Specific Test V. Exploring Transformers

Task: Use a vision transformer method of your choice to build a robust and efficient model for binary classification or unsupervised anomaly detection on the provided dataset. In the case of unsupervised anomaly detection, train your model to learn the distribution of the provided strong lensing images with no substructure. Please implement your approach in PyTorch or Keras and discuss your strategy.

Dataset: https://drive.google.com/file/d/16Y1taQoTeUTP5rGpB0tuPZ_S30acvnqr/view?usp=sharing

Dataset Description: A set of simulated strong gravitational lensing images with and without substructure.

Evaluation Metrics: ROC curve (Receiver Operating Characteristic curve) and AUC score (Area Under the ROC Curve)

Downloading the data

```
from google.colab import drive
drive.mount('/content/gdrive')
!tar --extract --file '/content/gdrive/MyDrive/lenses.tgz'
print('Extraction done.')
```

```
Mounted at /content/gdrive
Extraction done.
```

Setting up the imports:

```
pip install -U tensorflow-addons
```

```
Requirement already satisfied: tensorflow-addons in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: typeguard>=2.7 in /usr/local/lib/python3.7/dist-packages
```

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import auc, roc_curve
import tensorflow_addons as tfa
```

Extracting the data from the lense images:

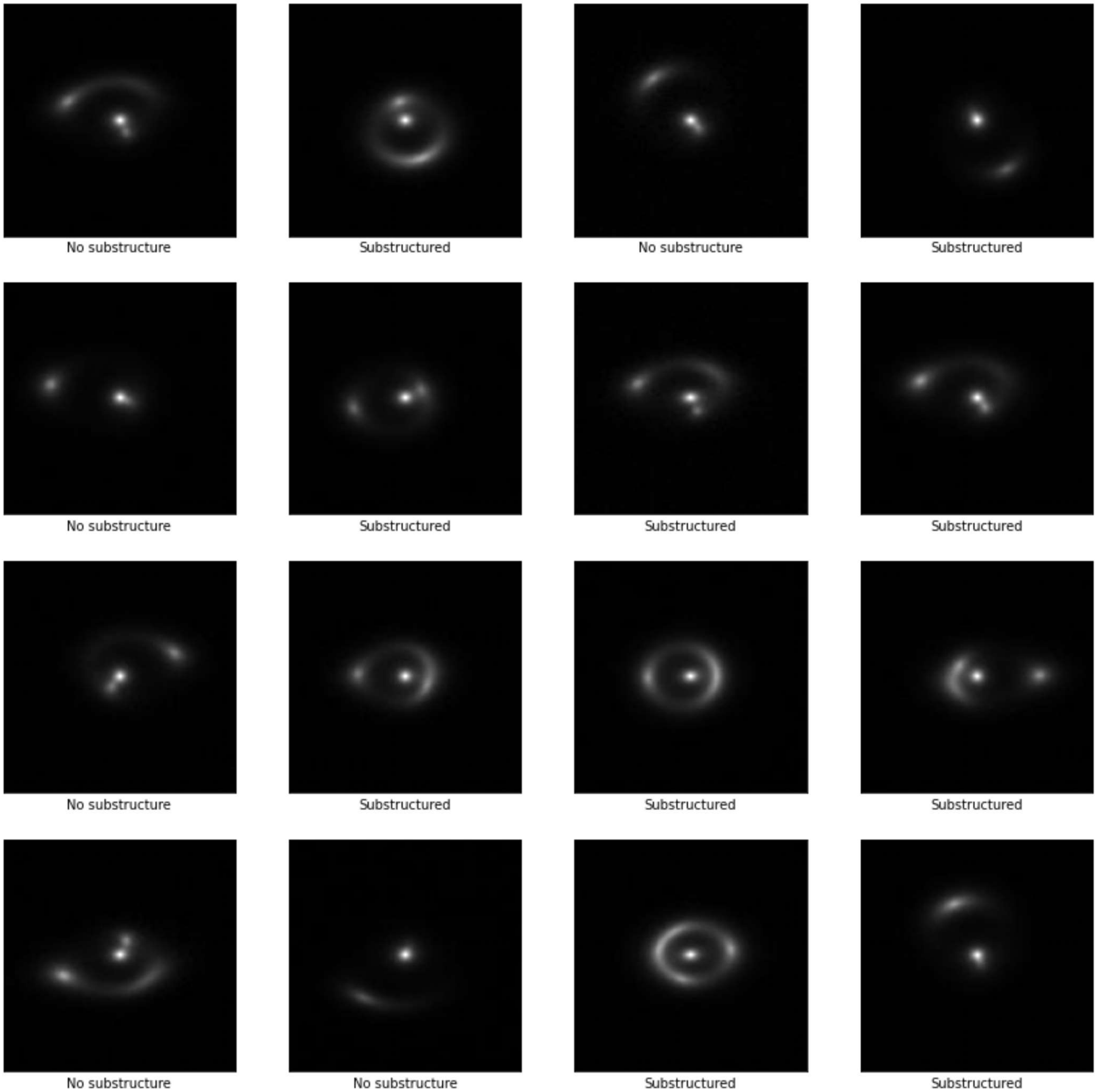
```
X_data = []
Y_data = []

#substructure data
sub = os.listdir('/content/lenses/sub')
for i in sub:
    img = cv2.imread('/content/lenses/sub/' + i)
    img = img / 255.0
    X_data.append(img)
    Y_data.append(1)

#no-substructure data
no_sub = os.listdir('/content/lenses/no_sub')
for i in no_sub:
    img = cv2.imread('/content/lenses/no_sub/' + i)
    img = img / 255.0
    X_data.append(img)
    Y_data.append(0)
```

Shuffling to introduce randomness in the data:

```
data = list(zip(X_data, Y_data))
np.random.shuffle(data)
X_data, Y_data = zip(*data)

#delete to free redundant space
del data

X_data = np.array(X_data)
Y_data = np.array(Y_data)
```

Visualising the images belonging to the two classes:

```
classes = ["Substructured", "No substructure"]
plt.figure(figsize=(15, 15))
for i in range(16):
    plt.subplot(4, 4, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    num = np.random.randint(0, len(X_data))
    plt.imshow(X_data[num])
    plt.xlabel(classes[Y_data[num]])
plt.show()
```



Splitting the data into training and validation:

```
X_train , X_test, Y_train, Y_test = train_test_split(X_data, Y_data, test_size=0.2, random_st
X_data.shape, Y_data.shape
```

```
((10000, 150, 150, 3), (10000,))
```

Deleting large variables to free up memory:

```
del X_data, Y_data
del img, no_sub, sub
```

Configuring the hyperparameters

```
num_classes = 2
input_shape = (150, 150, 3)
learning_rate = 0.001
weight_decay = 0.0001
batch_size = 32
num_epochs = 40
image_size = 72  # We'll resize input images to this size
patch_size = 6  # Size of the patches to be extract from the input images
num_patches = (image_size // patch_size) ** 2
projection_dim = 64
num_heads = 4
transformer_units = [
    projection_dim * 2,
    projection_dim,
]  # Size of the transformer layers
transformer_layers = 8
mlp_head_units = [2048, 1024]  # Size of the dense layers of the final classifier
```

Defining data augmentations:

```
data_augmentation = tf.keras.Sequential(
    [
        tf.keras.layers.Normalization(),
        tf.keras.layers.Resizing(image_size, image_size),
        tf.keras.layers.RandomFlip("horizontal"),
        tf.keras.layers.RandomRotation(factor=0.02),
        tf.keras.layers.RandomZoom(
            height_factor=0.2, width_factor=0.2
        ),
    ],
    name="data_augmentation",
)
# Compute the mean and the variance of the training data for normalization.
data_augmentation.layers[0].adapt(X_train)
```

Implementing multilayer perceptron (MLP):

```
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = tf.keras.layers.Dense(units, activation=tf.nn.gelu)(x)
        x = tf.keras.layers.Dropout(dropout_rate)(x)
    return x
```

Implementing patch creation as a layer:

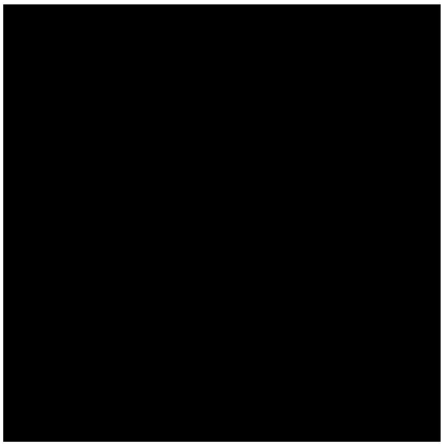
```
class Patches(tf.keras.layers.Layer):
    def __init__(self, patch_size):
        super(Patches, self).__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches
```

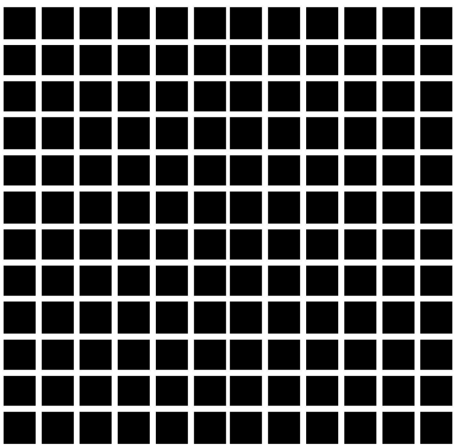
Displaying patches for a sample image:

```
plt.figure(figsize=(4, 4))
image = X_train[np.random.choice(range(X_train.shape[0]))]
plt.imshow(image.astype("uint8"))
plt.axis("off")

resized_image = tf.image.resize(
    tf.convert_to_tensor([image]), size=(image_size, image_size)
)
patches = Patches(patch_size)(resized_image)
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size} X {patch_size}")
print(f"Patches per image: {patches.shape[1]}")
print(f"Elements per patch: {patches.shape[-1]}")

n = int(np.sqrt(patches.shape[1]))
plt.figure(figsize=(4, 4))
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
```

```
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch_img.numpy().astype("uint8"))
    plt.axis("off")
```

```
Image size: 72 X 72
Patch size: 6 X 6
Patches per image: 144
Elements per patch: 108
```





Implementing the patch encoding layer:

```
class PatchEncoder(tf.keras.layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super(PatchEncoder, self).__init__()
        self.num_patches = num_patches
        self.projection = tf.keras.layers.Dense(units=projection_dim)
        self.position_embedding = tf.keras.layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded
```

Defining the Vision Transformer (ViT) model:

```
def create_vit_classifier():
    inputs = tf.keras.layers.Input(shape=input_shape)
    # Augment data.
    augmented = data_augmentation(inputs)
    # Create patches.
    patches = Patches(patch_size)(augmented)
    # Encode patches.
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.
        attention_output = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        x2 = tf.keras.layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        x3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(x2)
        # MLP.
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = tf.keras.layers.Add()([x3, x2])

    # Create a [batch_size, projection_dim] tensor.
    representation = tf.keras.layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = tf.keras.layers.Flatten()(representation)
    representation = tf.keras.layers.Dropout(0.5)(representation)
    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.5)
    # Classify outputs.
    logits = tf.keras.layers.Dense(num_classes)(features)
    # Create the Keras model.
    model = tf.keras.Model(inputs=inputs, outputs=logits)
    return model
```

Compiling, training and evaluating the model:

```
def run_experiment(model):
    optimizer = tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    )

    model.compile(
        optimizer=optimizer,
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
```

```
        metrics=[
            tf.keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
            tf.keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-5-accuracy"),
        ],
    )

    checkpoint_filepath = "/tmp/checkpoint"
    checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        checkpoint_filepath,
        monitor="val_accuracy",
        save_best_only=True,
        save_weights_only=True,
    )

    history = model.fit(
        x=X_train,
        y=Y_train,
        batch_size=batch_size,
        epochs=num_epochs,
        validation_split=0.1,
        callbacks=[checkpoint_callback],
    )

    model.load_weights(checkpoint_filepath)
    _, accuracy, top_5_accuracy = model.evaluate(X_test, Y_test)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    print(f"Test top 5 accuracy: {round(top_5_accuracy * 100, 2)}%")

    return history


vit_classifier = create_vit_classifier()
history = run_experiment(vit_classifier)
```

```
Epoch 2/40
225/225 [==============================] - 14s 63ms/step - loss: 0.7154 - accuracy: 0
Epoch 3/40
225/225 [==============================] - 15s 68ms/step - loss: 0.7099 - accuracy: 0
Epoch 4/40
225/225 [==============================] - 15s 68ms/step - loss: 0.7077 - accuracy: 0
Epoch 5/40
225/225 [==============================] - 14s 63ms/step - loss: 0.6977 - accuracy: 0
Epoch 6/40
225/225 [==============================] - 15s 68ms/step - loss: 0.6939 - accuracy: 0
Epoch 7/40
225/225 [==============================] - 15s 68ms/step - loss: 0.6920 - accuracy: 0
Epoch 8/40
225/225 [==============================] - 14s 64ms/step - loss: 0.6687 - accuracy: 0
Epoch 9/40
225/225 [==============================] - 14s 64ms/step - loss: 0.6514 - accuracy: 0
Epoch 10/40
225/225 [==============================] - 14s 63ms/step - loss: 0.6526 - accuracy: 0
Epoch 11/40
```

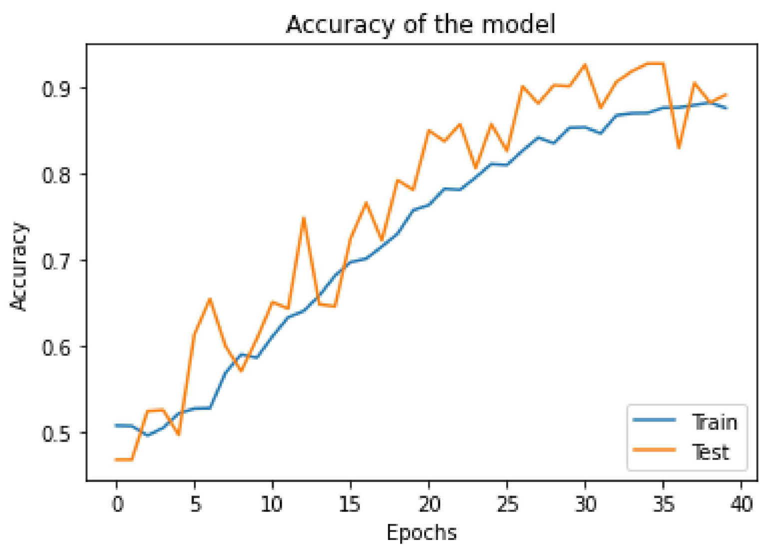
```
225/225 [==============================] - 14s 64ms/step - loss: 0.6191 - accuracy: 0
Epoch 12/40
225/225 [==============================] - 14s 63ms/step - loss: 0.5998 - accuracy: 0
Epoch 13/40
225/225 [==============================] - 15s 67ms/step - loss: 0.5888 - accuracy: 0
Epoch 14/40
225/225 [==============================] - 14s 64ms/step - loss: 0.5712 - accuracy: 0
Epoch 15/40
225/225 [==============================] - 14s 63ms/step - loss: 0.5529 - accuracy: 0
Epoch 16/40
225/225 [==============================] - 14s 64ms/step - loss: 0.5236 - accuracy: 0
Epoch 17/40
225/225 [==============================] - 15s 69ms/step - loss: 0.5260 - accuracy: 0
Epoch 18/40
225/225 [==============================] - 14s 64ms/step - loss: 0.5000 - accuracy: 0
Epoch 19/40
225/225 [==============================] - 15s 68ms/step - loss: 0.4936 - accuracy: 0
Epoch 20/40
225/225 [==============================] - 14s 63ms/step - loss: 0.4609 - accuracy: 0
Epoch 21/40
225/225 [==============================] - 15s 68ms/step - loss: 0.4511 - accuracy: 0
Epoch 22/40
225/225 [==============================] - 14s 64ms/step - loss: 0.4205 - accuracy: 0
Epoch 23/40
225/225 [==============================] - 15s 68ms/step - loss: 0.4193 - accuracy: 0
Epoch 24/40
225/225 [==============================] - 14s 64ms/step - loss: 0.3976 - accuracy: 0
Epoch 25/40
225/225 [==============================] - 14s 63ms/step - loss: 0.3943 - accuracy: 0
Epoch 26/40
225/225 [==============================] - 14s 64ms/step - loss: 0.3876 - accuracy: 0
Epoch 27/40
225/225 [==============================] - 15s 68ms/step - loss: 0.3587 - accuracy: 0
Epoch 28/40
225/225 [==============================] - 14s 64ms/step - loss: 0.3428 - accuracy: 0
Epoch 29/40
225/225 [==============================] - 15s 68ms/step - loss: 0.3474 - accuracy: 0
Epoch 30/40
```

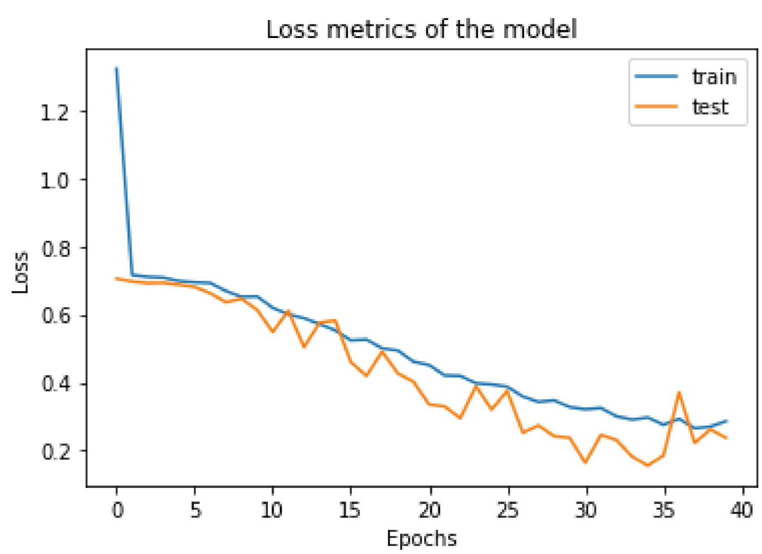
Predict on the validation data and load the best saved weight:

```
predictions = vit_classifier.predict(X_test)
temp_predictions = []
for i in range(len(predictions)):
    k = np.argmax(predictions[i])
    temp_predictions.append(k)

temp_predictions = np.array(temp_predictions)
```

Plotting accuracy and loss curves:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Accuracy of the model')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(['Train', 'Test'], loc='lower right')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss metrics of the model')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```

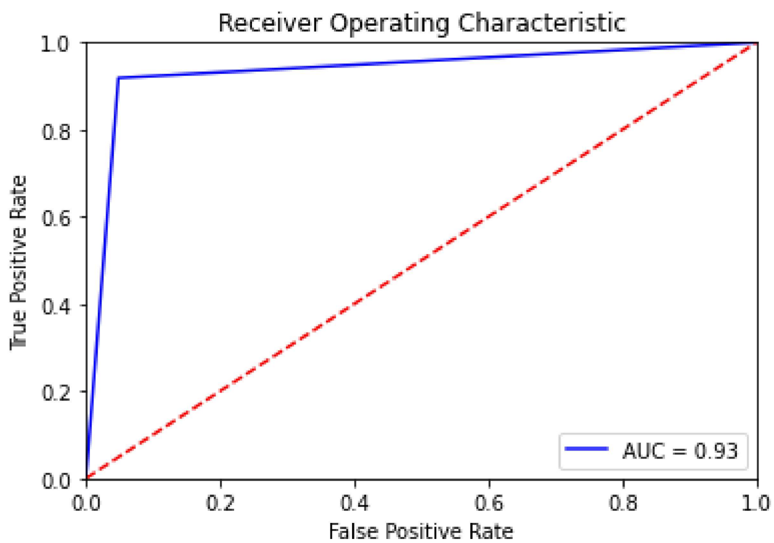




Plotting the ROC AUC curve:

```
fpr, tpr, thresholds = roc_curve(Y_test, temp_predictions)
```

```
roc_auc = auc(fpr, tpr)
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



✓ 0s completed at 1:46 AM