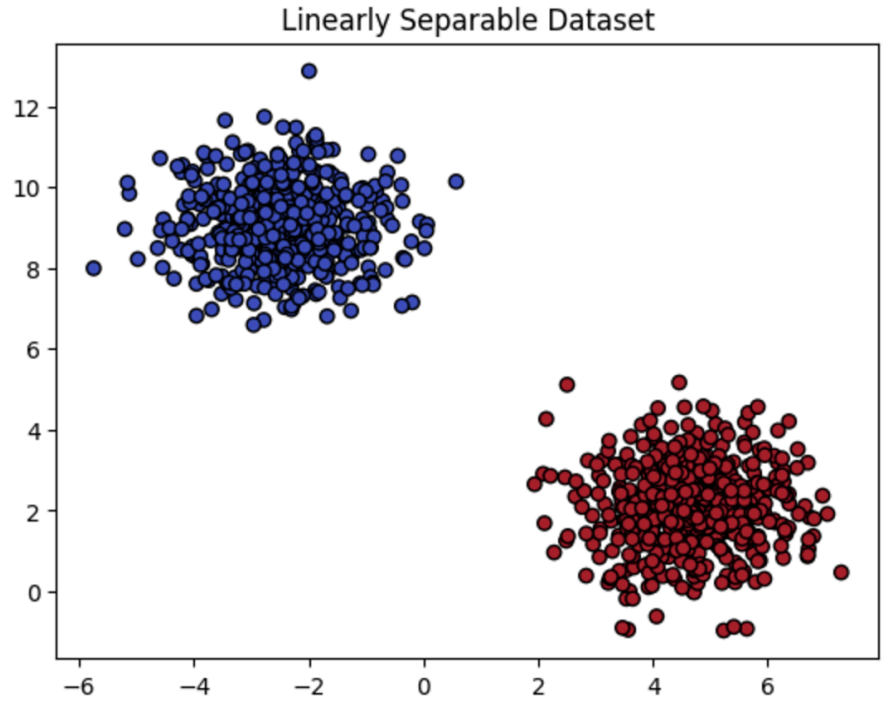
CS478: Introduction to Deep Learning
Assignment 1 (10 points)

Understanding Perceptrons and Multilayer Perceptrons

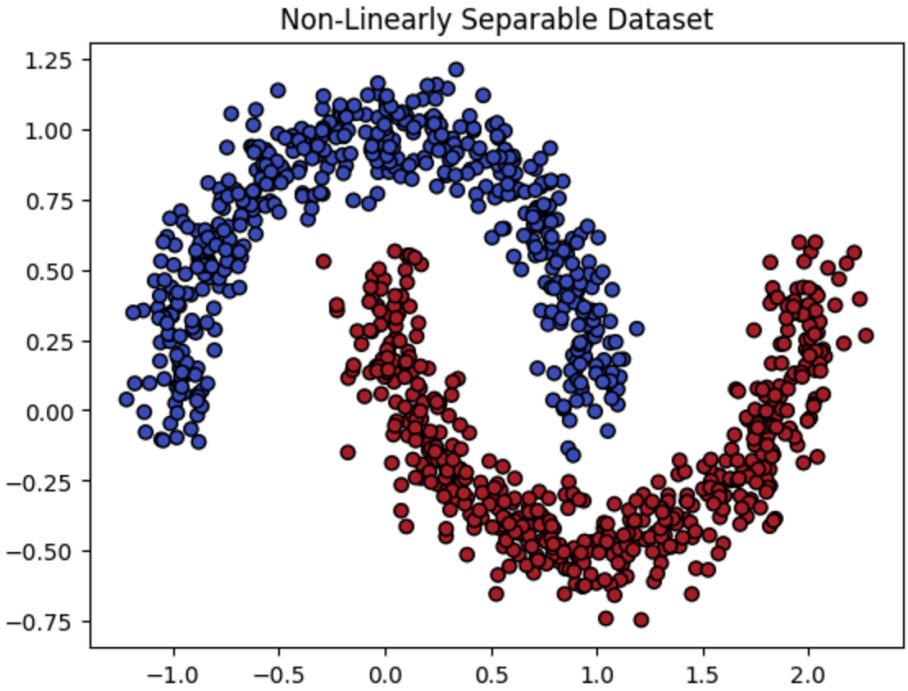
Implement a simple perceptron and a multilayer perceptron (MLP) to classify points in a 2D space.

Generate a simple dataset with linearly separable classes. See the starter code to generate the dataset



Task 1: Build a single-layer perceptron to classify this dataset. Train the model, and evaluate its performance. Report the accuracy.

Modify the dataset to include non-linearly separable classes. See the starter code to generate the dataset.



Task 2: Train the perceptron on this non-linear dataset, observe the results, and report the accuracy.

Task 3: Build a simple MLP with one hidden layer to classify the non-linearly separable dataset. Train the model, and evaluate its performance.

Task 4: Experiment with different activation functions (sigmoid, and ReLU) and report any changes in performance.

Summarize your findings regarding the impact of adding a hidden layer and activation functions on the MLP's ability to solve complex problems.

Requirements:

- Submit your code and a report summarizing your findings from the implementation task.
- Your code should be well-documented, with comments explaining each part of the model's setup and training process.
- Use Python and any popular deep learning library (Keras, TensorFlow, or PyTorch).

Colab Link:
Google Colab Link
GitHub Repo Link

Understanding the Starter Code:

Within the provided starter code, we use the *make_blobs* from the scikit-learn toolkit to generate a linearly separable dataset. We visualize the data using a scatter plot, as seen in the document. We also use the *make_moons* from the scikit-learn toolkit to generate a non-linearly separable dataset, and we visualize the data using a scatter plot.

Task 1:

We now must build a single-layered perceptron on the linearly separable dataset we have generated from the starter code. First, we must split the dataset into training and testing datasets, and we do this through the *train_test_split* from the scikit-learn toolkit. We then build our single-layered perceptron using the *Sequential* with a single perceptron as the output layer using a sigmoid activation function, so we get an output value between zero and one. We then compile our model using the *SGD optimizer (Stochastic Gradient Descent) with a learning rate of 0.1, the loss function used is* binary_crossentropy, and the metrics used are accuracy. Now we must train the model that we just built, we are going to use 50 epochs and a batch size of 32. After training we must evaluate our model, we do this by making a prediction, and lastly printing out the accuracy. The following code segment of the explanation given follows:

```
# Task 1.
# Split Dataset into Training & Testing.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Build Single-Layer Perceptron Model.
model = Sequential([
    Dense(1, activation='sigmoid') # Sigmoid Activation on Output Layer.
])

# Compile Model.
model.compile(optimizer=SGD(learning_rate=0.1), loss='binary_crossentropy',
metrics=['accuracy'])

# Train Model.
model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0)

# Evaluate Model.
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)

# Report Accuracy.
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy of the Single-Layered Perceptron: {accuracy * 100:.2f}%')
```

The model has an accuracy of 100% which tells us that the model has a linear decision boundary, which if you look at the graph of the data, it does based on observation, but our model proves this theory. It tells us there are no misclassifications since all training examples learned the correct decision boundary line. Here is the proof of the result:

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
Accuracy of the Single-Layered Perceptron: 100.00%
```

Task 2:

We now must build the same model (single-layered perceptron) but on the non-linearly separable data. For simplicity, the differences will be stated since it is the same code as Task 1. The only difference between the two models is the data that they are trained on. The following code segment shows the single-layered perceptron trained on non-linearly separable dataset generated from teh provided starter code:

```
# Task 2.
# Split Dataset into Training & Testing.
X_train_non_linear, X_test_non_linear, y_train_non_linear, y_test_non_linear =
train_test_split(X_non_linear, y_non_linear, test_size=0.2, random_state=42)

# Build Model.
model_non_linear = Sequential([
    Dense(1, activation='sigmoid') # Sigmoid Activation on Output Layer.
])

# Compile Model.
model_non_linear.compile(optimizer=SGD(learning_rate=0.1),
loss='binary_crossentropy', metrics=['accuracy'])

# Train Model.
model_non_linear.fit(X_train_non_linear, y_train_non_linear, epochs=50,
batch_size=32, verbose=0)

# Evaluate Model.
y_pred_non_linear = model_non_linear.predict(X_test_non_linear)
y_pred_non_linear = (y_pred_non_linear > 0.5).astype(int)

# Report Accuracy.
accuracy_non_linear = accuracy_score(y_test_non_linear, y_pred_non_linear)
print(f'Accuracy of the Single-Layered Perceptron: {accuracy_non_linear *
100:.2f}%')
```

The model has an accuracy of 86.5% on the non-linearly separable data. If we look at the graph this time, we notice it isn't linear, but we are using a linear model since we just have 1 single perceptron, based on the graph we can't necessarily draw a linear decision boundary to classify each class. Also, note the accuracy is still high, we can make the argument that since our model is trying to find a line to best fit to separate the classes, it finds the best

approximation for the line, although there may be overlap in the classes, 86.5% will be classified correctly. Here is the proof of the result:

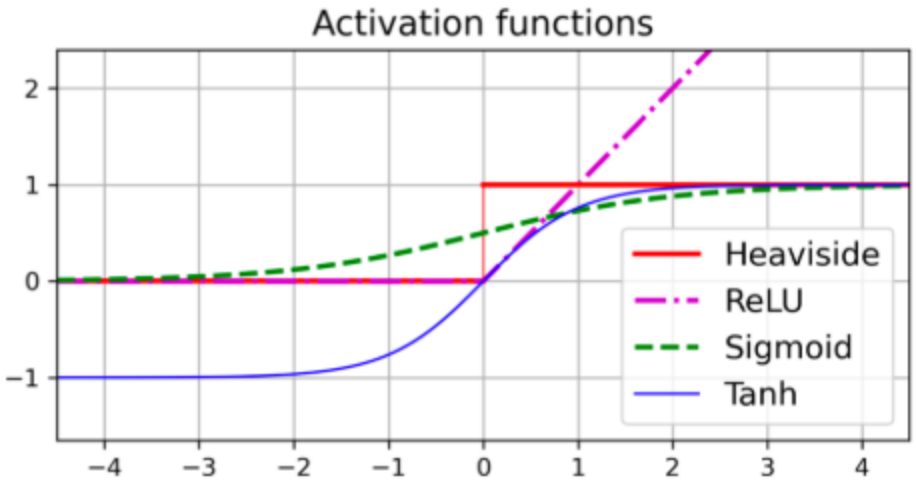
```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
Accuracy of the Single-Layered Perceptron: 86.50%
```

Task 3:

We must build a multi-layered perceptron (MLP) with one hidden layer to classify the non-linearly separable data. The only difference here is that we are adding a hidden layer from the previous model. Within our hidden layer, we are going to choose to use 8 neurons using the sigmoid activation function. Note that the number of neurons is important because we don't want to underfit or overfit our neural network. The number of hidden layers is also important because our data could overfit or underfit as well, but for this task, we are asked to build an MLP with one hidden layer. We chose 8 neurons since our dataset is not that big. The following code segment shows the network is implemented keeping in mind the information presented above:

```
# Task 3.
# Build  MLP Model.
model_mlp = Sequential([
    Dense(8, activation='sigmoid'), # Sigmoid Activation on Hidden Layer.
    Dense(1, activation='sigmoid') # Sigmoid Activation on Output Layer.
])

# Compile Model.
model_mlp.compile(optimizer=SGD(learning_rate=0.1), loss='binary_crossentropy',
metrics=['accuracy'])

# Train Model.
model_mlp.fit(X_train_non_linear, y_train_non_linear, epochs=50, batch_size=32,
verbose=0)

# Evaluate Model.
y_pred_mlp = model_mlp.predict(X_test_non_linear)
y_pred_mlp = (y_pred_mlp > 0.5).astype(int)

# Report Accuracy.
accuracy_mlp = accuracy_score(y_test_non_linear, y_pred_mlp)
print(f'Accuracy of the Multi-Layered Perceptron: {accuracy_mlp * 100:.2f}%')
```

The accuracy of the model is 87% which is barely better than our single-layered perceptron model. Note that our activation function is a sigmoid function which takes longer to learn, especially since we are using 8 neurons in the hidden layer, it is believed that relu will be much faster in learning than sigmoid due to this graph in the professor's slides:



Also, another factor for not improvement in performance is the number of hidden layers because through testing we tried raising the number of neurons and the accuracy was barely better. Using 50 neurons on one hidden layer raised the accuracy to around 88%, so increasing the number of hidden layers should achieve better accuracy. Here is the proof of the initial result:

```
7/7 ——————————————————— 0s 11ms/step
Accuracy of the Multi-Layered Perceptron: 87.00%
```

Task 4:

We now must change our model using different activation functions and adding more hidden layers to experiment with what our model was doing. For this task, we do two experiments where the first experiment we just change the activation function from sigmoid to relu and the second experiment will have 3 hidden layers using the relu activation function. Here is the code segment where we change the activation function from sigmoid to relu:

```
# Task 4.
# Build MLP Model.
exp_model1 = Sequential([
    Dense(8, activation='relu'), # Sigmoid Activation on Hidden Layer.
    Dense(1, activation='sigmoid') #Sigmoid Activation on Output Layer.
])

# Compile Model.
exp_model1.compile(optimizer=SGD(learning_rate=0.1),
loss='binary_crossentropy', metrics=['accuracy'])

# Train Model.
exp_model1.fit(X_train_non_linear, y_train_non_linear, epochs=50,
batch_size=32, verbose=0)

# Evaluate Model.
y_pred_exp_model1 = exp_model1.predict(X_test_non_linear)
```

```
y_pred_exp_model1 = (y_pred_exp_model1 > 0.5).astype(int)

# Report Accuracy.
accuracy_exp_model1 = accuracy_score(y_test_non_linear, y_pred_exp_model1)
print(f'Accuracy of the Multi-Layered Perceptron: {accuracy_exp_model1 *
100:.2f}%')
```

Our accuracy is much better as we now get 90% which increased due to changing the activation function in the hidden layer. Note that we keep the activation function in the output layer to be sigmoid since there is only one neuron in the output layer. Here's proof of the result: [insert image]

As stated before, for the second experiment we add more hidden layers to see if the performance gets any better still using relu as the activation function for those hidden layers. The following code provided shows that we added 2 additional hidden layers with 8 neurons:

```
# Build Another MLP Model.
exp_model2 = Sequential([
    Dense(8, activation='relu'), # Sigmoid Activation on Hidden Layer.
    Dense(8, activation='relu'), # Sigmoid Activation on Hidden Layer.
    Dense(8, activation='relu'), # Sigmoid Activation on Hidden Layer.
    Dense(1, activation='sigmoid') #Sigmoid Activation on Output Layer.
])

# Compile Model.
exp_model2.compile(optimizer=SGD(learning_rate=0.1),
loss='binary_crossentropy', metrics=['accuracy'])

# Train Model.
exp_model2.fit(X_train_non_linear, y_train_non_linear, epochs=50,
batch_size=32, verbose=0)

# Evaluate Model.
y_pred_exp_model2 = exp_model2.predict(X_test_non_linear)
y_pred_exp_model2 = (y_pred_exp_model2 > 0.5).astype(int)

# Report Accuracy.
accuracy_exp_model2 = accuracy_score(y_test_non_linear, y_pred_exp_model2)
print(f'Accuracy of the Multi-Layered Perceptron: {accuracy_exp_model2 *
100:.2f}%')
```

The accuracy for this model was 100%, which is much better than the previous model, the model is not overfitting since the accuracy is based on the test data rather than the training data affecting the testing data. However, we believe lowering the number of hidden layers to 2 may be more beneficial to limit the risk of overfitting the data. As mentioned the number of neurons and the number of hidden layers will affect the accuracy of the model. Here is the proof of the result:

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step
Accuracy of the Multi-Layered Perceptron: 91.00%
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
Accuracy of the Multi-Layered Perceptron: 100.00%
```