Notebook: Q1.ipynb

Task 1

Implement the OR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

```
import numpy as np
import tensorflow as tf
```

```
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 1, 1, 1])
w1 = 1
w2 = 1
b = 0
```

```
def step(z):
    return 1 if z>=1 else 0
```

```
for i in range(X.shape[0]):
    z = w1*X[i][0] + w2*X[i][1] + b
    print(f"Input: {X[i]} → Output:", step(z))
```

Using the updating weights and bias approach:

```
w = np.random.rand(2)
b = np.random.rand(1)

epochs = 5
learning_rate = 0.1

for epoch in range(epochs):
    for i in range(X.shape[0]):
        z = np.dot(w, X[i]) + b
        y_pred = step(z)
        error = y[i] - y_pred
        w += learning_rate * error * X[i]
        b += learning_rate * error

print("Trained weights:", w)
print("Trained bias:", b)
```

```
print("\nPredictions:")
for i in range(len(X)):
    z = np.dot(w, X[i]) + b
    print(f"Input: {X[i]} → Output:", step(z))
```

Task 2

- Use the iris dataset Encode the input and show the new representation
- Decode the lossy representation for the output
- Map the input to reconstruction and visualize

```
from sklearn.datasets import load_iris
```

```
iris =load_iris()
```

```
data = iris.data
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
data = scaler.fit_transform(data)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```
autoencoder = Sequential([
    Input(shape = (4,)),
    Dense(2, activation='relu'),
    Dense(4, activation='sigmoid')
])
```

```
autoencoder.compile(optimizer='adam', loss='mse')
```

```
autoencoder.summary()
```

```
autoencoder.fit(data, data, epochs=50, batch_size=8, shuffle=True)
```

```
encoder = Sequential([
    autoencoder.layers[0]
])

encoded_data = encoder.predict(data)

print("Encoded (Lossy) Representation - First 5 Samples:\n")
print(encoded_data[:5])
```

```
decoded_data = autoencoder.predict(data)

# Convert back to original scale
decoded_original = scaler.inverse_transform(decoded_data)
```

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,4))

# Original Sepal Length
plt.subplot(1,2,1)
plt.title("Original Sepal Length")
plt.plot(data[:, 0])
plt.xlabel("Samples")
plt.ylabel("Value")

# Reconstructed Sepal Length
plt.subplot(1,2,2)
plt.title("Reconstructed Sepal Length")
plt.plot(decoded_original[:, 0])
plt.xlabel("Samples")
plt.ylabel("Value")

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

```
import numpy as np

def mean_squared_error(original, reconstructed):
    return np.mean((original - reconstructed) ** 2)
```

```
for i in range(data.shape[1]):
    mse = mean_squared_error(data[:, i], decoded_data[:, i])
    print(f"MSE for feature {i+1}: {mse}")
```

Notebook: Q2.ipynb

Task 1

Implement the OR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

```
import numpy as np
```

```
def step(z):
    return 1 if z >= 1 else 0
```

```
w1 = 1
w2 = 1
b = 0

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

```
z = np.dot(X, np.array([w1, w2])) + b
predictions = np.array([step(i) for i in z])
```

```
predictions
```

Task 2

- Use the heart disease Dataset
- Create an Auto Encoder and fit it with our data using 3 neurons in the dense layer
- Display new reduced dimension values
- Plot loss for different Auto encoders

```
import pandas as pd
```

```
X = pd.read_csv('Data/heart.csv')
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```
autoencoder = Sequential([
    Input(shape = (X.shape[1], )),
    Dense(3, activation='relu'),
    Dense(X.shape[1], activation='sigmoid')
])
```

```
autoencoder.compile(optimizer='adam', loss='mse')
```

```
autoencoder.fit(X_scaled, X_scaled, epochs=50, batch_size=16, shuffle=True, validation_sp
```

```
encoder  =Sequential([autoencoder.layers[0]])
encoded_data = encoder.predict(X_scaled)
```

The reduced dimention values are as follows:

```
encoded_data
```

```
preds = []
loss = []
for i in range(1,5):
    autoencoder = Sequential([
        Input(shape = (X.shape[1], )),
        Dense(i, activation='relu'),
        Dense(X.shape[1], activation='sigmoid')
    ])
    autoencoder.compile(optimizer='adam', loss='mse')
    history = autoencoder.fit(X_scaled, X_scaled, epochs=20, batch_size=16, shuffle=True,
    preds.append(autoencoder.predict(X_scaled))
    loss.append(history.history['loss'])
    print(f'Encoding Dimension: {i}, Loss: {loss[-1]}')
```

```
loss
```

```
import matplotlib.pyplot as plt

for i in range(4):
    plt.plot(loss[i], label=f'Encoding Dim: {i+1}')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

Notebook: Q3.ipynb

Task 2

- Load the Intel Image dataset
- Train and test the dataset
- Create a model using CNN
- Evaluate the model using confusion matrix.

```
from tensorflow.keras.preprocessing import image_dataset_from_directory
```

```
dir = 'Data/Intel Image Data/seg_train/seg_train'
train = image_dataset_from_directory(
        dir,
        image_size=(150, 150),
        batch_size=32,
        subset = "training",
        validation_split=0.2,
        seed=42,
)
```

```
train.as_numpy_iterator().next()[0].shape
```

```
train.as_numpy_iterator().next()[0].shape[0]
```

```
val = image_dataset_from_directory(
        'Data/Intel Image Data/seg_test/seg_test',
        image_size=(150, 150),
        batch_size=32,
        subset = "validation",
        validation_split=0.2,
        seed=42
)
```

```
class_names = train.class_names
print(class_names)
```

```
import tensorflow as tf
AUTOTUNE = tf.data.AUTOTUNE
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
])
train = train.map(lambda x, y: (data_augmentation(x, training=True), y)).prefetch(AUTOTUNE
val = val.prefetch(AUTOTUNE)
```

```
for image_batch, labels_batch in train:
  print(image_batch.shape)
  print(labels_batch.shape)
  break
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Rescali
```

```
model = Sequential([
    Rescaling(1./255, input_shape=(150, 150, 3)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Dense(128, activation='relu'),
    Flatten(),
    Dropout(0.5),
    Dense(6, activation='softmax')
])
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy
model.fit(train, validation_data=val, epochs=10,)
```

```
preds = model.predict(val)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np
import matplotlib.pyplot as plt

true_labels = np.concatenate([y for x, y in val], axis=0)

predicted_labels = np.argmax(preds, axis=1)

cm = confusion_matrix(true_labels, predicted_labels)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp.plot(
)
plt.title('Confusion Matrix')
plt.show()
```

```python
# Plot ROC AUC Curves for Multi-class Classification (One-vs-Rest)
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.preprocessing import label_binarize
import numpy as np
import matplotlib.pyplot as plt

# Get probability predictions
y_true = np.concatenate([y for x, y in val], axis=0)
y_pred_proba = preds

# Number of classes
n_classes = len(class_names)

# Binarize the output for multi-class ROC AUC
y_bin = label_binarize(y_true, classes=list(range(n_classes)))

# Compute ROC curve and AUC for each class (One-vs-Rest)
fpr = dict()
tpr = dict()
roc_auc_dict = dict()
colors = plt.cm.Set1(np.linspace(0, 1, n_classes))

plt.figure(figsize=(12, 8))

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_proba[:, i])
    roc_auc_dict[i] = auc(fpr[i], tpr[i])
    plt.plot(fpr[i], tpr[i], color=colors[i], lw=2,
             label=f'{class_names[i]} (AUC = {roc_auc_dict[i]:.3f})')

# Plot random classifier baseline
plt.plot([0, 1], [0, 1], 'k--', lw=2, label='Random Classifier')

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', fontsize=12)
plt.ylabel('True Positive Rate', fontsize=12)
plt.title('ROC Curves - Intel Image Classification (One-vs-Rest)', fontsize=14, fontweigh
```

```
plt.legend(loc="lower right", fontsize=10)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Calculate and display macro-average AUC
macro_auc = np.mean(list(roc_auc_dict.values()))
print(f"\nROC AUC Scores per Class:")
for i in range(n_classes):
    print(f"  {class_names[i]}: {roc_auc_dict[i]:.4f}")
print(f"\nMacro-average AUC: {macro_auc:.4f}")

# Calculate weighted average AUC
weights = np.bincount(y_true) / len(y_true)
weighted_auc = np.average([roc_auc_dict[i] for i in range(n_classes)], weights=weights)
print(f"Weighted-average AUC: {weighted_auc:.4f}")
```

```
```

Notebook: Q4.ipynb

Task 1

Implement the NOT Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron

```
import numpy as np
import tensorflow as tf
```

```
X,y = np.array([
    [0, 1],
    [1, 0]
])
w1 = 1
w2 = -2
b = 1
X,y
```

```
def step(z):
    return 1 if z>=1 else 0
```

```
for i in range(X.shape[0]):
    z = w1*X[i] + w2*X[i] + b
    print(f"Input: {X[i]} → Output:", step(z))
```

Using the updating weights and bias approach:

```
w = np.random.rand(1)
b = np.random.rand(1)

epochs = 100
learning_rate = 0.1

for epoch in range(epochs):
    for i in range(X.shape[0]):
        z = np.dot(w, X[i]) + b
        y_pred = step(z)
        error = y[i] - y_pred
        w += learning_rate * error * X[i]
        b += learning_rate * error

print("Trained weights:", w)
print("Trained bias:", b)
```

```
print("\nPredictions:")
for i in range(len(X)):
    z = np.dot(w, X[i]) + b
    print(f"Input: {X[i]} → Output:", step(z))
```

Task 2

1. Use the Iris Dataset
2. Create an Auto Encoder and fit it with our data using 3 neurons in the dense layer
3. Display new reduced dimension values
4. Plot loss for different encoders

```
import pandas as pd
```

```
from sklearn.datasets import load_iris
X = load_iris().data
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```


```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```


```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```
autoencoder = Sequential([
    Input(shape = (X.shape[1], )),
    Dense(3, activation='relu'),
    Dense(X.shape[1], activation='sigmoid')
])
```

```
autoencoder.compile(optimizer='adam', loss='mse')
```

```
autoencoder.fit(X_scaled, X_scaled, epochs=50, batch_size=16, shuffle=True, validation_sp
```

```
encoder  =Sequential([autoencoder.layers[0]])
encoded_data = encoder.predict(X_scaled)
```

The reduced dimention values are as follows:

```
encoded_data
```

```
preds = []
loss = []
for i in range(1,5):
    autoencoder = Sequential([
        Input(shape = (X.shape[1], )),
        Dense(i, activation='relu'),
        Dense(X.shape[1], activation='sigmoid')
    ])
    autoencoder.compile(optimizer='adam', loss='mse')
    history = autoencoder.fit(X_scaled, X_scaled, epochs=20, batch_size=16, shuffle=True,
    preds.append(autoencoder.predict(X_scaled))
    loss.append(history.history['loss'])
    print(f'Encoding Dimension: {i}, Loss: {loss[-1]}')
```

```
import matplotlib.pyplot as plt

for i in range(4):
    plt.plot(loss[i], label=f'Encoding Dim: {i+1}')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

Notebook: Q5.ipynb

Task 1

Implement the OR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron

```
import numpy as np
import tensorflow as tf
```

```
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 1, 1, 1])
w1 = 1
w2 = 1
b = 0
```

```
def step(z):
    return 1 if z>=1 else 0
```

```
for i in range(X.shape[0]):
    z = w1*X[i][0] + w2*X[i][1] + b
    print(f"Input: {X[i]} → Output:", step(z))
```

Using the updating weights and bias approach:

```
w = np.random.rand(2)
b = np.random.rand(1)

epochs = 5
learning_rate = 0.1

for epoch in range(epochs):
    for i in range(X.shape[0]):
        z = np.dot(w, X[i]) + b
        y_pred = step(z)
        error = y[i] - y_pred
        w += learning_rate * error * X[i]
        b += learning_rate * error

print("Trained weights:", w)
print("Trained bias:", b)
```

```
print("\nPredictions:")
for i in range(len(X)):
    z = np.dot(w, X[i]) + b
    print(f"Input: {X[i]} → Output:", step(z))
```

Task 2

Use the heart disease dataset and do the following

- Use the Dataset
- Create an autoencoder and fit it with our data using 2 neurons in the dense layer
- Plot loss w.r.t. epochs
- Calculate reconstruction error using Mean Squared Error (MSE).

```
import pandas as pd
```

```
X = pd.read_csv('Data/heart.csv')
```

```
X
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```
autoencoder = Sequential([
    Input(shape = (X.shape[1], )),
    Dense(2, activation='relu'),
    Dense(X.shape[1], activation='sigmoid')
])
```

```
autoencoder.compile(optimizer='adam', loss='mse')
```

```
autoencoder.fit(X_scaled, X_scaled, epochs=50, batch_size=16, shuffle=True, validation_sp
```

```
encoder  =Sequential([autoencoder.layers[0]])
encoded_data = encoder.predict(X_scaled)
```

The reduced dimention values are as follows:

```
encoded_data
```

```
preds = []
loss = []
for i in range(1,8):
    autoencoder = Sequential([
        Input(shape = (X.shape[1], )),
        Dense(i, activation='relu'),
        Dense(X.shape[1], activation='sigmoid')
    ])
    autoencoder.compile(optimizer='adam', loss='mse')
    history = autoencoder.fit(X_scaled, X_scaled, epochs=20, batch_size=16, shuffle=True,
    preds.append(autoencoder.predict(X_scaled))
    loss.append(history.history['loss'])
    print(f'Encoding Dimension: {i}, Loss: {loss[-1]}')
```

```
loss
```

```
import matplotlib.pyplot as plt

for i in range(len(loss)):
    plt.plot(loss[i], label=f'Encoding Dim: {i+1}')
plt.title('Training Loss for Different Encoding Dimensions')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
```

Notebook: Q6.ipynb

Task 1

- Load California Housing dataset and select 2 features (e.g., Median Income, House Age) and 1 target (Median House Value).
- Normalize inputs and initialize a single-layer NN with random weights and bias.
- Perform forward propagation, calculate prediction error, Squared Error, and MSE.
- Update weights and bias using gradient descent.
- Plot Loss vs Weight, Loss vs Bias, and Error Surface.

```
import pandas as pd
import numpy as np
```

```
X = pd.read_csv('Data\housing.csv')
```

```
X
```

```
y = X[['median_house_value']].values
X = X[['median_income', 'households']].values
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
np.random.seed(42)
weights = np.random.rand(X.shape[1])
bias = np.random.rand(1)
initial_weights = weights.copy()
initial_bias = bias.copy()
```

```
def forwardprop(X,weights,bias):
    layer_input = np.dot(X,weights) + bias
    layer_output = layer_input #Regression
    return layer_output
```

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
y_preds = forwardprop(X_scaled, weights, bias)
pred_errors = y.flatten() - y_preds
squared_errors = pred_errors ** 2

print("Initial Metrics:")
print(f"Sample Prediction Errors: {pred_errors[:5]}")
print(f"Sample Squared Errors: {squared_errors[:5]}")
print(f"Initial MSE: {mean_squared_error(y, y_preds)}")
print(f"Initial MAE: {mean_absolute_error(y, y_preds)}")
```

```
epochs = 100
learning_rate = 0.01
loss_history = []
weight_history = []
bias_history = []

for i in range(epochs):
    y_preds = forwardprop(X_scaled, weights, bias)

    error = y.flatten() - y_preds
    dw = (-2/X.shape[0]) * np.dot(X_scaled.T, error)
    db = (-2/X.shape[0]) * np.sum(error)

    weights = weights - learning_rate * dw
    bias = bias - learning_rate * db

    mse = mean_squared_error(y, y_preds)
    loss_history.append(mse)
    weight_history.append(weights.copy())
    bias_history.append(bias.copy())

    if i % 10 == 0:
        mae = mean_absolute_error(y, y_preds)
        print(f'Epoch {i}: MSE={mse:.2f}, MAE={mae:.2f}')

print(f"\nFinal MSE: {loss_history[-1]:.2f}")
print(f"Final Weights: {weights}")

print(f"Final Bias: {bias}")
```

```
import matplotlib.pyplot as plt
# Plot 1: Loss vs Epochs
plt.figure(figsize=(10, 6))
plt.plot(loss_history)
plt.xlabel('Epoch')
plt.ylabel('MSE Loss')
plt.title('Loss vs Epochs')
plt.grid(True)
plt.show()

# Plot 2: Loss vs Weight (for first weight)
def calculate_loss_for_weight(w_val, w_idx):
    temp_weights = weights.copy()
    temp_weights[w_idx] = w_val
    y_pred = forwardprop(X_scaled, temp_weights, bias)
    return mean_squared_error(y, y_pred)

weight_range = np.linspace(weights[0] - 50, weights[0] + 50, 100)
losses_w0 = [calculate_loss_for_weight(w, 0) for w in weight_range]

plt.figure(figsize=(10, 6))
plt.plot(weight_range, losses_w0)
plt.xlabel('Weight[0] (Median Income)')
plt.ylabel('MSE Loss')
plt.title('Loss vs Weight[0]')
plt.axvline(x=weights[0], color='r', linestyle='--', label=f'Optimal Weight={weights[0]:.2
plt.legend()
plt.grid(True)
plt.show()

# Plot 3: Loss vs Bias
def calculate_loss_for_bias(b_val):
    y_pred = forwardprop(X_scaled, weights, b_val)
    return mean_squared_error(y, y_pred)

bias_range = np.linspace(bias[0] - 50000, bias[0] + 50000, 100)
losses_bias = [calculate_loss_for_bias(b) for b in bias_range]

plt.figure(figsize=(10, 6))
```

```
plt.plot(bias_range, losses_bias)
plt.xlabel('Bias')
plt.ylabel('MSE Loss')
plt.title('Loss vs Bias')
plt.axvline(x=bias[0], color='r', linestyle='--', label=f'Optimal Bias={bias[0]:.2f}')
plt.legend()
plt.grid(True)
plt.show()
```

```
```

Notebook: Q7.ipynb

Implement Self Organizing Map for anomaly Detection

- Use Credit Card Applications Dataset:
- Detect fraud customers in the dataset using SOM and perform hyperparameter tuning
- Show map and use markers to distinguish frauds

```
data = r'Data\Credit_Card_Applications.csv'
```

```
import pandas as pd
import numpy as np
from minisom import MiniSom
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

df = pd.read_csv(data)

print("Dataset shape:", df.shape)
print("\nFirst few rows:")
df.head()
```

```
X = df.drop(columns=df.columns[-1]).values
y = df.iloc[:, -1].values
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
X_scaled = scaler.fit_transform(X)
```

```
som_grid_size = 10
som_sigma = 1.0
som_learning_rate = 0.5
som_iterations = 10000
som = MiniSom(x=som_grid_size, y=som_grid_size,
              input_len=X_scaled.shape[1],
              sigma=som_sigma,
              learning_rate=som_learning_rate,
              random_seed=42)

som.random_weights_init(X_scaled)
som.train_random(X_scaled, som_iterations)
```

```
distance_map = som.distance_map()
fraud_markers = []
normal_markers = []

for i, x in enumerate(X_scaled):
    w = som.winner(x)
    if y[i] == 0:  # Assuming 0 is rejected/fraud
        fraud_markers.append(w)
    else:
        normal_markers.append(w)
```

```
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.heatmap(distance_map.T, cmap='bone_r', cbar_kws={'label': 'Distance'},
            square=True, xticklabels=False, yticklabels=False)

fraud_arr = np.asarray(fraud_markers)
normal_arr = np.asarray(normal_markers)

if fraud_arr.size:
    plt.scatter(fraud_arr[:, 0] + 0.5, fraud_arr[:, 1] + 0.5,
                s=50, c='red', edgecolors='k', linewidths=1.5,
                label='Rejected/Fraud', zorder=3)

if normal_arr.size:
    plt.scatter(normal_arr[:, 0] + 0.5, normal_arr[:, 1] + 0.5,
                s=50, c='green', alpha=0.35, edgecolors='none',
                label='Approved', zorder=2)

plt.legend(loc='upper right')
plt.title('Self-Organizing Map - Credit Card Applications\nFraud Detection')
plt.xlabel('SOM Grid X')
plt.ylabel('SOM Grid Y')
plt.show()

print(f"\nSOM trained with grid size: {som_grid_size}x{som_grid_size}")
print(f"Sigma: {som_sigma}, Learning rate: {som_learning_rate}")
print(f"Number of iterations: {som_iterations}")
```

Notebook: Q8.ipynb

Train a small neural network (dataset - MNIST classification)

Compare the optimizers:

1. SGD
2. SGD + Momentum
3. Adam

Plot:

1. Training loss vs epochs
2. Accuracy vs epochs

```
from tensorflow.keras.datasets import mnist
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

```
data = mnist.load_data()
(train_images, train_labels), (test_images, test_labels) = data
```

```
img_size =  train_images[0].shape[0]
```

```
num_classes = 10
```

```
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0
```

```
plt.imshow(train_images[0], cmap='gray')
plt.title(f'Label: {train_labels[0]}')
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten
```

```
from tensorflow.keras.optimizers import SGD, Adam
SGD_nomomentum = SGD()
SGD_momentum = SGD(momentum=0.6)
```

```
adam_model = Sequential([
    Input(shape =(img_size,img_size) ),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

adam_model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',metrics=[
adam_history = adam_model.fit(train_images,train_labels, epochs = 15)
```

```
SGD_model = Sequential([
    Input(shape =(img_size,img_size) ),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

SGD_model.compile(optimizer =SGD_nomomentum, loss = 'sparse_categorical_crossentropy',met
SGD_history = SGD_model.fit(train_images,train_labels, epochs = 15)
```

```
SGD_momentum_model = Sequential([
    Input(shape =(img_size,img_size) ),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

SGD_momentum_model.compile(optimizer =SGD_momentum, loss = 'sparse_categorical_crossentro
SGD_momentum_history = SGD_momentum_model.fit(train_images,train_labels, epochs = 15)
```

```
adam_losses = adam_history.history['loss']
SGD_losses = SGD_history.history['loss']
SGD_momentum_losses = SGD_momentum_history.history['loss']
```

```
x = np.arange(1, 16)
```

```
plt.plot(adam_losses, x)
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.title("Adam Epochs vs Loss")
plt.grid()
```

```
plt.plot(SGD_losses, x)
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.title("SGD Epochs vs Loss")
plt.grid()
```

```
plt.plot(SGD_momentum_losses, x)
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.title("SGD Momentum Epochs vs Loss")
plt.grid()
```

```
adam_acc = adam_history.history['accuracy']
SGD_acc = SGD_history.history['accuracy']
SGD_momentum_acc = SGD_momentum_history.history['accuracy']
```

```
plt.plot(adam_acc, x)
plt.xlabel('Accuracy')
plt.ylabel('Epochs')
plt.title("Adam Epochs vs Accuracy")
plt.grid()
```

```
plt.plot(SGD_acc, x)
plt.xlabel('Accuracy')
plt.ylabel('Epochs')
plt.title("SGD Epochs vs Accuracy")
plt.grid()
```

```
plt.plot(SGD_momentum_acc, x)
plt.xlabel('Accuracy')
plt.ylabel('Epochs')
plt.title("SGD Momentum Epochs vs Accuracy")
plt.grid()
```

```python
# Plot ROC AUC Curves for Multi-class Classification (One-vs-Rest)
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.preprocessing import label_binarize
import numpy as np
import matplotlib.pyplot as plt

# Get probability predictions for all three models
adam_preds = adam_model.predict(test_images)
sgd_preds = SGD_model.predict(test_images)
sgd_momentum_preds = SGD_momentum_model.predict(test_images)

n_classes = 10

# Binarize the test labels for multi-class ROC AUC
y_bin = label_binarize(test_labels, classes=list(range(n_classes)))

# Function to compute and plot ROC curves
def plot_roc_curves(y_pred_proba, y_bin, model_name, n_classes=10):
    fpr = dict()
    tpr = dict()
    roc_auc_dict = dict()

    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_proba[:, i])
        roc_auc_dict[i] = auc(fpr[i], tpr[i])

    # Calculate macro-average AUC
    macro_auc = np.mean(list(roc_auc_dict.values()))

    # Plot ROC curves for each class (showing first 5 classes for clarity)
    plt.figure(figsize=(12, 8))
    colors = plt.cm.tab10(np.linspace(0, 1, n_classes))

    for i in range(min(n_classes, 10)):
        plt.plot(fpr[i], tpr[i], color=colors[i], lw=1.5, alpha=0.7,
                 label=f'Class {i} (AUC = {roc_auc_dict[i]:.3f})')

    plt.plot([0, 1], [0, 1], 'k--', lw=2, label='Random Classifier')
```

```
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate', fontsize=12)
    plt.ylabel('True Positive Rate', fontsize=12)
    plt.title(f'ROC Curves - MNIST with {model_name}\n(Macro-average AUC = {macro_auc:.4f}
              fontsize=14, fontweight='bold')
    plt.legend(loc="lower right", fontsize=9)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    return macro_auc, roc_auc_dict

print("\n" + "="*70)
print("ROC AUC ANALYSIS FOR MNIST CLASSIFICATION")
print("="*70)

# Plot ROC for Adam
print("\n1. ADAM OPTIMIZER")
adam_macro_auc, adam_auc_dict = plot_roc_curves(adam_preds, y_bin, 'Adam')
print(f"Macro-average AUC (Adam): {adam_macro_auc:.4f}")

# Plot ROC for SGD
print("\n2. SGD WITHOUT MOMENTUM")
sgd_macro_auc, sgd_auc_dict = plot_roc_curves(sgd_preds, y_bin, 'SGD (No Momentum)')
print(f"Macro-average AUC (SGD): {sgd_macro_auc:.4f}")

# Plot ROC for SGD + Momentum
print("\n3. SGD WITH MOMENTUM")
sgd_momentum_macro_auc, sgd_momentum_auc_dict = plot_roc_curves(sgd_momentum_preds, y_bin
print(f"Macro-average AUC (SGD + Momentum): {sgd_momentum_macro_auc:.4f}")

# Comparison plot
print("\n" + "="*70)
print("OPTIMIZER COMPARISON - MACRO-AVERAGE AUC")
print("="*70)
plt.figure(figsize=(10, 6))
optimizers = ['Adam', 'SGD\n(No Momentum)', 'SGD +\nMomentum']
auc_scores = [adam_macro_auc, sgd_macro_auc, sgd_momentum_macro_auc]
```

```
colors_bar = ['#FF6B6B', '#4ECDC4', '#45B7D1']

bars = plt.bar(optimizers, auc_scores, color=colors_bar, alpha=0.8, edgecolor='black', li
plt.ylabel('Macro-average AUC', fontsize=12)
plt.title('MNIST Optimizer Comparison - ROC AUC Score', fontsize=14, fontweight='bold')
plt.ylim([0, 1.0])
plt.grid(True, alpha=0.3, axis='y')

# Add value labels on bars
for bar, score in zip(bars, auc_scores):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height,
            f'{score:.4f}',
            ha='center', va='bottom', fontsize=12, fontweight='bold')

plt.tight_layout()
plt.show()

print(f"\nBest Optimizer (by AUC): {optimizers[np.argmax(auc_scores)]} with AUC = {max(au
```

Notebook: Q9.ipynb

```
from tensorflow.keras.datasets import mnist
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

```
data = mnist.load_data()
(train_images, train_labels), (test_images, test_labels) = data
```

```
img_size =  train_images[0].shape[0]
```

```
num_classes = 10
```

```
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0
```

```
plt.imshow(train_images[0], cmap='gray')
plt.title(f'Label: {train_labels[0]}')
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten,RandomRotation, GaussianNoise
```

```
from tensorflow.keras.optimizers import SGD, Adam
```

```
Gen_model = Sequential([
    Input(shape =(img_size,img_size) ),
    RandomRotation(0.1),
    GaussianNoise(0.05),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

Gen_model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',metrics=['a
Gen_history = Gen_model.fit(train_images,train_labels, epochs = 50)
```

Without Generalization

```
data = mnist.load_data()
(train_images, train_labels), (test_images, test_labels) = data
img_size =  train_images[0].shape[0]
num_classes = 10
```

```
NonGen_model = Sequential([
    Input(shape =(img_size,img_size) ),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])
callbacks = tensorflow.k
NonGen_model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',metrics=
NonGen_history = NonGen_model.fit(train_images,train_labels, epochs = 50)
```

```
import matplotlib.pyplot as plt
epochs = range(1, len(NonGen_history.history['loss']) + 1)

plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(epochs, Gen_history.history['loss'], label='Gen loss')
plt.plot(epochs, NonGen_history.history['loss'], label='Non-gen loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.legend()

plt.subplot(1,2,2)
plt.plot(epochs, Gen_history.history['accuracy'], label='Gen acc')
plt.plot(epochs, NonGen_history.history['accuracy'], label='Non-gen acc')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy')
plt.legend()

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

Notebook: Q10.ipynb

SimCLR on CIFAR10 (with vs without augmentations)

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
tf.random.set_seed(42)
np.random.seed(42)
```

```
# Load cifar10 and keep a small subset for speed
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

subset = 12000
x_train_small = x_train[:subset]
y_train_small = y_train[:subset]
input_shape = x_train_small.shape[1:]
num_classes = 10
```

```
# Augmentation builders
strong_aug = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255),
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.15),
    tf.keras.layers.RandomTranslation(0.1, 0.1),
    tf.keras.layers.RandomContrast(0.2),
    tf.keras.layers.GaussianNoise(0.05),
])

def identity_aug(x):
    return x
```

```
AUTOTUNE = tf.data.AUTOTUNE
batch_size = 256
temperature = 0.1
epochs = 6

def make_simclr_ds(images, aug_fn):
    ds = tf.data.Dataset.from_tensor_slices(images)
    ds = ds.shuffle(10000)
    ds = ds.map(lambda x: (aug_fn(x, training=True), aug_fn(x, training=True)), num_paral
    ds = ds.batch(batch_size).prefetch(AUTOTUNE)
    return ds

ds_aug = make_simclr_ds(x_train_small, strong_aug)
ds_plain = make_simclr_ds(x_train_small, lambda x, training=True: identity_aug(x))
```

```
def build_encoder():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=input_shape),
        tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
        tf.keras.layers.Conv2D(64, 3, padding='same', strides=2, activation='relu'),
        tf.keras.layers.Conv2D(128, 3, padding='same', strides=2, activation='relu'),
        tf.keras.layers.GlobalAveragePooling2D(),
])

def build_projector():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(128,)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(64),
])
```

```
def nt_xent(z1, z2, temperature=0.1):
    z1 = tf.math.l2_normalize(z1, axis=1)
    z2 = tf.math.l2_normalize(z2, axis=1)
    z = tf.concat([z1, z2], axis=0)  # (2B, d)
    sim = tf.matmul(z, z, transpose_b=True) / temperature
    batch_size = tf.shape(z1)[0] * 2
    diag_mask = tf.eye(batch_size)
    sim = sim - 1e9 * diag_mask  # remove self-similarity
    # Positive mask: pairs (i, i+B) and (i+B, i)
    b = tf.shape(z1)[0]
    pos_mask_top = tf.concat([tf.zeros((b, b)), tf.eye(b)], axis=1)
    pos_mask_bottom = tf.concat([tf.eye(b), tf.zeros((b, b))], axis=1)
    pos_mask = tf.concat([pos_mask_top, pos_mask_bottom], axis=0)
    exp_sim = tf.exp(sim)
    log_prob = tf.math.log(tf.reduce_sum(exp_sim * pos_mask, axis=1) / tf.reduce_sum(exp_s
    loss = -tf.reduce_mean(log_prob)
    return loss
```

```
@tf.function
def train_step(encoder, projector, optimizer, x1, x2):
    with tf.GradientTape() as tape:
        h1 = encoder(x1, training=True)
        h2 = encoder(x2, training=True)
        z1 = projector(h1, training=True)
        z2 = projector(h2, training=True)
        loss = nt_xent(z1, z2, temperature)
    grads = tape.gradient(loss, encoder.trainable_variables + projector.trainable_variable
    optimizer.apply_gradients(zip(grads, encoder.trainable_variables + projector.trainable
    return loss
```

```
def train_simclr(ds):
    encoder = build_encoder()
    projector = build_projector()
    # build variables once to avoid creation inside tf.function
    dummy = tf.zeros((1,) + input_shape)
    h_dummy = encoder(dummy, training=False)
    _ = projector(h_dummy, training=False)
    optimizer = tf.keras.optimizers.Adam(1e-3)
    optimizer.build(encoder.trainable_variables + projector.trainable_variables)

    @tf.function
    def train_step(x1, x2):
        with tf.GradientTape() as tape:
            h1 = encoder(x1, training=True)
            h2 = encoder(x2, training=True)
            z1 = projector(h1, training=True)
            z2 = projector(h2, training=True)
            loss = nt_xent(z1, z2, temperature)
        grads = tape.gradient(loss, encoder.trainable_variables + projector.trainable_vari
        optimizer.apply_gradients(zip(grads, encoder.trainable_variables + projector.trai
        return loss

    history = []
    for epoch in range(epochs):
        losses = []
        for x1, x2 in ds:
            loss = train_step(x1, x2)
            losses.append(loss.numpy())
        history.append(np.mean(losses))
        print(f"Epoch {epoch+1}/{epochs} - loss: {history[-1]:.4f}")
    return encoder, projector, history

encoder_aug, projector_aug, loss_aug = train_simclr(ds_aug)
encoder_plain, projector_plain, loss_plain = train_simclr(ds_plain)
```

```
plt.figure(figsize=(8,4))
plt.plot(loss_aug, label='With augmentation')
plt.plot(loss_plain, label='Without augmentation')
plt.xlabel('Epoch')
plt.ylabel('NT-Xent loss')
plt.title('SimCLR contrastive loss')
plt.legend()
plt.grid(True)
plt.show()
```

Notebook: Q11.ipynb

Transfer learning on Intel Image dataset (frozen vs fine-tuned MobileNetV2)

```
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image_dataset_from_directory
tf.random.set_seed(42)
```

```
data_dir = 'Data/Intel Image Data/seg_train/seg_train'
img_size = (160, 160)
batch_size = 32

train_ds = image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset='training',
    seed=42,
    image_size=img_size,
    batch_size=batch_size
)

val_ds = image_dataset_from_directory(
    'Data/Intel Image Data/seg_test/seg_test',
    validation_split=0.2,
    subset='validation',
    seed=42,
    image_size=img_size,
    batch_size=batch_size
)

class_names = train_ds.class_names
num_classes = len(class_names)
class_names
```

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
])

train_ds = train_ds.map(lambda x, y: (data_augmentation(x, training=True), y))
```

```
# ----- Model 1: Frozen MobileNetV2 encoder (Sequential) -----
base_model_frozen = tf.keras.applications.MobileNetV2(
    input_shape=img_size + (3,),
    include_top=False,
    weights='imagenet'
)
base_model_frozen.trainable = False

model_frozen = tf.keras.Sequential([
    base_model_frozen,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

model_frozen.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
 )
model_frozen.summary()
```

```
history_frozen = model_frozen.fit(train_ds,
                                  epochs=5,
                                  validation_data=val_ds)
eval_frozen = model_frozen.evaluate(val_ds)
print('Frozen encoder val loss/acc:', eval_frozen)
```

```
# ----- Model 2: Separate model for fine-tuning (Sequential) -----
base_model_ft = tf.keras.applications.MobileNetV2(
    input_shape=img_size + (3,),
    include_top=False,
    weights='imagenet'
)
base_model_ft.trainable = True

# Freeze most of the encoder; fine-tune only the last ~30 layers
fine_tune_at = len(base_model_ft.layers) - 30
for layer in base_model_ft.layers[:fine_tune_at]:
    layer.trainable = False

model_ft = tf.keras.Sequential([
    base_model_ft,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

model_ft.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
 )

history_ft = model_ft.fit(
    train_ds,
    epochs=5,
    validation_data=val_ds,
 )
eval_ft = model_ft.evaluate(val_ds)
print('Fine-tuned val loss/acc:', eval_ft)
```

```
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(history_frozen.history['accuracy'], label='Frozen train acc')
plt.plot(history_frozen.history['val_accuracy'], label='Frozen val acc')
plt.plot([None]*len(history_frozen.history['accuracy']) + history_ft.history['accuracy'],
plt.plot([None]*len(history_frozen.history['val_accuracy']) + history_ft.history['val_accu
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Acc')
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_frozen.history['loss'], label='Frozen train loss')
plt.plot(history_frozen.history['val_loss'], label='Frozen val loss')
plt.plot([None]*len(history_frozen.history['loss']) + history_ft.history['loss'], label='
plt.plot([None]*len(history_frozen.history['val_loss']) + history_ft.history['val_loss'],
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()
```

```python
# Plot ROC AUC Curves for Multi-class Classification (One-vs-Rest)
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.preprocessing import label_binarize
import numpy as np
import matplotlib.pyplot as plt

# Get predictions from both models
y_true_list = []
y_frozen_proba = []
y_ft_proba = []

for x, y in val_ds:
    y_true_list.extend(y.numpy())
    y_frozen_proba.extend(model_frozen.predict(x, verbose=0))
    y_ft_proba.extend(model_ft.predict(x, verbose=0))

y_true = np.array(y_true_list)
y_frozen_proba = np.array(y_frozen_proba)
y_ft_proba = np.array(y_ft_proba)

# Binarize the output for multi-class ROC AUC
n_classes = num_classes
y_bin = label_binarize(y_true, classes=list(range(n_classes)))

# Function to compute and plot ROC curves
def plot_roc_multiclass(y_pred_proba, y_bin, model_name, n_classes, class_names):
    fpr = dict()
    tpr = dict()
    roc_auc_dict = dict()
    colors = plt.cm.Set1(np.linspace(0, 1, n_classes))

    plt.figure(figsize=(12, 8))

    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_proba[:, i])
        roc_auc_dict[i] = auc(fpr[i], tpr[i])
        plt.plot(fpr[i], tpr[i], color=colors[i], lw=2,
                 label=f'{class_names[i]} (AUC = {roc_auc_dict[i]:.3f})')
```

```
    # Plot random classifier baseline
    plt.plot([0, 1], [0, 1], 'k--', lw=2, label='Random Classifier')

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate', fontsize=12)
    plt.ylabel('True Positive Rate', fontsize=12)
    plt.title(f'ROC Curves - {model_name}', fontsize=14, fontweight='bold')
    plt.legend(loc="lower right", fontsize=9)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    # Calculate macro and weighted averages
    macro_auc = np.mean(list(roc_auc_dict.values()))
    weights = np.bincount(y_true) / len(y_true)
    weighted_auc = np.average([roc_auc_dict[i] for i in range(n_classes)], weights=weights

    return macro_auc, weighted_auc, roc_auc_dict

print("\n" + "="*70)
print("ROC AUC ANALYSIS - TRANSFER LEARNING MODELS")
print("="*70)

# Plot ROC for Frozen Model
print("\n1. FROZEN MOBILENETV2 ENCODER")
frozen_macro_auc, frozen_weighted_auc, frozen_auc_dict = plot_roc_multiclass(
    y_frozen_proba, y_bin, 'Frozen MobileNetV2', n_classes, class_names)
print(f"Macro-average AUC: {frozen_macro_auc:.4f}")
print(f"Weighted-average AUC: {frozen_weighted_auc:.4f}")

# Plot ROC for Fine-tuned Model
print("\n2. FINE-TUNED MOBILENETV2")
ft_macro_auc, ft_weighted_auc, ft_auc_dict = plot_roc_multiclass(
    y_ft_proba, y_bin, 'Fine-tuned MobileNetV2', n_classes, class_names)
print(f"Macro-average AUC: {ft_macro_auc:.4f}")
print(f"Weighted-average AUC: {ft_weighted_auc:.4f}")

# Comparison plot
```

```
print("\n" + "="*70)
print("MODEL COMPARISON - AUC SCORES")
print("="*70)

fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Macro-average comparison
models = ['Frozen', 'Fine-tuned']
macro_auc_scores = [frozen_macro_auc, ft_macro_auc]
colors_bar = ['#FF6B6B', '#45B7D1']

axes[0].bar(models, macro_auc_scores, color=colors_bar, alpha=0.8, edgecolor='black', line
axes[0].set_ylabel('Macro-average AUC', fontsize=12)
axes[0].set_title('Macro-average AUC Comparison', fontsize=12, fontweight='bold')
axes[0].set_ylim([0, 1.0])
axes[0].grid(True, alpha=0.3, axis='y')

# Add value labels
for i, (model, score) in enumerate(zip(models, macro_auc_scores)):
    axes[0].text(i, score + 0.02, f'{score:.4f}', ha='center', fontsize=11, fontweight='bo

# Weighted-average comparison
weighted_auc_scores = [frozen_weighted_auc, ft_weighted_auc]

axes[1].bar(models, weighted_auc_scores, color=colors_bar, alpha=0.8, edgecolor='black',
axes[1].set_ylabel('Weighted-average AUC', fontsize=12)
axes[1].set_title('Weighted-average AUC Comparison', fontsize=12, fontweight='bold')
axes[1].set_ylim([0, 1.0])
axes[1].grid(True, alpha=0.3, axis='y')

# Add value labels
for i, (model, score) in enumerate(zip(models, weighted_auc_scores)):
    axes[1].text(i, score + 0.02, f'{score:.4f}', ha='center', fontsize=11, fontweight='bo

plt.tight_layout()
plt.show()

# Determine winner
if ft_macro_auc > frozen_macro_auc:
    print(f"\n✓ Fine-tuned model performs better with macro-average AUC = {ft_macro_auc:.4
```

```
else:
    print(f"\n✓ Frozen model performs better with macro-average AUC = {frozen_macro_auc:.
```

Notebook: Q12.ipynb

Choose two datasets with different distributions (dogs &cats , cars).

1. Resize images to the required input size of the chosen pre-trained model.
2. Load Pre-trained Model (LeNet-5 or VGG-16 or MobileNetV2 or Resnet50 or AlexNet) . Compare the performances of all the models and visualize
3. Write down your observations and conclusions

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
data = tf.keras.datasets.cifar10.load_data()
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
```

```
from tensorflow.keras.applications.resnet_v2 import ResNet50V2
```

```
base_model_resnet = ResNet50V2(input_shape=(32,32,3), weights='imagenet',include_top=Fals
```

```
base_model_resnet
```

```
base_model_resnet.trainable = False
```

```
main_model_resnet = tf.keras.Sequential([
    base_model_resnet,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

```
main_model_resnet.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', met
```

```
main_model_resnet.summary()
```

```
main_model_resnet.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
resnet_losses = main_model_resnet.history.history['loss']
res_acc = main_model_resnet.history.history['accuracy']
```

```
test_loss, test_acc = main_model_resnet.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

```
preds = main_model_resnet.predict(x_test)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 's
true_labels = y_test.flatten()

predicted_labels = np.argmax(preds, axis=1)

cm = confusion_matrix(true_labels, predicted_labels)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp.plot()
plt.title('Confusion Matrix - ResNet50V2')
plt.tight_layout()
plt.show()
```

```
import gc
tf.keras.backend.clear_session()
gc.collect()
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2

base_model_mobilenet = MobileNetV2(input_shape=(32,32,3), weights='imagenet',include_top=F
base_model_mobilenet
base_model_mobilenet.trainable = False
main_model_mobilenet = tf.keras.Sequential([
    base_model_mobilenet,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

main_model_mobilenet.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', m
main_model_mobilenet.summary()
main_model_mobilenet.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

mobilenet_losses = main_model_mobilenet.history.history['loss']
mobilenet_acc = main_model_mobilenet.history.history['accuracy']
test_loss, test_acc = main_model_mobilenet.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

```
preds = main_model_mobilenet.predict(x_test, batch_size=32)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 's
true_labels = y_test.flatten()

predicted_labels = np.argmax(preds, axis=1)

cm = confusion_matrix(true_labels, predicted_labels)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp.plot()
plt.title('Confusion Matrix - MobileNetV2')
plt.tight_layout()
plt.show()
```

```
import gc
tf.keras.backend.clear_session()
gc.collect()
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
from tensorflow.keras.applications.vgg16 import VGG16

base_model_vgg16 = VGG16(input_shape=(32,32,3), weights='imagenet',include_top=False )
base_model_vgg16
base_model_vgg16.trainable = False
main_model_vgg16 = tf.keras.Sequential([
    base_model_vgg16,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

main_model_vgg16.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', metri
main_model_vgg16.summary()
main_model_vgg16.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

vgg16_losses = main_model_vgg16.history.history['loss']
vgg16_acc = main_model_vgg16.history.history['accuracy']

test_loss, test_acc = main_model_vgg16.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

```
y_preds = main_model_vgg16.predict(x_test, batch_size=32)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 's
true_labels = y_test.flatten()

predicted_labels = np.argmax(preds, axis=1)

cm = confusion_matrix(true_labels, predicted_labels)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp.plot()
plt.title('Confusion Matrix - VGG16')
plt.tight_layout()
plt.show()
```

```
import gc
tf.keras.backend.clear_session()
gc.collect()
```

```
x = range(1, 11)
plt.figure(figsize=(8,6))
plt.plot(x,resnet_losses, label='ResNet50V2 Loss', marker='x')
plt.plot(x,mobilenet_losses, label='MobileNetV2 Loss', marker='o')
plt.plot(x,vgg16_losses, label='VGG16 Loss', marker='s')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Comparison')
plt.legend()
plt.grid()
plt.title('Training Loss Comparison')
```

```
x = range(1, 11)
plt.figure(figsize=(8,6))
plt.plot(x,res_acc, label='ResNet50V2 Accuracy', marker='x')
plt.plot(x,mobilenet_acc, label='MobileNetV2 Accuracy', marker='o')
plt.plot(x,vgg16_acc, label='VGG16 Accuracy', marker='s')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training Accuracy Comparison')
plt.legend(loc='upper left')
plt.grid()
plt.title('Training Accuracy Comparison')
```

```
```

Notebook: Q13.ipynb

Task 1

1. Take the dataset of Iris.
2. Initialize a neural network with random weights.
3. Calculate output of Neural Network:
4. Calculate MSE
5. Plot error surface using loss function verses weight, bias
6. Perform this cycle in step c for every input output pair
7. Perform 10 epochs of step d
8. Update weights accordingly using stochastic gradient descend.
9. Plot the mean squared error for each iteration in stochastic Gradient Descent.
10. Similarly plot accuracy for iteration and note the results

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from mpl_toolkits.mplot3d import Axes3D

# 1. Take the dataset of Iris
iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)
print(X.shape, y.shape)
# Preprocessing
encoder = OneHotEncoder(sparse_output=False)
y_onehot = encoder.fit_transform(y)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split (using full dataset for training as implied by "dataset of Iris")
X_train, y_train = X_scaled, y_onehot
```

```python
# 2. Initialize a neural network with random weights
# Simple architecture: 4 Inputs -> 3 Outputs (Linear layer + Sigmoid activation)
input_size = 4
output_size = 3

np.random.seed(42)
weights = np.random.randn(input_size, output_size)
bias = np.random.randn(output_size)

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

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

def predict(inputs, w, b):
    # 3. Calculate output of Neural Network
    return sigmoid(np.dot(inputs, w) + b)

def calculate_mse(y_true, y_pred):
    # 4. Calculate MSE
    return np.mean((y_true - y_pred) ** 2)
```

```
# 5. Plot error surface using loss function verses weight, bias
# We pick one weight (w[0,0]) and one bias (b[0]) to vary for visualization
w_range = np.linspace(-5, 5, 50)
b_range = np.linspace(-5, 5, 50)
W_grid, B_grid = np.meshgrid(w_range, b_range)
loss_grid = np.zeros_like(W_grid)

w_temp = weights.copy()
b_temp = bias.copy()

for i in range(len(w_range)):
    for j in range(len(b_range)):
        w_temp[0, 0] = W_grid[i, j]
        b_temp[0] = B_grid[i, j]
        # Compute total MSE for dataset with these parameters
        y_out = predict(X_train, w_temp, b_temp)
        loss_grid[i, j] = calculate_mse(y_train, y_out)

fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(W_grid, B_grid, loss_grid, cmap='viridis')
ax.set_title('Error Surface (MSE vs Weight[0,0] vs Bias[0])')
ax.set_xlabel('Weight')
ax.set_ylabel('Bias')
ax.set_zlabel('MSE Loss')
plt.show()
```

```python
# Training loop parameters
learning_rate = 0.1
epochs = 10
mse_history = []
accuracy_history = []

print("Starting training...")

for epoch in range(epochs):
    epoch_errors = []
    correct_predictions = 0

    # 6. Perform this cycle in step c for every input output pair
    for i in range(len(X_train)):
        x_sample = X_train[i].reshape(1, -1)
        y_sample = y_train[i].reshape(1, -1)

        output = predict(x_sample, weights, bias)

        error = y_sample - output
        sample_mse = np.mean(error ** 2)
        epoch_errors.append(sample_mse)

        # Accuracy check
        if np.argmax(output) == np.argmax(y_sample):
            correct_predictions += 1


        d_output = error * sigmoid_derivative(output)

        weights += learning_rate * np.dot(x_sample.T, d_output)
        bias += learning_rate * np.sum(d_output, axis=0)

    avg_mse = np.mean(epoch_errors)
    epoch_acc = correct_predictions / len(X_train)

    mse_history.append(avg_mse)
    accuracy_history.append(epoch_acc)
```

```
    print(f"Epoch {epoch+1}/{epochs} - MSE: {avg_mse:.4f} - Accuracy: {epoch_acc:.4f}")
```

```
# 9. Plot the mean squared error for each iteration (epoch)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, epochs + 1), mse_history, marker='o', label='MSE')
plt.title('Mean Squared Error per Epoch')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.grid(True)
plt.legend()

# 10. Similarly plot accuracy for iteration and note the results
plt.subplot(1, 2, 2)
plt.plot(range(1, epochs + 1), accuracy_history, marker='o', color='orange', label='Accura
plt.title('Accuracy per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()

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

```
```

Notebook: Q14.ipynb

Task 1

1. Implement batch gradient descent optimizer function Take the dataset of Titanic
2. Initialize a neural network with random weights.

3. Calculate output of Neural Network:
4. Calculate squared error loss
5. Update network parameter using batch gradient descent optimizer function Implementation.
6. Display updated weight and bias values
7. Plot loss w.r.t. Iterations

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 1. Load Titanic dataset
titanic = pd.read_csv('/media/smayan/500GB SSD/Study/ML2/Practicals/Data/titanic.csv')

# Preprocessing
# Select relevant features and handle missing values
# Adjust column names based on actual CSV (common Titanic datasets use 'Pclass', 'Sex', 'A
df = titanic.copy()

# Standardize column names to lowercase for consistency
df.columns = df.columns.str.lower()

# Select relevant features - use actual column names from the dataset
feature_cols = ['pclass', 'sex', 'age', 'siblings/spouses aboard', 'parents/children aboa
target_col = 'survived'

# Keep only necessary columns
df = df[feature_cols + [target_col]].copy()

# Handle missing values
df['age'].fillna(df['age'].median(), inplace=True)
df['fare'].fillna(df['fare'].median(), inplace=True)
df.dropna(inplace=True)

# Encode categorical variable
df['sex'] = df['sex'].map({'male': 0, 'female': 1, 'Male': 0, 'Female': 1})

# Separate features and target
X = df[feature_cols].values
y = df[target_col].values.reshape(-1, 1)

# Standardize features
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
print(f"Dataset shape: X={X.shape}, y={y.shape}")
print(f"Features: {feature_cols}")
```

```
# 2. Initialize a neural network with random weights
input_size = X.shape[1]  # 6 features
hidden_size = 4
output_size = 1  # Binary classification (survived or not)

np.random.seed(42)

# Initialize weights and biases
W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size) * 0.01
b2 = np.zeros((1, output_size))

print("Initial Weights and Biases:")
print(f"W1 shape: {W1.shape}")
print(f"b1 shape: {b1.shape}")
print(f"W2 shape: {W2.shape}")
print(f"b2 shape: {b2.shape}")
```

```python
# Activation functions
def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

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

# 3. Forward pass - Calculate output of Neural Network
def forward_propagation(X, W1, b1, W2, b2):
    """
    Perform forward propagation through the network
    Returns: output predictions and intermediate values for backprop
    """
    # Hidden layer
    Z1 = np.dot(X, W1) + b1
    A1 = sigmoid(Z1)

    # Output layer
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)

    cache = {'Z1': Z1, 'A1': A1, 'Z2': Z2, 'A2': A2}
    return A2, cache
```

```python
# 4. Calculate squared error loss
def compute_loss(y_true, y_pred):
    """
    Compute mean squared error loss
    """
    m = y_true.shape[0]
    loss = (1 / (2 * m)) * np.sum((y_pred - y_true) ** 2)
    return loss

# Backward propagation
def backward_propagation(X, y, cache, W1, W2):
    """
    Compute gradients using backpropagation
    """
    m = X.shape[0]

    A1 = cache['A1']
    A2 = cache['A2']

    # Output layer gradients
    dZ2 = (A2 - y) * sigmoid_derivative(A2)
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)

    # Hidden layer gradients
    dZ1 = np.dot(dZ2, W2.T) * sigmoid_derivative(A1)
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    gradients = {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}
    return gradients
```

```python
# 5. Implement Batch Gradient Descent Optimizer with Momentum
def batch_gradient_descent(X, y, W1, b1, W2, b2, learning_rate=0.01, iterations=1000, mome
    """
    Batch Gradient Descent Optimizer with Momentum
    Updates weights using gradients computed on the entire dataset

    Parameters:
    - momentum: momentum coefficient (0 = no momentum, typically 0.9)
    """
    loss_history = []

    # Initialize velocity terms for momentum
    vW1 = np.zeros_like(W1)
    vb1 = np.zeros_like(b1)
    vW2 = np.zeros_like(W2)
    vb2 = np.zeros_like(b2)

    for i in range(iterations):
        # Forward propagation
        y_pred, cache = forward_propagation(X, W1, b1, W2, b2)

        # Compute loss
        loss = compute_loss(y, y_pred)
        loss_history.append(loss)

        # Backward propagation
        gradients = backward_propagation(X, y, cache, W1, W2)

        # Update velocities with momentum
        vW1 = momentum * vW1 + (1-momentum) * learning_rate * gradients['dW1']
        vb1 = momentum * vb1 + (1-momentum) * learning_rate * gradients['db1']
        vW2 = momentum * vW2 + (1-momentum) * learning_rate * gradients['dW2']
        vb2 = momentum * vb2 + (1-momentum) * learning_rate * gradients['db2']

        # Update parameters using velocity
        W1 = W1 - vW1
        b1 = b1 - vb1
        W2 = W2 - vW2
        b2 = b2 - vb2
```

```
        # Print progress every 100 iterations
        if (i + 1) % 100 == 0:
            print(f"Iteration {i+1}/{iterations}, Loss: {loss:.4f}")

    parameters = {'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
    return parameters, loss_history

# Train the network
print("Training Neural Network with Batch Gradient Descent...")
print("="*60)
parameters, loss_history = batch_gradient_descent(
    X, y, W1, b1, W2, b2,
    learning_rate=0.5,
    iterations=1000,
    momentum=0.9
)
```

```python
# 6. Display updated weight and bias values
print("\n" + "="*60)
print("UPDATED WEIGHTS AND BIASES AFTER TRAINING")
print("="*60)

print("\nLayer 1 (Input -> Hidden):")
print(f"W1 (shape {parameters['W1'].shape}):")
print(parameters['W1'])
print(f"\nb1 (shape {parameters['b1'].shape}):")
print(parameters['b1'])

print("\nLayer 2 (Hidden -> Output):")
print(f"W2 (shape {parameters['W2'].shape}):")
print(parameters['W2'])
print(f"\nb2 (shape {parameters['b2'].shape}):")
print(parameters['b2'])

print("\n" + "="*60)
print(f"Final Loss: {loss_history[-1]:.4f}")
print(f"Initial Loss: {loss_history[0]:.4f}")
print(f"Loss Reduction: {loss_history[0] - loss_history[-1]:.4f}")
print("="*60)
```

```python
# 7. Plot loss w.r.t. Iterations
plt.figure(figsize=(12, 5))

# Plot 1: Loss over all iterations
plt.subplot(1, 2, 1)
plt.plot(loss_history, linewidth=2, color='blue')
plt.title('Loss vs Iterations (Batch Gradient Descent)', fontsize=14, fontweight='bold')
plt.xlabel('Iteration', fontsize=12)
plt.ylabel('Mean Squared Error Loss', fontsize=12)
plt.grid(True, alpha=0.3)

# Plot 2: Loss over iterations (log scale for better visualization)
plt.subplot(1, 2, 2)
plt.plot(loss_history, linewidth=2, color='red')
plt.title('Loss vs Iterations (Log Scale)', fontsize=14, fontweight='bold')
plt.xlabel('Iteration', fontsize=12)
plt.ylabel('Mean Squared Error Loss (Log Scale)', fontsize=12)
plt.yscale('log')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Calculate accuracy
final_predictions, _ = forward_propagation(X, parameters['W1'], parameters['b1'],
                                           parameters['W2'], parameters['b2'])
predicted_classes = (final_predictions > 0.5).astype(int)
accuracy = np.mean(predicted_classes == y) * 100

print(f"\nFinal Model Accuracy: {accuracy:.2f}%")
```

```
# Plot ROC AUC Curve
from sklearn.metrics import roc_curve, auc, roc_auc_score

# Get probability predictions
y_pred_proba, _ = forward_propagation(X, parameters['W1'], parameters['b1'],
                                      parameters['W2'], parameters['b2'])

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y, y_pred_proba)
roc_auc = auc(fpr, tpr)

# Plot ROC Curve
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.3f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Classifier')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', fontsize=12)
plt.ylabel('True Positive Rate', fontsize=12)
plt.title('ROC Curve - Titanic Survival Prediction (Binary Classification)', fontsize=14,
plt.legend(loc="lower right", fontsize=11)
plt.grid(True, alpha=0.3)
plt.show()

print(f"\nROC AUC Score: {roc_auc:.4f}")
print(f"Model Performance: {'Excellent' if roc_auc >= 0.9 else 'Good' if roc_auc >= 0.8 e
```

Notebook: Q15.ipynb

Task 1.

Implement the NOR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

Task 2

Take the dataset of Diabetes 2

- Initialize a neural network with random weights.
- Calculate output of Neural Network:
    i. Calculate squared error loss
    ii. Update network parameter using batch Mini Batch gradient descent optimizer function Implementation.
    iii. Display updated weight and bias values
    iv. Plot loss w.r.t. bias values

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

# Task 1: Implement NOR Boolean Logic Gate using Perceptron

print("="*60)
print("TASK 1: NOR GATE USING PERCEPTRON")
print("="*60)

# NOR Truth Table:
# x1  x2  | y
# 0   0   | 1
# 0   1   | 0
# 1   0   | 0
# 1   1   | 0

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

print("\nNOR Truth Table:")
print("x1  x2  |  y")
print("-"*15)
for i in range(len(X)):
    print(f"{X[i][0]}   {X[i][1]}   |  {y[i]}")
```

```python
# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# Initialize weights and bias randomly
np.random.seed(42)
w = np.random.rand(2)
b = np.random.rand(1)

print(f"\nInitial weights: {w}")
print(f"Initial bias: {b}")

# Training parameters
epochs = 100
learning_rate = 0.1

print("\nTraining NOR Perceptron...")

# Training loop
for epoch in range(epochs):
    total_error = 0
    for i in range(X.shape[0]):
        # Forward pass
        z = np.dot(w, X[i]) + b
        y_pred = step(z)

        # Calculate error
        error = y[i] - y_pred
        total_error += abs(error)

        # Update weights and bias
        w += learning_rate * error * X[i]
        b += learning_rate * error

    # Stop if converged
    if total_error == 0:
        print(f"Converged at epoch {epoch + 1}")
        break
```

```python
# Display final weights and bias
print("\n" + "="*60)
print("FINAL WEIGHTS AND BIAS FOR NOR PERCEPTRON")
print("="*60)
print(f"\nTrained weights: {w}")
print(f"Trained bias: {b}")

# Test the trained perceptron
print("\nTesting NOR Perceptron:")
print("x1  x2  | Predicted | Actual")
print("-"*35)
for i in range(len(X)):
    z = np.dot(w, X[i]) + b
    prediction = step(z)
    print(f"{X[i][0]}   {X[i][1]}   |     {prediction}     |   {y[i]}")

# Verify all predictions are correct
correct = sum([step(np.dot(w, X[i]) + b) == y[i] for i in range(len(X))])
print(f"\nAccuracy: {correct}/{len(X)} = {(correct/len(X))*100:.0f}%")
```

```python
print("\n" + "="*60)
print("TASK 2: DIABETES DATASET WITH MINI-BATCH GRADIENT DESCENT")
print("="*60)

import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_diabetes

# Load Diabetes dataset
diabetes = load_diabetes()
X_diabetes = diabetes.data
y_diabetes = diabetes.target.reshape(-1, 1)

# Standardize features (already somewhat standardized, but ensuring consistency)
scaler_X = StandardScaler()
X_diabetes = scaler_X.fit_transform(X_diabetes)

# Standardize target for better training
scaler_y = StandardScaler()
y_diabetes = scaler_y.fit_transform(y_diabetes)

print(f"\nDataset shape: X={X_diabetes.shape}, y={y_diabetes.shape}")
print(f"Number of features: {X_diabetes.shape[1]}")
print(f"Number of samples: {X_diabetes.shape[0]}")
print(f"Feature names: {diabetes.feature_names}")
```

```python
# Initialize Neural Network with random weights
input_size = X_diabetes.shape[1]  # 10 features
hidden_size = 8
output_size = 1

np.random.seed(42)

# Layer 1: Input -> Hidden
W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))

# Layer 2: Hidden -> Output
W2 = np.random.randn(hidden_size, output_size) * 0.01
b2 = np.zeros((1, output_size))

print("\nInitial Network Architecture:")
print(f"Input Layer: {input_size} neurons")
print(f"Hidden Layer: {hidden_size} neurons")
print(f"Output Layer: {output_size} neuron")
print(f"\nW1 shape: {W1.shape}")
print(f"b1 shape: {b1.shape}")
print(f"W2 shape: {W2.shape}")
print(f"b2 shape: {b2.shape}")
```

```
# Define activation functions
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return (z > 0).astype(float)

# Forward propagation
def forward_pass(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = Z2  # Linear activation for regression

    cache = {'Z1': Z1, 'A1': A1, 'Z2': Z2, 'A2': A2}
    return A2, cache

# Calculate squared error loss (MSE)
def compute_loss(y_true, y_pred):
    m = y_true.shape[0]
    loss = (1 / (2 * m)) * np.sum((y_pred - y_true) ** 2)
    return loss

# Backward propagation
def backward_pass(X, y, cache, W1, W2):
    m = X.shape[0]

    A1 = cache['A1']
    A2 = cache['A2']
    Z1 = cache['Z1']

    # Output layer gradients
    dZ2 = (A2 - y)
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)

    # Hidden layer gradients
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
```

```
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    gradients = {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}
    return gradients

print("\nForward and Backward propagation functions defined.")
```

```python
# Mini-Batch Gradient Descent Optimizer with Momentum
def mini_batch_gradient_descent(X, y, W1, b1, W2, b2, batch_size=32, learning_rate=0.01, e
    """
    Mini-Batch Gradient Descent Optimizer with Momentum
    Updates weights using gradients computed on mini-batches of data
    """
    m = X.shape[0]
    loss_history = []

    # Initialize velocity for momentum
    vW1 = np.zeros_like(W1)
    vb1 = np.zeros_like(b1)
    vW2 = np.zeros_like(W2)
    vb2 = np.zeros_like(b2)

    print("\nTraining with Mini-Batch Gradient Descent with Momentum...")
    print(f"Batch size: {batch_size}")
    print(f"Learning rate: {learning_rate}")
    print(f"Momentum: {momentum}")
    print(f"Epochs: {epochs}")
    print("-" * 60)

    for epoch in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.random.permutation(m)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        epoch_loss = 0
        num_batches = 0

        # Process mini-batches
        for i in range(0, m, batch_size):
            # Get mini-batch
            X_batch = X_shuffled[i:i+batch_size]
            y_batch = y_shuffled[i:i+batch_size]

            # Forward propagation
            y_pred, cache = forward_pass(X_batch, W1, b1, W2, b2)
```

```
            # Compute loss
            batch_loss = compute_loss(y_batch, y_pred)
            epoch_loss += batch_loss
            num_batches += 1

            # Backward propagation
            gradients = backward_pass(X_batch, y_batch, cache, W1, W2)

            # Update velocity with momentum
            vW1 = momentum * vW1 - (1-momentum) * learning_rate * gradients['dW1']
            vb1 = momentum * vb1 - (1-momentum) * learning_rate * gradients['db1']
            vW2 = momentum * vW2 - (1-momentum) * learning_rate * gradients['dW2']
            vb2 = momentum * vb2 - (1-momentum) * learning_rate * gradients['db2']

            # Update parameters
            W1 += vW1
            b1 += vb1
            W2 += vW2
            b2 += vb2

        # Calculate average loss for the epoch
        avg_loss = epoch_loss / num_batches
        loss_history.append(avg_loss)

        # Print progress every 10 epochs
        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.6f}")

    parameters = {'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
    return parameters, loss_history

# Train the network with momentum
trained_params, loss_history = mini_batch_gradient_descent(
    X_diabetes, y_diabetes, W1, b1, W2, b2,
    batch_size=32,
    learning_rate=0.01,
    epochs=100,
    momentum=0.9
)
```

```
# Display updated weights and bias values
print("\n" + "="*60)
print("UPDATED WEIGHTS AND BIASES AFTER TRAINING")
print("="*60)

print("\nLayer 1 (Input -> Hidden):")
print(f"W1 shape: {trained_params['W1'].shape}")
print(f"W1:\n{trained_params['W1']}")
print(f"\nb1 shape: {trained_params['b1'].shape}")
print(f"b1:\n{trained_params['b1']}")

print("\nLayer 2 (Hidden -> Output):")
print(f"W2 shape: {trained_params['W2'].shape}")
print(f"W2:\n{trained_params['W2']}")
print(f"\nb2 shape: {trained_params['b2'].shape}")
print(f"b2:\n{trained_params['b2']}")

print("\n" + "="*60)
print(f"Initial Loss: {loss_history[0]:.6f}")
print(f"Final Loss: {loss_history[-1]:.6f}")
print(f"Loss Reduction: {loss_history[0] - loss_history[-1]:.6f}")
print("="*60)
```

```python
# Plot loss w.r.t. bias values
# We'll vary one bias value and see how loss changes

print("\nGenerating loss surface with respect to bias values...")

# Select one bias from b2 to vary (output layer bias)
bias_range = np.linspace(-2, 2, 100)
loss_vs_bias = []

# Keep all parameters constant except one bias value
W1_temp = trained_params['W1'].copy()
b1_temp = trained_params['b1'].copy()
W2_temp = trained_params['W2'].copy()
b2_temp = trained_params['b2'].copy()

# Store original bias value
original_bias = b2_temp[0, 0]

for bias_val in bias_range:
    # Set the bias to test value
    b2_temp[0, 0] = bias_val

    # Forward pass with this bias
    y_pred, _ = forward_pass(X_diabetes, W1_temp, b1_temp, W2_temp, b2_temp)

    # Compute loss
    loss = compute_loss(y_diabetes, y_pred)
    loss_vs_bias.append(loss)

# Restore original bias
b2_temp[0, 0] = original_bias

# Create the plot
plt.figure(figsize=(14, 5))

# Plot 1: Loss vs Epochs
plt.subplot(1, 2, 1)
plt.plot(range(1, len(loss_history) + 1), loss_history, linewidth=2, color='blue')
plt.title('Loss vs Epochs (Mini-Batch GD)', fontsize=14, fontweight='bold')
```

```
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Mean Squared Error Loss', fontsize=12)
plt.grid(True, alpha=0.3)

# Plot 2: Loss vs Bias Value
plt.subplot(1, 2, 2)
plt.plot(bias_range, loss_vs_bias, linewidth=2, color='red')
plt.axvline(x=original_bias, color='green', linestyle='--', linewidth=2, label=f'Trained b
plt.title('Loss vs Output Bias Value (b2[0,0])', fontsize=14, fontweight='bold')
plt.xlabel('Bias Value', fontsize=12)
plt.ylabel('Mean Squared Error Loss', fontsize=12)
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nOptimal bias value (after training): {original_bias:.6f}")
print(f"Loss at optimal bias: {compute_loss(y_diabetes, forward_pass(X_diabetes, trained_p
```

Notebook: Q16.ipynb

Task 1

- Implement the XOR Boolean logic gate using perceptron Neural Network.
- Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

Task 2

Take the dataset of Penguin

- Initialize a neural network with random weights.
- Calculate output of Neural Network:
  i. Calculate squared error loss
  ii. Update network parameter using batch Adaptive delta gradient descent optimizer function

Implementation.

iii. Display updated weight and bias values

iv. Plot accuracy w.r.t. epoch values

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

# Task 1: XOR Gate Implementation

print("="*60)
print("TASK 1: XOR GATE USING PERCEPTRON")
print("="*60)

# XOR Truth Table:
# x1  x2  | y
# 0   0   | 0
# 0   1   | 1
# 1   0   | 1
# 1   1   | 0

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

print("\nXOR Truth Table:")
print("x1  x2  |  y")
print("-"*15)
for i in range(len(X)):
    print(f"{X[i][0]}   {X[i][1]}   |  {y[i]}")

# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# For XOR, we need multiple perceptrons
# Perceptron 1: learns OR
# Perceptron 2: learns NAND
```

```
# Perceptron 3: learns AND of outputs from 1 and 2

print("\n" + "-"*60)
print("Training Perceptrons for XOR:")
print("-"*60)

# Initialize weights and bias for first two perceptrons
np.random.seed(42)
w1 = np.random.rand(2)
b1 = np.random.rand(1)
w2 = np.random.rand(2)
b2 = np.random.rand(1)
w3 = np.random.rand(2)
b3 = np.random.rand(1)

epochs = 100
learning_rate = 0.1

# Training for XOR decomposition
for epoch in range(epochs):
    total_error = 0
    for i in range(X.shape[0]):
        # First perceptron (OR-like)
        z1 = np.dot(w1, X[i]) + b1
        y1 = step(z1)

        # Second perceptron (NAND-like)
        z2 = np.dot(w2, X[i]) + b2
        y2 = step(z2)

        # Third perceptron (AND of y1 and y2)
        hidden = np.array([y1, y2])
        z3 = np.dot(w3, hidden) + b3
        y_pred = step(z3)

        # Errors
        error3 = y[i] - y_pred
        total_error += abs(error3)

        # Update third perceptron
```

```
        w3 += learning_rate * error3 * hidden
        b3 += learning_rate * error3

        # Backprop errors to first two perceptrons
        error1 = error3 * y2 * (1 if z1 >= 0 else 0)
        error2 = error3 * y1 * (1 if z2 >= 0 else 0)

        w1 += learning_rate * error1 * X[i]
        b1 += learning_rate * error1
        w2 += learning_rate * error2 * X[i]
        b2 += learning_rate * error2

    if total_error == 0:
        print(f"Converged at epoch {epoch + 1}")
        break

print("\n" + "="*60)
print("FINAL WEIGHTS AND BIASES FOR XOR PERCEPTRONS")
print("="*60)
print(f"\nPerceptron 1 (OR-like):")
print(f"  Weights: {w1}, Bias: {b1}")
print(f"\nPerceptron 2 (NAND-like):")
print(f"  Weights: {w2}, Bias: {b2}")
print(f"\nPerceptron 3 (AND of hidden):")
print(f"  Weights: {w3}, Bias: {b3}")

# Test XOR
print("\n" + "="*60)
print("Testing XOR Perceptrons:")
print("="*60)
print("x1  x2  | Predicted | Actual")
print("-"*35)
correct = 0
for i in range(len(X)):
    z1 = np.dot(w1, X[i]) + b1
    y1 = step(z1)
    z2 = np.dot(w2, X[i]) + b2
    y2 = step(z2)
    hidden = np.array([y1, y2])
    z3 = np.dot(w3, hidden) + b3
```

```
    y_pred = step(z3)
    print(f"{X[i][0]}   {X[i][1]}   |     {y_pred}     |   {y[i]}")
    if y_pred == y[i]:
        correct += 1

print(f"\nAccuracy: {correct}/{len(X)} = {(correct/len(X))*100:.0f}%")
```

```
# Task 2: Penguin Dataset with AdaDelta Optimizer

print("\n\n" + "="*60)
print("TASK 2: PENGUIN DATASET WITH ADADELTA OPTIMIZER")
print("="*60)

# Load penguin dataset from CSV
penguins = pd.read_csv('/media/smayan/500GB SSD/Study/ML2/Practicals/Data/penguins.csv')

# Preprocessing
penguins = penguins.dropna()
X_penguin = penguins[['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g
y_penguin = (penguins['species'] == 'Adelie').astype(int).values.reshape(-1, 1)

# Standardize
scaler_X = StandardScaler()
X_penguin = scaler_X.fit_transform(X_penguin)

print(f"\nDataset shape: X={X_penguin.shape}, y={y_penguin.shape}")
print(f"Classes: Adelie vs Others (Binary classification)")
```

```python
# Initialize Neural Network
input_size = X_penguin.shape[1]
hidden_size = 8
output_size = 1

np.random.seed(42)

W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size) * 0.01
b2 = np.zeros((1, output_size))

print("\nInitial Network Architecture:")
print(f"Input Layer: {input_size} neurons")
print(f"Hidden Layer: {hidden_size} neurons")
print(f"Output Layer: {output_size} neuron")

# Sigmoid activation
def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

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

# Forward pass
def forward_pass(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    cache = {'Z1': Z1, 'A1': A1, 'Z2': Z2, 'A2': A2}
    return A2, cache

# Calculate loss
def compute_loss(y_true, y_pred):
    m = y_true.shape[0]
    loss = -np.mean(y_true * np.log(y_pred + 1e-8) + (1 - y_true) * np.log(1 - y_pred + 1e
    return loss
```

```python
# Backward pass
def backward_pass(X, y, cache, W1, W2):
    m = X.shape[0]
    A1 = cache['A1']
    A2 = cache['A2']
    Z1 = cache['Z1']

    dZ2 = (A2 - y)
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * (1 - A1 ** 2)
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    return {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}
```

```python
# AdaDelta Optimizer Implementation
def adadelta_optimizer(X, y, W1, b1, W2, b2, epochs=100, rho=0.95, epsilon=1e-7):
    """
    AdaDelta Optimizer - Adaptive Learning Rate Method
    Accumulates squared gradients and parameter updates
    """
    m = X.shape[0]

    # Initialize accumulators
    E_dW1 = np.zeros_like(W1)
    E_dW2 = np.zeros_like(W2)
    E_db1 = np.zeros_like(b1)
    E_db2 = np.zeros_like(b2)

    E_ΔW1 = np.zeros_like(W1)
    E_ΔW2 = np.zeros_like(W2)
    E_Δb1 = np.zeros_like(b1)
    E_Δb2 = np.zeros_like(b2)

    loss_history = []
    accuracy_history = []

    print("\nTraining with AdaDelta Optimizer...")
    print(f"Rho (decay rate): {rho}, Epsilon: {epsilon}")
    print("-" * 60)

    for epoch in range(epochs):
        # Forward pass on full batch
        y_pred, cache = forward_pass(X, W1, b1, W2, b2)

        # Compute loss
        loss = compute_loss(y, y_pred)
        loss_history.append(loss)

        # Backward pass
        gradients = backward_pass(X, y, cache, W1, W2)

        # AdaDelta update for W1
        E_dW1 = rho * E_dW1 + (1 - rho) * (gradients['dW1'] ** 2)
```

```
        ΔW1 = -np.sqrt(E_ΔW1 + epsilon) / np.sqrt(E_dW1 + epsilon) * gradients['dW1']
        E_ΔW1 = rho * E_ΔW1 + (1 - rho) * (ΔW1 ** 2)
        W1 += ΔW1

        # AdaDelta update for b1
        E_db1 = rho * E_db1 + (1 - rho) * (gradients['db1'] ** 2)
        Δb1 = -np.sqrt(E_Δb1 + epsilon) / np.sqrt(E_db1 + epsilon) * gradients['db1']
        E_Δb1 = rho * E_Δb1 + (1 - rho) * (Δb1 ** 2)
        b1 += Δb1

        # AdaDelta update for W2
        E_dW2 = rho * E_dW2 + (1 - rho) * (gradients['dW2'] ** 2)
        ΔW2 = -np.sqrt(E_ΔW2 + epsilon) / np.sqrt(E_dW2 + epsilon) * gradients['dW2']
        E_ΔW2 = rho * E_ΔW2 + (1 - rho) * (ΔW2 ** 2)
        W2 += ΔW2

        # AdaDelta update for b2
        E_db2 = rho * E_db2 + (1 - rho) * (gradients['db2'] ** 2)
        Δb2 = -np.sqrt(E_Δb2 + epsilon) / np.sqrt(E_db2 + epsilon) * gradients['db2']
        E_Δb2 = rho * E_Δb2 + (1 - rho) * (Δb2 ** 2)
        b2 += Δb2

        # Calculate accuracy
        y_pred_binary = (y_pred > 0.5).astype(int)
        accuracy = np.mean(y_pred_binary == y) * 100
        accuracy_history.append(accuracy)

        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.6f}, Accuracy: {accuracy:.2f}%

    parameters = {'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
    return parameters, loss_history, accuracy_history

# Train with AdaDelta
trained_params, loss_history, accuracy_history = adadelta_optimizer(
    X_penguin, y_penguin, W1, b1, W2, b2,
    epochs=100, rho=0.95, epsilon=1e-7
)
```

```python
# Display updated weights and bias values
print("\n" + "="*60)
print("UPDATED WEIGHTS AND BIASES AFTER TRAINING")
print("="*60)

print("\nLayer 1 (Input -> Hidden):")
print(f"W1 shape: {trained_params['W1'].shape}")
print(f"W1:\n{trained_params['W1']}")
print(f"\nb1 shape: {trained_params['b1'].shape}")
print(f"b1:\n{trained_params['b1']}")

print("\nLayer 2 (Hidden -> Output):")
print(f"W2 shape: {trained_params['W2'].shape}")
print(f"W2:\n{trained_params['W2']}")
print(f"\nb2 shape: {trained_params['b2'].shape}")
print(f"b2:\n{trained_params['b2']}")

print("\n" + "="*60)
print(f"Initial Loss: {loss_history[0]:.6f}")
print(f"Final Loss: {loss_history[-1]:.6f}")
print(f"Initial Accuracy: {accuracy_history[0]:.2f}%")
print(f"Final Accuracy: {accuracy_history[-1]:.2f}%")
print("="*60)
```

```
# Plot Accuracy vs Epoch
plt.figure(figsize=(14, 5))

# Plot 1: Loss vs Epochs
plt.subplot(1, 2, 1)
plt.plot(range(1, len(loss_history) + 1), loss_history, linewidth=2, color='blue')
plt.title('Loss vs Epochs (AdaDelta)', fontsize=14, fontweight='bold')
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Binary Crossentropy Loss', fontsize=12)
plt.grid(True, alpha=0.3)

# Plot 2: Accuracy vs Epochs
plt.subplot(1, 2, 2)
plt.plot(range(1, len(accuracy_history) + 1), accuracy_history, linewidth=2, color='green
plt.title('Accuracy vs Epochs (AdaDelta)', fontsize=14, fontweight='bold')
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Accuracy (%)', fontsize=12)
plt.ylim([0, 105])
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nTraining Summary:")
print(f"Accuracy improved from {accuracy_history[0]:.2f}% to {accuracy_history[-1]:.2f}%"
print(f"Loss decreased from {loss_history[0]:.6f} to {loss_history[-1]:.6f}")
```

```
# Plot ROC AUC Curve
from sklearn.metrics import roc_curve, auc, roc_auc_score

# Get probability predictions on training data
y_pred_proba, _ = forward_pass(X_penguin, trained_params['W1'], trained_params['b1'],
                               trained_params['W2'], trained_params['b2'])

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_penguin, y_pred_proba)
roc_auc = auc(fpr, tpr)

# Plot ROC Curve
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.3f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Classifier')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', fontsize=12)
plt.ylabel('True Positive Rate', fontsize=12)
plt.title('ROC Curve - Penguin Classification (Adelie vs Others)', fontsize=14, fontweigh
plt.legend(loc="lower right", fontsize=11)
plt.grid(True, alpha=0.3)
plt.show()

print(f"\nROC AUC Score: {roc_auc:.4f}")
print(f"Model Performance: {'Excellent' if roc_auc >= 0.9 else 'Good' if roc_auc >= 0.8 e
```

Notebook: Q17.ipynb

Task 1

Implement backpropagation algorithm from scratch.

a) Take Iris Dataset

b) Initialize a neural network with random weights.

c) Calculate Squared Error (SE)

d) Perform multiple iterations.

e) Update weights accordingly.

f) Plot accuracy for iterations and note the results

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

# a) Take Iris Dataset
print("="*70)
print("TASK: BACKPROPAGATION ALGORITHM FROM SCRATCH")
print("="*70)

iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)

# Preprocessing
encoder = OneHotEncoder(sparse_output=False)
y_onehot = encoder.fit_transform(y)

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

print(f"\nDataset Shape:")
print(f"X: {X.shape} (samples, features)")
print(f"y (one-hot): {y_onehot.shape} (samples, classes)")
print(f"Classes: {iris.target_names}")

# b) Initialize Neural Network with Random Weights
input_size = X.shape[1]
hidden_size = 8
output_size = y_onehot.shape[1]

np.random.seed(42)

# Layer 1: Input -> Hidden
W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))

# Layer 2: Hidden -> Output
W2 = np.random.randn(hidden_size, output_size) * 0.01
```

```
b2 = np.zeros((1, output_size))

print(f"\nNetwork Architecture:")
print(f"Input Layer: {input_size} neurons")
print(f"Hidden Layer: {hidden_size} neurons (ReLU)")
print(f"Output Layer: {output_size} neurons (Softmax)")

# Activation Functions
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return (z > 0).astype(float)

def softmax(z):
    z = z - np.max(z, axis=1, keepdims=True)  # Numerical stability
    return np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)

# Forward Propagation
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softmax(Z2)

    cache = {'X': X, 'Z1': Z1, 'A1': A1, 'Z2': Z2, 'A2': A2}
    return A2, cache

# c) Calculate Squared Error (SE)
def calculate_se_loss(y_true, y_pred):
    m = y_true.shape[0]
    se_loss = np.sum((y_true - y_pred) ** 2) / (2 * m)
    return se_loss

# Categorical Crossentropy (for better classification)
def calculate_ce_loss(y_true, y_pred):
    m = y_true.shape[0]
    ce_loss = -np.sum(y_true * np.log(y_pred + 1e-8)) / m
    return ce_loss
```

```python
# Backward Propagation (Backpropagation Algorithm)
def backward_propagation(cache, y_true, W1, W2):
    m = cache['X'].shape[0]

    A1 = cache['A1']
    A2 = cache['A2']
    Z1 = cache['Z1']
    X = cache['X']

    # Output layer gradient
    dZ2 = A2 - y_true
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)

    # Hidden layer gradient (backpropagating error)
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    gradients = {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}
    return gradients

print("\nForward and Backward Propagation functions defined.")
```

```python
# d) & e) Training Loop: Multiple Iterations with Weight Updates
def train_with_backpropagation(X, y_true, W1, b1, W2, b2, epochs=200, learning_rate=0.01)
    """
    d) Perform multiple iterations
    e) Update weights accordingly using backpropagation
    """
    loss_history = []
    se_loss_history = []
    accuracy_history = []

    print("\n" + "="*70)
    print("TRAINING WITH BACKPROPAGATION")
    print("="*70)
    print(f"Epochs: {epochs}, Learning Rate: {learning_rate}")
    print("-"*70)

    for epoch in range(epochs):
        # Forward pass
        y_pred, cache = forward_propagation(X, W1, b1, W2, b2)

        # Calculate losses
        ce_loss = calculate_ce_loss(y_true, y_pred)
        se_loss = calculate_se_loss(y_true, y_pred)
        loss_history.append(ce_loss)
        se_loss_history.append(se_loss)

        # Calculate accuracy
        y_pred_class = np.argmax(y_pred, axis=1)
        y_true_class = np.argmax(y_true, axis=1)
        accuracy = np.mean(y_pred_class == y_true_class) * 100
        accuracy_history.append(accuracy)

        # Backward propagation
        gradients = backward_propagation(cache, y_true, W1, W2)

        # Update weights and biases (e)
        W1 = W1 - learning_rate * gradients['dW1']
        b1 = b1 - learning_rate * gradients['db1']
        W2 = W2 - learning_rate * gradients['dW2']
```

```
        b2 = b2 - learning_rate * gradients['db2']

        # Print progress
        if (epoch + 1) % 20 == 0:
            print(f"Epoch {epoch+1:3d}/{epochs} | CE Loss: {ce_loss:.6f} | SE Loss: {se_l

    parameters = {'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
    return parameters, loss_history, se_loss_history, accuracy_history

# Train the network
parameters, loss_history, se_loss_history, accuracy_history = train_with_backpropagation(
    X, y_onehot, W1, b1, W2, b2,
    epochs=200,
    learning_rate=0.01
)
```

```python
# Display Training Results
print("\n" + "="*70)
print("TRAINING RESULTS")
print("="*70)

print(f"\nFinal Metrics:")
print(f"Initial Accuracy: {accuracy_history[0]:.2f}%")
print(f"Final Accuracy: {accuracy_history[-1]:.2f}%")
print(f"Accuracy Improvement: {accuracy_history[-1] - accuracy_history[0]:.2f}%")

print(f"\nInitial CE Loss: {loss_history[0]:.6f}")
print(f"Final CE Loss: {loss_history[-1]:.6f}")
print(f"Loss Reduction: {loss_history[0] - loss_history[-1]:.6f}")

print(f"\nInitial SE Loss: {se_loss_history[0]:.6f}")
print(f"Final SE Loss: {se_loss_history[-1]:.6f}")

# Final evaluation on training set
y_pred_final, _ = forward_propagation(X, parameters['W1'], parameters['b1'], parameters['W
y_pred_class = np.argmax(y_pred_final, axis=1)
y_true_class = np.argmax(y_onehot, axis=1)
final_accuracy = np.mean(y_pred_class == y_true_class) * 100

print(f"\nFinal Training Accuracy: {final_accuracy:.2f}%")
print(f"Correctly Classified: {np.sum(y_pred_class == y_true_class)}/{len(y_true_class)}"
```

```
# f) Plot Accuracy for Iterations
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Plot 1: Accuracy vs Epochs
axes[0, 0].plot(range(1, len(accuracy_history) + 1), accuracy_history, linewidth=2, color=
axes[0, 0].set_title('Accuracy vs Epochs', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Accuracy (%)')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].set_ylim([0, 105])

# Plot 2: Categorical Crossentropy Loss vs Epochs
axes[0, 1].plot(range(1, len(loss_history) + 1), loss_history, linewidth=2, color='blue')
axes[0, 1].set_title('Categorical Crossentropy Loss vs Epochs', fontsize=12, fontweight='
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Loss')
axes[0, 1].grid(True, alpha=0.3)

# Plot 3: Squared Error (SE) Loss vs Epochs
axes[1, 0].plot(range(1, len(se_loss_history) + 1), se_loss_history, linewidth=2, color='
axes[1, 0].set_title('Squared Error (SE) Loss vs Epochs', fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('SE Loss')
axes[1, 0].grid(True, alpha=0.3)

# Plot 4: Accuracy with Log Scale
axes[1, 1].semilogy(range(1, len(accuracy_history) + 1),
                    100 - np.array(accuracy_history) + 0.01,
                    linewidth=2, color='purple')
axes[1, 1].set_title('Classification Error vs Epochs (Log Scale)', fontsize=12, fontweigh
axes[1, 1].set_xlabel('Epoch')
axes[1, 1].set_ylabel('Error (%)')
axes[1, 1].grid(True, alpha=0.3, which='both')

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("NOTES AND OBSERVATIONS")
```

```
print("="*70)
print(f"""
1. Backpropagation Algorithm:
   - Forward pass: Compute predictions through network layers
   - Calculate loss (SE or CE)
   - Backward pass: Propagate errors back through layers
   - Gradient computation for each layer
   - Weight updates using gradients

2. Training Performance:
   - Accuracy improved from {accuracy_history[0]:.2f}% to {accuracy_history[-1]:.2f}%
   - Loss decreased from {loss_history[0]:.6f} to {loss_history[-1]:.6f}
   - SE Loss decreased from {se_loss_history[0]:.6f} to {se_loss_history[-1]:.6f}

3. Convergence:
   - Network converged well with stable accuracy improvement
   - ReLU activation in hidden layer helps with deep learning
   - Softmax ensures valid probability distribution for 3 classes

4. Key Components:
   - Forward Propagation: Z = W·X + b, A = activation(Z)
   - Backward Propagation: dW = (1/m) · X^T · dZ
   - Weight Update: W_new = W - learning_rate · dW
""")
```

Notebook: Q18.ipynb

Build a multiclass image categorization CNN network which correctly classifies different categories of images in the dataset.(handwritten digits from Mnist digit dataset

- Split original dataset to train and test set
- Build CNN Model
- Generate the accuracy of the built model using Adam Optimizer and Adagrad Optimizer.
- Compare performance of different optimizer on Digit categorization.
- Plot training vs validation accuracy ⮚ Evaluate the model using confusion matrix, precision, recall.

```
from tensorflow.keras.datasets import mnist
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

```
data = mnist.load_data()
(train_images, train_labels), (test_images, test_labels) = data
```

```
img_size =  train_images[0].shape[0]
```

```
num_classes = 10
```

```
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0
```

```
plt.imshow(train_images[0], cmap='gray')
plt.title(f'Label: {train_labels[0]}')
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D, MaxPooling2D
```

```
from tensorflow.keras.optimizers import SGD, Adam, Adagrad, RMSprop
SGD_nomomentum = SGD()
SGD_momentum = SGD(momentum=0.6)
AdaGrad_momentum = Adagrad()
RMSprop_momentum = RMSprop()
```

```
adam_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape =(img_size,img_size, 1))
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

adam_model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',metrics=[
adam_history = adam_model.fit(train_images,train_labels, epochs = 15)
```

```
SGD_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape =(img_size,img_size, 1))
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

SGD_model.compile(optimizer =SGD_nomomentum, loss = 'sparse_categorical_crossentropy',met
SGD_history = SGD_model.fit(train_images,train_labels, epochs = 15)
```

```
SGD_momentum_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape =(img_size,img_size, 1))
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

SGD_momentum_model.compile(optimizer =SGD_momentum, loss = 'sparse_categorical_crossentrop
SGD_momentum_history = SGD_momentum_model.fit(train_images,train_labels, epochs = 15)
```

```
AdaGrad_momentum_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape =(img_size,img_size, 1))
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

AdaGrad_momentum_model.compile(optimizer =AdaGrad_momentum, loss = 'sparse_categorical_cr
AdaGrad_momentum_history = AdaGrad_momentum_model.fit(train_images,train_labels, epochs =
```

```
RmsProp_momentum_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape =(img_size,img_size, 1))
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(num_classes, activation='softmax')
])

RmsProp_momentum_model.compile(optimizer =RMSprop_momentum, loss = 'sparse_categorical_cr
RmsProp_momentum_history = RmsProp_momentum_model.fit(train_images,train_labels, epochs =
```

```
adam_losses = adam_history.history['loss']
SGD_losses = SGD_history.history['loss']
SGD_momentum_losses = SGD_momentum_history.history['loss']
AdaGrad_momentum_losses = AdaGrad_momentum_history.history['loss']
RMsProp_momentum_losses = RmsProp_momentum_history.history['loss']
```

```
x = np.arange(1, 16)
```

```
plt.plot(x, adam_losses, label='Adam')
plt.plot(x, SGD_losses, label='SGD')
plt.plot(x, SGD_momentum_losses, label='SGD with Momentum')
plt.plot(x, AdaGrad_momentum_losses, label='AdaGrad')
plt.plot(x, RMsProp_momentum_losses, label='RMSProp')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title("Optimizer Comparison: Epochs vs Loss")
plt.grid()
```

```
adam_acc = adam_history.history['accuracy']
SGD_acc = SGD_history.history['accuracy']
SGD_momentum_acc = SGD_momentum_history.history['accuracy']
AdamGrad_momentum_acc = AdaGrad_momentum_history.history['accuracy']
RmsProp_momentum_acc = RmsProp_momentum_history.history['accuracy']
```

```
plt.plot(x, adam_acc, label='Adam')
plt.plot(x, SGD_acc, label='SGD')
plt.plot(x, SGD_momentum_acc, label='SGD with Momentum')
plt.plot(x, AdamGrad_momentum_acc, label='AdaGrad')
plt.plot(x, RmsProp_momentum_acc, label='RMSProp')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title("Optimizer Comparison: Epochs vs Accuracy")
plt.grid()
```

```
```

Notebook: Q19.ipynb

- Use alpaca dataset
- CNN must include : Convolution layer, Pooling layer, Flatten layer,Dense layer Plot:
- Accuracy vs Epochs
- Loss (Error) vs Epochs

```
import numpy as np
import pandas as pd
import tensorflow as tf
```

```
from tensorflow.keras.preprocessing import image_dataset_from_directory
```

```
train_dataset = image_dataset_from_directory(
    'Data/alpaca_dataset',
    image_size=(224, 224),
    batch_size=32,
    label_mode='int'
)

classes = train_dataset.class_names
print("Classes:", classes)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import Adam
```

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(2, activation='softmax')
])
```

```
model.compile(optimizer=Adam(), loss='sparse_categorical_crossentropy', metrics=['accuracy
```

```
history = model.fit(train_dataset, epochs=10)
```

```
losses = history.history['loss']
accuracies = history.history['accuracy']
```

```
x = np.arange(1, len(losses) + 1)
```

```
import matplotlib.pyplot as plt
plt.plot(x, losses, label='Loss', marker='o')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.legend()
plt.grid()
plt.show()
```

```
plt.plot(x, accuracies, label='Accuracy', marker='o')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.legend()
plt.grid()
plt.show()
```

```
```

Notebook: Q20.ipynb

1. Load the Corn 3-Classes image dataset.
2. Preprocess the images:

   a. Resize images to a fixed size (e.g., 224×224)
   b. Normalize pixel values.
3. Split the dataset into training and testing sets.
4. Create a CNN model using:
   a. Convolution layer
   b. Max Pooling layer
   c. Flatten layer
   d. Dense layer
5. Train the CNN model for multi-class classification.
6. Test the model on unseen images.
7. Plot graphs:
   a. Training vs Validation Accuracy
   b. Training vs Validation Loss (Error)

```
import numpy as np
import pandas as pd
import tensorflow as tf
```

```
from tensorflow.keras.preprocessing import image_dataset_from_directory
```

```
train_dataset = image_dataset_from_directory(
    'Data/Corn_3_Classes_Image_Dataset',
    image_size=(224, 224),
    batch_size=32,
    label_mode='int',
    validation_split=0.2,
    subset='training',
    seed=123
)

val_dataset = image_dataset_from_directory(
    'Data/Corn_3_Classes_Image_Dataset',
    image_size=(224, 224),
    batch_size=32,
    label_mode='int',
    validation_split=0.2,
    subset='validation',
    seed=123
)

classes = train_dataset.class_names
print("Classes:", classes)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten , BatchNormaliza
from tensorflow.keras.optimizers import Adam
```

```
model = Sequential([
    tf.keras.layers.Input(shape=(224, 224, 3)),
    tf.keras.layers.Rescaling(1./255),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(len(classes), activation='softmax')
])
```

```
model.compile(optimizer=Adam(), loss='sparse_categorical_crossentropy', metrics=['accuracy
```

```
history = model.fit(train_dataset, validation_data=val_dataset, epochs=10)
```

```
train_losses = history.history['loss']
train_accuracies = history.history['accuracy']

val_losses = history.history['val_loss']
val_accuracies = history.history['val_accuracy']
```

```
x = np.arange(1, len(train_losses) + 1)
```

```
import matplotlib.pyplot as plt
plt.plot(x, train_losses, label='Train Loss', marker='o')
plt.plot(x, val_losses, label='Validation Loss', marker='o')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.legend()
plt.grid()
plt.show()
```

```
plt.plot(x, train_accuracies, label='Accuracy', marker='o')
plt.plot(x, val_accuracies, label='Validation Accuracy', marker='o')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.legend()
plt.grid()
plt.show()
```

```
```

Notebook: Q21.ipynb

Task 1

Implement the NAND Boolean Logic Gate using a Perceptron Neural Network.

- Inputs: x1, x2, bias
- Train using perceptron learning rule
- Output: y
- Display final weights and bias
- Verify truth table results

```
import numpy as np
import tensorflow as tf
import pandas as pd
```

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

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

```
weights = np.random.rand(2,1)
bias = np.random.rand(1)
```

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

```
def perceptron(X, weights, bias):
    z = np.dot(X,weights) + bias
    a = sigmoid(z)
    return a
```

```
w = np.random.rand(2)
b = np.random.rand(1)

epochs = 50
learning_rate = 0.1

for epoch in range(epochs):
    for i in range(X.shape[0]):
        y_pred = perceptron(X[i], w, b)
        error = y[i] - y_pred
        w += learning_rate * error * X[i]
        b += learning_rate * error

print("Trained weights:", w)
print("Trained bias:", b)
```

```
def step(x):
    return 1 if x >= 0.5 else 0
```

```
print("\nPredictions:")
for i in range(len(X)):
    z = np.dot(w, X[i]) + b
    print(f"Input: {X[i]} → Output:", step(sigmoid(z)))
```

```
import matplotlib.pyplot as plt
for i in range(len(X)):
    plt.scatter(X[i][0], X[i][1], c='r' if y[i] == 1 else 'b')
plt.grid()
```

Task 2

Use the Iris Dataset

- Normalize the input features
- Perform Min–Max scaling
- Visualize original vs normalized features

Load Iris Dataset

```
# Standardization (Normalization) - mean=0, std=1
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Load Iris dataset
iris = load_iris()
X_original = pd.DataFrame(iris.data, columns=iris.feature_names)

print("Original Iris Dataset:")
print(X_original.head())
print(f"\nShape: {X_original.shape}")
print(f"\nStatistics:\n{X_original.describe()}")

print("\n" + "="*70 + "\n")

scaler_standard = StandardScaler()
X_normalized = pd.DataFrame(
    scaler_standard.fit_transform(X_original),
    columns=X_original.columns
)

print("Normalized Dataset (Standardization):")
print(X_normalized.head())
print(f"\nStatistics:\n{X_normalized.describe()}")
```

```
# Min-Max Scaling - scale to [0, 1] range
scaler_minmax = MinMaxScaler()
X_minmax = pd.DataFrame(
    scaler_minmax.fit_transform(X_original),
    columns=X_original.columns
)

print("Min-Max Scaled Dataset:")
print(X_minmax.head())
print(f"\nStatistics:\n{X_minmax.describe()}")
```

Visualize Original vs Normalized Features

```
# Visualize distributions for all features
fig, axes = plt.subplots(4, 3, figsize=(15, 12))
fig.suptitle('Comparison: Original vs Normalized vs Min-Max Scaled Features', fontsize=16

for i, feature in enumerate(X_original.columns):
    # Original
    sns.histplot(X_original[feature], bins=20, color='blue', alpha=0.7, edgecolor='black'
    axes[i, 0].set_title(f'{feature}\n(Original)')
    axes[i, 0].set_ylabel('Frequency')

    # Normalized (Standardized)
    sns.histplot(X_normalized[feature], bins=20, color='green', alpha=0.7, edgecolor='bla
    axes[i, 1].set_title(f'{feature}\n(Normalized)')

    # Min-Max Scaled
    sns.histplot(X_minmax[feature], bins=20, color='red', alpha=0.7, edgecolor='black', ax
    axes[i, 2].set_title(f'{feature}\n(Min-Max Scaled)')

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

```
# Box plots comparison
fig, axes = plt.subplots(1, 3, figsize=(18, 5))
import seaborn as sns

# Original
sns.boxplot(data=X_original, ax=axes[0])
axes[0].set_ylabel('Value')
axes[0].grid(True, alpha=0.3)

# Normalized
sns.boxplot(data=X_normalized, ax=axes[1])
axes[1].set_title('Normalized Features (Standardized)', fontsize=14)
axes[1].set_ylabel('Value')
axes[1].grid(True, alpha=0.3)

# Min-Max Scaled
sns.boxplot(data=X_minmax, ax=axes[2])
axes[2].set_title('Min-Max Scaled Features', fontsize=14)
axes[2].set_ylabel('Value')
axes[2].grid(True, alpha=0.3)

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

```
# Summary comparison
print("="*70)
print("COMPARISON SUMMARY")
print("="*70)

print("\n1. ORIGINAL DATA:")
print(f"   Range: [{X_original.min().min():.2f}, {X_original.max().max():.2f}]")
print(f"   Mean: {X_original.mean().mean():.2f}")
print(f"   Std: {X_original.std().mean():.2f}")

print("\n2. NORMALIZED (STANDARDIZED):")
print(f"   Range: [{X_normalized.min().min():.2f}, {X_normalized.max().max():.2f}]")
print(f"   Mean: {X_normalized.mean().mean():.2f}")
print(f"   Std: {X_normalized.std().mean():.2f}")

print("\n3. MIN-MAX SCALED:")
print(f"   Range: [{X_minmax.min().min():.2f}, {X_minmax.max().max():.2f}]")
print(f"   Mean: {X_minmax.mean().mean():.2f}")
print(f"   Std: {X_minmax.std().mean():.2f}")

print("\n" + "="*70)
```

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Load Iris dataset
iris = load_iris()
X_original = pd.DataFrame(iris.data, columns=iris.feature_names)

print("Original Iris Dataset:")
print(X_original.head())
print(f"\nShape: {X_original.shape}")
print(f"\nStatistics:\n{X_original.describe()}")
```

Notebook: Q22.ipynb

Task 1 Implement Multi-output Perceptron for

- AND gate
- OR gate
- Display weight matrix and bias vector

Task 2 Load Flowers Dataset

- Train CNN model with 3 kernals.
- Plot training and validation accuracy using graph.

Task 1: Multi-output Perceptron for AND & OR Gates

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.models import Sequential
```

Prepare input data for AND and OR gates

```
# Input data (same for both gates)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Output for AND gate
y_and = np.array([0, 0, 0, 1])

# Output for OR gate
y_or = np.array([0, 1, 1, 1])

# Combined output (multi-output)
y_combined = np.column_stack([y_and, y_or])

print("Input X:")
print(X)
print("\nAND Output:", y_and)
print("OR Output:", y_or)
print("\nCombined Output (AND, OR):")
print(y_combined)
```

Define step function

```
def step(z):
    """Step activation function"""
    return 1 if z >= 1 else 0
```

Train multi-output perceptron

```
# Initialize weight matrix and bias vector
# Weight matrix: 2 inputs x 2 outputs (AND, OR)
W = np.random.rand(2, 2)
b = np.random.rand(2)

epochs = 50
learning_rate = 0.1

print("Initial Weight Matrix:")
print(W)
print("\nInitial Bias Vector:")
print(b)
print("\n" + "="*50)

# Training loop
for epoch in range(epochs):
    for i in range(X.shape[0]):
        # Forward pass for both outputs
        z = np.dot(X[i], W) + b
        y_pred = np.array([step(z[0]), step(z[1])])

        # Calculate error for both outputs
        error = y_combined[i] - y_pred

        # Update weights and biases
        W += learning_rate * np.outer(X[i], error)
        b += learning_rate * error

print("\nFinal Weight Matrix:")
print(W)
print("\nFinal Bias Vector:")
print(b)
```

Test the perceptron

```
print("\n" + "="*50)
print("PREDICTIONS:")
print("="*50)
print("Input\t\tAND\tOR")
print("-"*50)

for i in range(len(X)):
    z = np.dot(W.T, X[i]) + b
    and_out = step(z[0])
    or_out = step(z[1])
    print(f"{X[i]}\t\t{and_out}\t{or_out}")

print("="*50)
```

Task 2: CNN Model for Flowers Dataset

Load and prepare Flowers dataset

```
# Load the tf_flowers dataset
import tensorflow_datasets as tfds
z
# Load the flowers dataset
(train_ds, val_ds), info = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:]'],
    as_supervised=True,
    with_info=True
)

# Get dataset info
num_classes = info.features['label'].num_classes
print(f"Number of classes: {num_classes}")
print(f"Class names: {info.features['label'].names}")
```

Preprocess images

```
IMG_SIZE = 128
BATCH_SIZE = 32

def preprocess(image, label):
    """Resize and normalize images"""
    image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
    image = image / 255.0  # Normalize to [0,1]
    return image, label

# Apply preprocessing
train_ds = train_ds.map(preprocess).shuffle(1000).batch(BATCH_SIZE).prefetch(2)
val_ds = val_ds.map(preprocess).batch(BATCH_SIZE).prefetch(2)

print("Dataset prepared successfully!")
```

Build CNN model with 3 convolutional layers (kernels)

```
model = Sequential([
    Input(shape=(IMG_SIZE, IMG_SIZE, 3)),

    # First Convolutional Layer (Kernel 1)
    Conv2D(32, 3, padding='same', activation='relu'),
    MaxPooling2D(2),

    # Second Convolutional Layer (Kernel 2)
    Conv2D(64, 3, padding='same', activation='relu'),
    MaxPooling2D(2),

    # Third Convolutional Layer (Kernel 3)
    Conv2D(128, 3, padding='same', activation='relu'),
    MaxPooling2D(2),

    # Flatten and Dense layers
    Flatten(),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])

model.summary()
```

Compile and train the model

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Train the model
EPOCHS = 10

history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS,
    verbose=1
)

print("\nTraining completed!")
```

Plot training and validation accuracy

```
plt.figure(figsize=(12, 5))

# Plot accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy', marker='o')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker='s')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss', marker='o')
plt.plot(history.history['val_loss'], label='Validation Loss', marker='s')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Print final metrics
print("\n" + "="*50)
print("FINAL METRICS:")
print("="*50)
print(f"Training Accuracy: {history.history['accuracy'][-1]:.4f}")
print(f"Validation Accuracy: {history.history['val_accuracy'][-1]:.4f}")
print(f"Training Loss: {history.history['loss'][-1]:.4f}")
print(f"Validation Loss: {history.history['val_loss'][-1]:.4f}")
print("="*50)
```

Notebook: Q23.ipynb

Implement XOR gate using 2-layer Neural Network

- Use Adadelta optimizer
- Plot accuracy vs epoch

```
import numpy as np
import tensorflow as tf
import pandas as pd
```

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

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

```
w1 = np.random.rand(2,2)
b1 = np.random.rand(1)
w2 = np.random.rand(1,1)
b2 = np.random.rand(1)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```
model = Sequential([
    Input(shape=(2,)),
    Dense(2, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

```
model.compile(optimizer='adadelta', loss='binary_crossentropy', metrics=['accuracy'])
```

```
history = model.fit(X, y, epochs=100)
```

```python
import matplotlib.pyplot as plt

# Plot accuracy vs epoch
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], marker='o')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], marker='o', color='orange')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)

plt.tight_layout()
plt.show()

# Test predictions
print("\n" + "="*40)
print("XOR Gate Predictions:")
print("="*40)
predictions = model.predict(X)
for i in range(len(X)):
    predicted = 1 if predictions[i] > 0.5 else 0
    print(f"Input: {X[i]} → Predicted: {predicted}, Actual: {y[i]}")
print("="*40)
```

Use dataset with initial values X = [1.0, 2.0], Y = [0.5, 1.5]

- Initialize neural network with random weights
- Compute output using linear activation
- Calculate MAE and MSE

- Plot loss surface (weight vs loss)

Initialize dataset and neural network

```
# Dataset
X_data = np.array([1.0, 2.0])
Y_data = np.array([0.5, 1.5])

# Initialize random weights and bias
np.random.seed(42)
w = np.random.randn()
b = np.random.randn()

print("Dataset:")
print(f"X = {X_data}")
print(f"Y = {Y_data}")
print(f"\nInitial weights:")
print(f"w = {w:.4f}")
print(f"b = {b:.4f}")
```

Compute output using linear activation

```
# Linear activation: y = w*x + b
Y_pred = w * X_data + b

print("Predictions (Linear Activation):")
print(f"Y_pred = {Y_pred}")
print(f"\nActual:")
print(f"Y = {Y_data}")
```

Calculate MAE and MSE

```
# Calculate errors
errors = Y_data - Y_pred

# Mean Absolute Error (MAE)
mae = np.mean(np.abs(errors))

# Mean Squared Error (MSE)
mse = np.mean(errors ** 2)

print("Error Metrics:")
print("="*40)
print(f"MAE (Mean Absolute Error): {mae:.4f}")
print(f"MSE (Mean Squared Error): {mse:.4f}")
print(f"RMSE (Root Mean Squared Error): {np.sqrt(mse):.4f}")
print("="*40)
```

Plot loss surface (weight vs loss)

```python
# Create a range of weight values
weights = np.linspace(-2, 2, 100)

# Calculate MSE for each weight value (keeping bias constant)
mse_values = []
for weight in weights:
    predictions = weight * X_data + b
    mse_val = np.mean((Y_data - predictions) ** 2)
    mse_values.append(mse_val)

# Plot loss surface
plt.figure(figsize=(12, 5))

# Plot 1: MSE vs Weight
plt.subplot(1, 2, 1)
plt.plot(weights, mse_values, linewidth=2)
plt.axvline(w, color='red', linestyle='--', label=f'Current w={w:.4f}')
plt.scatter([w], [mse], color='red', s=100, zorder=5, label=f'Current MSE={mse:.4f}')
plt.xlabel('Weight (w)')
plt.ylabel('MSE Loss')
plt.title('Loss Surface: MSE vs Weight')
plt.legend()
plt.grid(True, alpha=0.3)

# Plot 2: MAE vs Weight
mae_values = []
for weight in weights:
    predictions = weight * X_data + b
    mae_val = np.mean(np.abs(Y_data - predictions))
    mae_values.append(mae_val)

plt.subplot(1, 2, 2)
plt.plot(weights, mae_values, linewidth=2, color='orange')
plt.axvline(w, color='red', linestyle='--', label=f'Current w={w:.4f}')
plt.scatter([w], [mae], color='red', s=100, zorder=5, label=f'Current MAE={mae:.4f}')
plt.xlabel('Weight (w)')
plt.ylabel('MAE Loss')
plt.title('Loss Surface: MAE vs Weight')
plt.legend()
```

```
plt.grid(True, alpha=0.3)

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

3D Loss Surface (Weight vs Bias vs Loss)

```
# Create mesh grid for weight and bias
w_range = np.linspace(-2, 2, 50)
b_range = np.linspace(-2, 2, 50)
W_grid, B_grid = np.meshgrid(w_range, b_range)

# Calculate MSE for each combination
MSE_grid = np.zeros_like(W_grid)
for i in range(W_grid.shape[0]):
    for j in range(W_grid.shape[1]):
        predictions = W_grid[i, j] * X_data + B_grid[i, j]
        MSE_grid[i, j] = np.mean((Y_data - predictions) ** 2)

# Create 3D surface plot
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(14, 5))

# 3D Surface
ax1 = fig.add_subplot(121, projection='3d')
surf = ax1.plot_surface(W_grid, B_grid, MSE_grid, cmap='viridis', alpha=0.8)
ax1.scatter([w], [b], [mse], color='red', s=100, label='Current point')
ax1.set_xlabel('Weight (w)')
ax1.set_ylabel('Bias (b)')
ax1.set_zlabel('MSE Loss')
ax1.set_title('3D Loss Surface')
fig.colorbar(surf, ax=ax1, shrink=0.5)

# Contour plot
ax2 = fig.add_subplot(122)
contour = ax2.contour(W_grid, B_grid, MSE_grid, levels=20, cmap='viridis')
ax2.scatter([w], [b], color='red', s=100, zorder=5, label='Current point')
ax2.set_xlabel('Weight (w)')
ax2.set_ylabel('Bias (b)')
ax2.set_title('Loss Contour Plot')
ax2.legend()
fig.colorbar(contour, ax=ax2)

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

```
print(f"\nCurrent position: w={w:.4f}, b={b:.4f}, MSE={mse:.4f}")
```

Fashion-MNIST Classification

- CNN with RMSProp & Adam
- Compare confusion matrices

```
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import RMSprop, Adam
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# Load Fashion-MNIST dataset
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Class names for Fashion-MNIST
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

print(f"Training set shape: {train_images.shape}")
print(f"Test set shape: {test_images.shape}")
print(f"Number of classes: {len(class_names)}")
```

Preprocess data

```
# Normalize pixel values to [0, 1]
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0

# Get image dimensions
img_size = train_images.shape[1]
num_classes = len(class_names)

# Display sample images
plt.figure(figsize=(10, 4))
for i in range(5):
    plt.subplot(1, 5, i+1)
    plt.imshow(train_images[i], cmap='gray')
    plt.title(f'{class_names[train_labels[i]]}')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Build and train CNN with Adam optimizer

```
adam_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape=(img_size, img_size, 1)),
    MaxPooling2D(pool_size=(2,2)),
    Conv2D(64, kernel_size=(3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])

adam_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['ac
print("Training with Adam optimizer...")
adam_history = adam_model.fit(train_images, train_labels, epochs=10, validation_split=0.2
```

Build and train CNN with RMSProp optimizer

```
rmsprop_model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape=(img_size, img_size, 1)),
    MaxPooling2D(pool_size=(2,2)),
    Conv2D(64, kernel_size=(3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])

rmsprop_model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics
print("Training with RMSProp optimizer...")
rmsprop_history = rmsprop_model.fit(train_images, train_labels, epochs=10, validation_spli
```

Compare training performance

```
# Extract history
adam_acc = adam_history.history['accuracy']
adam_val_acc = adam_history.history['val_accuracy']
adam_loss = adam_history.history['loss']
adam_val_loss = adam_history.history['val_loss']

rmsprop_acc = rmsprop_history.history['accuracy']
rmsprop_val_acc = rmsprop_history.history['val_accuracy']
rmsprop_loss = rmsprop_history.history['loss']
rmsprop_val_loss = rmsprop_history.history['val_loss']

epochs = range(1, len(adam_acc) + 1)

# Plot comparison
plt.figure(figsize=(14, 5))

# Accuracy comparison
plt.subplot(1, 2, 1)
plt.plot(epochs, adam_acc, 'b-', label='Adam Training')
plt.plot(epochs, adam_val_acc, 'b--', label='Adam Validation')
plt.plot(epochs, rmsprop_acc, 'r-', label='RMSProp Training')
plt.plot(epochs, rmsprop_val_acc, 'r--', label='RMSProp Validation')
plt.title('Training vs Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Loss comparison
plt.subplot(1, 2, 2)
plt.plot(epochs, adam_loss, 'b-', label='Adam Training')
plt.plot(epochs, adam_val_loss, 'b--', label='Adam Validation')
plt.plot(epochs, rmsprop_loss, 'r-', label='RMSProp Training')
plt.plot(epochs, rmsprop_val_loss, 'r--', label='RMSProp Validation')
plt.title('Training vs Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
```

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

Evaluate models on test set

```
# Evaluate Adam model
adam_test_loss, adam_test_acc = adam_model.evaluate(test_images, test_labels, verbose=0)
print("Adam Model:")
print(f"  Test Accuracy: {adam_test_acc:.4f}")
print(f"  Test Loss: {adam_test_loss:.4f}")

# Evaluate RMSProp model
rmsprop_test_loss, rmsprop_test_acc = rmsprop_model.evaluate(test_images, test_labels, ve
print("\nRMSProp Model:")
print(f"  Test Accuracy: {rmsprop_test_acc:.4f}")
print(f"  Test Loss: {rmsprop_test_loss:.4f}")
```

Generate predictions and confusion matrices

```
# Get predictions
adam_predictions = np.argmax(adam_model.predict(test_images), axis=1)
rmsprop_predictions = np.argmax(rmsprop_model.predict(test_images), axis=1)

# Generate confusion matrices
adam_cm = confusion_matrix(test_labels, adam_predictions)
rmsprop_cm = confusion_matrix(test_labels, rmsprop_predictions)

print("Predictions generated successfully!")
```

Compare confusion matrices

```
# Plot confusion matrices side by side
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Adam confusion matrix
sns.heatmap(adam_cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names, ax=axes[0])
axes[0].set_title(f'Adam Optimizer\nTest Accuracy: {adam_test_acc:.4f}')
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('Actual')

# RMSProp confusion matrix
sns.heatmap(rmsprop_cm, annot=True, fmt='d', cmap='Greens',
            xticklabels=class_names, yticklabels=class_names, ax=axes[1])
axes[1].set_title(f'RMSProp Optimizer\nTest Accuracy: {rmsprop_test_acc:.4f}')
axes[1].set_xlabel('Predicted')
axes[1].set_ylabel('Actual')

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

Classification reports

```
print("="*70)
print("ADAM OPTIMIZER - CLASSIFICATION REPORT")
print("="*70)
print(classification_report(test_labels, adam_predictions, target_names=class_names))

print("\n" + "="*70)
print("RMSPROP OPTIMIZER - CLASSIFICATION REPORT")
print("="*70)
print(classification_report(test_labels, rmsprop_predictions, target_names=class_names))

# Summary comparison
print("\n" + "="*70)
print("OPTIMIZER COMPARISON SUMMARY")
print("="*70)
print(f"Adam    - Test Accuracy: {adam_test_acc:.4f}, Test Loss: {adam_test_loss:.4f}")
print(f"RMSProp - Test Accuracy: {rmsprop_test_acc:.4f}, Test Loss: {rmsprop_test_loss:.4
print("="*70)
```

Notebook: Q24.ipynb

Implement the backpropagation algorithm.

1. Take Iris Dataset
2. Initialize a neural network with random weights.
3. Calculate error
4. Perform multiple iterations of NN
5. Update weights accordingly.
6. Plot accuracy for iterations and note the results.

Step 1: Import libraries and load Iris dataset

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

# Load Iris dataset
iris = load_iris()
X = iris.data  # Features (4 dimensions)
y = iris.target  # Labels (3 classes)

print(f"Dataset shape: {X.shape}")
print(f"Number of features: {X.shape[1]}")
print(f"Number of samples: {X.shape[0]}")
print(f"Number of classes: {len(np.unique(y))}")
```

Step 2: Preprocess data (normalize and split)

```
from sklearn.preprocessing import OneHotEncoder

# Normalize features to have mean 0 and std 1
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# Convert labels to one-hot encoding (for 3 classes)
one_hot_encoder = OneHotEncoder(sparse_output=False)
def one_hot_encode(y, num_classes=3):
    """Convert class labels to one-hot encoding"""
    return one_hot_encoder.fit_transform(y.reshape(-1, 1))

y_train_encoded = one_hot_encode(y_train)
y_test_encoded = one_hot_encode(y_test)

print(f"Training set: {X_train.shape}")
print(f"Test set: {X_test.shape}")
print(f"One-hot encoded labels shape: {y_train_encoded.shape}")
```

Step 3: Initialize neural network with random weights

```
# Network architecture
np.random.seed(42)
input_size = X_train.shape[1]    # 4 features
hidden_size = 8                  # Hidden layer neurons
output_size = 3                  # 3 classes

# Initialize weights with random values (Xavier initialization)
W1 = np.random.randn(input_size, hidden_size) * 0.01  # Input to hidden
b1 = np.zeros((1, hidden_size))                        # Hidden layer bias

W2 = np.random.randn(hidden_size, output_size) * 0.01 # Hidden to output
b2 = np.zeros((1, output_size))                        # Output layer bias

print("Network Architecture:")
print(f"  Input layer: {input_size} neurons")
print(f"  Hidden layer: {hidden_size} neurons")
print(f"  Output layer: {output_size} neurons")
print(f"\nWeight matrices shapes:")
print(f"  W1 (Input → Hidden): {W1.shape}")
print(f"  W2 (Hidden → Output): {W2.shape}")
```

Step 4: Define activation functions

```
# Activation functions
def sigmoid(z):
    """Sigmoid activation function"""
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

def sigmoid_derivative(z):
    """Derivative of sigmoid"""
    return z * (1 - z)

def softmax(z):
    """Softmax activation for output layer"""
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

def relu(z):
    """ReLU activation function"""
    return np.maximum(0, z)

def relu_derivative(z):
    """Derivative of ReLU"""
    return (z > 0).astype(float)

print("Activation functions defined successfully!")
```

Step 5: Forward propagation

```
def forward_propagation(X, W1, b1, W2, b2):
    """
    Forward pass through the network

    Returns:
        z1, a1, z2, a2: Intermediate values needed for backprop
    """
    # Input to hidden layer
    z1 = np.dot(X, W1) + b1           # Linear transformation
    a1 = sigmoid(z1)                  # Sigmoid activation

    # Hidden to output layer
    z2 = np.dot(a1, W2) + b2          # Linear transformation
    a2 = softmax(z2)                  # Softmax activation (probabilities)

    return z1, a1, z2, a2

# Test forward propagation
z1_test, a1_test, z2_test, a2_test = forward_propagation(
    X_train[:1], W1, b1, W2, b2
)

print("Forward propagation test (single sample):")
print(f"  Hidden layer output shape: {a1_test.shape}")
print(f"  Output layer predictions: {a2_test}")
print(f"  Output probabilities sum to 1: {np.sum(a2_test):.4f}")
```

Step 6: Calculate loss and accuracy

```
def calculate_loss(y_true, y_pred):
    """
    Cross-entropy loss function
    """
    m = y_true.shape[0]
    # Add small epsilon to avoid log(0)
    loss = -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))
    return loss

def calculate_accuracy(y_true, y_pred):
    """
    Accuracy: percentage of correct predictions
    """
    # Get predicted class (argmax)
    predictions = np.argmax(y_pred, axis=1)
    # Get true class
    true_labels = np.argmax(y_true, axis=1)
    # Calculate accuracy
    accuracy = np.mean(predictions == true_labels)
    return accuracy

# Test on initial weights
_, _, _, a2 = forward_propagation(X_train, W1, b1, W2, b2)
initial_loss = calculate_loss(y_train_encoded, a2)
initial_acc = calculate_accuracy(y_train_encoded, a2)

print(f"Initial Loss: {initial_loss:.4f}")
print(f"Initial Accuracy: {initial_acc:.4f}")
```

Step 7: Backward propagation (Backpropagation algorithm)

```
def backward_propagation(X, y, z1, a1, z2, a2, W1, W2):
    """
    Backward pass through the network (Backpropagation)

    Computes gradients of loss with respect to weights and biases
    """
    m = X.shape[0]

    # Output layer error
    dz2 = a2 - y  # Derivative of cross-entropy loss + softmax

    # Gradients for W2 and b2
    dW2 = np.dot(a1.T, dz2) / m
    db2 = np.sum(dz2, axis=0, keepdims=True) / m

    # Hidden layer error
    da1 = np.dot(dz2, W2.T)
    dz1 = da1 * sigmoid_derivative(a1)

    # Gradients for W1 and b1
    dW1 = np.dot(X.T, dz1) / m
    db1 = np.sum(dz1, axis=0, keepdims=True) / m

    return dW1, db1, dW2, db2

print("Backpropagation function defined successfully!")
```

Step 8: Update weights using gradient descent

```
def update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    """
    Update weights and biases using gradient descent

    New weight = Old weight - learning_rate * gradient
    """
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    return W1, b1, W2, b2

print("Weight update function defined successfully!")
```

Step 9: Training loop with multiple iterations

```
# Training hyperparameters
learning_rate = 0.1
num_iterations = 100
batch_size = 20

# Reset weights
W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size) * 0.01
b2 = np.zeros((1, output_size))

# Track metrics
train_losses = []
train_accuracies = []
test_accuracies = []

print("Training started...")
print(f"Learning rate: {learning_rate}")
print(f"Iterations: {num_iterations}")
print(f"Batch size: {batch_size}\n")

# Training loop
for iteration in range(num_iterations):
    # Mini-batch gradient descent
    indices = np.random.choice(len(X_train), batch_size)
    X_batch = X_train[indices]
    y_batch = y_train_encoded[indices]

    # Forward pass
    z1, a1, z2, a2 = forward_propagation(X_batch, W1, b1, W2, b2)

    # Calculate loss and accuracy
    loss = calculate_loss(y_batch, a2)
    train_losses.append(loss)

    # Backward pass
    dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, z1, a1, z2, a2, W1, W2)

    # Update weights
```

```
    W1, b1, W2, b2 = update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

    # Calculate accuracies on full sets (every 10 iterations)
    if iteration % 10 == 0:
        _, _, _, train_a2 = forward_propagation(X_train, W1, b1, W2, b2)
        train_acc = calculate_accuracy(y_train_encoded, train_a2)
        train_accuracies.append(train_acc)

        _, _, _, test_a2 = forward_propagation(X_test, W1, b1, W2, b2)
        test_acc = calculate_accuracy(y_test_encoded, test_a2)
        test_accuracies.append(test_acc)

        print(f"Iteration {iteration:3d} | Loss: {loss:.4f} | Train Acc: {train_acc:.4f}

print("\nTraining completed!")
```

Step 10: Plot accuracy and loss for iterations

```
plt.figure(figsize=(14, 5))

# Plot 1: Loss over iterations
plt.subplot(1, 3, 1)
plt.plot(train_losses)
plt.title('Training Loss Over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Cross-Entropy Loss')
plt.grid(True, alpha=0.3)

# Plot 2: Accuracy over iterations
plt.subplot(1, 3, 2)
iterations_points = list(range(0, num_iterations, 10))
plt.plot(iterations_points, train_accuracies, 'b-o', label='Training Accuracy', markersize
plt.plot(iterations_points, test_accuracies, 'r-s', label='Test Accuracy', markersize=6)
plt.title('Accuracy Over Training Iterations')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True, alpha=0.3)

# Plot 3: Loss evolution
plt.subplot(1, 3, 3)
plt.plot(train_losses, alpha=0.7)
plt.title('Loss Convergence')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.grid(True, alpha=0.3)

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

Step 11: Results and conclusions

```
# Final evaluation
_, _, _, final_train_pred = forward_propagation(X_train, W1, b1, W2, b2)
_, _, _, final_test_pred = forward_propagation(X_test, W1, b1, W2, b2)

final_train_acc = calculate_accuracy(y_train_encoded, final_train_pred)
final_test_acc = calculate_accuracy(y_test_encoded, final_test_pred)

print("\n" + "="*60)
print("BACKPROPAGATION ALGORITHM - RESULTS")
print("="*60)
print(f"\nNetwork Architecture:")
print(f"  Input Layer: {input_size} neurons")
print(f"  Hidden Layer: {hidden_size} neurons (Sigmoid activation)")
print(f"  Output Layer: {output_size} neurons (Softmax activation)")

print(f"\nTraining Configuration:")
print(f"  Learning Rate: {learning_rate}")
print(f"  Total Iterations: {num_iterations}")
print(f"  Batch Size: {batch_size}")

print(f"\nFinal Results:")
print(f"  Initial Train Accuracy: {initial_acc:.4f}")
print(f"  Final Train Accuracy:   {final_train_acc:.4f}")
print(f"  Final Test Accuracy:    {final_test_acc:.4f}")
print(f"  Improvement:            {final_train_acc - initial_acc:.4f} ({((final_train_acc

print(f"\nKey Observations:")
print(f"  - Loss decreased from {train_losses[0]:.4f} to {train_losses[-1]:.4f}")
print(f"  - Training accuracy improved from {train_accuracies[0]:.4f} to {train_accuracie
print(f"  - Test accuracy: {final_test_acc:.4f}")
print(f"  - Model successfully learned through backpropagation!")

print("="*60)
```

```python
# Plot ROC AUC Curves for Multi-class Classification (One-vs-Rest)
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import numpy as np
import matplotlib.pyplot as plt

# Get predictions
_, _, _, train_pred_proba = forward_propagation(X_train, W1, b1, W2, b2)
_, _, _, test_pred_proba = forward_propagation(X_test, W1, b1, W2, b2)

# Binarize the output for multi-class ROC AUC
n_classes = 3
y_train_bin = label_binarize(np.argmax(y_train_encoded, axis=1), classes=list(range(n_cla
y_test_bin = label_binarize(np.argmax(y_test_encoded, axis=1), classes=list(range(n_classe

# Function to compute and plot ROC curves
def plot_iris_roc_curves(y_pred_proba, y_bin, dataset_name, iris_target_names):
    fpr = dict()
    tpr = dict()
    roc_auc_dict = dict()
    colors = plt.cm.Set1(np.linspace(0, 1, n_classes))

    plt.figure(figsize=(10, 8))

    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_proba[:, i])
        roc_auc_dict[i] = auc(fpr[i], tpr[i])
        plt.plot(fpr[i], tpr[i], color=colors[i], lw=2,
                 label=f'{iris_target_names[i]} (AUC = {roc_auc_dict[i]:.3f})')

    # Plot random classifier baseline
    plt.plot([0, 1], [0, 1], 'k--', lw=2, label='Random Classifier')

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate', fontsize=12)
    plt.ylabel('True Positive Rate', fontsize=12)
    plt.title(f'ROC Curves - Iris {dataset_name} Set', fontsize=14, fontweight='bold')
```

```
    plt.legend(loc="lower right", fontsize=11)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    # Calculate averages
    macro_auc = np.mean(list(roc_auc_dict.values()))
    return macro_auc, roc_auc_dict

# Iris class names
iris_target_names = iris.target_names

print("\n" + "="*70)
print("ROC AUC ANALYSIS - IRIS DATASET (BACKPROPAGATION)")
print("="*70)

# Plot ROC for Training set
print("\nTraining Set ROC Curves:")
train_macro_auc, train_auc_dict = plot_iris_roc_curves(
    train_pred_proba, y_train_bin, 'Training', iris_target_names)
print(f"Macro-average AUC: {train_macro_auc:.4f}")
for i in range(n_classes):
    print(f"  {iris_target_names[i]}: {train_auc_dict[i]:.4f}")

# Plot ROC for Test set
print("\nTest Set ROC Curves:")
test_macro_auc, test_auc_dict = plot_iris_roc_curves(
    test_pred_proba, y_test_bin, 'Test', iris_target_names)
print(f"Macro-average AUC: {test_macro_auc:.4f}")
for i in range(n_classes):
    print(f"  {iris_target_names[i]}: {test_auc_dict[i]:.4f}")

# Summary comparison
print("\n" + "="*70)
print("PERFORMANCE SUMMARY")
print("="*70)
print(f"Training set macro-average AUC: {train_macro_auc:.4f}")
print(f"Test set macro-average AUC:     {test_macro_auc:.4f}")
print(f"\nModel generalization: {'Good' if abs(train_macro_auc - test_macro_auc) < 0.1 el
```

Notebook: Q25.ipynb

Activation Functions Analysis

This notebook explores various activation functions, their derivatives, and error metrics.

Task 1: Sigmoid and Tanh Activation Functions

- Input range: (-10, +10)
- Plot activation functions and their derivatives
- Observe behavior and characteristics
- Calculate MSE and MAE with sample predictions

Task 2: Tanh and ReLU Activation Functions

- Input range: (-5, +5)
- Plot activation functions and their derivatives
- Observe behavior and characteristics
- Calculate MSE and MAE with sample predictions

Task 3: Sigmoid, ReLU and Softmax Activation Functions

- Input range: (-10, +10)

- Plot activation functions and their derivatives
- Observe behavior and characteristics
- Calculate MSE and MAE with sample predictions

Import Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, mean_absolute_error
import warnings
warnings.filterwarnings('ignore')
```

Define Activation Functions

```
def sigmoid(x):
    return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

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

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

def tanh_derivative(x):
    return 1 - np.tanh(x) ** 2

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

def softmax(x):
    exp_x = np.exp(x - np.max(x))
    return exp_x / np.sum(exp_x, axis=0)

def softmax_derivative(x):
    s = softmax(x)
    return s * (1 - s)
```

Task 1: Sigmoid and Tanh Functions (-10 to +10)

```
x_task1 = np.linspace(-10, 10, 300)

sigmoid_output = sigmoid(x_task1)
sigmoid_deriv = sigmoid_derivative(x_task1)

tanh_output = tanh_function(x_task1)
tanh_deriv = tanh_derivative(x_task1)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

axes[0, 0].plot(x_task1, sigmoid_output, 'b-', linewidth=2, label='Sigmoid')
axes[0, 0].set_title('Sigmoid Activation Function', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Input')
axes[0, 0].set_ylabel('Output')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].legend()

axes[0, 1].plot(x_task1, sigmoid_deriv, 'r-', linewidth=2, label="Sigmoid'")
axes[0, 1].set_title('Sigmoid Derivative', fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel('Input')
axes[0, 1].set_ylabel('Derivative')
axes[0, 1].grid(True, alpha=0.3)
axes[0, 1].legend()

axes[1, 0].plot(x_task1, tanh_output, 'g-', linewidth=2, label='Tanh')
axes[1, 0].set_title('Tanh Activation Function', fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Input')
axes[1, 0].set_ylabel('Output')
axes[1, 0].grid(True, alpha=0.3)
axes[1, 0].legend()

axes[1, 1].plot(x_task1, tanh_deriv, 'orange', linewidth=2, label="Tanh'")
axes[1, 1].set_title('Tanh Derivative', fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel('Input')
axes[1, 1].set_ylabel('Derivative')
axes[1, 1].grid(True, alpha=0.3)
axes[1, 1].legend()

plt.tight_layout()
```

```
plt.show()

print("\nTask 1 - Sigmoid and Tanh Analysis:")
print("=" * 50)
print(f"Sigmoid output range: [{sigmoid_output.min():.4f}, {sigmoid_output.max():.4f}]")
print(f"Sigmoid derivative max: {sigmoid_deriv.max():.4f} at x = {x_task1[sigmoid_deriv.a
print(f"\nTanh output range: [{tanh_output.min():.4f}, {tanh_output.max():.4f}]")
print(f"Tanh derivative max: {tanh_deriv.max():.4f} at x = {x_task1[tanh_deriv.argmax()]:
```

Task 1: Error Metrics (MSE and MAE)

```
y_true_task1 = np.random.rand(50)
y_pred_sigmoid = sigmoid(np.random.uniform(-10, 10, 50))
y_pred_tanh = tanh_function(np.random.uniform(-10, 10, 50))

mse_sigmoid = mean_squared_error(y_true_task1, y_pred_sigmoid)
mae_sigmoid = mean_absolute_error(y_true_task1, y_pred_sigmoid)

mse_tanh = mean_squared_error(y_true_task1, y_pred_tanh)
mae_tanh = mean_absolute_error(y_true_task1, y_pred_tanh)

print("\nTask 1 - Error Metrics:")
print("=" * 50)
print(f"Sigmoid - MSE: {mse_sigmoid:.6f}, MAE: {mae_sigmoid:.6f}")
print(f"Tanh    - MSE: {mse_tanh:.6f}, MAE: {mae_tanh:.6f}")
```

Task 2: Tanh and ReLU Functions (-5 to +5)

```
x_task2 = np.linspace(-5, 5, 300)

tanh_output_t2 = tanh_function(x_task2)
tanh_deriv_t2 = tanh_derivative(x_task2)

relu_output = relu(x_task2)
relu_deriv = relu_derivative(x_task2)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

axes[0, 0].plot(x_task2, tanh_output_t2, 'g-', linewidth=2, label='Tanh')
axes[0, 0].set_title('Tanh Activation Function', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Input')
axes[0, 0].set_ylabel('Output')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].legend()

axes[0, 1].plot(x_task2, tanh_deriv_t2, 'orange', linewidth=2, label="Tanh'")
axes[0, 1].set_title('Tanh Derivative', fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel('Input')
axes[0, 1].set_ylabel('Derivative')
axes[0, 1].grid(True, alpha=0.3)
axes[0, 1].legend()

axes[1, 0].plot(x_task2, relu_output, 'purple', linewidth=2, label='ReLU')
axes[1, 0].set_title('ReLU Activation Function', fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Input')
axes[1, 0].set_ylabel('Output')
axes[1, 0].grid(True, alpha=0.3)
axes[1, 0].legend()

axes[1, 1].plot(x_task2, relu_deriv, 'brown', linewidth=2, label="ReLU'")
axes[1, 1].set_title('ReLU Derivative', fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel('Input')
axes[1, 1].set_ylabel('Derivative')
axes[1, 1].grid(True, alpha=0.3)
axes[1, 1].legend()

plt.tight_layout()
```

```
plt.show()

print("\nTask 2 - Tanh and ReLU Analysis:")
print("=" * 50)
print(f"Tanh output range: [{tanh_output_t2.min():.4f}, {tanh_output_t2.max():.4f}]")
print(f"ReLU output range: [{relu_output.min():.4f}, {relu_output.max():.4f}]")
print(f"ReLU is zero for x < 0: {np.all(relu_output[x_task2 < 0] == 0)}")
```

Task 2: Error Metrics (MSE and MAE)

```
y_true_task2 = np.random.rand(50)
y_pred_tanh_t2 = tanh_function(np.random.uniform(-5, 5, 50))
y_pred_relu = relu(np.random.uniform(-5, 5, 50))

mse_tanh_t2 = mean_squared_error(y_true_task2, y_pred_tanh_t2)
mae_tanh_t2 = mean_absolute_error(y_true_task2, y_pred_tanh_t2)

mse_relu = mean_squared_error(y_true_task2, y_pred_relu)
mae_relu = mean_absolute_error(y_true_task2, y_pred_relu)

print("\nTask 2 - Error Metrics:")
print("=" * 50)
print(f"Tanh - MSE: {mse_tanh_t2:.6f}, MAE: {mae_tanh_t2:.6f}")
print(f"ReLU - MSE: {mse_relu:.6f}, MAE: {mae_relu:.6f}")
```

Task 3: Sigmoid, ReLU and Softmax Functions (-10 to +10)

```
x_task3 = np.linspace(-10, 10, 300)

sigmoid_output_t3 = sigmoid(x_task3)
sigmoid_deriv_t3 = sigmoid_derivative(x_task3)

relu_output_t3 = relu(x_task3)
relu_deriv_t3 = relu_derivative(x_task3)

x_softmax = np.linspace(-10, 10, 300).reshape(-1, 1)
softmax_output = softmax(x_softmax).flatten()
softmax_deriv = softmax_derivative(x_softmax).flatten()

fig, axes = plt.subplots(3, 2, figsize=(14, 14))

axes[0, 0].plot(x_task3, sigmoid_output_t3, 'b-', linewidth=2, label='Sigmoid')
axes[0, 0].set_title('Sigmoid Activation Function', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Input')
axes[0, 0].set_ylabel('Output')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].legend()

axes[0, 1].plot(x_task3, sigmoid_deriv_t3, 'r-', linewidth=2, label="Sigmoid'")
axes[0, 1].set_title('Sigmoid Derivative', fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel('Input')
axes[0, 1].set_ylabel('Derivative')
axes[0, 1].grid(True, alpha=0.3)
axes[0, 1].legend()

axes[1, 0].plot(x_task3, relu_output_t3, 'purple', linewidth=2, label='ReLU')
axes[1, 0].set_title('ReLU Activation Function', fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Input')
axes[1, 0].set_ylabel('Output')
axes[1, 0].grid(True, alpha=0.3)
axes[1, 0].legend()

axes[1, 1].plot(x_task3, relu_deriv_t3, 'brown', linewidth=2, label="ReLU'")
axes[1, 1].set_title('ReLU Derivative', fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel('Input')
axes[1, 1].set_ylabel('Derivative')
```

```
axes[1, 1].grid(True, alpha=0.3)
axes[1, 1].legend()

axes[2, 0].plot(x_task3, softmax_output, 'cyan', linewidth=2, label='Softmax')
axes[2, 0].set_title('Softmax Activation Function', fontsize=12, fontweight='bold')
axes[2, 0].set_xlabel('Input')
axes[2, 0].set_ylabel('Output')
axes[2, 0].grid(True, alpha=0.3)
axes[2, 0].legend()

axes[2, 1].plot(x_task3, softmax_deriv, 'magenta', linewidth=2, label="Softmax'")
axes[2, 1].set_title('Softmax Derivative', fontsize=12, fontweight='bold')
axes[2, 1].set_xlabel('Input')
axes[2, 1].set_ylabel('Derivative')
axes[2, 1].grid(True, alpha=0.3)
axes[2, 1].legend()

plt.tight_layout()
plt.show()

print("\nTask 3 - Sigmoid, ReLU and Softmax Analysis:")
print("=" * 50)
print(f"Sigmoid output range: [{sigmoid_output_t3.min():.4f}, {sigmoid_output_t3.max():.4
print(f"ReLU output range: [{relu_output_t3.min():.4f}, {relu_output_t3.max():.4f}]")
print(f"Softmax output range: [{softmax_output.min():.4f}, {softmax_output.max():.4f}]")
print(f"Softmax output sum: {np.sum(softmax_output):.6f}")
```

Task 3: Error Metrics (MSE and MAE)

```
y_true_task3 = np.random.rand(50)
y_pred_sigmoid_t3 = sigmoid(np.random.uniform(-10, 10, 50))
y_pred_relu_t3 = relu(np.random.uniform(-10, 10, 50))
y_pred_softmax = softmax(np.random.uniform(-10, 10, 50).reshape(-1, 1)).flatten()

mse_sigmoid_t3 = mean_squared_error(y_true_task3, y_pred_sigmoid_t3)
mae_sigmoid_t3 = mean_absolute_error(y_true_task3, y_pred_sigmoid_t3)

mse_relu_t3 = mean_squared_error(y_true_task3, y_pred_relu_t3)
mae_relu_t3 = mean_absolute_error(y_true_task3, y_pred_relu_t3)

mse_softmax = mean_squared_error(y_true_task3, y_pred_softmax)
mae_softmax = mean_absolute_error(y_true_task3, y_pred_softmax)

print("\nTask 3 - Error Metrics:")
print("=" * 50)
print(f"Sigmoid - MSE: {mse_sigmoid_t3:.6f}, MAE: {mae_sigmoid_t3:.6f}")
print(f"ReLU    - MSE: {mse_relu_t3:.6f}, MAE: {mae_relu_t3:.6f}")
print(f"Softmax - MSE: {mse_softmax:.6f}, MAE: {mae_softmax:.6f}")
```

Summary and Observations

```
summary = """
KEY OBSERVATIONS AND CHARACTERISTICS:

Sigmoid Function:
- Output range: (0, 1) - suitable for binary classification
- S-shaped curve with smooth transition
- Maximum derivative at x=0 (0.25)
- Suffers from vanishing gradient problem in deep networks
- Used in output layer for binary classification

Tanh Function:
- Output range: (-1, 1) - centered around zero
- S-shaped curve, smoother transitions than sigmoid
- Maximum derivative at x=0 (1.0) - stronger gradients
- Better for hidden layers than sigmoid
- Converges faster than sigmoid

ReLU Function:
- Output range: [0, ∞) - allows unbounded activation
- Linear for positive inputs, zero for negative
- Derivative: 1 for x > 0, 0 for x < 0
- Computationally efficient (simple comparison)
- Reduces vanishing gradient problem
- Most popular in modern deep networks

Softmax Function:
- Output: Probability distribution (sum = 1)
- Converts logits to probabilities
- Used in multi-class classification
- Output range: (0, 1) for each element
- Maintains relative ordering of inputs

ERROR METRICS INSIGHTS:
- MSE penalizes larger errors more (quadratic)
- MAE treats all errors equally (linear)
- Lower values indicate better predictions
- Choice depends on problem requirements and outlier sensitivity
"""
```

```
print(summary)
```

Notebook: Q26.ipynb

Task 1

1. Use the titanic Dataset
   2.Create an Auto Encoder and fit it with our data using 3 neurons in the dense layer
2. Display new reduced dimension values
3. Plot loss for different encoders [Sparse Autoencoder, Noise Autoencoder]

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.regularizers import l1
```

```
X = pd.read_csv('/media/smayan/500GB SSD/Study/ML2/Practicals/Data/titanic.csv')
X = X.drop(['Name'], axis=1)
X = X.dropna()
X = pd.get_dummies(X, columns=['Sex'], drop_first=True)
X.head()
```

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled.shape
```

Standard Autoencoder with 3 neurons

```
autoencoder_standard = Sequential([
    Input(shape=(X_scaled.shape[1],)),
    Dense(3, activation='relu'),
    Dense(X_scaled.shape[1], activation='sigmoid')
])
autoencoder_standard.compile(optimizer='adam', loss='mse')
history_standard = autoencoder_standard.fit(X_scaled, X_scaled, epochs=50, batch_size=16,
print("Standard Autoencoder trained")
```

```
encoder_standard = Sequential([autoencoder_standard.layers[0]])
encoded_data_standard = encoder_standard.predict(X_scaled, verbose=0)
encoded_data_standard
```

Sparse Autoencoder with L1 Regularization

```
autoencoder_sparse = Sequential([
    Input(shape=(X_scaled.shape[1],)),
    Dense(3, activation='relu', activity_regularizer=l1(0.001)),
    Dense(X_scaled.shape[1], activation='sigmoid')
])
autoencoder_sparse.compile(optimizer='adam', loss='mse')
history_sparse = autoencoder_sparse.fit(X_scaled, X_scaled, epochs=50, batch_size=16, shu
print("Sparse Autoencoder trained")
```

```
encoder_sparse = Sequential([autoencoder_sparse.layers[0]])
encoded_data_sparse = encoder_sparse.predict(X_scaled, verbose=0)
encoded_data_sparse
```

Noise Autoencoder (Denoising Autoencoder)

```
noise_factor = 0.2
X_noisy = X_scaled + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_scaled.sha

autoencoder_noise = Sequential([
    Input(shape=(X_scaled.shape[1],)),
    Dense(3, activation='relu'),
    Dense(X_scaled.shape[1], activation='sigmoid')
])
autoencoder_noise.compile(optimizer='adam', loss='mse')
history_noise = autoencoder_noise.fit(X_noisy, X_scaled, epochs=50, batch_size=16, shuffle
print("Noise Autoencoder trained")
```

```
encoder_noise = Sequential([autoencoder_noise.layers[0]])
encoded_data_noise = encoder_noise.predict(X_scaled, verbose=0)
encoded_data_noise
```

Loss Comparison Plot

```
fig, axes = plt.subplots(1, 3, figsize=(16, 4))

axes[0].plot(history_standard.history['loss'], label='Training Loss', linewidth=2)
axes[0].plot(history_standard.history['val_loss'], label='Validation Loss', linewidth=2)
axes[0].set_title('Standard Autoencoder', fontsize=12, fontweight='bold')
axes[0].set_xlabel('Epochs')
axes[0].set_ylabel('Loss')
axes[0].legend()
axes[0].grid()

axes[1].plot(history_sparse.history['loss'], label='Training Loss', linewidth=2)
axes[1].plot(history_sparse.history['val_loss'], label='Validation Loss', linewidth=2)
axes[1].set_title('Sparse Autoencoder (L1)', fontsize=12, fontweight='bold')
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss')
axes[1].legend()
axes[1].grid()

axes[2].plot(history_noise.history['loss'], label='Training Loss', linewidth=2)
axes[2].plot(history_noise.history['val_loss'], label='Validation Loss', linewidth=2)
axes[2].set_title('Noise Autoencoder', fontsize=12, fontweight='bold')
axes[2].set_xlabel('Epochs')
axes[2].set_ylabel('Loss')
axes[2].legend()
axes[2].grid()

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

Notebook: load.ipynb

```
import numpy as np
```

```
dir = '/media/smayan/500GB SSD/Study/ML2/Practicals/Data/Intel Image Data/seg_train/seg_t
```

```
import os
classes = os.listdir(dir)
```

```
classes
```

```
files = {}
for class_name in os.listdir(dir):
    class_path = os.path.join(dir, class_name)
    if not os.path.isdir(class_path):
        continue
    print(class_path)
    files[class_name] = [f for f in os.listdir(class_path)]
```

```
import glob
from PIL import Image
import numpy as np

def iter_image_paths(root_dir, exts=(".jpg", ".jpeg", ".png", ".bmp")):
    """Yield (path, class_name) for images under root_dir/class_name/*."""
    class_names = [d for d in os.listdir(root_dir) if os.path.isdir(os.path.join(root_dir
    class_names.sort()
    for class_name in class_names:
        class_dir = os.path.join(root_dir, class_name)
        for ext in exts:
            for path in glob.glob(os.path.join(class_dir, f"*{ext}")):
                yield path, class_name

def load_image_pil(path, size=(150, 150), mode="RGB"):
    """Load an image as a NumPy array using PIL (no TensorFlow)."""
    with Image.open(path) as img:
        img = img.convert(mode)
        if size is not None:
            img = img.resize(size)
        arr = np.asarray(img, dtype=np.uint8)
    return arr

def load_folder_dataset(root_dir, size=(150, 150), max_per_class=None, normalize=False):
    """
    Load images from a folder structured as:
      root_dir/
        classA/*.jpg
        classB/*.jpg
    Returns: X (N,H,W,C), y (N,), class_to_index (dict).
    """
    class_names = [d for d in os.listdir(root_dir) if os.path.isdir(os.path.join(root_dir
    class_names.sort()
    class_to_index = {name: i for i, name in enumerate(class_names)}

    X_list = []
    y_list = []
    per_class_count = {name: 0 for name in class_names}
```

```
    for path, class_name in iter_image_paths(root_dir):
        if class_name not in class_to_index:
            continue
        if max_per_class is not None and per_class_count[class_name] >= max_per_class:
            continue
        try:
            arr = load_image_pil(path, size=size, mode="RGB")
        except Exception as e:
            # Skip unreadable/corrupt files
            continue
        X_list.append(arr)
        y_list.append(class_to_index[class_name])
        per_class_count[class_name] += 1

    if len(X_list) == 0:
        raise ValueError(f"No images found under: {root_dir}")

    X = np.stack(X_list, axis=0)
    y = np.asarray(y_list, dtype=np.int64)
    if normalize:
        X = X.astype(np.float32) / 255.0
    return X, y, class_to_index

# Example: limit per class to avoid OOM; set max_per_class=None to load everything
X, y, class_to_index = load_folder_dataset(dir, size=(150, 150), max_per_class=200, norma
X.shape, y.shape, class_to_index
```

```
import matplotlib.pyplot as plt

# Quick sanity check: show a few samples
inv_map = {v: k for k, v in class_to_index.items()}
n_show = min(6, len(X))
plt.figure(figsize=(12, 4))
for i in range(n_show):
    plt.subplot(1, n_show, i + 1)
    plt.imshow(X[i])
    plt.title(inv_map[int(y[i])])
    plt.axis("off")
plt.tight_layout()
plt.show()
```

Notebook: simCLR.ipynb

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
tf.random.set_seed(42)
np.random.seed(42)
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
```

```
subset = 10000
x_train_small = x_train[:subset]
y_train_small = y_train[:subset]
input_shape = x_train_small.shape[1:]
num_classes = 10
```

```
input_shape
```

```
strong_aug = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255),
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.15),
    tf.keras.layers.RandomTranslation(0.1, 0.1),
    tf.keras.layers.RandomContrast(0.2),
    tf.keras.layers.GaussianNoise(0.05),
])
```

```
def identity_aug(x):
    return x
```

```
AUTOTUNE = tf.data.AUTOTUNE
batch_size = 256
temperature = 0.1
epochs = 6

def make_simclr_ds(images, aug_fn):
    ds = tf.data.Dataset.from_tensor_slices(images)
    ds = ds.shuffle(10000)
    ds = ds.map(lambda x :( aug_fn(x, training=True), aug_fn(x, training=True)), num_para
    return ds
```

```
ds_aug = make_simclr_ds(x_train_small, strong_aug)
ds_plain = make_simclr_ds(x_train_small, lambda x : identity_aug(x))
```

```
```