Lab 1: Implementing a Simple Perceptron using TensorFlow

Objective:

Understand the basics of neural networks by implementing a simple perceptron and its limitations on linearly separable vs. non-linearly separable data.

Tasks:

1. **Implement a single-layer perceptron** using TensorFlow.
2. **Train the perceptron on the OR and XOR logic gates**: XOR is a non-linearly separable problem.
3. **Evaluate the performance** using accuracy, precision, recall, and F1 score.
4. **Analyze the model limitations**: Explain why the perceptron fails on XOR and suggest possible solutions (e.g., adding hidden layers).

Steps:

1. **Set up the environment**:

```
pip install tensorflow
```

2. **Define the OR and XOR gate datasets**:

```
# Define OR and XOR datasets
X_OR = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
y_OR = np.array([[0], [1], [1], [1]], dtype=np.float32)

X_XOR = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
y_XOR = np.array([[0], [1], [1], [0]], dtype=np.float32)
```

3. **Create a single-layer perceptron model**:

```
import tensorflow as tf

# Function to create and train a single-layer perceptron with improved
accuracy
def train_perceptron(X, y, epochs=100, learning_rate=0.1):
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(1, activation='sigmoid', input_shape=(2,))
    ])
    model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=
    learning_rate),loss='binary_crossentropy',
    metrics=['accuracy'])
    model.fit(X, y, epochs=epochs, verbose=0)
    return model
```

4. **Train the model on OR and XOR datasets**:

```
# Train on OR gate with improved accuracy
model_OR = train_perceptron(X_OR, y_OR, epochs=500, learning_rate=0.5)

# Evaluate on OR
loss_OR, accuracy_OR = model_OR.evaluate(X_OR, y_OR)
print(f"OR Gate Accuracy: {accuracy_OR}")

# Train on XOR gate with improved accuracy (not possible with a single-
layer perceptron)
```

```
# However, we can try increasing epochs and learning rate
model_XOR = train_perceptron(X_XOR, y_XOR, epochs=1000, learning_rate=0.8)

# Evaluate on XOR
loss_XOR, accuracy_XOR = model_XOR.evaluate(X_XOR, y_XOR)
print(f"XOR Gate Accuracy: {accuracy_XOR}")
```

5. **Analyze the results**:

   #Make a prediction using model_OR

```
input1 = 0
input2 = 0

user_input = np.array([[input1, input2]])

prediction = model_OR.predict(user_input)

if prediction > 0.5:
    print("The model predicts 1 for your input.")
else:
    print("The model predicts 0 for your input.")

#Do the same using model_XOR
```

   - Discuss the success on the OR gate and failure on the XOR gate.
   - Explain that the XOR gate cannot be solved by a single-layer perceptron due to non-linearity, and suggest adding hidden layers as a solution.

Expected Outcome:

Students will learn the limitations of a single-layer perceptron and understand why it fails on non-linearly separable problems like XOR.

Lab 2: Building a Multilayer Perceptron (MLP)

Objective:

Learn to build and train a multilayer perceptron (MLP) for more complex classification tasks using advanced optimization techniques.

Tasks:

1. **Create an MLP with additional hidden layers**: Introduce batch normalization and dropout.
2. **Train the MLP on the Fashion-MNIST dataset**, which is more challenging than MNIST.
3. **Apply learning rate scheduling** to improve training stability.
4. **Visualize training progress**: Track and plot accuracy, loss, and overfitting behavior.

Steps:

1. **Load the Fashion-MNIST dataset**:

```
from tensorflow.keras.datasets import fashion_mnist

# Load dataset
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

# Normalize data
X_train, X_test = X_train / 255.0, X_test / 255.0
```

2. **Define the MLP model** with batch normalization and dropout:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

3. **Implement learning rate scheduling**:

```
lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-3
* 10 ** (epoch / 20))

history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test,
y_test), callbacks=[lr_schedule])
```

4. **Plot training and validation performance**:

```
import matplotlib.pyplot as plt

# Plot accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Expected Outcome:

Students will learn how to use batch normalization, dropout, and learning rate scheduling to improve MLP performance. They will also visualize model training to detect overfitting.

Lab 3: Exploring Convolutional Neural Networks (CNN)

Objective:

Explore more advanced convolutional neural network (CNN) architectures and interpret their behavior.

Tasks:

1. **Build a deeper CNN** with multiple convolutional layers, max-pooling, and data augmentation.
2. **Train the CNN on the CIFAR-10 dataset**, which contains 32x32 color images..
3. **Evaluate the model** using precision, recall.

Steps:

1. **Load and preprocess the CIFAR-10 dataset**:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Normalize pixel values to be between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
# Convert class vectors to binary class matrices (one-hot encoding)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
```

2. **Define a deeper CNN model**:

```
# Define the CNN model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', padding='same',
input_shape=(32, 32, 3)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

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

3. **Evaluate the model using precision, recall, and confusion matrix**:

```
# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=64,
validation_data=(X_test, y_test))

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print('Test accuracy:', accuracy)

# Make predictions on the test set
y_pred_prob = model.predict(X_test)
y_pred = np.argmax(y_pred_prob, axis=1)
y_true = np.argmax(y_test, axis=1)

# Calculate precision and recall
precision = precision_score(y_true, y_pred, average='macro')
recall = recall_score(y_true, y_pred, average='macro')

print('Precision:', precision)
print('Recall:', recall)
```

Expected Outcome:

Students will understand CNN architecture in greater depth and gain experience working with a more difficult dataset.

Lab 4: Introduction to Recurrent Neural Networks (RNN)

Objective:

Explore the workings of recurrent neural networks (RNNs) and implement them on sequential data, including more advanced RNN architectures such as LSTMs or GRUs.

Tasks:

1. **Implement an advanced RNN architecture**: Use Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU) for handling sequential dependencies.
2. **Train the model on sequential data**: Use a real-world sequential dataset such as the IMDB sentiment analysis dataset.
3. **Use word embeddings**: Apply an embedding layer to convert text into numerical representations.
4. **Evaluate and visualize model predictions** using accuracy, precision, recall, and F1 score.

Steps:

1. **Load the IMDB dataset**:

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Load dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)

# Pad sequences to ensure equal length inputs
X_train = pad_sequences(X_train, maxlen=200)
X_test = pad_sequences(X_test, maxlen=200)
```

2. **Define the LSTM model with an embedding layer**:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=10000, output_dim=128,
input_length=200),
    tf.keras.layers.LSTM(128, return_sequences=False),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

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

3. **Train the model on the IMDB dataset**:

```
history = model.fit(X_train, y_train, epochs=10, batch_size=64,
validation_data=(X_test, y_test))
```

4. **Visualize training progress**: Plot accuracy and loss for both training and validation sets.

```
import matplotlib.pyplot as plt

# Plot accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.show()
```

5. **Evaluate the model using precision, recall, and F1 score**:

```
from sklearn.metrics import precision_score, recall_score, f1_score

# Get predictions and round them to nearest integer
y_pred = (model.predict(X_test) > 0.5).astype("int32")

# Evaluate performance
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Precision: {precision}, Recall: {recall}, F1 Score: {f1}")
```

6. **Analyze model performance on sequential data**: Discuss how LSTMs/GRUs handle sequential dependencies and compare the results with standard RNNs.

```
def preprocess_input(text):
    tokenizer = Tokenizer(num_words=max_features)
    tokenizer.fit_on_texts([text])  # Fit only on the user input
    sequence = tokenizer.texts_to_sequences([text])
    padded_sequence = pad_sequences(sequence, maxlen=maxlen)
    return padded_sequence

# Get user input
user_input = input("Enter a movie review: ")

# Preprocess the user input
processed_input = preprocess_input(user_input)

# Make prediction
prediction = model.predict(processed_input)
sentiment = "Positive" if prediction[0][0] > 0.5 else "Negative"

# Output the result
print(f"Predicted Sentiment: {sentiment} (Probability: {prediction[0]
[0]:.2f})")
```

Expected Outcome:

Students will gain experience using LSTMs or GRUs, and they will understand how recurrent neural networks can handle sequential data like text. They will learn to evaluate model performance using advanced metrics such as precision, recall, and F1 score.

Lab 5: Autoencoders for Data Compression

Objective:

Understand autoencoders and their applications for data compression. Implement more complex architectures, such as convolutional autoencoders (CAE) and denoising autoencoders.

Tasks:

1. **Build a convolutional autoencoder (CAE)**: Use convolutional layers instead of fully connected layers to handle image data more effectively.
2. **Train the autoencoder on the CIFAR-10 dataset** to handle more complex image data.
3. **Implement a denoising autoencoder**: Train the autoencoder to reconstruct images from noisy inputs.
4. **Evaluate and visualize the original, noisy, and reconstructed images**. Measure reconstruction quality using mean squared error (MSE).

Steps:

1. **Load and preprocess the CIFAR-10 dataset**:

```
from tensorflow.keras.datasets import cifar10

# Load dataset
(X_train, _), (X_test, _) = cifar10.load_data()

# Normalize the data
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

2. **Add noise to the CIFAR-10 images**:

```
import numpy as np

noise_factor = 0.5
X_train_noisy = X_train + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=X_train.shape)
X_test_noisy = X_test + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=X_test.shape)

# Clip to keep pixel values between 0 and 1
X_train_noisy = np.clip(X_train_noisy, 0., 1.)
X_test_noisy = np.clip(X_test_noisy, 0., 1.)
```

3. **Define the convolutional autoencoder (CAE)**:

```
model = tf.keras.Sequential([
    # Encoder
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same',
input_shape=(32, 32, 3)),
    tf.keras.layers.MaxPooling2D((2, 2), padding='same'),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    tf.keras.layers.MaxPooling2D((2, 2), padding='same'),

    # Decoder
    tf.keras.layers.Conv2DTranspose(64, (3, 3), activation='relu',
padding='same'),
    tf.keras.layers.UpSampling2D((2, 2)),
```

```
    tf.keras.layers.Conv2DTranspose(32, (3, 3), activation='relu',
padding='same'),
    tf.keras.layers.UpSampling2D((2, 2)),
    tf.keras.layers.Conv2D(3, (3, 3), activation='sigmoid',
padding='same')
])

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

4. **Train the autoencoder** on noisy CIFAR-10 images:

```
history = model.fit(X_train_noisy, X_train, epochs=50, batch_size=128,
validation_data=(X_test_noisy, X_test))
```

5. **Visualize original, noisy, and reconstructed images**:

```
import matplotlib.pyplot as plt

# Reconstruct images
decoded_imgs = model.predict(X_test_noisy)

# Display original, noisy, and reconstructed images
n = 10
plt.figure(figsize=(20, 6))
for i in range(n):
    # Display original
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(X_test[i])
    plt.title("Original")
    plt.axis('off')

    # Display noisy
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(X_test_noisy[i])
    plt.title("Noisy")
    plt.axis('off')

    # Display reconstruction
    ax = plt.subplot(3, n, i + 1 + 2*n)
    plt.imshow(decoded_imgs[i])
    plt.title("Reconstructed")
    plt.axis('off')
plt.show()
```

6. **Evaluate the performance using Mean Squared Error (MSE)**:

```
mse = np.mean(np.square(X_test - decoded_imgs))
print(f"Reconstruction MSE: {mse}")
```

Expected Outcome:

Students will understand the concept of convolutional autoencoders and how they can be applied for data compression and denoising. They will also gain hands-on experience in evaluating the performance of autoencoders using MSE and visualizing the reconstructed images.