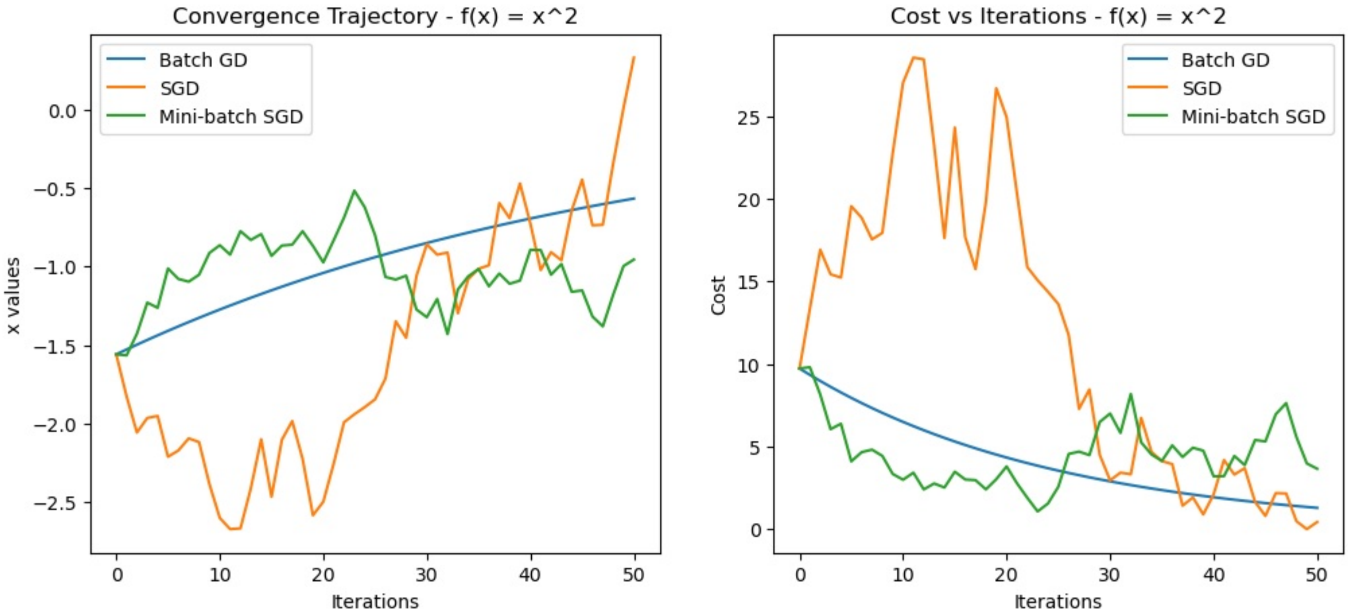
Deep Learning Lab Experiment No. 3

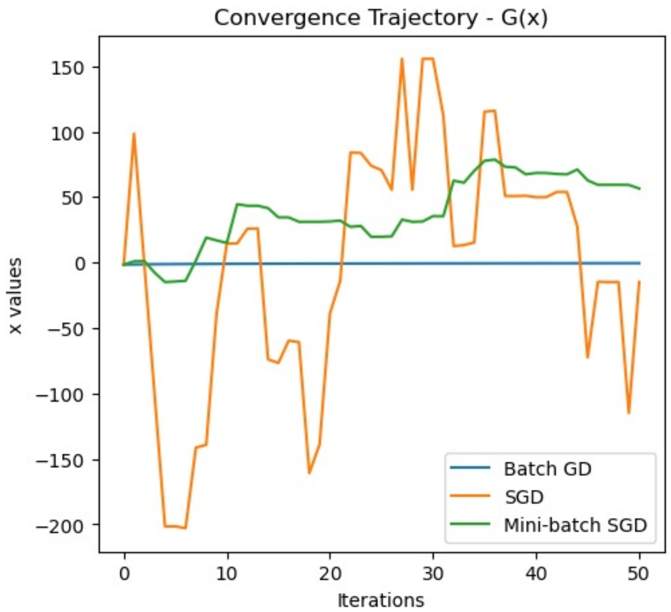
# 1. Demonstrate the behavior of Gradient Descent and its variants —Batch Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Mini-batch Stochastic Gradient Descent (Mini-batch SGD) on both simple and complex functions.

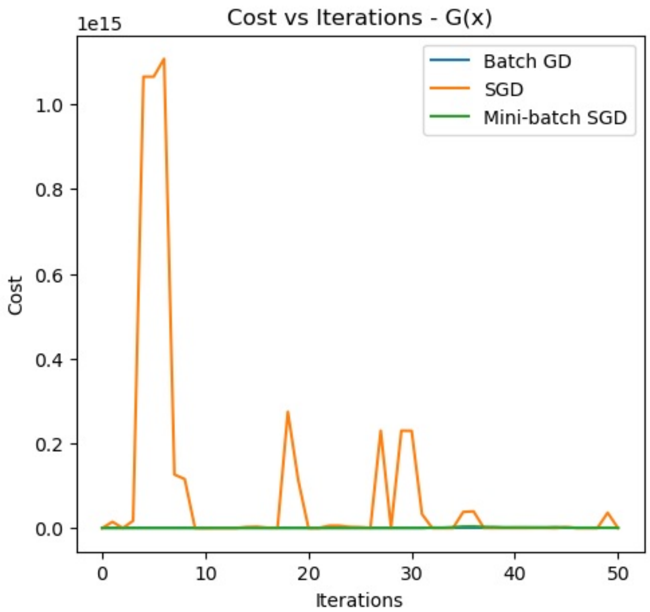
In [1]:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the functions and their gradients
def f(x):
    return x**2

def df(x):
    return 2*x

def G(x):
    return x**4 + x**3 + x**2

def dG(x):
    return 4*x**3 + 3*x**2 + 2*x

# Gradient Descent Implementation with overflow handling
def gradient_descent(grad, x_init, lr=0.01, epochs=50, method='batch', batch_size=5, max_value=1e4):
    x = x_init
    trajectory = [x]
    cost = [grad(x)**2]

    for _ in range(epochs):
        if method == 'sgd':
            x_sample = np.random.uniform(-20, 20)  # Random sample for SGD
        elif method == 'mini-batch':
            x_sample = np.mean(np.random.uniform(-20, 20, batch_size))  # Random batch for Mini-batch
        else:
            x_sample = x  # Batch GD uses the full dataset

        gradient = grad(x_sample)

        # Prevent overflow by clamping gradient
        if abs(gradient) > max_value:
            gradient = np.sign(gradient) * max_value

        x -= lr * gradient

        # Prevent runaway values
        if abs(x) > max_value:
            print(f"Warning: Value exceeded {max_value}, stopping optimization.")
            break

        trajectory.append(x)
        cost.append(grad(x)**2)

    return x, trajectory, cost

# Initialize parameters
x_init = np.random.uniform(-20, 20)
learning_rate = 0.01
epochs = 50
batch_size = 5

# Run Batch Gradient Descent for f(x) and G(x)
x_final_batch_f, traj_batch_f, cost_batch_f = gradient_descent(df, x_init, learning_rate, epochs, method='batch
x_final_batch_G, traj_batch_G, cost_batch_G = gradient_descent(dG, x_init, learning_rate, epochs, method='batch

# Run Stochastic Gradient Descent (SGD)
x_final_sgd_f, traj_sgd_f, cost_sgd_f = gradient_descent(df, x_init, learning_rate, epochs, method='sgd')
x_final_sgd_G, traj_sgd_G, cost_sgd_G = gradient_descent(dG, x_init, learning_rate, epochs, method='sgd')

# Run Mini-batch Gradient Descent
x_final_mini_f, traj_mini_f, cost_mini_f = gradient_descent(df, x_init, learning_rate, epochs, method='mini-bat
x_final_mini_G, traj_mini_G, cost_mini_G = gradient_descent(dG, x_init, learning_rate, epochs, method='mini-bat

# Plot convergence trajectories for f(x) = x^2
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(traj_batch_f, label="Batch GD")
plt.plot(traj_sgd_f, label="SGD")
plt.plot(traj_mini_f, label="Mini-batch SGD")
```

```python
plt.title("Convergence Trajectory - f(x) = x^2")
plt.xlabel("Iterations")
plt.ylabel("x values")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(cost_batch_f, label="Batch GD")
plt.plot(cost_sgd_f, label="SGD")
plt.plot(cost_mini_f, label="Mini-batch SGD")
plt.title("Cost vs Iterations - f(x) = x^2")
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.legend()

plt.show()

# Plot convergence trajectories for G(x)
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(traj_batch_G, label="Batch GD")
plt.plot(traj_sgd_G, label="SGD")
plt.plot(traj_mini_G, label="Mini-batch SGD")
plt.title("Convergence Trajectory - G(x)")
plt.xlabel("Iterations")
plt.ylabel("x values")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(cost_batch_G, label="Batch GD")
plt.plot(cost_sgd_G, label="SGD")
plt.plot(cost_mini_G, label="Mini-batch SGD")
plt.title("Cost vs Iterations - G(x)")
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.legend()

plt.show()

# Print final optimal values
print("Final optimal values for f(x) = x^2:")
print(f"Batch GD: {x_final_batch_f:.4f}, SGD: {x_final_sgd_f:.4f}, Mini-batch SGD: {x_final_mini_f:.4f}")

print("\nFinal optimal values for G(x) = x^4 + x^3 + x^2:")
print(f"Batch GD: {x_final_batch_G:.4f}, SGD: {x_final_sgd_G:.4f}, Mini-batch SGD: {x_final_mini_G:.4f}")
```





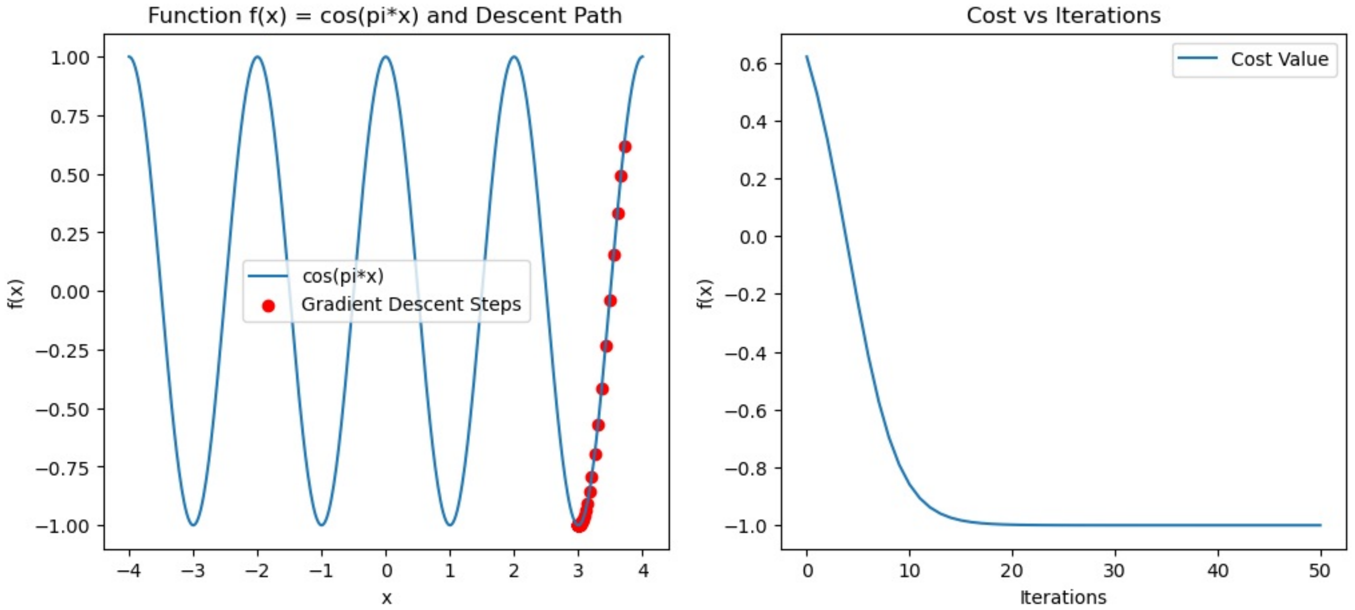

```
Final optimal values for f(x) = x^2:
Batch GD: -0.5678, SGD: 0.3290, Mini-batch SGD: -0.9558

Final optimal values for G(x) = x^4 + x^3 + x^2:
Batch GD: -0.4629, SGD: -14.8585, Mini-batch SGD: 56.5782
```

# 2. Consider this function also cos(pi.x) compute for x -4 to 4

In [3]:

```
# Given function and its derivative
def f(x):
    return np.cos(np.pi * x)

def df(x):
    return -2 * np.pi * np.sin(np.pi * x)

# Gradient Descent function
def gradient_descent(grad, x_init, lr=0.01, epochs=50):
    x = x_init
    trajectory = [x]
    cost = [f(x)]

    for _ in range(epochs):
        x -= lr * grad(x)
        trajectory.append(x)
        cost.append(f(x))

    return x, trajectory, cost

# Initial point and parameters
x_init = np.random.uniform(-4, 4)
learning_rate = 0.01
epochs = 50

# Perform gradient descent
x_final, x_trajectory, cost_trajectory = gradient_descent(df, x_init, learning_rate, epochs)

# Plot the results
x_vals = np.linspace(-4, 4, 400)
y_vals = f(x_vals)

plt.figure(figsize=(12, 5))

# Function plot
```

```python
plt.subplot(1, 2, 1)
plt.plot(x_vals, y_vals, label="cos(pi*x)")
plt.scatter(x_trajectory, f(np.array(x_trajectory)), color='red', label="Gradient Descent Steps")
plt.title("Function f(x) = cos(pi*x) and Descent Path")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()

# Cost plot
plt.subplot(1, 2, 2)
plt.plot(cost_trajectory, label="Cost Value")
plt.title("Cost vs Iterations")
plt.xlabel("Iterations")
plt.ylabel("f(x)")
plt.legend()

plt.show()

# Print final optimized x value
print(f"Initial x: {x_init:.4f}")
print(f"Final optimized x: {x_final:.4f}")
print(f"Final function value: {f(x_final):.4f}")
```




```
Initial x: 3.7129
Final optimized x: 3.0000
Final function value: -1.0000
```

# 3. Develop and train a deep neural network to predict the onset of diabetes using the Pima Indians Diabetes dataset.

In [18]:

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE

# Load dataset
data = pd.read_csv('diabetes.csv')

# Handle missing values and anomalies
columns_with_zeros = ["Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI"]
for col in columns_with_zeros:
    median = data[col].median()
    data[col] = data[col].replace(0, median)

# Handle class imbalance
X = data.drop("Outcome", axis=1)
y = data["Outcome"]
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Apply feature engineering (scaling features)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_resampled)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_resampled, test_size=0.2, random_state=42)
```

```python
# Summary of preprocessing
print("Data preprocessing complete!")
print(f"Training set size: {X_train.shape[0]} samples")
print(f"Test set size: {X_test.shape[0]} samples")
```

```
Data preprocessing complete!
Training set size: 800 samples
Test set size: 200 samples
```

In [27]:

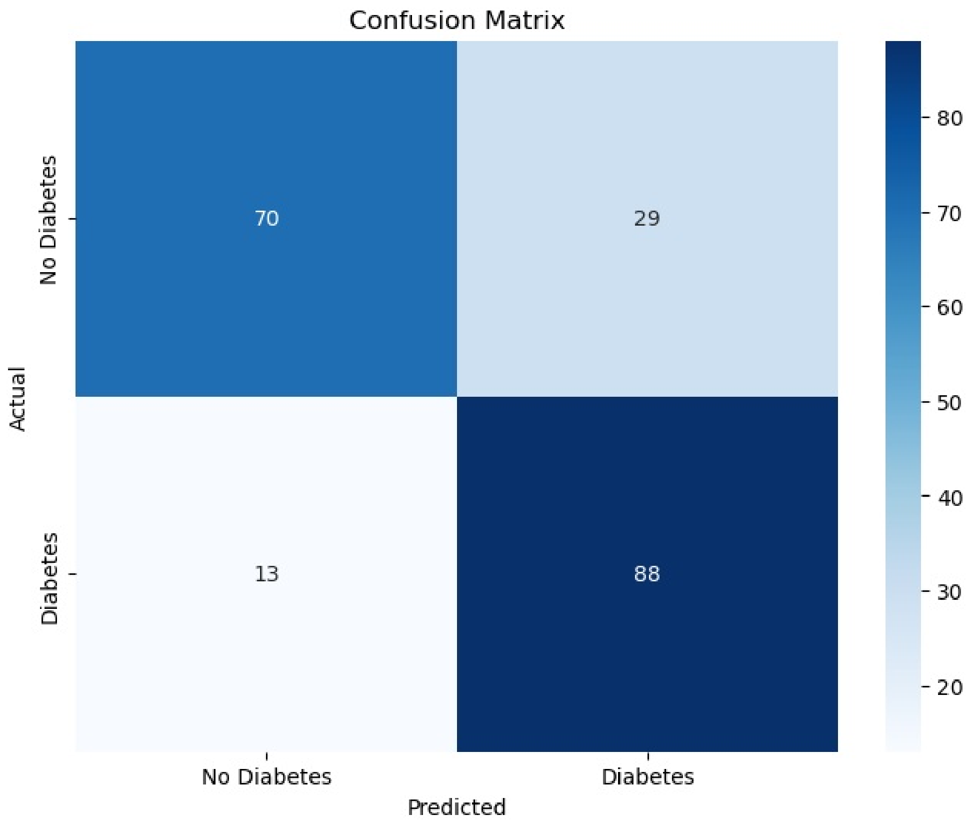
```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
                             confusion_matrix, roc_curve, auc)
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping

# Load dataset
data = pd.read_csv('diabetes.csv')

# Handle missing values and anomalies
columns_with_zeros = ["Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI"]
for col in columns_with_zeros:
    median = data[col].median()
    data[col] = data[col].replace(0, median)

# Handle class imbalance
X = data.drop("Outcome", axis=1)
y = data["Outcome"]
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Apply feature engineering (scaling features)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_resampled)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_resampled, test_size=0.2, random_state=42)

# Build a Deep Neural Network model
model = Sequential([
    Dense(64, input_dim=X_train.shape[1], activation='relu'),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Train the model
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=100,
                    batch_size=32,
                    callbacks=[early_stopping],
                    verbose=1)

# Evaluate the model
eval_results = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {eval_results[0]:.4f}")
print(f"Test Accuracy: {eval_results[1]:.4f}")

# Generate classification metrics
y_pred = (model.predict(X_test) > 0.5).astype("int32")
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-score: {f1:.2f}")

# Confusion Matrix
```

```python
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=["No Diabetes", "Diabetes"],
            yticklabels=["No Diabetes", "Diabetes"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# ROC Curve
y_pred_proba = model.predict(X_test).ravel()
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color="blue", label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], color="gray", linestyle="--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic (ROC) Curve")
plt.legend()
plt.show()

# Plot Training Curves
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()

plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()
```

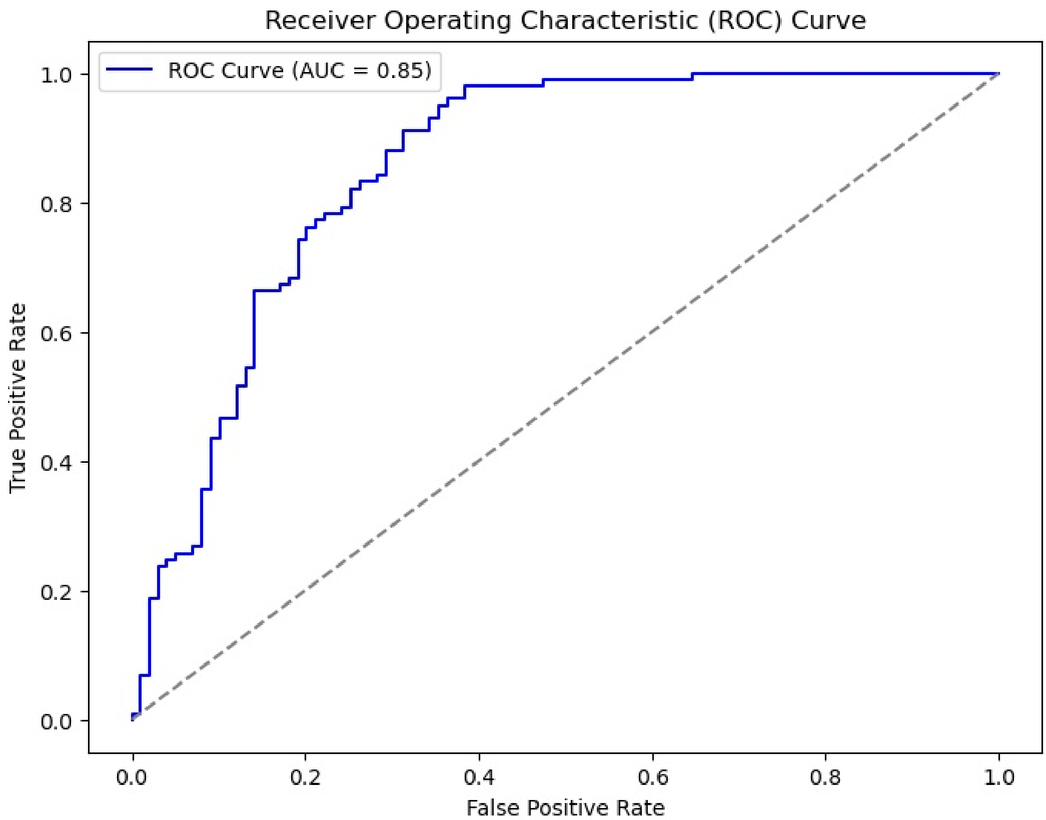
```
Epoch 1/100
C:\Users\shubh\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input
_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as
the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 27ms/step - accuracy: 0.4907 - loss: 0.7101 - val_accuracy: 0.7000 - val_loss: 0.6
274
Epoch 2/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.6713 - loss: 0.6269 - val_accuracy: 0.7200 - val_loss: 0.5
625
Epoch 3/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.7274 - loss: 0.5677 - val_accuracy: 0.7550 - val_loss: 0.5
254
Epoch 4/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.7290 - loss: 0.5526 - val_accuracy: 0.7350 - val_loss: 0.5
059
Epoch 5/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7466 - loss: 0.5164 - val_accuracy: 0.7400 - val_loss: 0.4
954
Epoch 6/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7411 - loss: 0.5163 - val_accuracy: 0.7400 - val_loss: 0.4
901
Epoch 7/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7301 - loss: 0.5148 - val_accuracy: 0.7550 - val_loss: 0.4
865
Epoch 8/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7557 - loss: 0.4934 - val_accuracy: 0.7600 - val_loss: 0.4
836
Epoch 9/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7375 - loss: 0.4913 - val_accuracy: 0.7700 - val_loss: 0.4
796
Epoch 10/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7779 - loss: 0.4803 - val_accuracy: 0.7750 - val_loss: 0.4
788
Epoch 11/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.7685 - loss: 0.4972 - val_accuracy: 0.7700 - val_loss: 0.47
74
Epoch 12/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7609 - loss: 0.4782 - val_accuracy: 0.7700 - val_loss: 0.4
768
Epoch 13/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7844 - loss: 0.4770 - val_accuracy: 0.7600 - val_loss: 0.4
```

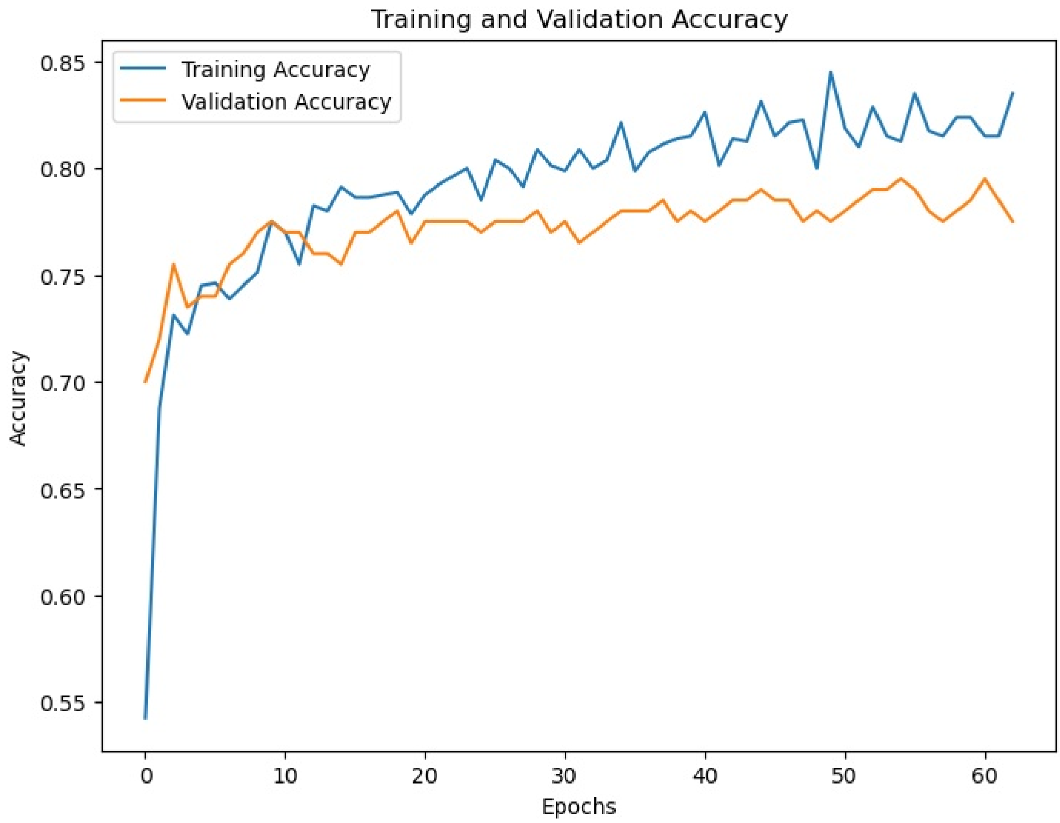
```
777
Epoch 14/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7924 - loss: 0.4544 - val_accuracy: 0.7600 - val_loss: 0.4
757
Epoch 15/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8073 - loss: 0.4477 - val_accuracy: 0.7550 - val_loss: 0.4
764
Epoch 16/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7720 - loss: 0.4707 - val_accuracy: 0.7700 - val_loss: 0.4
766
Epoch 17/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7943 - loss: 0.4652 - val_accuracy: 0.7700 - val_loss: 0.4
750
Epoch 18/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7897 - loss: 0.4741 - val_accuracy: 0.7750 - val_loss: 0.4
738
Epoch 19/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7727 - loss: 0.4583 - val_accuracy: 0.7800 - val_loss: 0.4
727
Epoch 20/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7711 - loss: 0.4764 - val_accuracy: 0.7650 - val_loss: 0.4
740
Epoch 21/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7729 - loss: 0.4549 - val_accuracy: 0.7750 - val_loss: 0.4
717
Epoch 22/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.7959 - loss: 0.4283 - val_accuracy: 0.7750 - val_loss: 0.4
708
Epoch 23/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7713 - loss: 0.4730 - val_accuracy: 0.7750 - val_loss: 0.4
695
Epoch 24/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8091 - loss: 0.4156 - val_accuracy: 0.7750 - val_loss: 0.4
682
Epoch 25/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8069 - loss: 0.4392 - val_accuracy: 0.7700 - val_loss: 0.4
697
Epoch 26/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7907 - loss: 0.4382 - val_accuracy: 0.7750 - val_loss: 0.4
687
Epoch 27/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7791 - loss: 0.4705 - val_accuracy: 0.7750 - val_loss: 0.4
669
Epoch 28/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7880 - loss: 0.4478 - val_accuracy: 0.7750 - val_loss: 0.46
65
Epoch 29/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8091 - loss: 0.4257 - val_accuracy: 0.7800 - val_loss: 0.4
667
Epoch 30/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8145 - loss: 0.4052 - val_accuracy: 0.7700 - val_loss: 0.4
683
Epoch 31/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7757 - loss: 0.4514 - val_accuracy: 0.7750 - val_loss: 0.46
90
Epoch 32/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7899 - loss: 0.4496 - val_accuracy: 0.7650 - val_loss: 0.4
722
Epoch 33/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.7972 - loss: 0.4455 - val_accuracy: 0.7700 - val_loss: 0.4
678
Epoch 34/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8107 - loss: 0.4179 - val_accuracy: 0.7750 - val_loss: 0.4
653
Epoch 35/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8213 - loss: 0.4109 - val_accuracy: 0.7800 - val_loss: 0.4
625
Epoch 36/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8037 - loss: 0.4017 - val_accuracy: 0.7800 - val_loss: 0.4
634
Epoch 37/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8012 - loss: 0.4277 - val_accuracy: 0.7800 - val_loss: 0.4
624
Epoch 38/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8242 - loss: 0.3971 - val_accuracy: 0.7850 - val_loss: 0.4
620
Epoch 39/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8131 - loss: 0.4250 - val_accuracy: 0.7750 - val_loss: 0.46
10
Epoch 40/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8229 - loss: 0.4056 - val_accuracy: 0.7800 - val_loss: 0.4
614
Epoch 41/100
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8392 - loss: 0.4042 - val_accuracy: 0.7750 - val_loss: 0.4
624
Epoch 42/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7881 - loss: 0.4413 - val_accuracy: 0.7800 - val_loss: 0.4
616
Epoch 43/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8011 - loss: 0.4211 - val_accuracy: 0.7850 - val_loss: 0.46
02
Epoch 44/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8031 - loss: 0.4088 - val_accuracy: 0.7850 - val_loss: 0.4
589
Epoch 45/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8323 - loss: 0.4010 - val_accuracy: 0.7900 - val_loss: 0.4
586
Epoch 46/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7929 - loss: 0.4394 - val_accuracy: 0.7850 - val_loss: 0.4
596
Epoch 47/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.8259 - loss: 0.4453 - val_accuracy: 0.7850 - val_loss: 0.4
587
Epoch 48/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.8266 - loss: 0.4005 - val_accuracy: 0.7750 - val_loss: 0.4
587
Epoch 49/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7936 - loss: 0.4276 - val_accuracy: 0.7800 - val_loss: 0.4
583
Epoch 50/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8301 - loss: 0.4228 - val_accuracy: 0.7750 - val_loss: 0.4
571
Epoch 51/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.8192 - loss: 0.4279 - val_accuracy: 0.7800 - val_loss: 0.45
52
Epoch 52/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8123 - loss: 0.4042 - val_accuracy: 0.7850 - val_loss: 0.4
546
Epoch 53/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8341 - loss: 0.3942 - val_accuracy: 0.7900 - val_loss: 0.45
21
Epoch 54/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8186 - loss: 0.4114 - val_accuracy: 0.7900 - val_loss: 0.45
54
Epoch 55/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7941 - loss: 0.4310 - val_accuracy: 0.7950 - val_loss: 0.4
545
Epoch 56/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8497 - loss: 0.3683 - val_accuracy: 0.7900 - val_loss: 0.4
574
Epoch 57/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.8068 - loss: 0.4137 - val_accuracy: 0.7800 - val_loss: 0.4
596
Epoch 58/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8052 - loss: 0.4257 - val_accuracy: 0.7750 - val_loss: 0.4
601
Epoch 59/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8267 - loss: 0.3821 - val_accuracy: 0.7800 - val_loss: 0.4
555
Epoch 60/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8199 - loss: 0.4085 - val_accuracy: 0.7850 - val_loss: 0.4
559
Epoch 61/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8176 - loss: 0.4117 - val_accuracy: 0.7950 - val_loss: 0.4
562
Epoch 62/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.8037 - loss: 0.4007 - val_accuracy: 0.7850 - val_loss: 0.4
588
Epoch 63/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8525 - loss: 0.3845 - val_accuracy: 0.7750 - val_loss: 0.4
568
Test Loss: 0.4521
Test Accuracy: 0.7900
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
Accuracy: 0.79
Precision: 0.75
Recall: 0.87
F1-score: 0.81
```

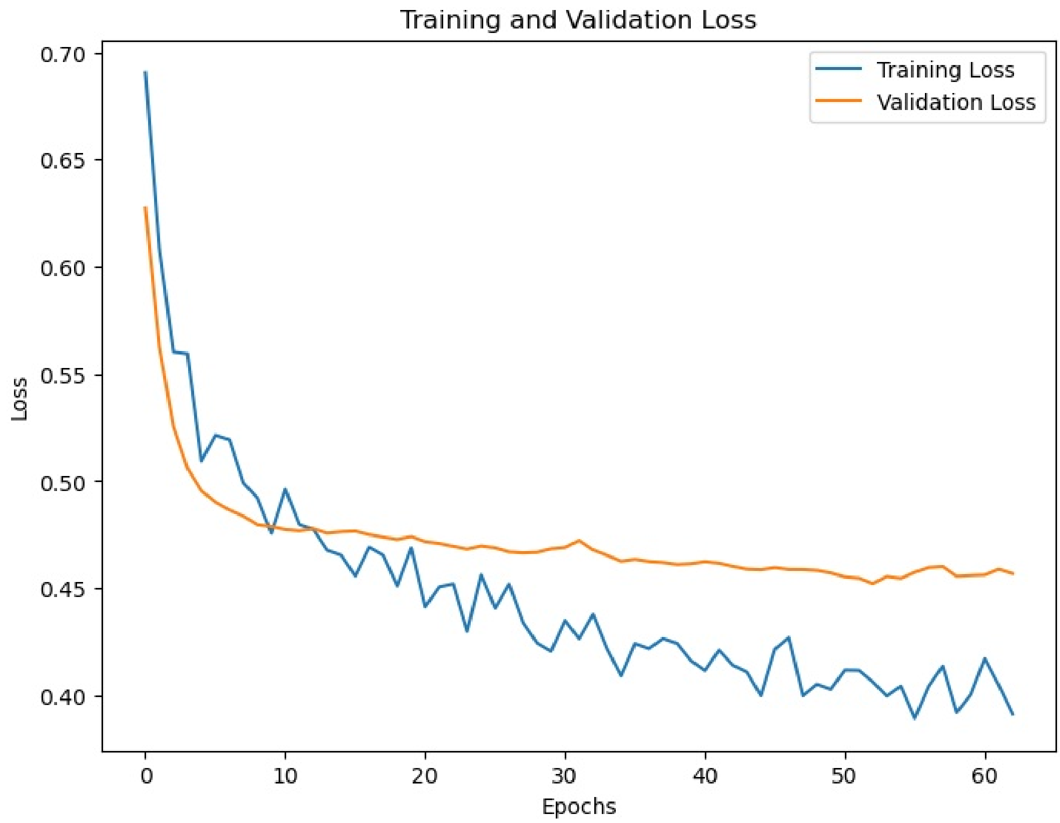



**7/7** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step





Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs


In [ ]: