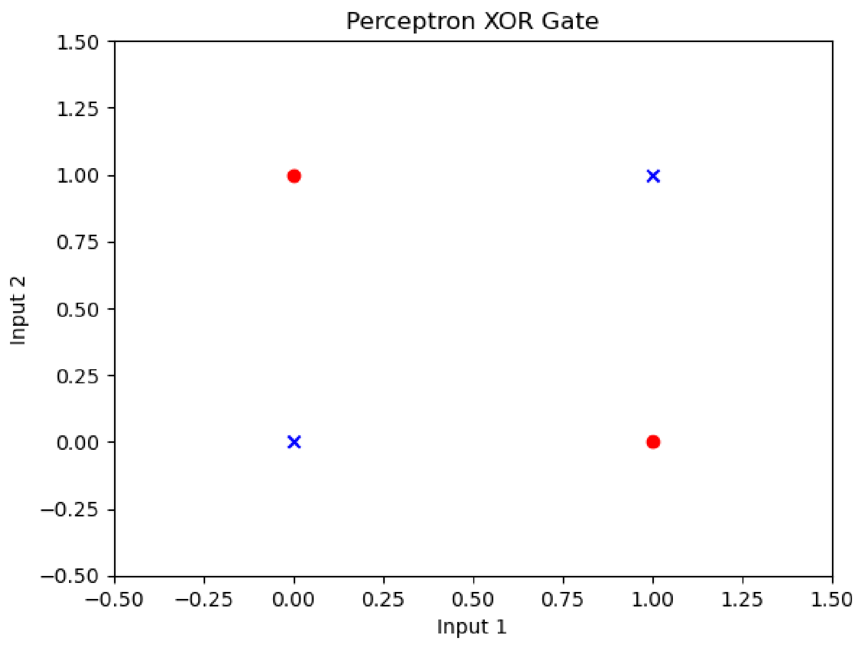


2.3 XOR Gate

```
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, learning_rate=0.1):
        self.weights = np.zeros(3)
        self.learning_rate = learning_rate

    def predict(self, inputs):
        # Add bias to inputs
        inputs = np.append(inputs, 1)
        summation = np.dot(inputs, self.weights)
        return self.activation_function(summation)

    def activation_function(self, x):
        return 1 if x > 0 else 0
    def train(self, training_inputs, labels, epochs):
        for epoch in range(epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                error = label - prediction

                inputs = np.append(inputs, 1)
                self.weights += self.learning_rate * error * inputs

training_inputs = np.array([[0, 0],
                            [0, 1],
                            [1, 0],
                            [1, 1]])
labels = np.array([0, 1, 1, 0])

perceptron = Perceptron(learning_rate=0.1)
perceptron.train(training_inputs, labels, epochs=50)

predictions = []
for inputs in training_inputs:
    prediction = perceptron.predict(inputs)
    predictions.append(prediction)
    print(f"Input: {inputs}, Prediction: {prediction}")

plt.figure()
for point, label in zip(training_inputs, labels):
    plt.scatter(point[0], point[1], marker='o' if label == 1 else 'x', c='r' if label == 1 else 'b')

plt.xlim(-0.5, 1.5)
plt.ylim(-0.5, 1.5)
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('Perceptron XOR Gate')
plt.show()
```

Output:

```
Input: [0 0], Prediction: 1
Input: [0 1], Prediction: 1
Input: [1 0], Prediction: 0
Input: [1 1], Prediction: 0
```



3. Implementation of Multilayer Perceptron Algorithm using Iris dataset

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A multi-layer perceptron has one input layer and for each input, there is one neuron (or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.

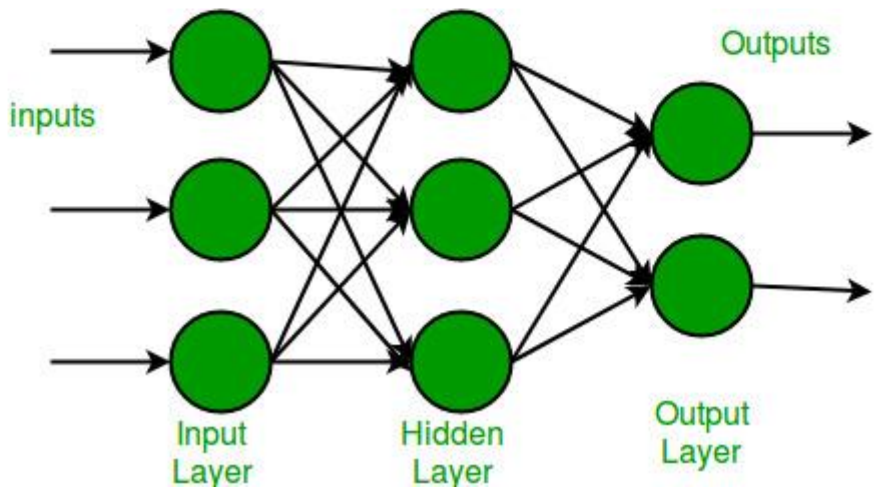
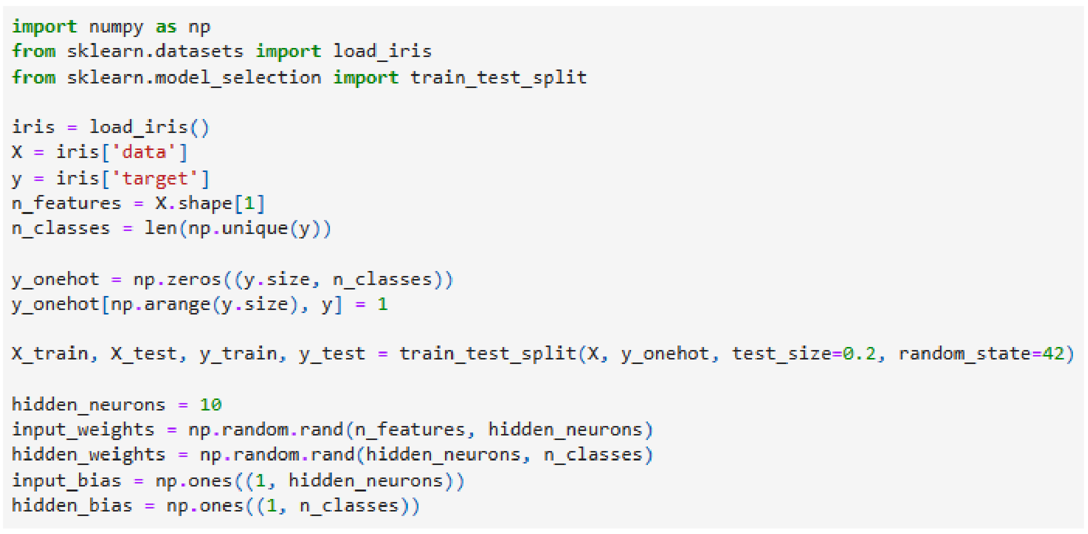


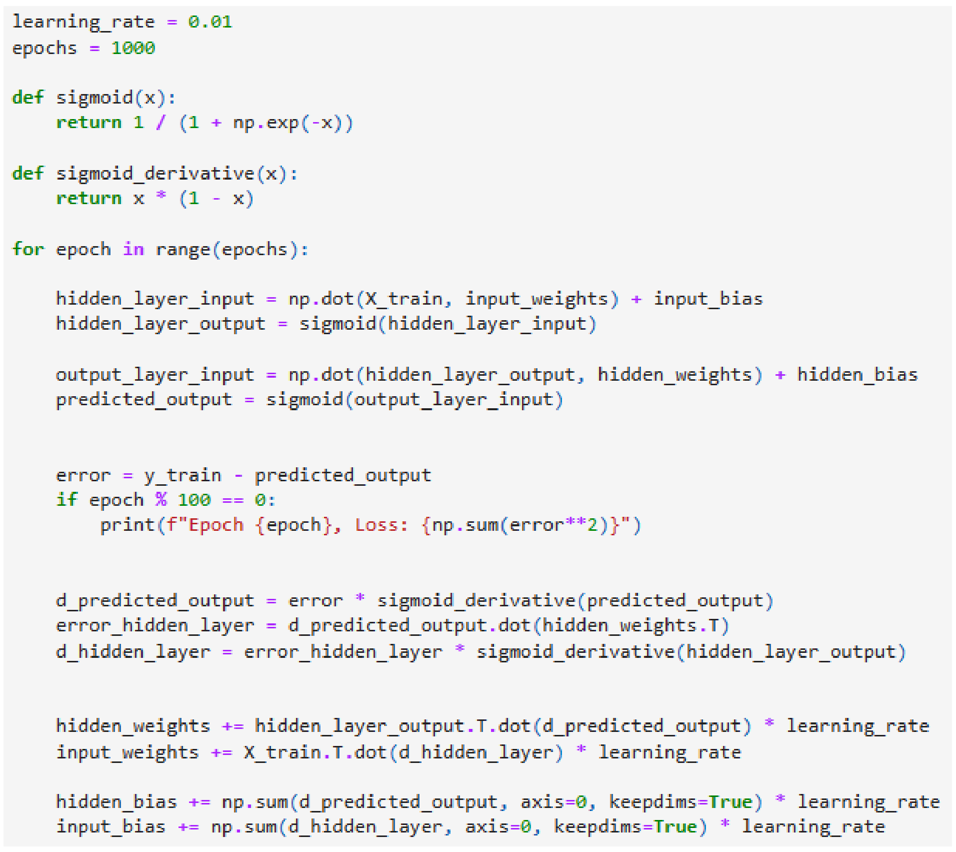
Fig. 3.1 Multilayer Perceptron Visualization

In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

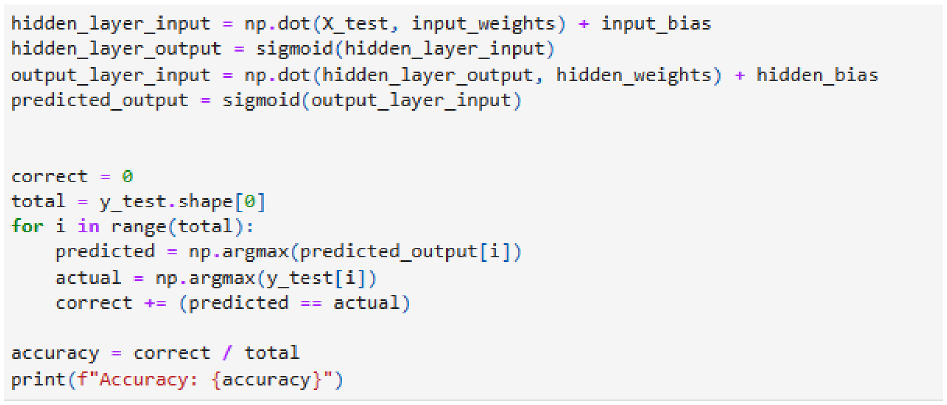
Code and Output:



```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris['data']
y = iris['target']
n_features = X.shape[1]
n_classes = len(np.unique(y))

y_onehot = np.zeros((y.size, n_classes))
y_onehot[np.arange(y.size), y] = 1

X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=42)

hidden_neurons = 10
input_weights = np.random.rand(n_features, hidden_neurons)
hidden_weights = np.random.rand(hidden_neurons, n_classes)
input_bias = np.ones((1, hidden_neurons))
hidden_bias = np.ones((1, n_classes))
```

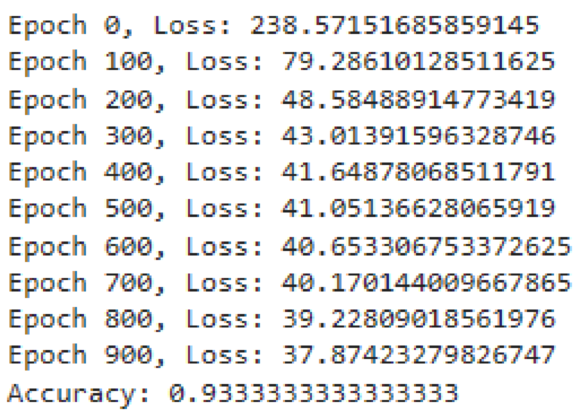


```python
learning_rate = 0.01
epochs = 1000

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

def sigmoid_derivative(x):
    return x * (1 - x)

for epoch in range(epochs):

    hidden_layer_input = np.dot(X_train, input_weights) + input_bias
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, hidden_weights) + hidden_bias
    predicted_output = sigmoid(output_layer_input)


    error = y_train - predicted_output
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {np.sum(error**2)}")


    d_predicted_output = error * sigmoid_derivative(predicted_output)
    error_hidden_layer = d_predicted_output.dot(hidden_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)


    hidden_weights += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
    input_weights += X_train.T.dot(d_hidden_layer) * learning_rate

    hidden_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
    input_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
```



```python
hidden_layer_input = np.dot(X_test, input_weights) + input_bias
hidden_layer_output = sigmoid(hidden_layer_input)
output_layer_input = np.dot(hidden_layer_output, hidden_weights) + hidden_bias
predicted_output = sigmoid(output_layer_input)


correct = 0
total = y_test.shape[0]
for i in range(total):
    predicted = np.argmax(predicted_output[i])
    actual = np.argmax(y_test[i])
    correct += (predicted == actual)

accuracy = correct / total
print(f"Accuracy: {accuracy}")
```

Output:

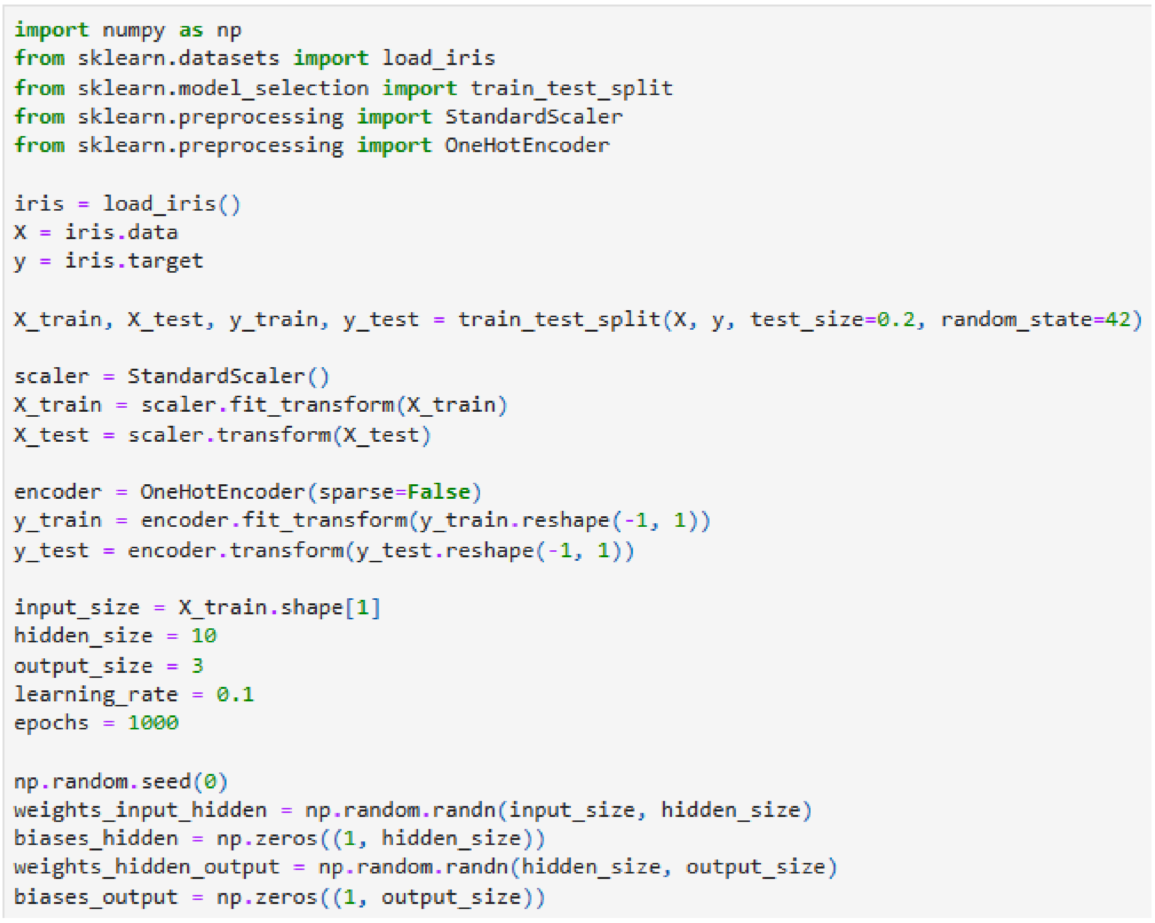


```
Epoch 0, Loss: 238.57151685859145
Epoch 100, Loss: 79.28610128511625
Epoch 200, Loss: 48.58488914773419
Epoch 300, Loss: 43.01391596328746
Epoch 400, Loss: 41.64878068511791
Epoch 500, Loss: 41.05136628065919
Epoch 600, Loss: 40.653306753372625
Epoch 700, Loss: 40.170144009667865
Epoch 800, Loss: 39.22809018561976
Epoch 900, Loss: 37.87423279826747
Accuracy: 0.9333333333333333
```

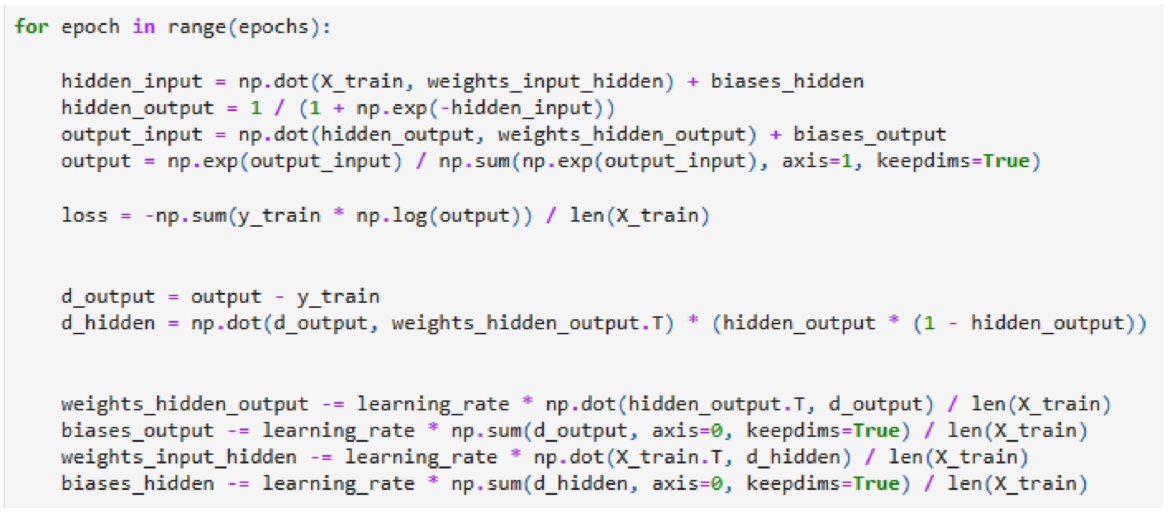
4. Implementation of Kolmogorov Theorem for 3 layer and 12 layer

The Kolmogorov-Arnold representation theorem, also known simply as Kolmogorov's theorem, is a foundational result in the field of function approximation and functional analysis. It was proven by Andrey Kolmogorov in 1957, with extensions and generalizations later provided by Vladimir Arnold. The theorem provides a theoretical underpinning for the approximation capabilities of artificial neural networks, particularly feedforward neural networks with a single hidden layer.

4.1 For 3 Layer



```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

encoder = OneHotEncoder(sparse=False)
y_train = encoder.fit_transform(y_train.reshape(-1, 1))
y_test = encoder.transform(y_test.reshape(-1, 1))

input_size = X_train.shape[1]
hidden_size = 10
output_size = 3
learning_rate = 0.1
epochs = 1000

np.random.seed(0)
weights_input_hidden = np.random.randn(input_size, hidden_size)
biases_hidden = np.zeros((1, hidden_size))
weights_hidden_output = np.random.randn(hidden_size, output_size)
biases_output = np.zeros((1, output_size))
```



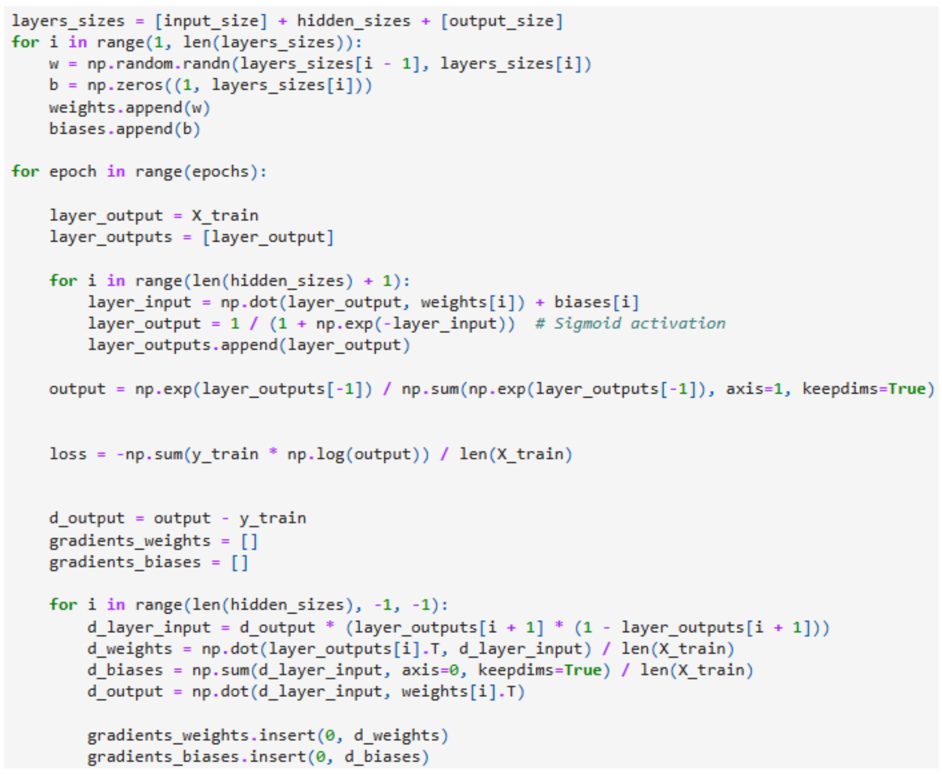
```python
for epoch in range(epochs):

    hidden_input = np.dot(X_train, weights_input_hidden) + biases_hidden
    hidden_output = 1 / (1 + np.exp(-hidden_input))
    output_input = np.dot(hidden_output, weights_hidden_output) + biases_output
    output = np.exp(output_input) / np.sum(np.exp(output_input), axis=1, keepdims=True)

    loss = -np.sum(y_train * np.log(output)) / len(X_train)


    d_output = output - y_train
    d_hidden = np.dot(d_output, weights_hidden_output.T) * (hidden_output * (1 - hidden_output))


    weights_hidden_output -= learning_rate * np.dot(hidden_output.T, d_output) / len(X_train)
    biases_output -= learning_rate * np.sum(d_output, axis=0, keepdims=True) / len(X_train)
    weights_input_hidden -= learning_rate * np.dot(X_train.T, d_hidden) / len(X_train)
    biases_hidden -= learning_rate * np.sum(d_hidden, axis=0, keepdims=True) / len(X_train)
    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.4f}")

hidden_input_test = np.dot(X_test, weights_input_hidden) + biases_hidden
hidden_output_test = 1 / (1 + np.exp(-hidden_input_test))
output_input_test = np.dot(hidden_output_test, weights_hidden_output) + biases_output
output_test = np.exp(output_input_test) / np.sum(np.exp(output_input_test), axis=1, keepdims=True)

predictions = np.argmax(output_test, axis=1)

accuracy = np.mean(predictions == y_test.argmax(axis=1))
print(f"Test Accuracy: {accuracy:.4f}")
```

Output:

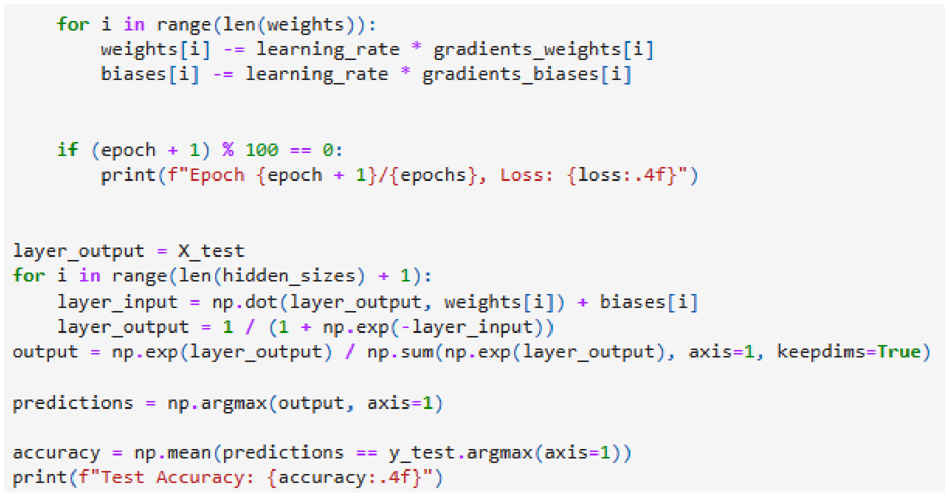
```
Epoch 100/1000, Loss: 0.4699
Epoch 200/1000, Loss: 0.3648
Epoch 300/1000, Loss: 0.3086
Epoch 400/1000, Loss: 0.2674
Epoch 500/1000, Loss: 0.2344
Epoch 600/1000, Loss: 0.2071
Epoch 700/1000, Loss: 0.1845
Epoch 800/1000, Loss: 0.1658
Epoch 900/1000, Loss: 0.1503
Epoch 1000/1000, Loss: 0.1375
Test Accuracy: 1.0000
```

4.2 For 12 Layer

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

encoder = OneHotEncoder(sparse=False)
y_train = encoder.fit_transform(y_train.reshape(-1, 1))
y_test = encoder.transform(y_test.reshape(-1, 1))

input_size = X_train.shape[1]
hidden_sizes = [32, 64, 64, 64, 64, 32, 32, 32, 32, 32, 32, 32]
output_size = 3
learning_rate = 0.1
epochs = 1000

np.random.seed(0)
weights = []
biases = []
```

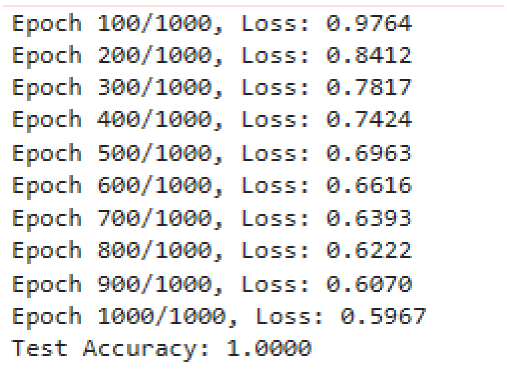


```python
layers_sizes = [input_size] + hidden_sizes + [output_size]
for i in range(1, len(layers_sizes)):
    w = np.random.randn(layers_sizes[i - 1], layers_sizes[i])
    b = np.zeros((1, layers_sizes[i]))
    weights.append(w)
    biases.append(b)

for epoch in range(epochs):

    layer_output = X_train
    layer_outputs = [layer_output]

    for i in range(len(hidden_sizes) + 1):
        layer_input = np.dot(layer_output, weights[i]) + biases[i]
        layer_output = 1 / (1 + np.exp(-layer_input))  # Sigmoid activation
        layer_outputs.append(layer_output)

    output = np.exp(layer_outputs[-1]) / np.sum(np.exp(layer_outputs[-1]), axis=1, keepdims=True)


    loss = -np.sum(y_train * np.log(output)) / len(X_train)


    d_output = output - y_train
    gradients_weights = []
    gradients_biases = []

    for i in range(len(hidden_sizes), -1, -1):
        d_layer_input = d_output * (layer_outputs[i + 1] * (1 - layer_outputs[i + 1]))
        d_weights = np.dot(layer_outputs[i].T, d_layer_input) / len(X_train)
        d_biases = np.sum(d_layer_input, axis=0, keepdims=True) / len(X_train)
        d_output = np.dot(d_layer_input, weights[i].T)

        gradients_weights.insert(0, d_weights)
        gradients_biases.insert(0, d_biases)
```



```python
    for i in range(len(weights)):
        weights[i] -= learning_rate * gradients_weights[i]
        biases[i] -= learning_rate * gradients_biases[i]


    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.4f}")


layer_output = X_test
for i in range(len(hidden_sizes) + 1):
    layer_input = np.dot(layer_output, weights[i]) + biases[i]
    layer_output = 1 / (1 + np.exp(-layer_input))
output = np.exp(layer_output) / np.sum(np.exp(layer_output), axis=1, keepdims=True)

predictions = np.argmax(output, axis=1)

accuracy = np.mean(predictions == y_test.argmax(axis=1))
print(f"Test Accuracy: {accuracy:.4f}")
```

Output:



```
Epoch 100/1000, Loss: 0.9764
Epoch 200/1000, Loss: 0.8412
Epoch 300/1000, Loss: 0.7817
Epoch 400/1000, Loss: 0.7424
Epoch 500/1000, Loss: 0.6963
Epoch 600/1000, Loss: 0.6616
Epoch 700/1000, Loss: 0.6393
Epoch 800/1000, Loss: 0.6222
Epoch 900/1000, Loss: 0.6070
Epoch 1000/1000, Loss: 0.5967
Test Accuracy: 1.0000
```

5. Implementation of K Means Clustering

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabelled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.

It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabelled dataset on its own without the need for any training.

It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

The algorithm takes the unlabelled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.

The k-means clustering algorithm mainly performs two tasks:

Determines the best value for K center points or centroids by an iterative process.

Assigns each data point to its closest k-center. Those data points which are near to the particular k-center, create a cluster.

Hence each cluster has datapoints with some commonalities, and it is away from other clusters.

The below diagram explains the working of the K-means Clustering Algorithm:

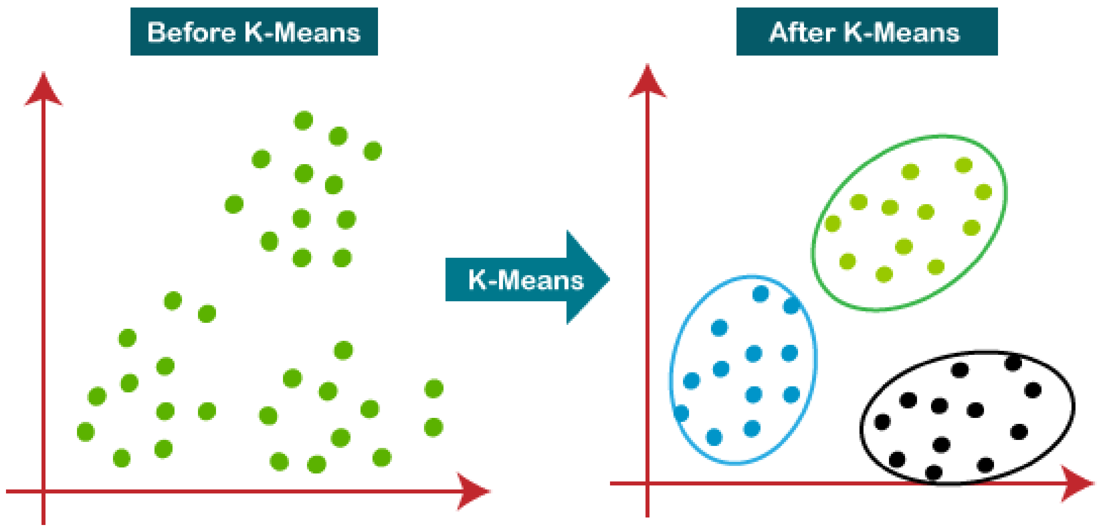
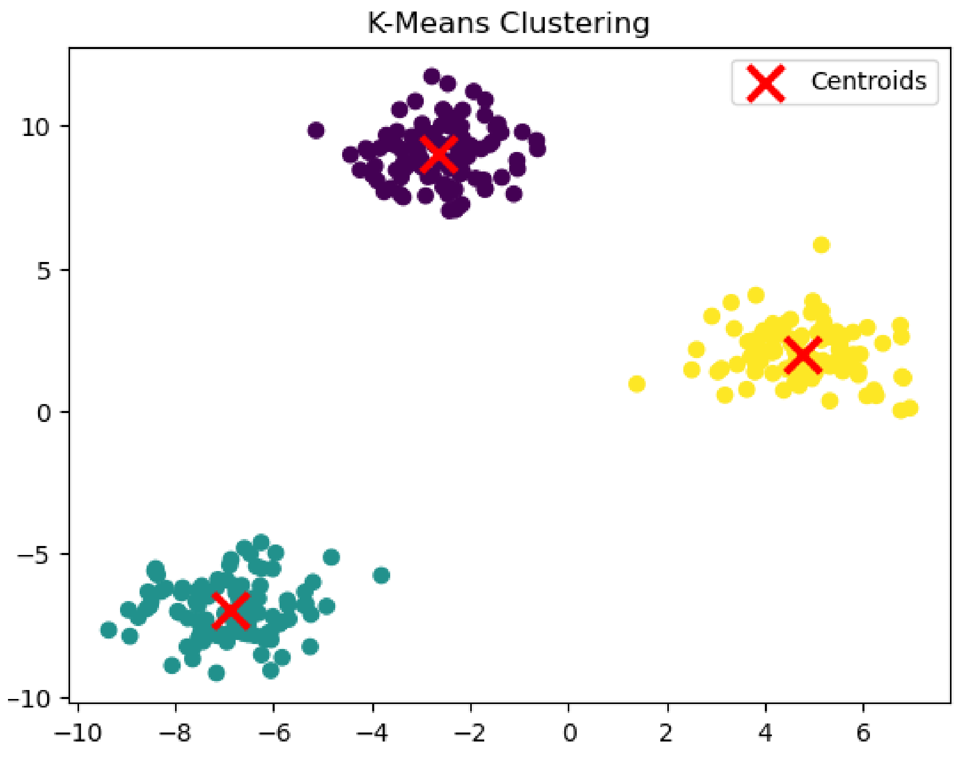


Fig. 5.1 K-Means Clustering Visualization

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

n_samples = 300
n_features = 2
n_clusters = 3

X, _ = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_clusters, random_state=42)

k = 3

np.random.seed(42)
centroids = X[np.random.choice(X.shape[0], k, replace=False)]

max_iterations = 100

for iteration in range(max_iterations):

    distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)


    new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(k)])


    if np.all(centroids == new_centroids):
        break

    centroids = new_centroids

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=200, linewidths=3, color='r', label='Centroids')
plt.legend()
plt.title('K-Means Clustering')
plt.show()
```

Output:



6. Implementation of Radial Basis Function

Radial basis function (RBF) networks are a common type of use in artificial neural networks for function approximation problems. Radial-based function networks are distinguished from other neural networks due to their global approximation and fast learning speed. and RBF neural networks are also a type of feed-forward network trained using a supervised training algorithm. The main advantage of the RBF network is that it has only one hidden layer and uses the radial basis function as the activation function. These functions are very powerful in approximation.

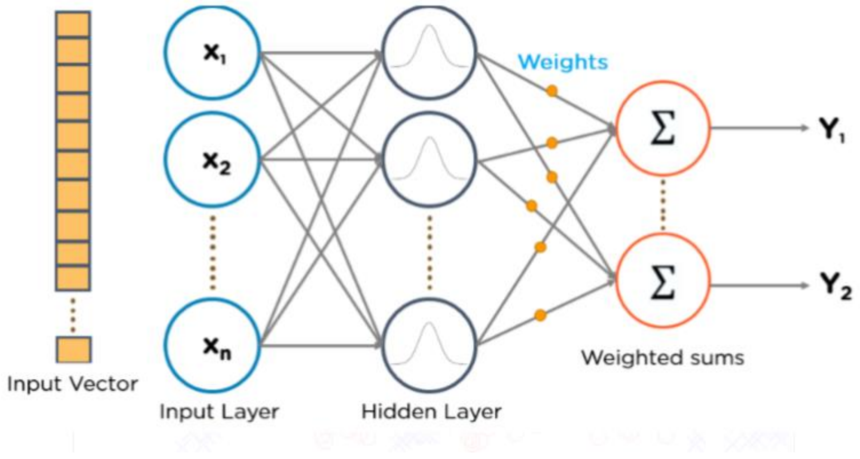
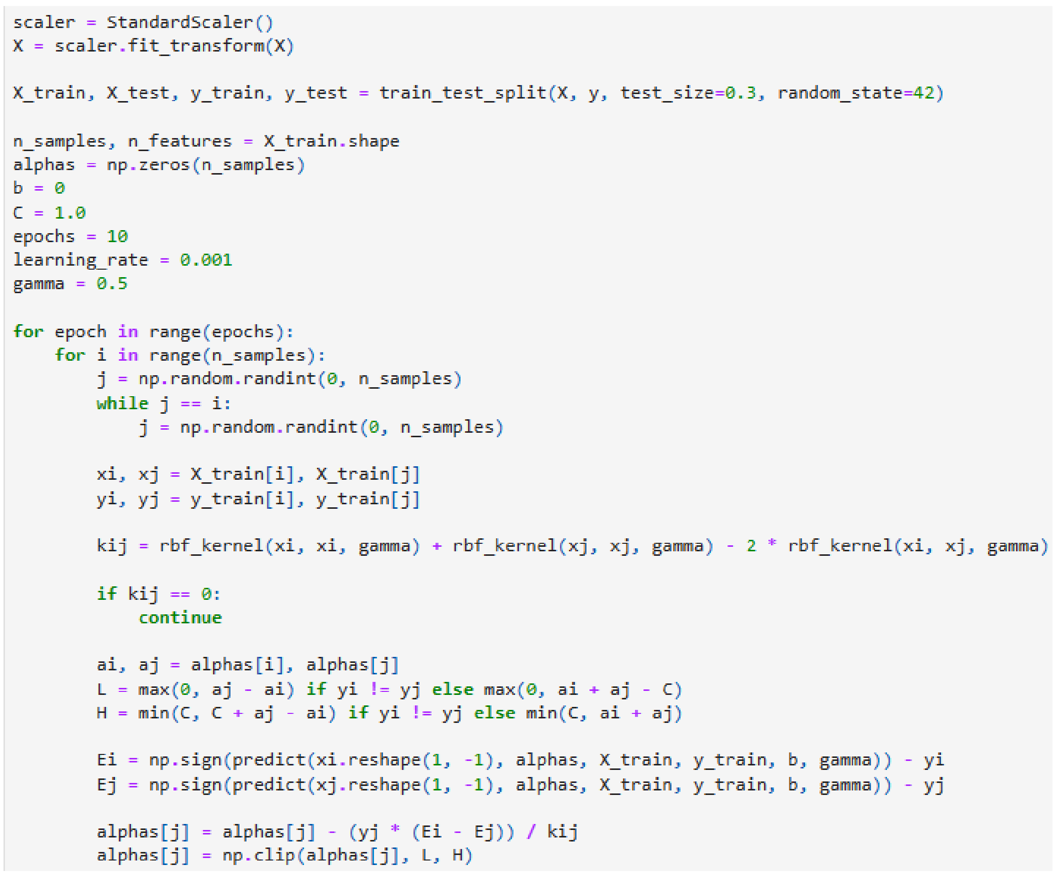


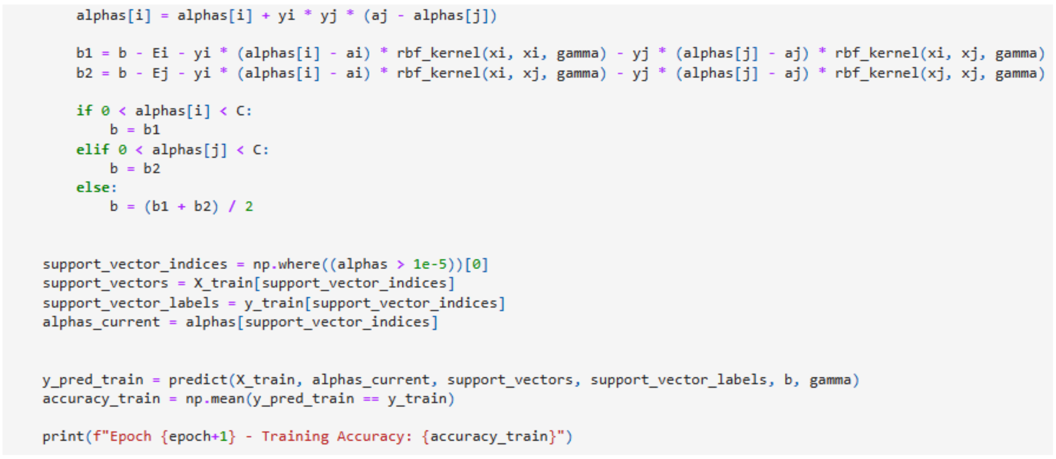
Fig. 6.1 Radial Basis Function Network Visualization

Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

def rbf_kernel(x1, x2, gamma=0.5):
    return np.exp(-gamma * np.linalg.norm(x1 - x2) ** 2)

def predict(X, alphas, support_vectors, support_vector_labels, b, gamma):
    y_pred = []
    for i in range(len(X)):
        prediction = 0
        for alpha, sv, sv_label in zip(alphas, support_vectors, support_vector_labels):
            prediction += alpha * sv_label * rbf_kernel(X[i], sv, gamma)
        prediction += b
        y_pred.append(np.sign(prediction))
    return np.array(y_pred)

iris = load_iris()
X, y = iris.data, iris.target

X, y = X[y != 2], y[y != 2]

y[y == 0] = -1
```

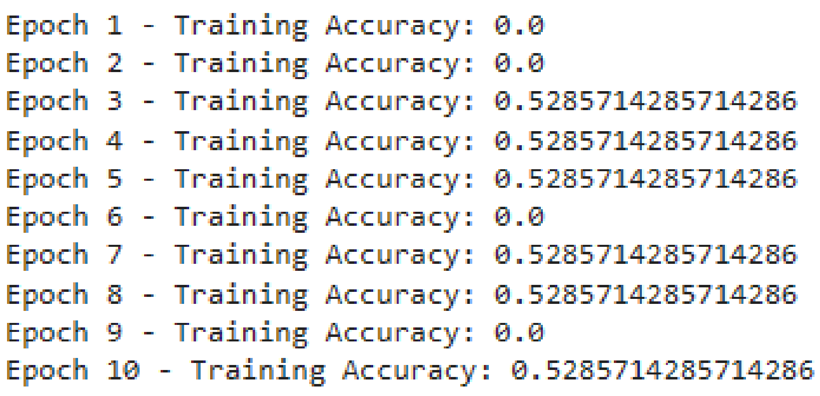


```python
scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

n_samples, n_features = X_train.shape
alphas = np.zeros(n_samples)
b = 0
C = 1.0
epochs = 10
learning_rate = 0.001
gamma = 0.5

for epoch in range(epochs):
    for i in range(n_samples):
        j = np.random.randint(0, n_samples)
        while j == i:
            j = np.random.randint(0, n_samples)

        xi, xj = X_train[i], X_train[j]
        yi, yj = y_train[i], y_train[j]

        kij = rbf_kernel(xi, xi, gamma) + rbf_kernel(xj, xj, gamma) - 2 * rbf_kernel(xi, xj, gamma)

        if kij == 0:
            continue

        ai, aj = alphas[i], alphas[j]
        L = max(0, aj - ai) if yi != yj else max(0, ai + aj - C)
        H = min(C, C + aj - ai) if yi != yj else min(C, ai + aj)

        Ei = np.sign(predict(xi.reshape(1, -1), alphas, X_train, y_train, b, gamma)) - yi
        Ej = np.sign(predict(xj.reshape(1, -1), alphas, X_train, y_train, b, gamma)) - yj

        alphas[j] = alphas[j] - (yj * (Ei - Ej)) / kij
        alphas[j] = np.clip(alphas[j], L, H)
```



```python
        alphas[i] = alphas[i] + yi * yj * (aj - alphas[j])

        b1 = b - Ei - yi * (alphas[i] - ai) * rbf_kernel(xi, xi, gamma) - yj * (alphas[j] - aj) * rbf_kernel(xi, xj, gamma)
        b2 = b - Ej - yi * (alphas[i] - ai) * rbf_kernel(xi, xj, gamma) - yj * (alphas[j] - aj) * rbf_kernel(xj, xj, gamma)

        if 0 < alphas[i] < C:
            b = b1
        elif 0 < alphas[j] < C:
            b = b2
        else:
            b = (b1 + b2) / 2


    support_vector_indices = np.where((alphas > 1e-5))[0]
    support_vectors = X_train[support_vector_indices]
    support_vector_labels = y_train[support_vector_indices]
    alphas_current = alphas[support_vector_indices]


    y_pred_train = predict(X_train, alphas_current, support_vectors, support_vector_labels, b, gamma)
    accuracy_train = np.mean(y_pred_train == y_train)

    print(f"Epoch {epoch+1} - Training Accuracy: {accuracy_train}")
```

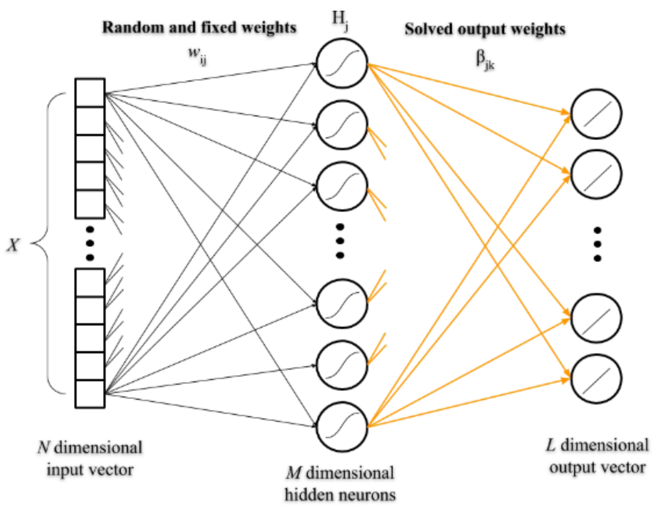
Output:



```
Epoch 1 - Training Accuracy: 0.0
Epoch 2 - Training Accuracy: 0.0
Epoch 3 - Training Accuracy: 0.5285714285714286
Epoch 4 - Training Accuracy: 0.5285714285714286
Epoch 5 - Training Accuracy: 0.5285714285714286
Epoch 6 - Training Accuracy: 0.0
Epoch 7 - Training Accuracy: 0.5285714285714286
Epoch 8 - Training Accuracy: 0.5285714285714286
Epoch 9 - Training Accuracy: 0.0
Epoch 10 - Training Accuracy: 0.5285714285714286
```

7. Implementation of Extreme Learning Machine

Extreme Learning Machines (ELMs) are single-hidden layer feedforward neural networks (SLFNs) capable to learn faster compared to gradient-based learning techniques. It's like a classical one hidden layer neural network without a learning process. This kind of neural network does not perform iterative tuning, making it faster with better generalization performance than networks trained using backpropagation method.

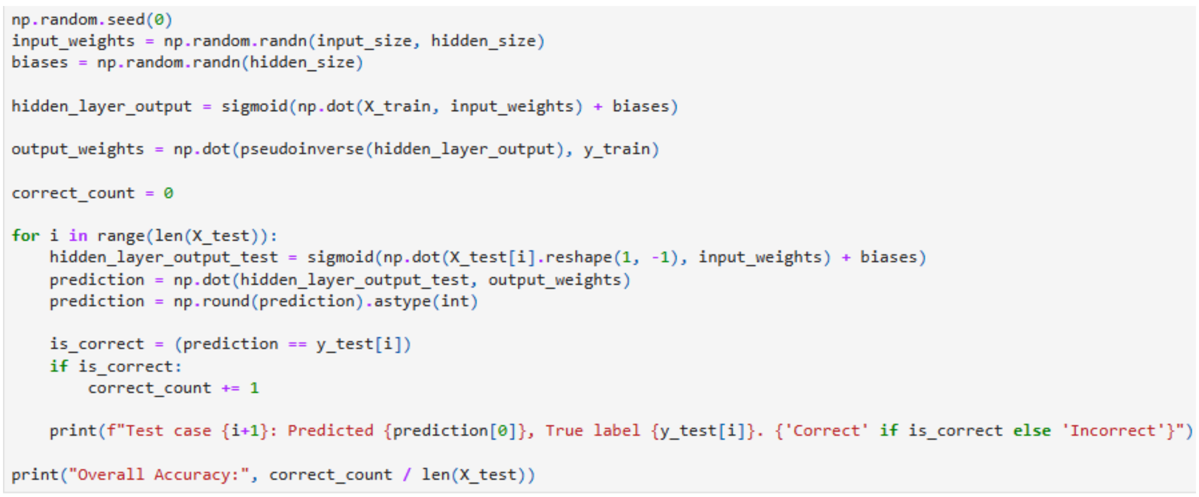


Random and fixed weights
w_{ij}
H_j
Solved output weights
β_{jk}
X
N dimensional input vector
M dimensional hidden neurons
L dimensional output vector

Fig. 7.1 Extreme Learning Machine Illustration

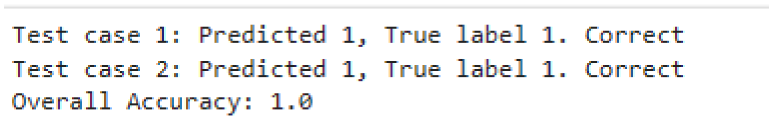
Code:

```
import numpy as np
import matplotlib.pyplot as plt

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

def standardize(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    return (X - mean) / std

def pseudoinverse(A):
    U, S, Vt = np.linalg.svd(A)
    S_inv = np.zeros(A.shape).T
    S_inv[:S.shape[0], :S.shape[0]] = np.diag(1 / S)
    return np.dot(Vt.T, np.dot(S_inv, U.T))

X = np.array([[5.1, 3.5], [4.9, 3.0], [7.0, 3.2], [6.4, 3.2], [5.9, 3.0], [6.7, 3.1], [6.3, 2.3], [5.8, 2.7]])
y = np.array([0, 0, 1, 1, 1, 1, 1, 1])

X = standardize(X)

X_train = X[:6]
y_train = y[:6]
X_test = X[6:]
y_test = y[6:]

input_size = X_train.shape[1]
hidden_size = 5
```



```
np.random.seed(0)
input_weights = np.random.randn(input_size, hidden_size)
biases = np.random.randn(hidden_size)

hidden_layer_output = sigmoid(np.dot(X_train, input_weights) + biases)

output_weights = np.dot(pseudoinverse(hidden_layer_output), y_train)

correct_count = 0

for i in range(len(X_test)):
    hidden_layer_output_test = sigmoid(np.dot(X_test[i].reshape(1, -1), input_weights) + biases)
    prediction = np.dot(hidden_layer_output_test, output_weights)
    prediction = np.round(prediction).astype(int)

    is_correct = (prediction == y_test[i])
    if is_correct:
        correct_count += 1

    print(f"Test case {i+1}: Predicted {prediction[0]}, True label {y_test[i]}. {'Correct' if is_correct else 'Incorrect'}")

print("Overall Accuracy:", correct_count / len(X_test))
```

Output:



```
Test case 1: Predicted 1, True label 1. Correct
Test case 2: Predicted 1, True label 1. Correct
Overall Accuracy: 1.0
```

8. Implementation of SVM

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression. Though we say regression problems as well its best suited for classification. The main objective of the SVM algorithm is to find the optimal hyperplane in an N-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries that the margin between the closest points of different classes should be as maximum as possible. The dimension of the hyperplane depends upon the number of features. If the number of input features is two, then the hyperplane is just a line. If the number of input features is three, then the hyperplane becomes a 2-D plane. It becomes difficult to imagine when the number of features exceeds three.
Let's consider two independent variables x_1, x_2, and one dependent variable which is either a blue circle or a red circle.

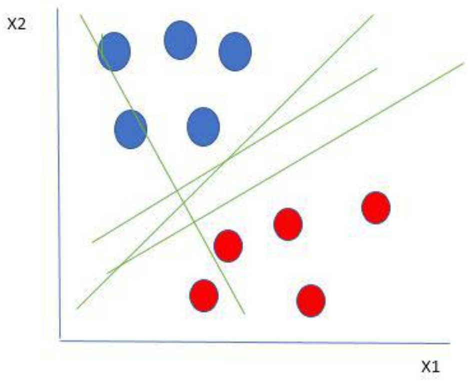


Fig. 8.1 Support Vector Machine Visualization

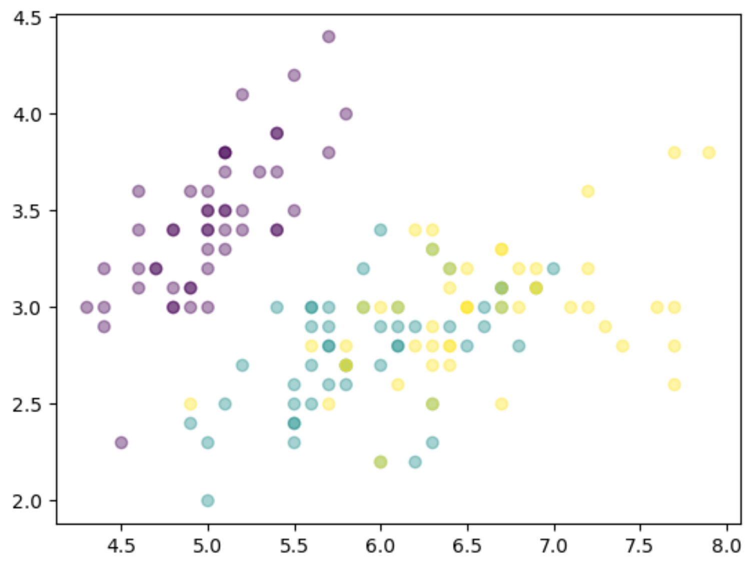
From the figure above it's very clear that there are multiple lines (our hyperplane here is a line because we are considering only two input features x_1, x_2) that segregate our data points or do a classification between red and blue circles.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import datasets, svm
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target
```

```
plt.scatter(X[:, 0], X[:, 1],  alpha=0.4, c=y)
```

```
<matplotlib.collections.PathCollection at 0x15639ec4f40>
```



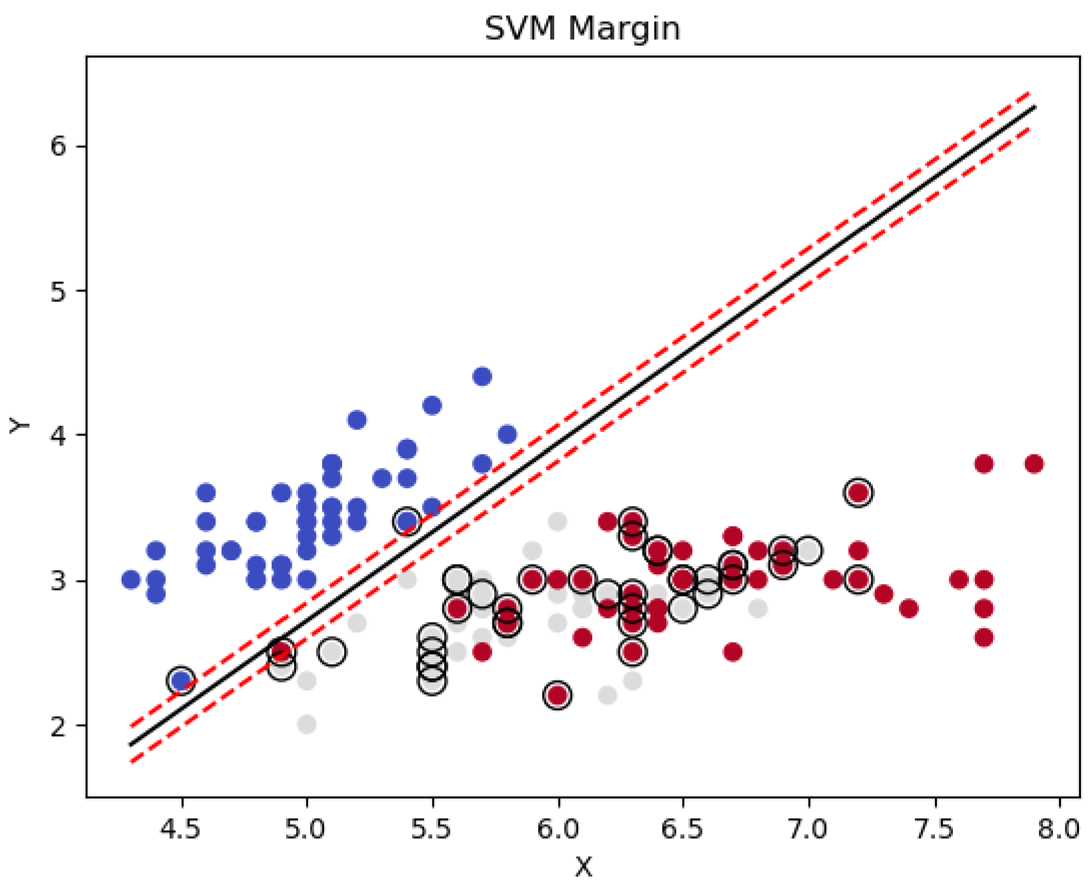
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)



print ('Train X Shape: ',  X_train.shape)
print ('Train Y Shape: ', y_train.shape)
print ('Test X Shape: ', X_test.shape)
```

```
Train X Shape:  (120, 2)
Train Y Shape:  (120,)
Test X Shape:  (30, 2)
```

```
svm= svm.SVC(kernel='linear', C=1e6)
svm.fit(X_train, y_train)

y_pred = svm.predict(X_test)

accuracy = np.mean(y_pred == y_test)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9
```

```
support_vectors = svm.support_vectors_
support_vector_indices = svm.support_

w = svm.coef_[0]

b = svm.intercept_

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm')

plt.scatter(support_vectors[:, 0], support_vectors[:, 1], facecolors='none', edgecolors='k', s=100)

x1 = np.linspace(np.min(X[:, 0]), np.max(X[:, 0]), 100)
x2 = (-w[0] * x1 - b[0]) / w[1]
plt.plot(x1, x2, 'k-')

margin = 1 / np.linalg.norm(w)
upper_margin = x2 + margin
lower_margin = x2 - margin
plt.plot(x1, upper_margin, 'r--')
plt.plot(x1, lower_margin, 'r--')

plt.xlabel('X')
plt.ylabel('Y')
plt.title('SVM Margin')

plt.show()
```



9. Implementation of Self Organizing Map

Self Organizing Map (or Kohonen Map or SOM) is a type of Artificial Neural Network which is also inspired by biological models of neural systems from the 1970s. It follows an unsupervised learning approach and trained its network through a competitive learning algorithm. SOM is used for clustering and mapping (or dimensionality reduction) techniques to map multidimensional data onto lower-dimensional which allows people to reduce complex problems for easy interpretation. SOM has two layers, one is the Input layer and the other one is the Output layer.
The architecture of the Self Organizing Map with two clusters and n input features of any sample is given below:

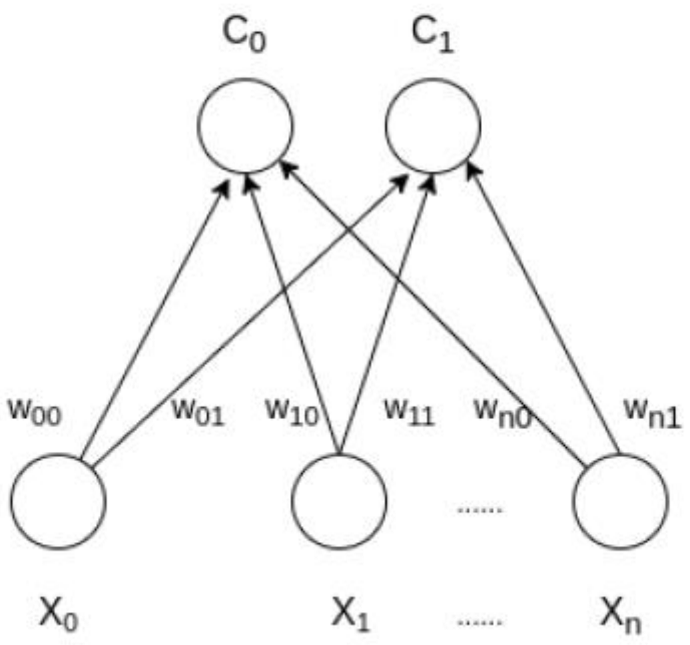


Fig.9.1 Self-Organizing Map Visualization

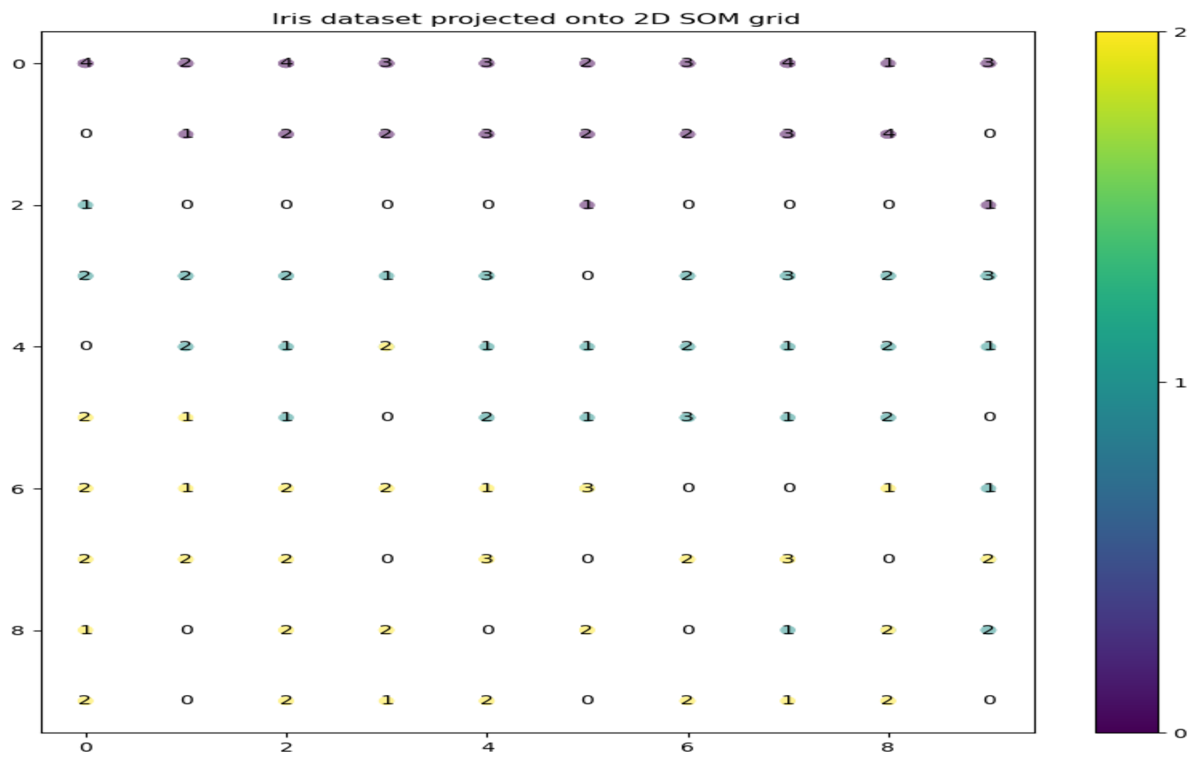
Code:

```python
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

def find_bmu(x, neurons):
    min_dist = np.inf
    bmu_idx = None
    for i in range(neurons.shape[0]):
        for j in range(neurons.shape[1]):
            dist = np.linalg.norm(x - neurons[i, j])
            if dist < min_dist:
                min_dist = dist
                bmu_idx = (i, j)
    return bmu_idx

iris = datasets.load_iris()
X = iris.data
X = StandardScaler().fit_transform(X)
y = iris.target
```

```
grid_size = (10, 10)
num_epochs = 200
initial_lr = 0.1
initial_radius = max(grid_size) / 2

neurons = np.random.rand(grid_size[0], grid_size[1], X.shape[1])

for epoch in range(num_epochs):

    lr = initial_lr * (1 - epoch / num_epochs)
    radius = initial_radius * (1 - epoch / num_epochs)

    for x in X:
        bmu_idx = find_bmu(x, neurons)


        for i in range(neurons.shape[0]):
            for j in range(neurons.shape[1]):
                d = np.linalg.norm([i - bmu_idx[0], j - bmu_idx[1]])
                if d <= radius:
                    influence = np.exp(-d / (2 * (radius ** 2)))
                    neurons[i, j] += lr * influence * (x - neurons[i, j])
```

```
projection = []
for x in X:
    bmu_idx = find_bmu(x, neurons)
    projection.append(bmu_idx)

projection = np.array(projection)

plt.figure(figsize=(10, 10))

for i in range(grid_size[0]):
    for j in range(grid_size[1]):
        plt.text(j, i, str(np.sum((projection == (i, j)).all(axis=1))),
                 ha='center', va='center', bbox=dict(facecolor='white', alpha=0.5, lw=0))

scatter = plt.scatter(projection[:, 1], projection[:, 0], c=y, cmap='viridis', label=y)
plt.colorbar(scatter, ticks=[0, 1, 2])
plt.gca().invert_yaxis()
plt.title('Iris dataset projected onto 2D SOM grid')
plt.show()
```

Output:



10. Implementation of Hopfield Network

A Hopfield network which operates in a discrete line fashion or in other words, it can be said the input and output patterns are discrete vector, which can be either binary 0,1 or bipolar +1,−1+1,−1 in nature. The network has symmetrical weights with no self-connections i.e., $\mathbf{w_{ij} = w_{ji}}$ and $\mathbf{w_{ii} = 0}$.

Following are some important points to keep in mind about discrete Hopfield network −

- This model consists of neurons with one inverting and one non-inverting output.
- The output of each neuron should be the input of other neurons but not the input of self.
- Weight/connection strength is represented by $\mathbf{w_{ij}}$.
- Connections can be excitatory as well as inhibitory. It would be excitatory, if the output of the neuron is same as the input, otherwise inhibitory.
- Weights should be symmetrical, i.e. $\mathbf{w_{ij} = w_{ji}}$

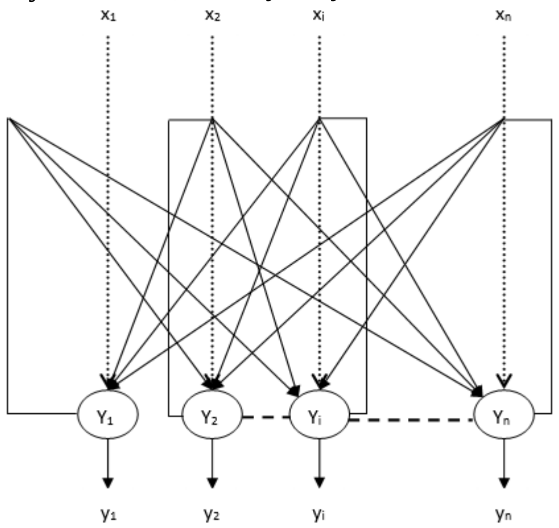


Fig. 9.1 Hopfield network Visualization

The output from $\mathbf{Y_1}$ going to $\mathbf{Y_2}$, $\mathbf{Y_i}$ and $\mathbf{Y_n}$ have the weights $\mathbf{w_{12}}$, $\mathbf{w_{1i}}$ and $\mathbf{w_{1n}}$ respectively. Similarly, other arcs have the weights on them.

Code:

```
import numpy as np

def to_bipolar(pattern):
    return 2 * np.array(pattern) - 1

def train_hopfield(patterns):
    n = patterns.shape[1]
    W = np.zeros((n, n))
    for p in patterns:
        W += np.outer(p, p)
    np.fill_diagonal(W, 0)
    return W / n

def recall(W, patterns, steps=10):
    s = patterns.copy()
    for _ in range(steps):
        s = np.sign(np.dot(s, W))
    return s

patterns = np.array([
    [1, 0, 1, 0, 1],  # 10101
    [0, 0, 1, 1, 1],  # 00111
])
```

```python
bipolar_patterns = np.array([to_bipolar(p) for p in patterns])
W = train_hopfield(bipolar_patterns)

test_pattern = np.array([1, 0, 1, 1, 1])  # 10111
bipolar_test = to_bipolar(test_pattern)
recalled_pattern = recall(W, bipolar_test)

recalled_pattern = (recalled_pattern + 1) // 2

print(f"Test pattern: {test_pattern}")
print(f"Recalled pattern: {recalled_pattern}")
```

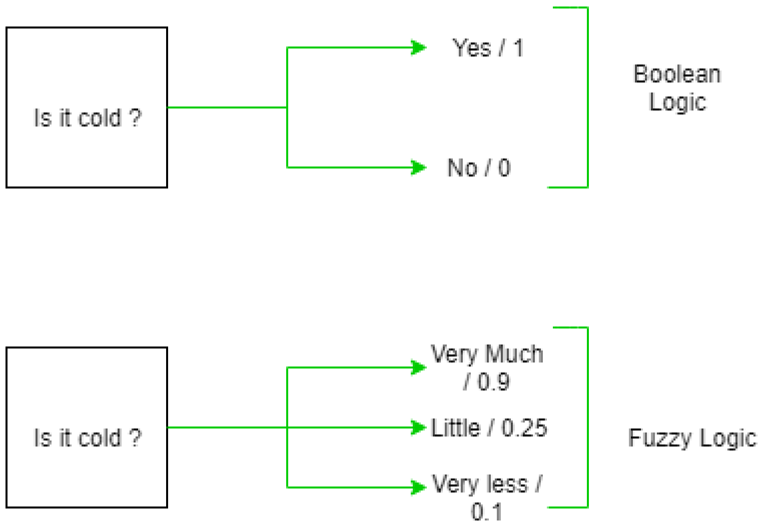
Output:

```
Test pattern: [1 0 1 1 1]
Recalled pattern: [1. 0. 1. 1. 1.]
```

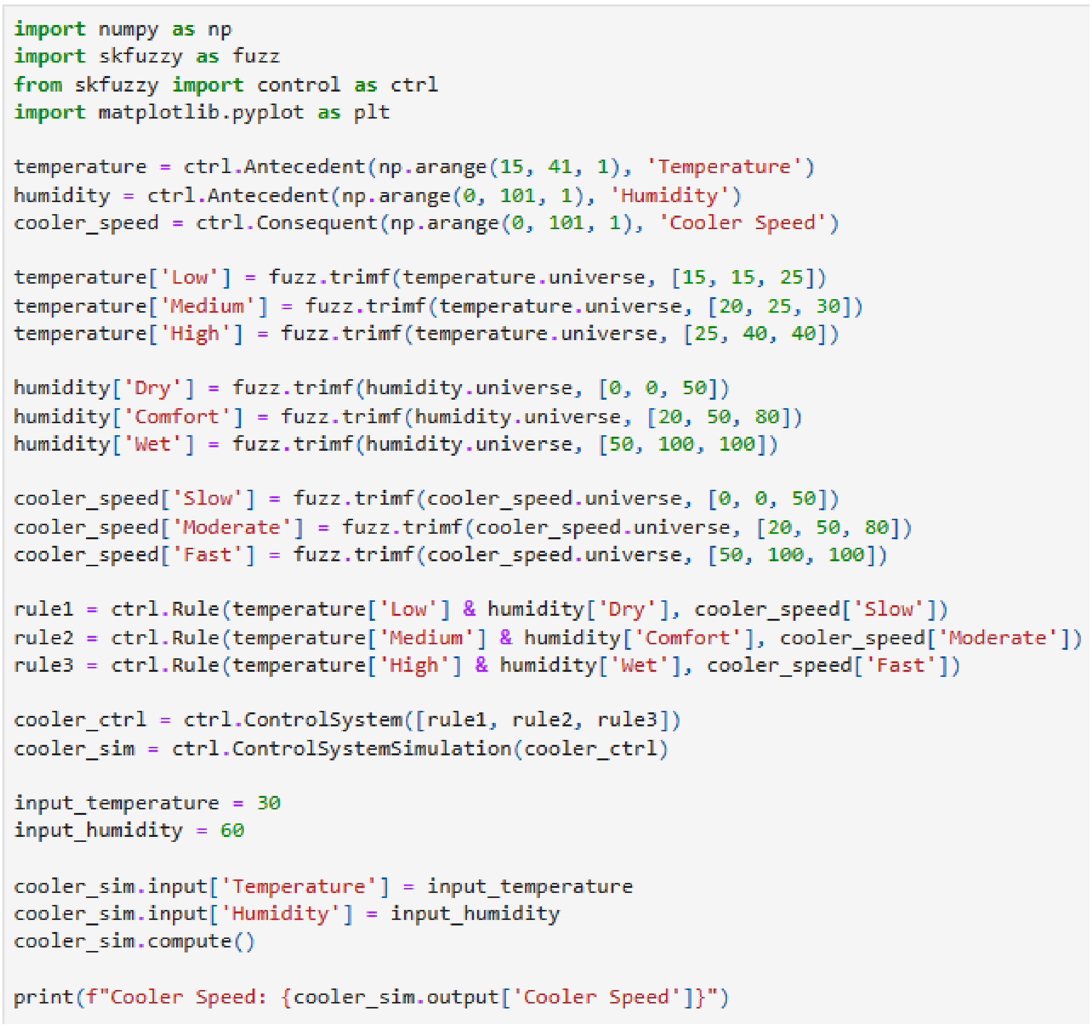
11. Implementation of Fuzzy Logic

The term **fuzzy** refers to things that are not clear or are vague. In the real world many times we encounter a situation when we can't determine whether the state is true or false, their fuzzy logic provides very valuable flexibility for reasoning. In this way, we can consider the inaccuracies and uncertainties of any situation.
Fuzzy Logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1, instead of just the traditional values of true or false. It is used to deal with imprecise or uncertain information and is a mathematical method for representing vagueness and uncertainty in decision-making.

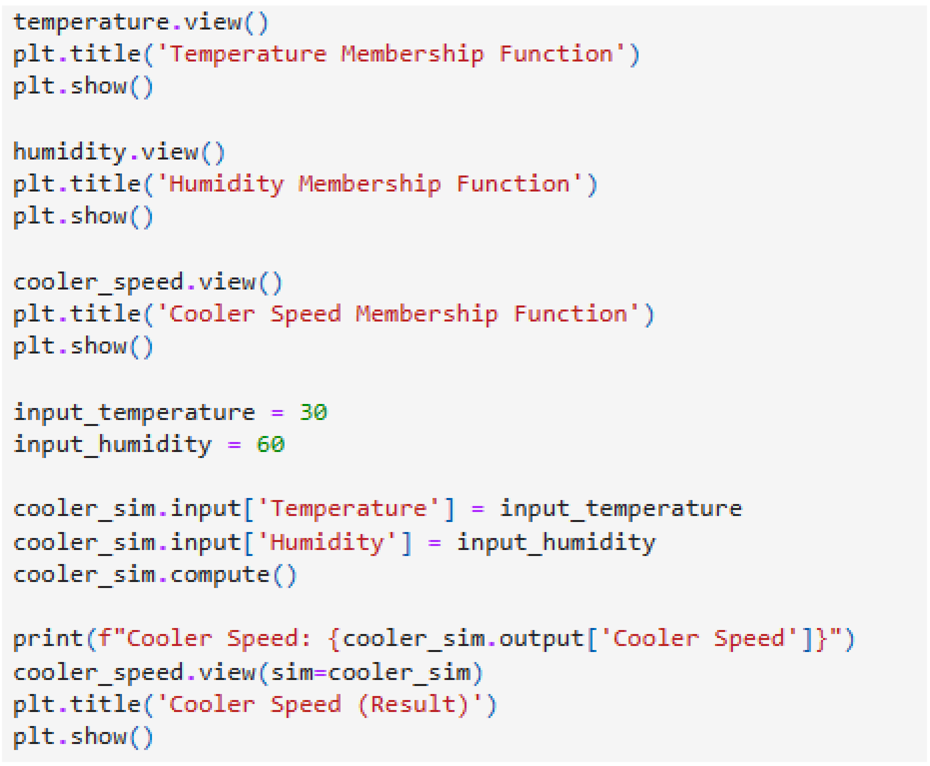
In the boolean system truth value, 1.0 represents the absolute truth value and 0.0 represents the absolute false value. But in the fuzzy system, there is no logic for the absolute truth and absolute false value. But in fuzzy logic, there is an intermediate value too present which is partially true and partially false.



Code:



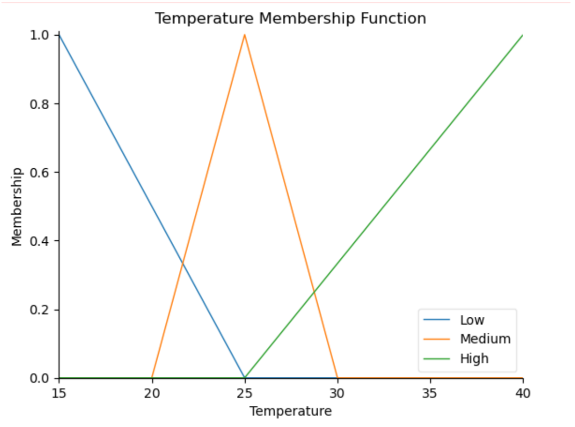
```python
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

temperature = ctrl.Antecedent(np.arange(15, 41, 1), 'Temperature')
humidity = ctrl.Antecedent(np.arange(0, 101, 1), 'Humidity')
cooler_speed = ctrl.Consequent(np.arange(0, 101, 1), 'Cooler Speed')

temperature['Low'] = fuzz.trimf(temperature.universe, [15, 15, 25])
temperature['Medium'] = fuzz.trimf(temperature.universe, [20, 25, 30])
temperature['High'] = fuzz.trimf(temperature.universe, [25, 40, 40])

humidity['Dry'] = fuzz.trimf(humidity.universe, [0, 0, 50])
humidity['Comfort'] = fuzz.trimf(humidity.universe, [20, 50, 80])
humidity['Wet'] = fuzz.trimf(humidity.universe, [50, 100, 100])

cooler_speed['Slow'] = fuzz.trimf(cooler_speed.universe, [0, 0, 50])
cooler_speed['Moderate'] = fuzz.trimf(cooler_speed.universe, [20, 50, 80])
cooler_speed['Fast'] = fuzz.trimf(cooler_speed.universe, [50, 100, 100])

rule1 = ctrl.Rule(temperature['Low'] & humidity['Dry'], cooler_speed['Slow'])
rule2 = ctrl.Rule(temperature['Medium'] & humidity['Comfort'], cooler_speed['Moderate'])
rule3 = ctrl.Rule(temperature['High'] & humidity['Wet'], cooler_speed['Fast'])

cooler_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
cooler_sim = ctrl.ControlSystemSimulation(cooler_ctrl)

input_temperature = 30
input_humidity = 60

cooler_sim.input['Temperature'] = input_temperature
cooler_sim.input['Humidity'] = input_humidity
cooler_sim.compute()

print(f"Cooler Speed: {cooler_sim.output['Cooler Speed']}")
```

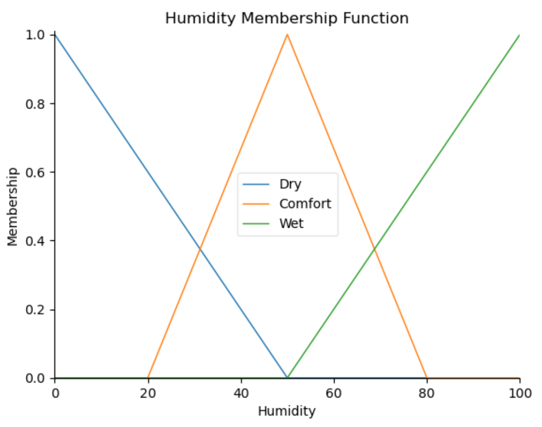


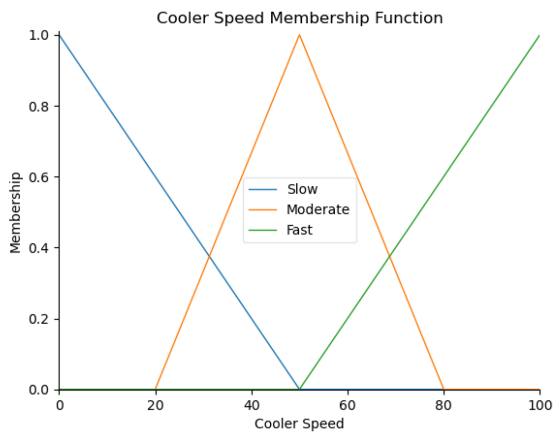
```python
temperature.view()
plt.title('Temperature Membership Function')
plt.show()

humidity.view()
plt.title('Humidity Membership Function')
plt.show()

cooler_speed.view()
plt.title('Cooler Speed Membership Function')
plt.show()

input_temperature = 30
input_humidity = 60

cooler_sim.input['Temperature'] = input_temperature
cooler_sim.input['Humidity'] = input_humidity
cooler_sim.compute()

print(f"Cooler Speed: {cooler_sim.output['Cooler Speed']}")
cooler_speed.view(sim=cooler_sim)
plt.title('Cooler Speed (Result)')
plt.show()
```

Output:

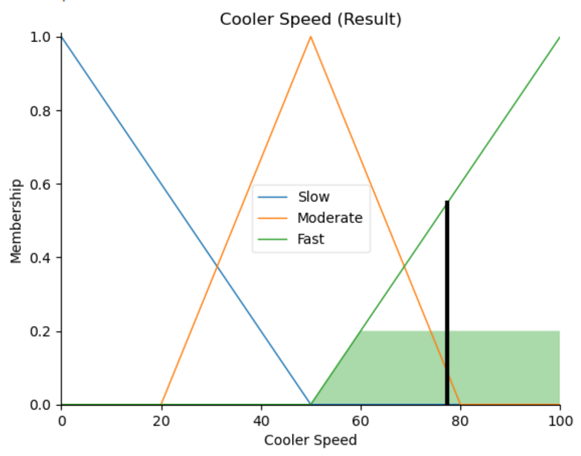
```
Cooler Speed: 77.4074074074074
```







```
Cooler Speed: 77.4074074074074
```


Cooler Speed (Result)
Membership
Cooler Speed
Slow
Moderate
Fast