Neural Network Playground Clone

Objective

To simulate a basic version of the TensorFlow Neural Network Playground by:

- Training a neural network on a 2D non-linear dataset.
- Visualizing the model's learning progress and decision boundaries interactively.

Tools and Technologies

- **TensorFlow** — for building and training the neural network.
- **scikit-learn** — for generating synthetic datasets.
- **Matplotlib** — for plotting and visualizing results.
- **NumPy** and **Pandas** — for data manipulation and support.

Project Workflow

1. Setup and Imports

Logic:
Prepare the environment by installing required libraries and importing them.

```
!pip install tensorflow

from sklearn.datasets import make_circles

from sklearn.model_selection import train_test_split

import tensorflow as tf

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

2. Dataset Creation

Logic:
We use the make_circles function to create a 2D binary classification dataset shaped like two concentric circles — this problem is not linearly separable, so it tests the model's ability to learn non-linear decision boundaries.

```
X, y = make_circles(n_samples=1000, noise=0.03, factor=0.5, random_state=42)

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

3. Decision Boundary Visualization Function

Logic:
The plot_decision_boundary function creates a grid of points across the 2D space, feeds them into the model to predict classes, and then colors the regions based on predictions. It overlays the original data points for comparison.

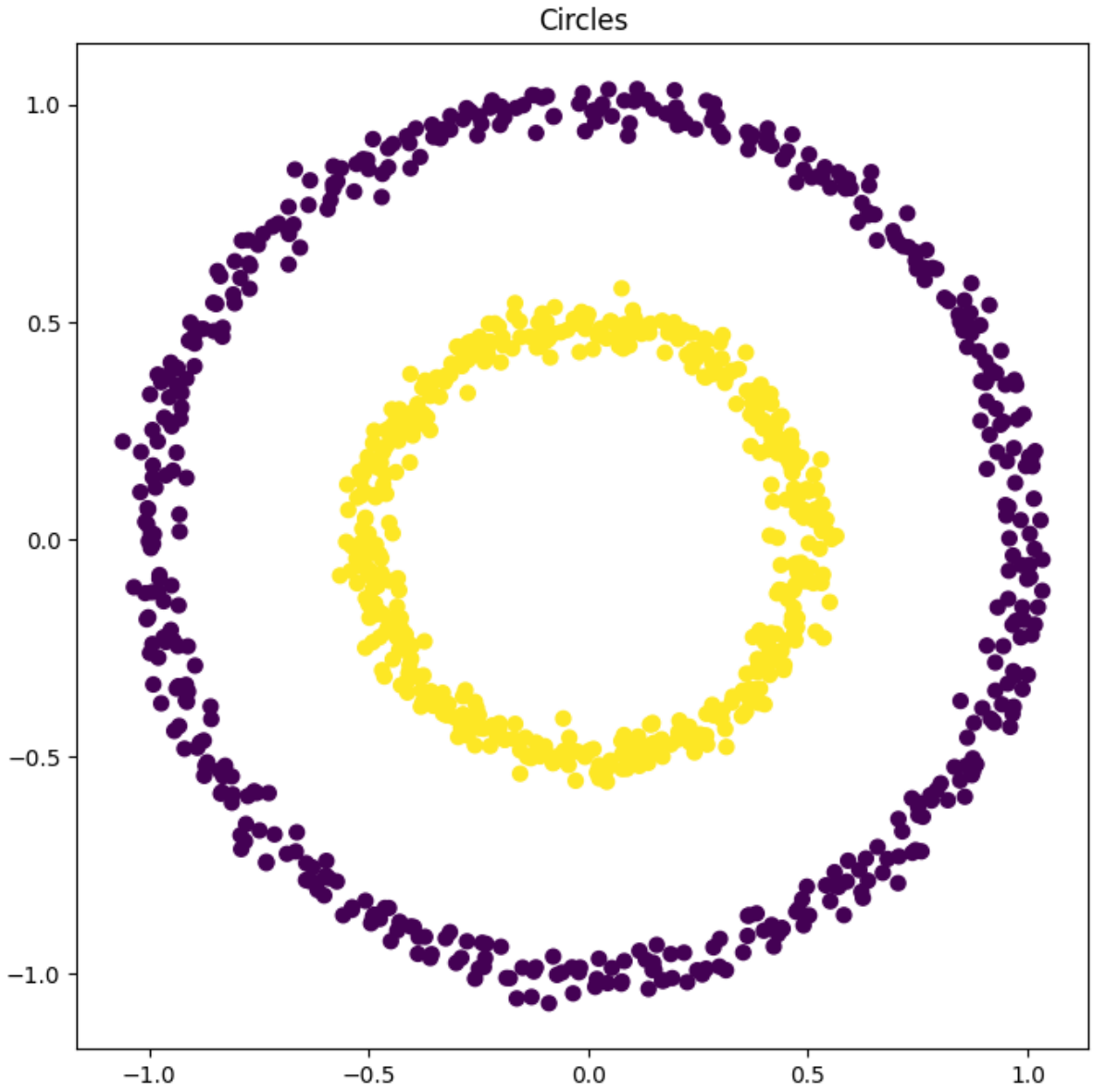
```
def plot_decision_boundary(model, x, y):
  x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
  y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
  xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                       np.linspace(y_min, y_max, 100))
  grid = np.c_[xx.ravel(), yy.ravel()]
  pred_func = model.predict(grid)
  pred_func = np.round(pred_func).reshape(xx.shape)
  plt.contourf(xx, yy, pred_func, alpha=0.8, cmap='viridis')
  plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap='viridis')
  plt.xlim(xx.min(), xx.max())
  plt.ylim(yy.min(), yy.max())
  plt.show()
```

4. Visualizing the Initial Dataset

Logic:
Before training, we visualize the data to understand its structure.

```
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap='viridis')
plt.title("Circles")
plt.show()
```

Initial Data Scatter Plot



5. Model Building

Logic:
We build a simple feed-forward neural network:

- **Input layer** (2 nodes for x and y coordinates)
- **Hidden layers** with 8 neurons and ReLU activation
- **Output layer** with 1 neuron and sigmoid activation (for binary classification)

```
base_model_circles = tf.keras.Sequential([
  tf.keras.layers.Dense(8, activation='relu', input_shape=(2,)),
  tf.keras.layers.Dense(8, activation='relu'),
```

```
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

6. Model Compilation

Logic:
We compile the model using:

- **Adam Optimizer**: an adaptive learning rate optimization algorithm.
- **Binary Crossentropy Loss**: suitable for binary classification problems.
- **Accuracy Metric**: to measure performance.

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

7. Model Training

Logic:
The model is trained for **50 epochs** on the training data.
We also monitor performance on validation (test) data at each epoch.

```
base_model_circles_loss = base_model_circles.fit(X_train, y_train, epochs=50,
validation_data=(X_test, y_test))
```

8. Model Evaluation

Logic:
After training, we evaluate the model to see how well it generalizes to unseen data (test set).

```
loss, accuracy = base_model_circles.evaluate(X_test, y_test)
print("Test accuracy: {accuracy:.2f}")
```

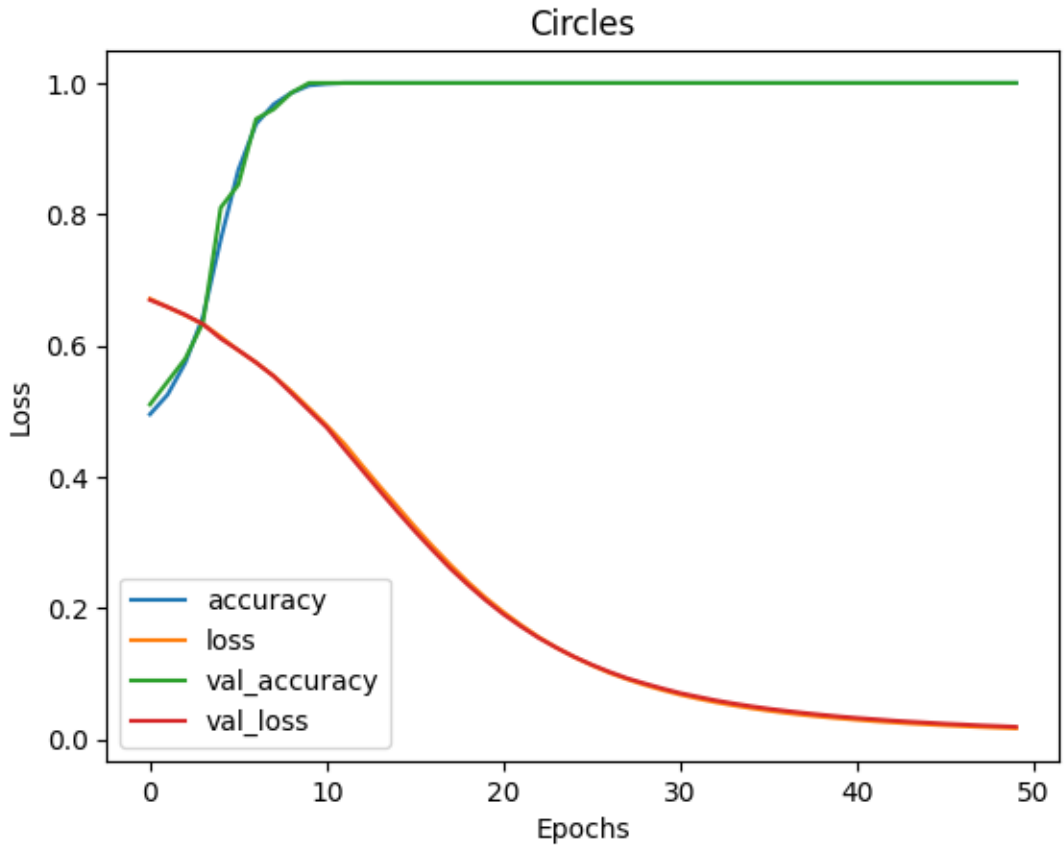
9. Training Progress Visualization

Logic:
Plot the loss curve over epochs to see if the model converged properly and if overfitting/underfitting occurred.

```
pd.DataFrame(base_model_circles_loss.history).plot()
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

```
plt.title("Circles")

plt.show()
```



Conclusion Based on Loss vs Epochs Plot

- Training loss steadily decreases over time, showing that the model is successfully learning the patterns in the training data.
- Validation loss also generally decreases and stays close to the training loss, meaning the model is not overfitting significantly.
- There are no major spikes, sharp increases, or irregular patterns, suggesting that:
    - The learning rate is appropriate.
    - The model architecture is well-suited for the task.
- Both training and validation losses stabilize toward the end of the training, indicating that the model converged properly.
- The final test accuracy is so close to 1, confirming that the model performs very well on unseen data.

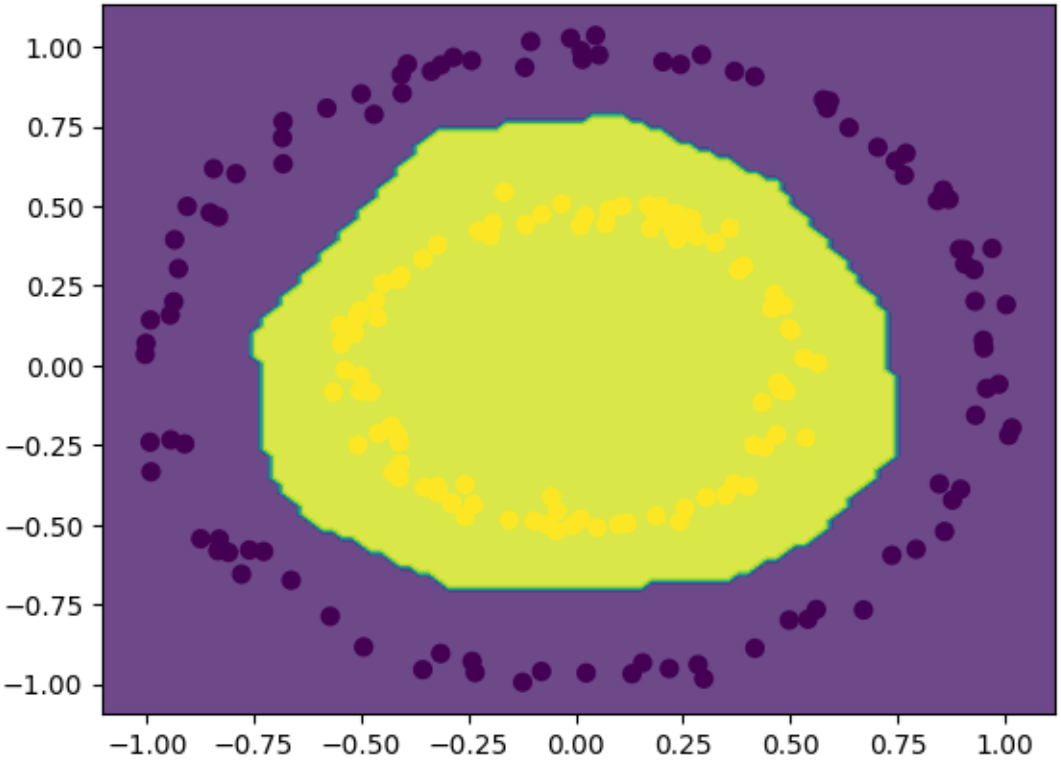
Overall, the training process was smooth and successful.
The model learned meaningful decision boundaries without underfitting or overfitting.

10. Final Decision Boundary Visualization

Logic:
After the model has learned, we use the decision boundary function to visualize how well the model separates the two classes.

```
plot_decision_boundary(base_model_circles, X_test, y_test)
```

Decision Boundary After Training



The decision boundary plot shows that the model has successfully learned the non-linear pattern of the dataset (the two concentric circles).

- The boundary smoothly wraps around the inner circle, correctly separating the two classes.
- There are no major misclassified regions — the transition between classes looks clean and accurate.
- This indicates that:
    - The neural network captured the complex relationship between the features (x, y).
    - The hidden layers and non-linear activations (ReLU) allowed the model to learn flexible, non-linear boundaries.