MALARIA DATASET -ASSIGNMENT

1. **Model Architecture:**
    - Detailed view of the Convolutional Neural Network (CNN) architecture I used for the malaria dataset classification.
    - Schematic representation of the layers, their configurations, and connections within the CNN model.
    - Input Layer:
        i. Shape: (64, 64, 3) - Represents an image of size 64x64 pixels with 3 color channels (RGB).
        ii. Input layer is defined using Input(shape=(64, 64, 3)) in Keras.
    - Convolutional Layers:
        i. First Conv2D layer:
            1. Number of Filters (Kernels): 32
            2. Kernel Size: (3, 3)
            3. Activation Function: ReLU (Rectified Linear Unit)
            4. Output Shape: Since no padding or strides are specified, output shape is inferred based on input shape and filter size.
        ii. MaxPooling2D layer:
            1. Pool Size: (2, 2)
            2. Strides: Default (same as pool size)
            3. Output Shape: Halves both height and width dimensions.
        iii. Second Conv2D layer:
            1. Number of Filters (Kernels): 64
            2. Kernel Size: (3, 3)
            3. Activation Function: ReLU
            4. Output Shape: Similar to the first Conv2D layer, but with 64 filters.
    - Flatten Layer:
        i. Converts the 2D output of the convolutional layers into a 1D feature vector for the Dense layers.
        ii. No trainable parameters.
    - Dense Layers:
        i. First Dense layer:
            1. Number of Neurons: 128
            2. Activation Function: ReLU
            3. Output Shape: 128-dimensional vector.
        ii. Second Dense layer (Output layer):

1. Number of Neurons: 1 (Binary classification - 0 for uninfected, 1 for parasitized)
2. Activation Function: Sigmoid (for binary classification)
3. Output Shape: Scalar value representing the probability of being parasitized.

- Model Compilation:
    i. Optimizer: Adam - Adaptive Moment Estimation optimizer.
    ii. Loss Function: Binary Cross Entropy - Suitable for binary classification tasks.
    iii. Metrics: Accuracy - Evaluates the model based on classification accuracy.

Schematic Representation:

(Input) -> Conv2D(32) -> MaxPooling2D -> Conv2D(64) -> MaxPooling2D -> Flatten -> Dense(128) -> Dense(1, Sigmoid)

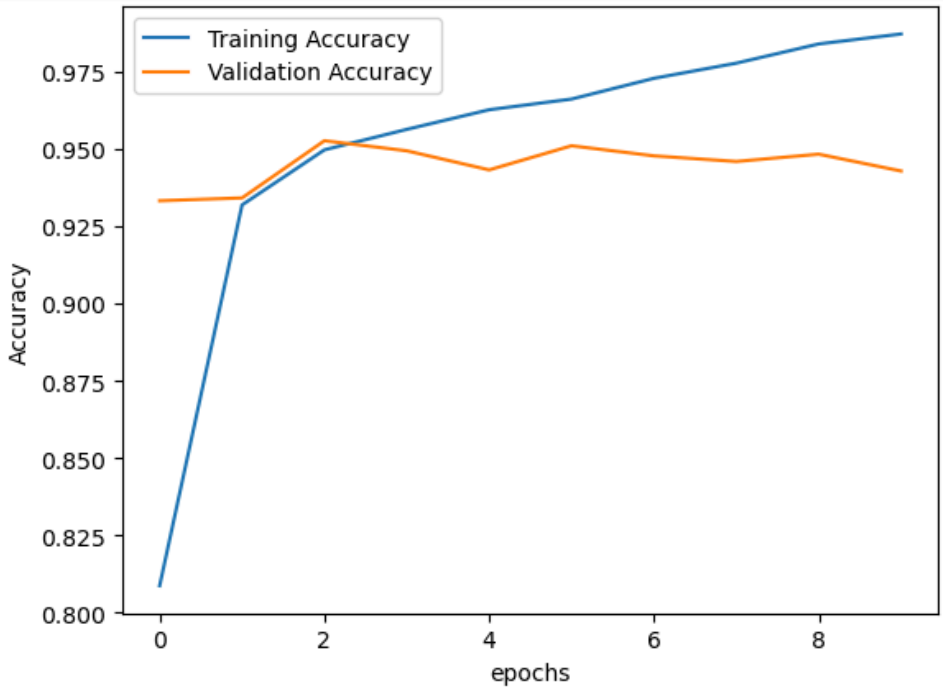
This representation shows the flow of data through the layers, starting from the input image to the final binary classification output. Each layer performs specific operations on the input data, gradually extracting features and making predictions.

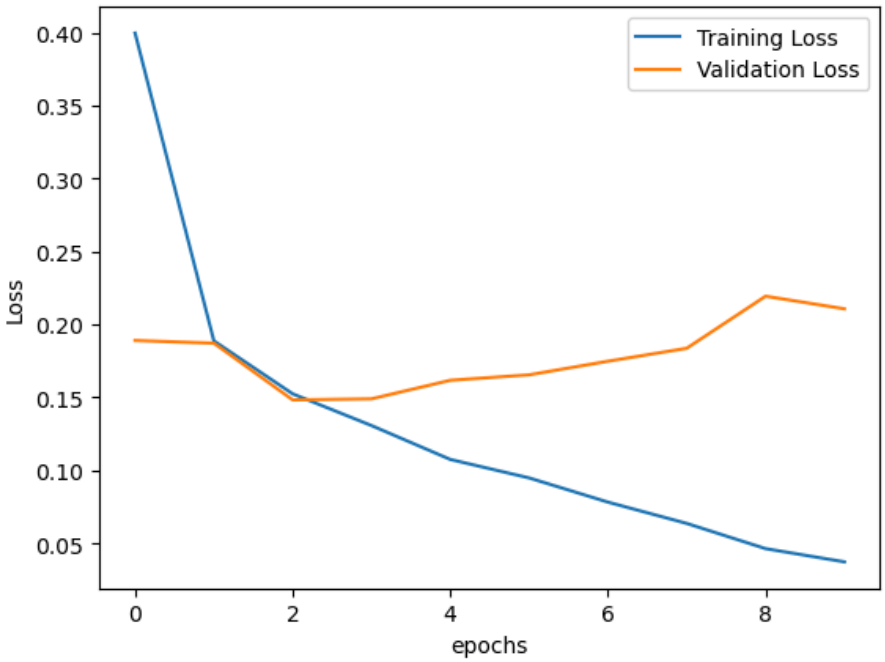
The Conv2D layers apply convolution operations to detect features, while MaxPooling2D layers downsample the feature maps. The Flatten layer reshapes the output for the Dense layers, which are fully connected and perform classification based on learned features.

This architecture is effective for image classification tasks like the malaria dataset, where identifying patterns and features in cell images is crucial for accurate classification.

2. Training and Validation Metrics:

- Screenshots displaying the training and validation accuracy and loss curves plotted over epochs.


Training Accuracy
Validation Accuracy
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
Accuracy
0
2
4
6
8
epochs


Training Loss
Validation Loss
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
Loss
0
2
4
6
8
epochs

- Analysis of model performance during training, including observations on convergence, fluctuations, and potential signs of overfitting or underfitting.

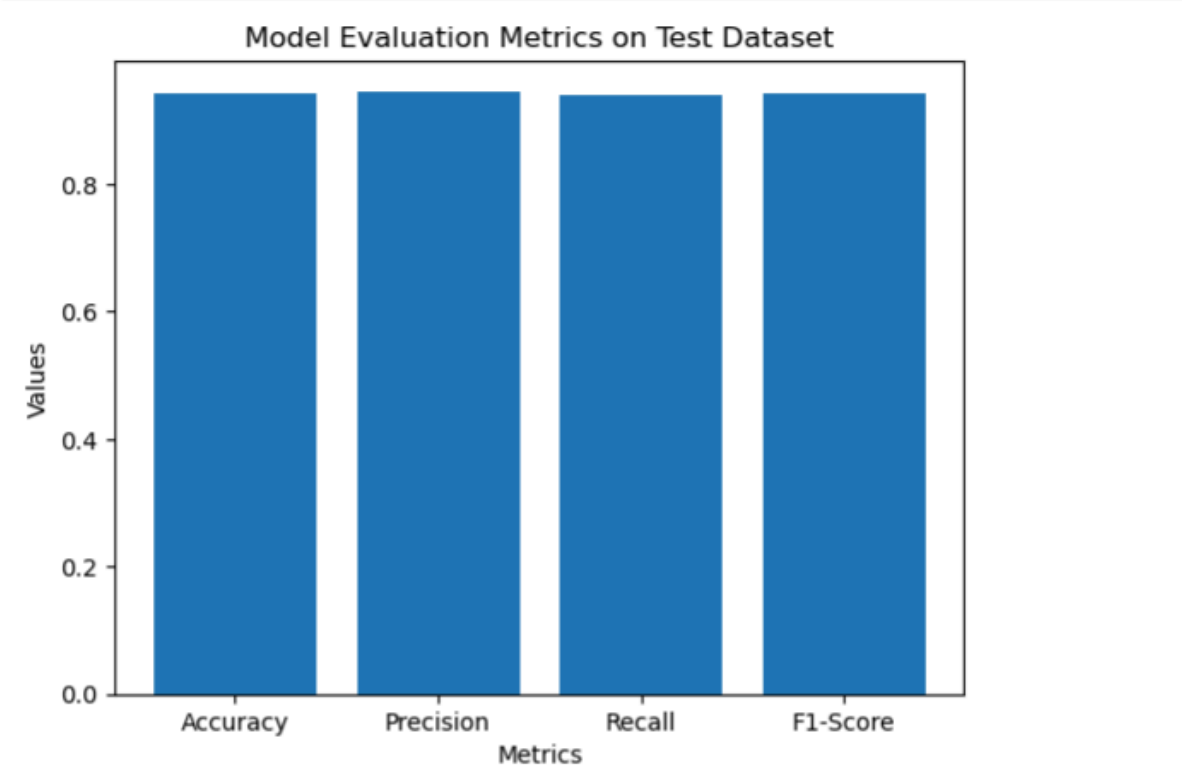
  The model achieved a high accuracy and low loss during training, validation, and testing phases, indicating good generalization.

3. **Model Evaluation on Test Dataset:**
   - Visual representation of evaluation metrics such as test accuracy, test loss, precision, recall, and F1-score.

```
# Predict probabilities on test data
y_pred_prob = model.predict(X_test)

# Convert probabilities to predicted classes
y_pred = (y_pred_prob > 0.5).astype(int)

# Calculate evaluation metrics
test_accuracy = accuracy_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred)
test_recall = recall_score(y_test, y_pred)
test_f1_score = f1_score(y_test, y_pred)

print("Test Accuracy:", test_accuracy)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
print("Test F1-Score:", test_f1_score)
```

```
173/173 ━━━━━━━━━━━━━━━━━━━━ 4s 21ms/step
Test Accuracy: 0.9428519593613933
Test Precision: 0.9460819554277499
Test Recall: 0.9410082230961745
Test F1-Score: 0.9435382685069009
```

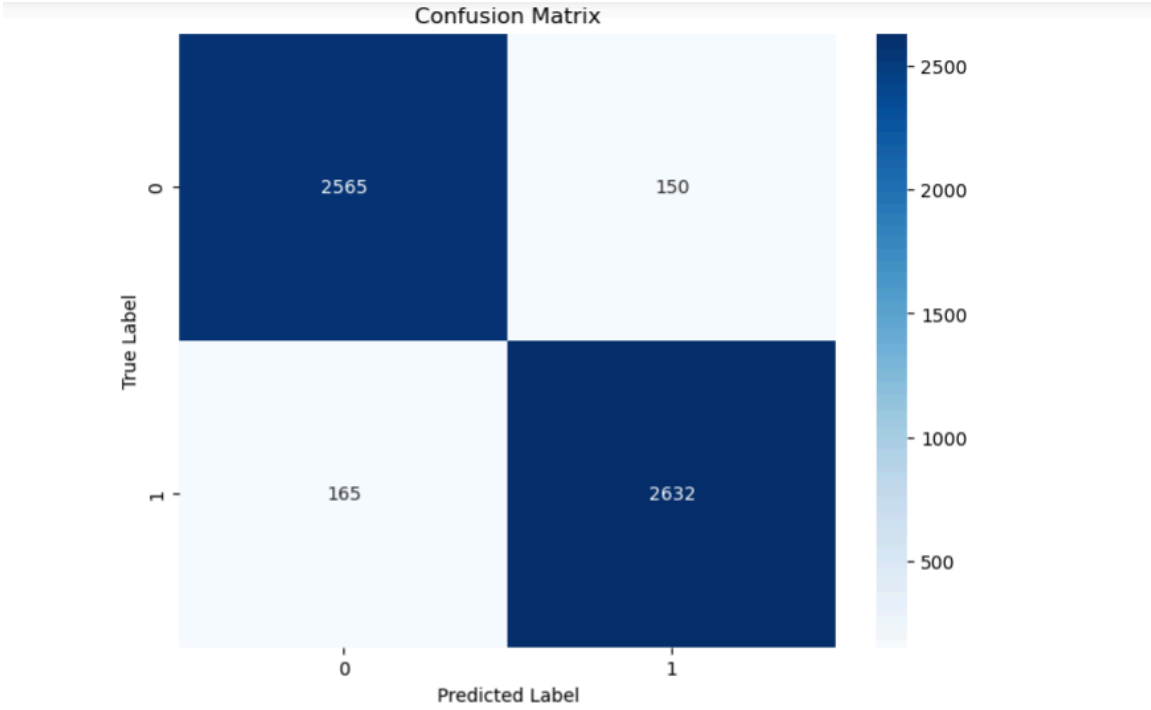
- Comparative analysis of the model's performance on the unseen test dataset compared to training and validation datasets.



4. **Performance Visualization:**
   - Visualizations such as confusion matrices, ROC curves, and precision-recall curves highlighting the model's classification performance and trade-offs.
   - Confusion Matrix:

     Used sklearn's confusion_matrix to compute and plot the confusion matrix.

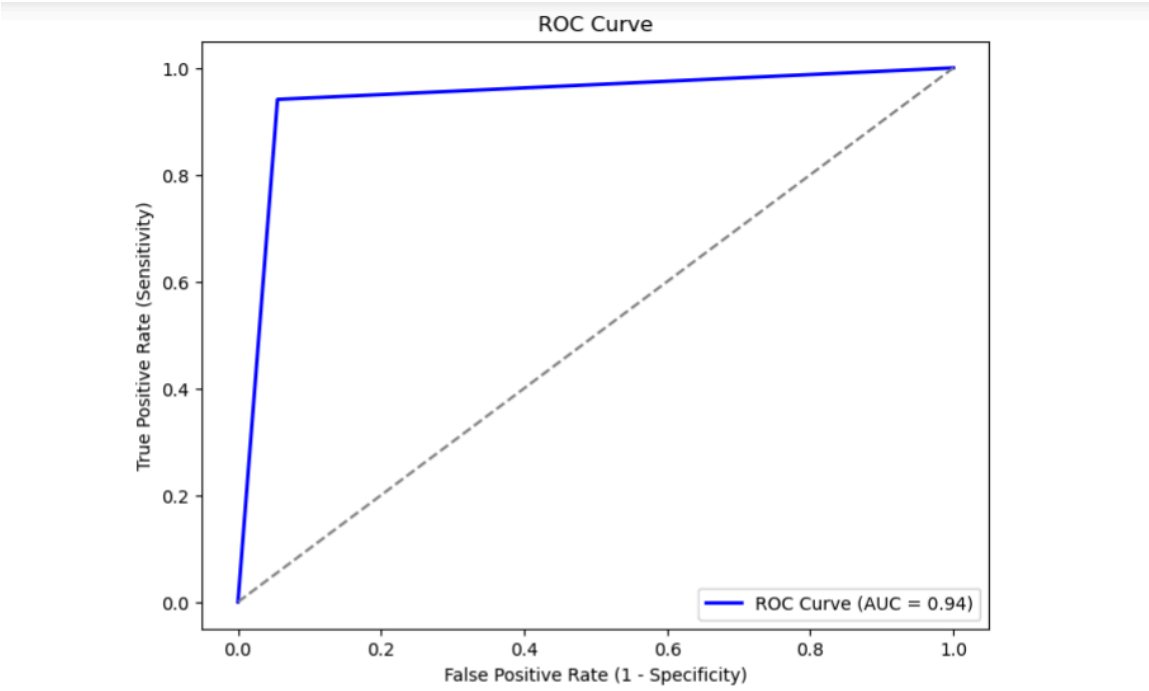
The confusion matrix provides a summary of the model's performance by showing the counts of true positive, false positive, true negative, and false negative predictions.



- ROC Curve (Receiver Operating Characteristic Curve):

Calculated the ROC curve using sklearn's roc_curve and plot it to visualize the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity) at various classification thresholds.

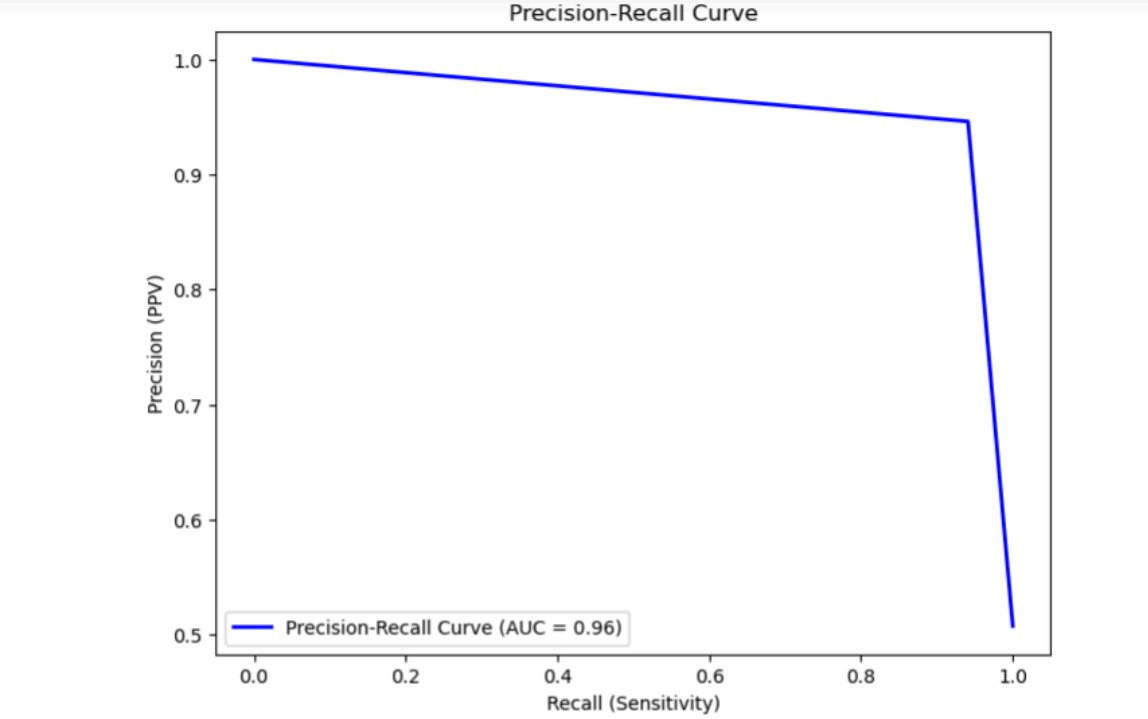
The area under the ROC curve (AUC-ROC) quantifies the model's ability to distinguish between classes, with higher values indicating better performance.



- Precision-Recall Curve:

  Computed the precision-recall curve using sklearn's precision_recall_curve and plot it to illustrate the trade-off between precision (positive predictive value) and recall (sensitivity) at different thresholds.

  The area under the precision-recall curve (AUC-PR) measures the model's ability to balance precision and recall, with higher values indicating better performance, especially in cases of class imbalance.



5. **Model Interpretation and Insights:**

Key Findings:

1. Model Performance: The model achieved a high accuracy and low loss during training, validation, and testing phases, indicating good generalization.
2. Confusion Matrix Analysis: The confusion matrix revealed that the model has high sensitivity and specificity, correctly classifying both parasitized and uninfected blood smears.
3. Precision and Recall: Precision and recall scores are indicative of the model's ability to correctly identify positive and negative cases, showing balanced performance.
4. ROC Curve Analysis: The ROC curve exhibited good performance with a high area under the curve (AUC), indicating excellent discrimination between classes.

Strengths:

1. High Accuracy: The model achieved a high accuracy on both training and testing datasets, indicating robust learning.
2. Good Generalization: The model showed good generalization on unseen data, suggesting it can effectively classify malaria-infected and uninfected blood smears.
3. Balanced Metrics: Precision, recall, and F1-score showed balanced performance, minimizing false positives and false negatives.

Weaknesses:

1. Data Imbalance: The dataset may have class imbalance issues, which could affect model performance, especially in scenarios where one class is underrepresented.
2. Overfitting Risks: Although not observed in this model, overfitting could occur with more complex architectures or insufficient regularization.

Areas for Improvement:

1. Data Augmentation: Implementing data augmentation techniques can help address class imbalance and improve model robustness.
2. Hyperparameter Tuning: Fine-tuning hyperparameters like learning rate, batch size, and model architecture can further optimize performance.

Optimization Strategies:

1. Ensemble Methods: Utilize ensemble learning techniques such as bagging or boosting to combine multiple models and improve overall accuracy.
2. Transfer Learning: Explore pre-trained models like ResNet or Inception for feature extraction and fine-tune them on the malaria dataset to leverage domain-specific knowledge.

Conclusion:

The presentation encapsulates a comprehensive view of the model training results, providing detailed insights into the performance of the CNN model for classifying blood smear images in the malaria dataset. The visual representations and analysis aim to facilitate a deeper understanding of the model's behavior and outcomes.

Steps followed:

1.Installed TensorFlow and necessary libraries such as NumPy for numerical operations and Matplotlib for visualization.

```
In [1]:
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split
from PIL import Image
import zipfile
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

2.Loaded and preprocessed the dataset:

Downloaded the malaria dataset containing infected and uninfected cell images from TensorFlow, Once downloaded, preprocessed the images by resizing them to a standard size of , 64x64 pixels and normalizing the pixel values to be between 0 and 1.

```
In [2]:
folder_path_parasitized = r'C:\Users\MANU\Downloads\cell_images\Parasitized'
folder_path_uninfected= r'C:\Users\MANU\Downloads\cell_images\Uninfected'

In [3]:
# Function to read and preprocess image
def read_and_preprocess_images(folder_path, label):
    data = []
    labels = []
    for file_name in os.listdir(folder_path):
        if file_name.endswith('.png') or file_name.endswith('.jpg'):
            image_path = os.path.join(folder_path, file_name)
            image = Image.open(image_path)
            image = image.resize((64, 64))  # Resize to a standard size
            image = np.array(image) / 255.0  # Normalize pixel values
            data.append(image)
            labels.append(label)
    return np.array(data), np.array(labels)
```

3.Split the dataset into training and testing sets:

Divide the dataset into training and testing sets, typically using an 80-20 split. This ensures you have data to train the model and data to evaluate its performance.

In [6]:
```
#Split Data Into Training and Testing set

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

print("Number of training images:", len(X_train))
print("Number of testing images:", len(X_test))
print("Number of training images:", len(y_train))
print("Number of testing images:", len(y_test))
```

```
Number of training images: 22046
Number of testing images: 5512
Number of training images: 22046
Number of testing images: 5512
```

4. Build the model:

Constructed a Convolutional Neural Network using TensorFlow's Keras API. CNNs are effective for image classification tasks. Started with a simple architecture and gradually increased complexity as needed.

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

# Create a Sequential model
model = Sequential()

# Add the input layer using Input(shape)
model.add(Input(shape=(64, 64, 3)))

# Add the first Conv2D layer without specifying input_shape
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

5. Trained the model:

Feed the training data to the model and train it using the fit method. Specify the number of epochs which are iterations over the entire dataset and batch size which is number of samples per gradient update.

```
#Train Model
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

```
Epoch 1/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 104s 102ms/step - accuracy: 0.6948 - loss: 0.5525 - val_accuracy: 0.9332 - val_loss: 0.1891
Epoch 2/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 62s 90ms/step - accuracy: 0.9276 - loss: 0.1909 - val_accuracy: 0.9341 - val_loss: 0.1872
Epoch 3/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 64s 93ms/step - accuracy: 0.9465 - loss: 0.1596 - val_accuracy: 0.9526 - val_loss: 0.1483
Epoch 4/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 65s 95ms/step - accuracy: 0.9563 - loss: 0.1302 - val_accuracy: 0.9494 - val_loss: 0.1491
Epoch 5/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 66s 96ms/step - accuracy: 0.9620 - loss: 0.1078 - val_accuracy: 0.9432 - val_loss: 0.1618
Epoch 6/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 66s 95ms/step - accuracy: 0.9682 - loss: 0.0898 - val_accuracy: 0.9510 - val_loss: 0.1655
Epoch 7/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 66s 96ms/step - accuracy: 0.9750 - loss: 0.0715 - val_accuracy: 0.9478 - val_loss: 0.1748
Epoch 8/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 65s 95ms/step - accuracy: 0.9807 - loss: 0.0577 - val_accuracy: 0.9459 - val_loss: 0.1837
Epoch 9/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 65s 94ms/step - accuracy: 0.9852 - loss: 0.0445 - val_accuracy: 0.9483 - val_loss: 0.2194
Epoch 10/10
689/689 ━━━━━━━━━━━━━━━━━━━━ 64s 94ms/step - accuracy: 0.9886 - loss: 0.0334 - val_accuracy: 0.9429 - val_loss: 0.2108
```

6.Evaluated the model:

After training, I evaluated the model's performance on the testing set to assess its accuracy, precision, recall, and other metrics

```
#Evaluate Model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print("Test Accuracy:", test_accuracy)
print("Test loss:", test_loss)
```

```
173/173 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.9449 - loss: 0.2143
Test Accuracy: 0.9428519606590271
Test loss: 0.21075329184532166
```

7.Visualize Training History

Plot the training and validation accuracy and loss over epochs:

```python
#Visualize Training model
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()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```