```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

A1) Write your own functions for the following modules: a) Summation unit b) Activation Unit – Step, Bipolar Step, Sigmoid, TanH, ReLU and Leaky ReLU functions c) Comparator unit for Error calculation

```
import numpy as np

# a
def summation(inputs, weights, bias):
    return np.dot(inputs, weights) + bias

# b
def step_activation(x):
    return 1 if x >= 0 else 0

def bipolar_step(x):
    return 1 if x >= 0 else -1

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

def tanh_activation(x):
    return np.tanh(x)

def relu(x):
    return max(0, x)

def leaky_relu(x, alpha=0.01):
    return x if x > 0 else alpha * x

# c
def calculate_error(predicted, target):
    return target - predicted
```

A2) Develop the above perceptron in your own code (don't use the perceptron model available from package). Use the initial weights as provided below.
W0 = 10, W1 = 0.2, w2 = -0.75, learning rate (α) = 0.05 Write a function for Activation function. Develop & Use the code for Step activation function to learn the weights of the network to implement above provided AND gate logic. The activation function is demonstrated below.

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

def step_activation(x):
    return 1 if x >= 0 else 0

def train_perceptron_AND(X, y, w0=10, w1=0.2, w2=-0.75, learning_rate=0.05, max_epochs=1000, threshold=0.002):
    weights = np.array([w0, w1, w2], dtype=float)
    errors = []

    for epoch in range(max_epochs):
        total_error = 0

        for i in range(len(X)):
            x_input = np.insert(X[i], 0, 1)
            weighted_sum = np.dot(weights, x_input)
            prediction = step_activation(weighted_sum)
            error = y[i] - prediction
            total_error += error ** 2

            weights += learning_rate * error * x_input

        errors.append(total_error)

        if total_error <= threshold:
            break

    return weights, errors, epoch + 1

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

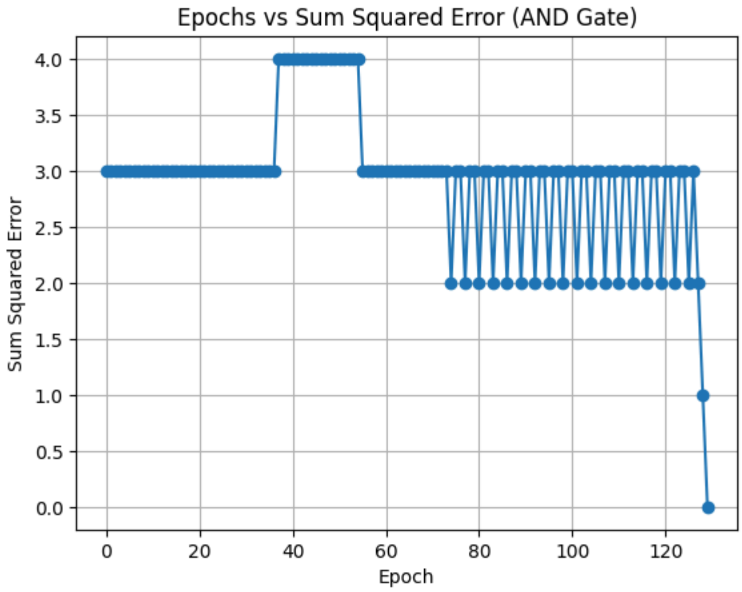
```
y = np.array([0, 0, 0, 1])

weights_final, errors_over_epochs, epochs_run = train_perceptron_AND(X, y)

print("Final Weights:", weights_final)
print("Epochs Taken:", epochs_run)

plt.plot(errors_over_epochs, marker='o')
plt.title("Epochs vs Sum Squared Error (AND Gate)")
plt.xlabel("Epoch")
plt.ylabel("Sum Squared Error")
plt.grid(True)
plt.show()
```

```
Final Weights: [-0.1   0.1   0.05]
Epochs Taken: 130
```



A3) Repeat the above A1 experiment with following activation functions (write your own code for activation functions). Compare the iterations taken to converge against each of the activation functions. Keep the learning rate same as A1. • Bi-Polar Step function • Sigmoid function • ReLU function

```
def bipolar_step_activation(x):
    return 1 if x >= 0 else -1

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

def relu_activation(x):
    return max(0, x)


def train_perceptron_custom(X, y, activation_func, is_binary_output=True,
                            w0=10, w1=0.2, w2=-0.75, learning_rate=0.05,
                            max_epochs=1000, threshold=0.002):
    weights = np.array([w0, w1, w2], dtype=float)
    errors = []

    for epoch in range(max_epochs):
        total_error = 0

        for i in range(len(X)):
            x_input = np.insert(X[i], 0, 1)
            net_input = np.dot(weights, x_input)
            output = activation_func(net_input)
            output = 1 if output >= 0.5 and is_binary_output else output
            error = y[i] - output
            total_error += error ** 2
            weights += learning_rate * error * x_input

        errors.append(total_error)

        if total_error <= threshold:
            break

    return weights, errors, epoch + 1
```

```
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
y = np.array([0, 0, 0, 1])

w_bipolar, err_bipolar, ep_bipolar = train_perceptron_custom(X, y, bipolar_step_activation)
print(f"Bipolar Step -> Epochs: {ep_bipolar}")

w_sigmoid, err_sigmoid, ep_sigmoid = train_perceptron_custom(X, y, sigmoid_activation)
print(f"Sigmoid -> Epochs: {ep_sigmoid}")

w_relu, err_relu, ep_relu = train_perceptron_custom(X, y, relu_activation)
print(f"ReLU -> Epochs: {ep_relu}")
```

```
Bipolar Step -> Epochs: 1000
Sigmoid -> Epochs: 1000
ReLU -> Epochs: 257
```

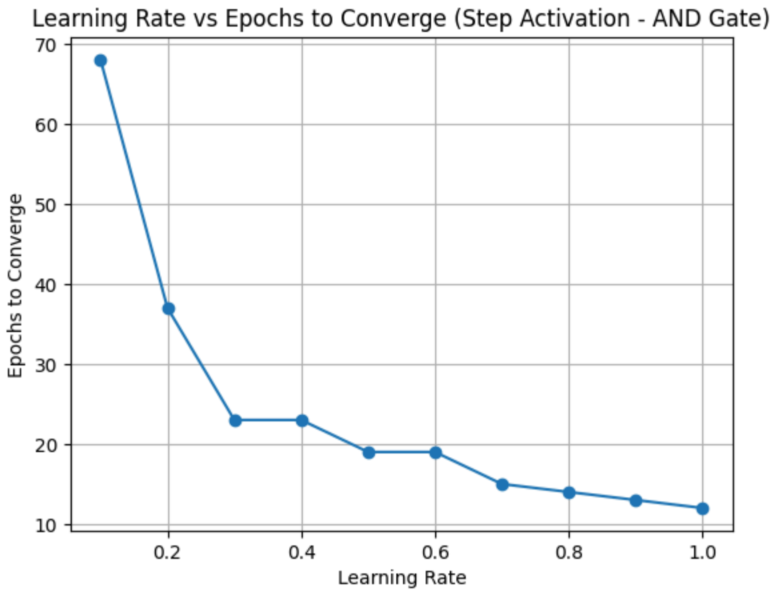
A4) Repeat exercise A1 with varying the learning rate, keeping the initial weights same. Take learning rate = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}. Make a plot of the number of iterations taken for learning to converge against the learning rates.

```
learning_rates = np.arange(0.1, 1.1, 0.1)
epoch_results = []

for lr in learning_rates:
    _, _, epochs = train_perceptron_AND(X, y, w0=10, w1=0.2, w2=-0.75, learning_rate=lr)
    epoch_results.append(epochs)
    print(f"Learning Rate: {lr:.1f} -> Epochs: {epochs}")
```

```
Learning Rate: 0.1 -> Epochs: 68
Learning Rate: 0.2 -> Epochs: 37
Learning Rate: 0.3 -> Epochs: 23
Learning Rate: 0.4 -> Epochs: 23
Learning Rate: 0.5 -> Epochs: 19
Learning Rate: 0.6 -> Epochs: 19
Learning Rate: 0.7 -> Epochs: 15
Learning Rate: 0.8 -> Epochs: 14
Learning Rate: 0.9 -> Epochs: 13
Learning Rate: 1.0 -> Epochs: 12
```

```
plt.plot(learning_rates, epoch_results, marker='o')
plt.title("Learning Rate vs Epochs to Converge (Step Activation - AND Gate)")
plt.xlabel("Learning Rate")
plt.ylabel("Epochs to Converge")
plt.grid(True)
plt.show()
```



A5). Repeat the above exercises, A1 to A3, for XOR gate logic.

```
X_xor = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
```

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

_, _, ep_step_xor = train_perceptron_AND(X_xor, y_xor)
print(f"Step Activation (XOR) -> Epochs: {ep_step_xor}")

_, _, ep_bipolar_xor = train_perceptron_custom(X_xor, y_xor, bipolar_step_activation)
print(f"Bipolar Step (XOR) -> Epochs: {ep_bipolar_xor}")

_, _, ep_sigmoid_xor = train_perceptron_custom(X_xor, y_xor, sigmoid_activation)
print(f"Sigmoid (XOR) -> Epochs: {ep_sigmoid_xor}")

_, _, ep_relu_xor = train_perceptron_custom(X_xor, y_xor, relu_activation)
print(f"ReLU (XOR) -> Epochs: {ep_relu_xor}")
```

```
Step Activation (XOR) -> Epochs: 1000
Bipolar Step (XOR) -> Epochs: 1000
Sigmoid (XOR) -> Epochs: 1000
ReLU (XOR) -> Epochs: 1000
```

A6) Use customer data provided. Build a perceptron & learn to classify the transactions as high or low value as provided in the below table. Use sigmoid as the activation function. Initialize the weights & learning rate with your choice.

```
def train_perceptron_custom(X, y, activation_func, is_binary_output=True,
                            initial_weights=None, learning_rate=0.05,
                            max_epochs=1000, threshold=0.002):
    n_features = X.shape[1]

    if initial_weights is None:
        weights = np.random.randn(n_features + 1) * 0.01
    else:
        weights = np.array(initial_weights, dtype=float)

    errors = []

    for epoch in range(max_epochs):
        total_error = 0

        for i in range(len(X)):
            x_input = np.insert(X[i], 0, 1)
            net_input = np.dot(weights, x_input)
            output = activation_func(net_input)
            output = 1 if output >= 0.5 and is_binary_output else 0
            error = y[i] - output
            total_error += error ** 2
            weights += learning_rate * error * x_input

        errors.append(total_error)
        if total_error <= threshold:
            break

    return weights, errors, epoch + 1


X_customers = np.array([
    [20, 6, 2, 386],
    [16, 3, 6, 289],
    [27, 6, 2, 393],
    [19, 1, 2, 110],
    [24, 4, 2, 280],
    [22, 1, 5, 167],
    [15, 4, 2, 271],
    [18, 4, 2, 274],
    [21, 1, 4, 148],
    [16, 2, 4, 198],
])

y_customers = np.array([1, 1, 1, 0, 1, 0, 1, 1, 0, 0])

X_norm = (X_customers - X_customers.mean(axis=0)) / X_customers.std(axis=0)

initial_weights = [0.1] * (X_norm.shape[1] + 1)

w_customers, err_customers, ep_customers = train_perceptron_custom(
    X_norm, y_customers, sigmoid_activation, is_binary_output=True,
    initial_weights=initial_weights,
    learning_rate=0.05
)

print("Customer Data Perceptron")
```

```
print("Final Weights:", w_customers)
print("Epochs Taken:", ep_customers)
```

```
Customer Data Perceptron
Final Weights: [0.05       0.06967752 0.16000992 0.03428713 0.14672553]
Epochs Taken: 2
```

A7) Compare the results obtained from above perceptron learning to the ones obtained with matrix pseudo-inverse.

```
X_bias = np.hstack((np.ones((X_norm.shape[0], 1)), X_norm))

w_pinv = np.dot(np.linalg.pinv(X_bias), y_customers)

print("Pseudo-inverse Weights:", w_pinv)

y_pred_pinv = sigmoid_activation(np.dot(X_bias, w_pinv))
y_pred_labels = (y_pred_pinv >= 0.5).astype(int)

accuracy = np.mean(y_pred_labels == y_customers)
print("Pseudo-inverse Accuracy:", accuracy)
```

```
Pseudo-inverse Weights: [ 0.6        -0.09436819  0.21713405 -0.01342766  0.23416864]
Pseudo-inverse Accuracy: 0.6
```

A8) Develop the below Neural Network. Use learning rate (α) = 0.05 with a Sigmoid activation function. Learn the weights of the network using back-propagation algorithm to implement above provided AND gate logic.

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

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

def train_mlp_AND(X, y, hidden_neurons=2, lr=0.05, max_epochs=1000, threshold=0.002):
    np.random.seed(42)

    input_dim = X.shape[1]
    output_dim = 1

    W1 = np.random.randn(input_dim, hidden_neurons)
    b1 = np.zeros((1, hidden_neurons))
    W2 = np.random.randn(hidden_neurons, output_dim)
    b2 = np.zeros((1, output_dim))

    errors = []

    for epoch in range(max_epochs):
        z1 = np.dot(X, W1) + b1
        a1 = sigmoid(z1)
        z2 = np.dot(a1, W2) + b2
        a2 = sigmoid(z2)

        error = y - a2
        loss = np.sum(error ** 2)
        errors.append(loss)

        if loss <= threshold:
            break

        d_output = error * sigmoid_derivative(z2)
        d_hidden = np.dot(d_output, W2.T) * sigmoid_derivative(z1)

        W2 += lr * np.dot(a1.T, d_output)
        b2 += lr * np.sum(d_output, axis=0, keepdims=True)
        W1 += lr * np.dot(X.T, d_hidden)
        b1 += lr * np.sum(d_hidden, axis=0, keepdims=True)

    return W1, W2, b1, b2, errors, epoch + 1


X_and = np.array([[0,0],[0,1],[1,0],[1,1]])
y_and = np.array([[0],[0],[0],[1]])

W1, W2, b1, b2, error_list, epochs_taken = train_mlp_AND(X_and, y_and)

print("Epochs:", epochs_taken)
print("Final Loss:", error_list[-1])
```

```
Epochs: 1000
Final Loss: 0.7276006058402114
```

A9) Repeat the above A1 experiment for XOR Gate logic. Keep the learning rate & activation function same as A1.

```
def step_activation(x):
    return 1 if x >= 0 else 0

X_xor = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
y_xor = np.array([0, 1, 1, 0])

def train_perceptron_AND(X, y, w0=10, w1=0.2, w2=-0.75, learning_rate=0.05, max_epochs=1000, threshold=0.002):
    weights = np.array([w0, w1, w2], dtype=float)
    errors = []

    for epoch in range(max_epochs):
        total_error = 0
        for i in range(len(X)):
            x_input = np.insert(X[i], 0, 1)
            weighted_sum = np.dot(weights, x_input)
            prediction = step_activation(weighted_sum)
            error = y[i] - prediction
            total_error += error ** 2
            weights += learning_rate * error * x_input
        errors.append(total_error)
        if total_error <= threshold:
            break
    return weights, errors, epoch + 1

weights_xor, errors_xor, epochs_xor = train_perceptron_AND(X_xor, y_xor)
print("Final Weights (XOR):", weights_xor)
print("Epochs Taken (XOR):", epochs_xor)
```

```
Final Weights (XOR): [ 0.1 -0.1 -0.1]
Epochs Taken (XOR): 1000
```

A10) Repeat exercise A1 & A2 with 2 output nodes (as shown below). A zero output of logic gate maps to [O1 O2] = [1 0] from output layer while a one output from logic gate maps to [0 1].

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

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

def train_dual_output_perceptron(X, y_encoded, lr=0.05, max_epochs=1000, threshold=0.002):
    np.random.seed(42)
    input_dim = X.shape[1]
    output_dim = y_encoded.shape[1]

    W = np.random.randn(input_dim, output_dim)
    b = np.zeros((1, output_dim))

    errors = []

    for epoch in range(max_epochs):
        z = np.dot(X, W) + b
        y_pred = sigmoid(z)

        error = y_encoded - y_pred
        loss = np.sum(error ** 2)
        errors.append(loss)

        if loss <= threshold:
            break

        d_output = error * sigmoid_derivative(z)

        W += lr * np.dot(X.T, d_output)
        b += lr * np.sum(d_output, axis=0, keepdims=True)

    return W, b, errors, epoch + 1
```

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

y_encoded = np.array([
    [1, 0],
    [1, 0],
    [1, 0],
    [0, 1]
])

W_dual, b_dual, loss_dual, epochs_dual = train_dual_output_perceptron(X, y_encoded)

print("Epochs Taken:", epochs_dual)
print("Final Loss:", loss_dual[-1])
print("Final Weights:\n", W_dual)
print("Final Bias:\n", b_dual)

output_pred = sigmoid(np.dot(X, W_dual) + b_dual)
print("Predictions:\n", np.round(output_pred, 2))
```

```
Epochs Taken: 1000
Final Loss: 0.47632889414320256
Final Weights:
 [[-1.70689256  1.89103153]
 [-1.70249178  1.95644745]]
Final Bias:
 [[ 2.72252113 -3.03782234]]
Predictions:
 [[0.94 0.05]
 [0.73 0.25]
 [0.73 0.24]
 [0.33 0.69]]
```

A11) Learn using a MLP network from Sci-Kit manual available at https://scikit learn.org/stable/modules/neural_networks_supervised.html. Repeat the AND Gate and XOR Gate exercises using MLPClassifier() function.

```
from sklearn.neural_network import MLPClassifier

X_and = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
y_and = np.array([0, 0, 0, 1])

clf_and = MLPClassifier(hidden_layer_sizes=(2,), activation='logistic', solver='sgd', learning_rate_init=0.05, max_iter=1000, random_sta
clf_and.fit(X_and, y_and)

print("AND Gate - MLPClassifier Predictions:", clf_and.predict(X_and))
print("AND Gate - Loss:", clf_and.loss_)
print("AND Gate - Converged in:", clf_and.n_iter_, "iterations\n")

y_xor = np.array([0, 1, 1, 0])

clf_xor = MLPClassifier(hidden_layer_sizes=(2,), activation='logistic', solver='sgd', learning_rate_init=0.05, max_iter=1000, random_sta
clf_xor.fit(X_and, y_xor)

print("XOR Gate - MLPClassifier Predictions:", clf_xor.predict(X_and))
print("XOR Gate - Loss:", clf_xor.loss_)
print("XOR Gate - Converged in:", clf_xor.n_iter_, "iterations")
```

```
AND Gate - MLPClassifier Predictions: [0 0 0 1]
AND Gate - Loss: 0.027865494926248548
AND Gate - Converged in: 513 iterations

XOR Gate - MLPClassifier Predictions: [0 0 1 0]
XOR Gate - Loss: 0.6940242447623771
XOR Gate - Converged in: 23 iterations
```

A12) Use the MLPClassifier() function on your project dataset.

```
import numpy as np
import pandas as pd
import networkx as nx
from sklearn.model_selection import train_test_split


file_path = '/content/drive/MyDrive/Colab Notebooks/ML_PROJECT/DWI_with_Labels.xlsx'
df = pd.read_excel(file_path)
```

```
flattened_size = 82 * 82

X_flat = df.iloc[:, :flattened_size].values
labels = df.iloc[:, -1].values

correlation_matrices = X_flat.reshape(-1, 82, 82)

def extract_graph_features(matrix):
    G = nx.from_numpy_array(matrix)
    features = {
        'avg_degree': np.mean([d for n, d in G.degree()]),
        'avg_clustering': nx.average_clustering(G),
        'density': nx.density(G),
        'transitivity': nx.transitivity(G),
        'avg_shortest_path_length': nx.average_shortest_path_length(G) if nx.is_connected(G) else 0,
        'diameter': nx.diameter(G) if nx.is_connected(G) else 0,
        'radius': nx.radius(G) if nx.is_connected(G) else 0,
        'assortativity': nx.degree_assortativity_coefficient(G),
        'number_of_edges': G.number_of_edges(),
        'number_of_nodes': G.number_of_nodes(),
        'eccentricity_mean': np.mean(list(nx.eccentricity(G).values())) if nx.is_connected(G) else 0,
        'pagerank_mean': np.mean(list(nx.pagerank(G).values())),
        'betweenness_mean': np.mean(list(nx.betweenness_centrality(G).values())),
        'closeness_mean': np.mean(list(nx.closeness_centrality(G).values())),
    }
    return features

feature_list = [extract_graph_features(mat) for mat in correlation_matrices]
df_features = pd.DataFrame(feature_list)
df_features['label'] = labels


from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report

X = df_features.drop(columns=['label'])
y = df_features['label']

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

clf = MLPClassifier(hidden_layer_sizes=(64, 32), activation='relu',
                    solver='adam', learning_rate_init=0.001, max_iter=500, random_state=42)

clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
acc = accuracy_score(y_test, y_pred)

print("Project Dataset Classification")
print(f"Accuracy: {acc:.4f}")
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Project Dataset Classification
Accuracy: 0.4647
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        91
           1       0.46      1.00      0.63        79

    accuracy                           0.46       170
   macro avg       0.23      0.50      0.32       170
weighted avg       0.22      0.46      0.29       170

/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```