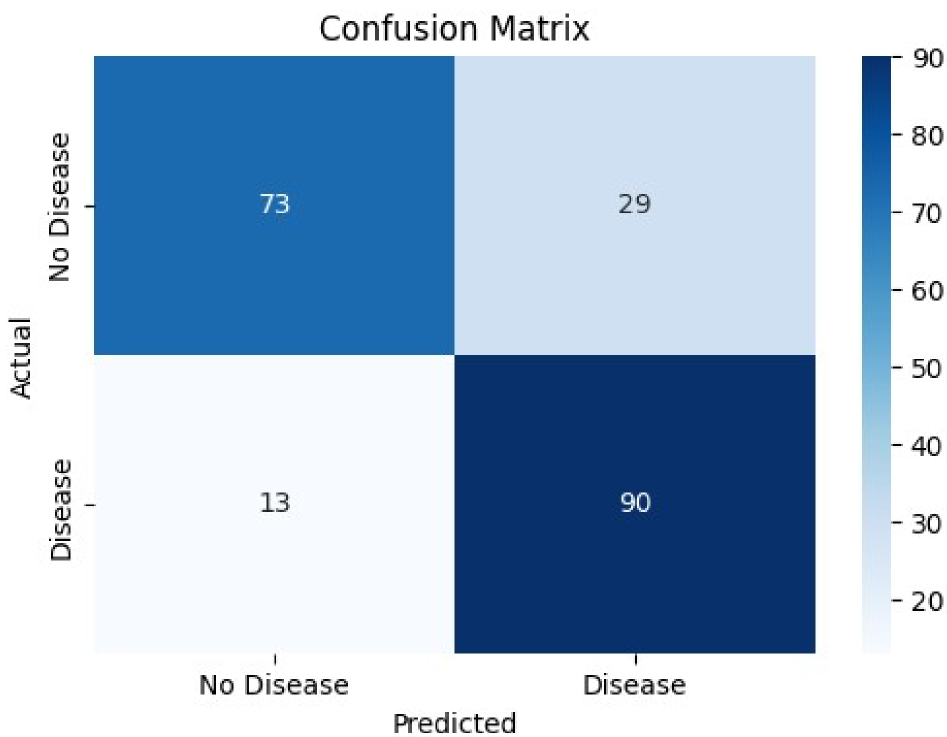
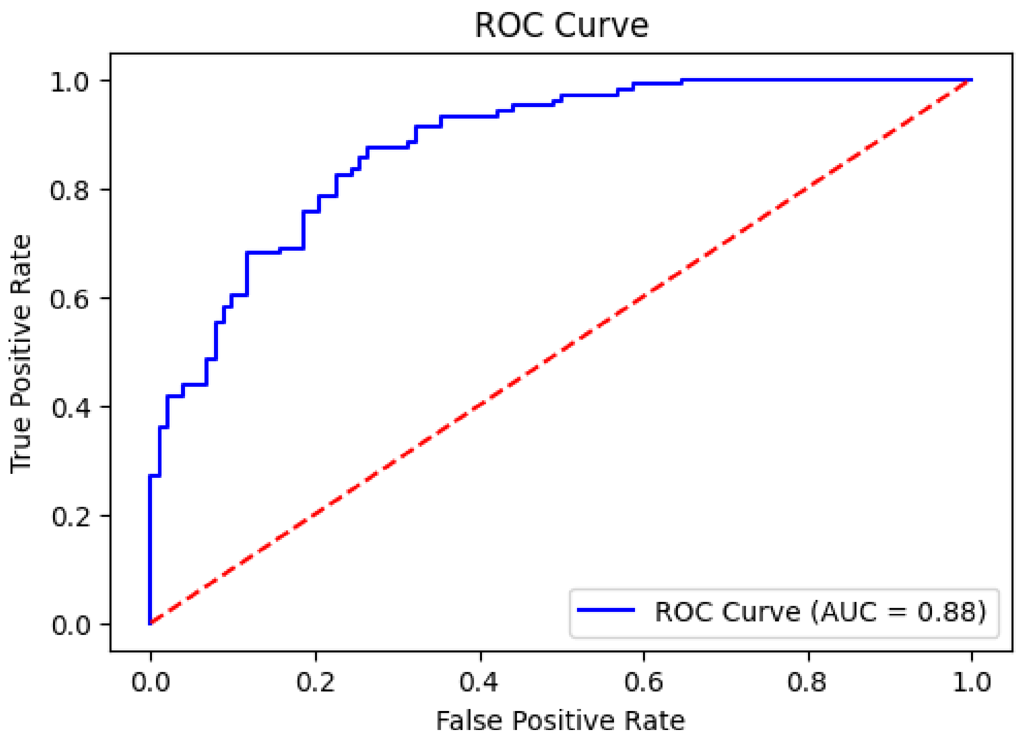
Logistic Regression (SKLearn)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
roc_curve, auc

# Load the CSV file (Ensure the correct path is given)
df = pd.read_csv("heart1.csv")

X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Split the dataset into training and testing sets (80% train, 20%
test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardizing features for better model performance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict on the test data
y_pred = model.predict(X_test)
y_pred_proba = model.predict_proba(X_test)[:, 1]  # Probabilities for
ROC curve

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy using Scikit-Learn: {accuracy:.4f}")

# 1. Confusion Matrix
plt.figure(figsize=(6, 4))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["No
Disease", "Disease"], yticklabels=["No Disease", "Disease"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```

```
# 2. ROC Curve
# Plots True Positive Rate (TPR) vs False Positive Rate (FPR).
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, color="blue", label=f"ROC Curve (AUC =
{roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle="--", color="red")  # Diagonal
baseline
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()

Accuracy using Scikit-Learn: 0.7951
```





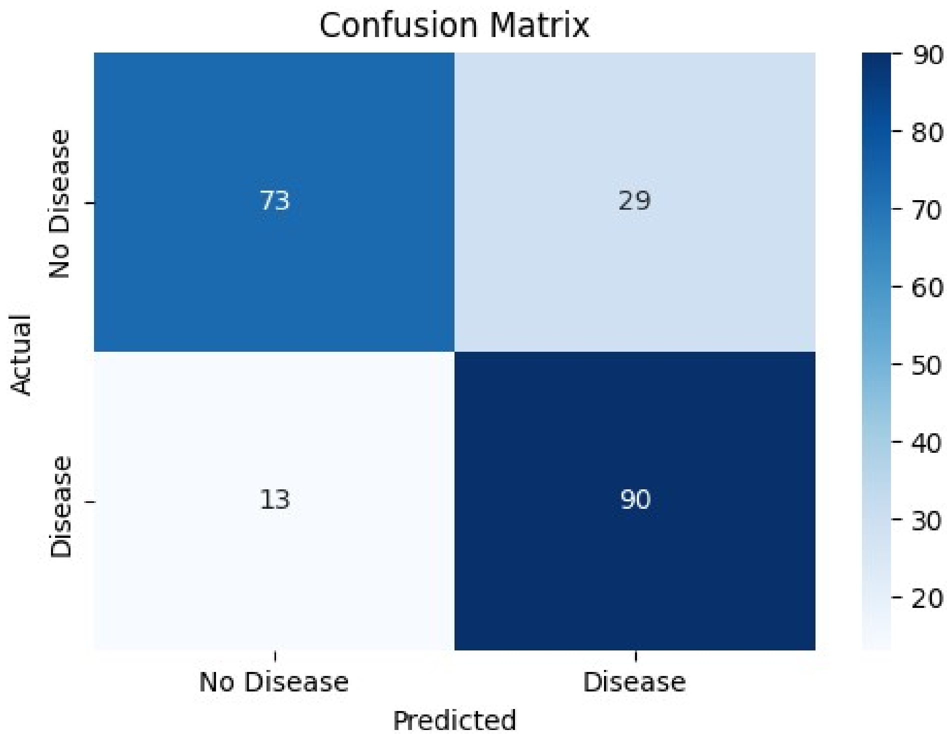
LOGISTIC REGRESSION (SCRATCH)

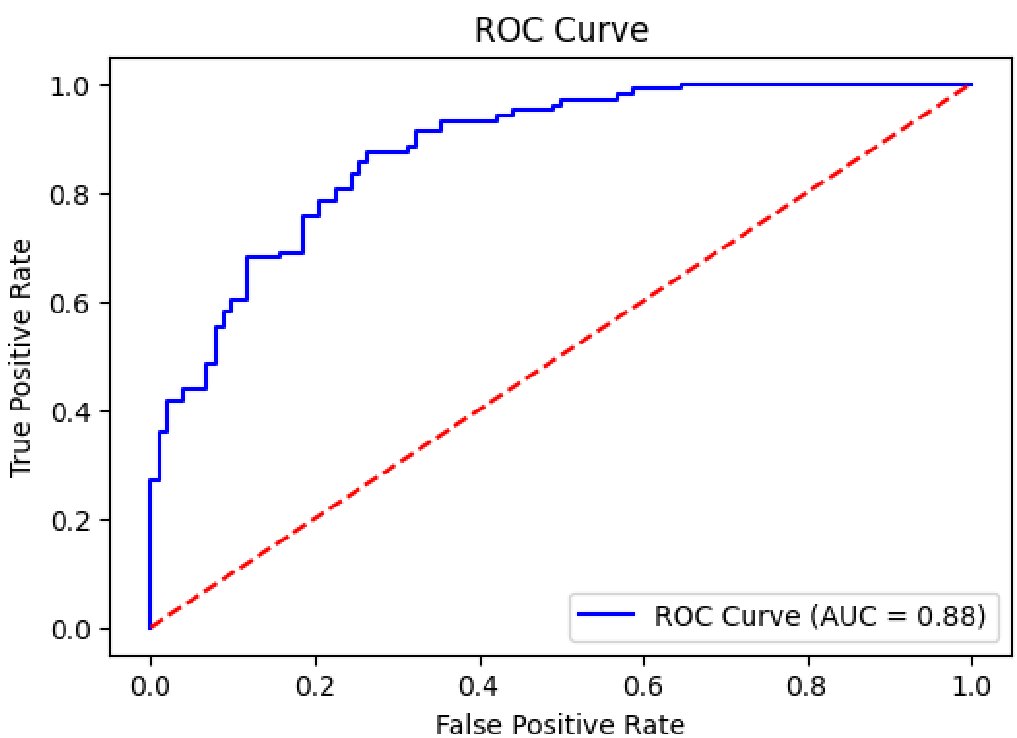
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, roc_curve, auc

# Sigmoid function to map values to probabilities
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Function to compute cost (log loss)
def compute_cost(X, y, weights):
    m = len(y)
    predictions = sigmoid(np.dot(X, weights))
    cost = (-1/m) * np.sum(y * np.log(predictions) + (1 - y) *
np.log(1 - predictions))
    return cost

# Function to perform gradient descent
def gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    cost_history = []
```

```
    for i in range(iterations):
        predictions = sigmoid(np.dot(X, weights))
        error = predictions - y
        gradient = (1/m) * np.dot(X.T, error)
        weights -= learning_rate * gradient

        cost = compute_cost(X, y, weights)
        cost_history.append(cost)

        if i % 1000 == 0:
            print(f"Iteration {i}: Cost {cost:.4f}")

    return weights, cost_history

# Load the dataset
df = pd.read_csv("heart1.csv")

# Assume the last column is the target variable and the rest are
features
X = df.iloc[:, :-1].values  # Features
y = df.iloc[:, -1].values   # Target variable

# Split data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardizing the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Add a bias term (column of ones) to X_train and X_test
X_train = np.c_[np.ones((X_train.shape[0], 1)), X_train]
X_test = np.c_[np.ones((X_test.shape[0], 1)), X_test]

# Initialize weights with zeros
weights = np.zeros(X_train.shape[1])

# Hyperparameters
learning_rate = 0.01
iterations = 10000

# Train the model
weights, cost_history = gradient_descent(X_train, y_train, weights,
learning_rate, iterations)

# Make predictions
y_pred_proba = sigmoid(np.dot(X_test, weights))  # Probabilities for
ROC curve
y_pred = y_pred_proba >= 0.5  # Convert probabilities to 0 or 1
```

```python
# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Accuracy from scratch: {accuracy:.4f}")


# 1. Confusion Matrix
plt.figure(figsize=(6, 4))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["No
Disease", "Disease"], yticklabels=["No Disease", "Disease"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# 2. ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, color="blue", label=f"ROC Curve (AUC =
{roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle="--", color="red")  # Diagonal
baseline
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

```
Iteration 0: Cost 0.6898
Iteration 1000: Cost 0.3453
Iteration 2000: Cost 0.3339
Iteration 3000: Cost 0.3311
Iteration 4000: Cost 0.3301
Iteration 5000: Cost 0.3298
Iteration 6000: Cost 0.3296
Iteration 7000: Cost 0.3295
Iteration 8000: Cost 0.3295
Iteration 9000: Cost 0.3295
Accuracy from scratch: 0.7951
```





LOGLOSS

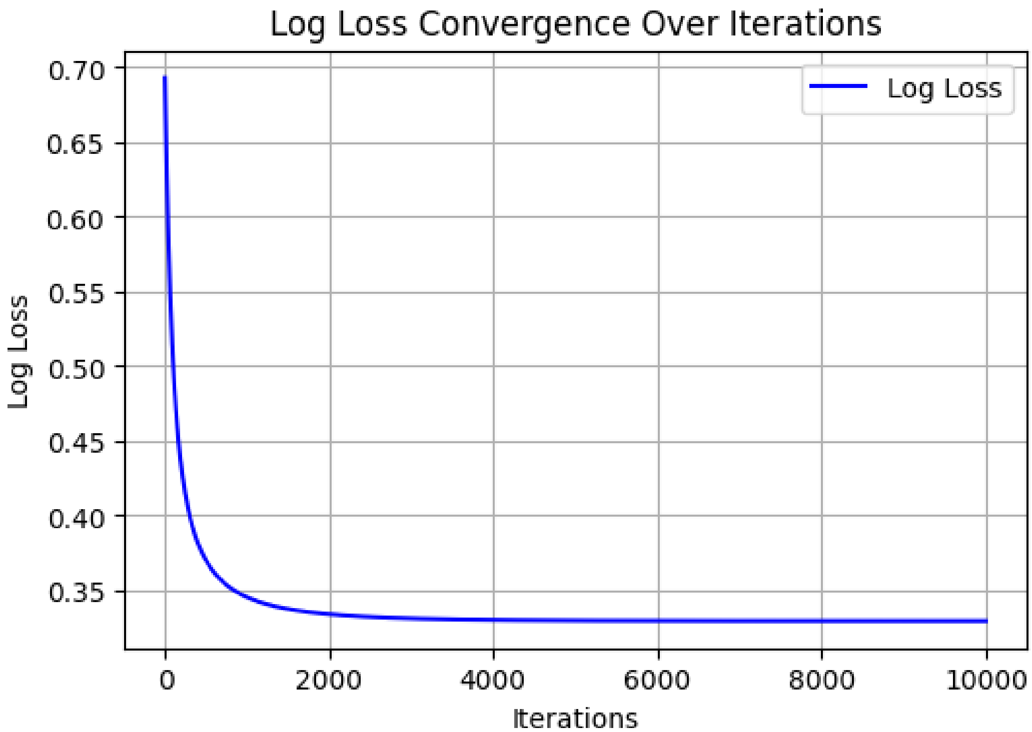
```
import numpy as np
import pandas as pd
```

```python
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, roc_curve, auc

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Log Loss function
def compute_log_loss(y_true, y_pred_proba):
    m = len(y_true)
    epsilon = 1e-15  # Avoid log(0)
    y_pred_proba = np.clip(y_pred_proba, epsilon, 1 - epsilon)
    loss = -np.mean(y_true * np.log(y_pred_proba) + (1 - y_true) *
np.log(1 - y_pred_proba))
    return loss

# Gradient Descent function
def gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    log_loss_history = []

    for i in range(iterations):
        predictions = sigmoid(np.dot(X, weights))
        error = predictions - y
        gradient = (1/m) * np.dot(X.T, error)
        weights -= learning_rate * gradient

        # Compute and store log loss
        loss = compute_log_loss(y, predictions)
        log_loss_history.append(loss)

        if i % 1000 == 0:
            print(f"Iteration {i}: Log Loss = {loss:.4f}")

    return weights, log_loss_history

# Load the dataset
df = pd.read_csv("heart1.csv")

# Assume the last column is the target variable and the rest are
features
X = df.iloc[:, :-1].values  # Features
y = df.iloc[:, -1].values   # Target variable

# Split data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

```python
# Standardizing the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Add bias term (column of ones)
X_train = np.c_[np.ones((X_train.shape[0], 1)), X_train]
X_test = np.c_[np.ones((X_test.shape[0], 1)), X_test]

# Initialize weights with zeros
weights = np.zeros(X_train.shape[1])

# Hyperparameters
learning_rate = 0.01
iterations = 10000

# Train model
weights, log_loss_history = gradient_descent(X_train, y_train,
weights, learning_rate, iterations)

# Predictions
y_pred_proba = sigmoid(np.dot(X_test, weights))
y_pred = y_pred_proba >= 0.5  # Convert to 0 or 1

# Compute Accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Final Accuracy: {accuracy:.4f}")

# ---------------- Plot Graphs ----------------

# 1. Log Loss Curve (Cost Function)
plt.figure(figsize=(6, 4))
plt.plot(range(len(log_loss_history)), log_loss_history, color="blue",
label="Log Loss")
plt.xlabel("Iterations")
plt.ylabel("Log Loss")
plt.title("Log Loss Convergence Over Iterations")
plt.legend()
plt.grid()
plt.show()
```

```
Iteration 0: Log Loss = 0.6931
Iteration 1000: Log Loss = 0.3453
Iteration 2000: Log Loss = 0.3339
Iteration 3000: Log Loss = 0.3311
Iteration 4000: Log Loss = 0.3301
Iteration 5000: Log Loss = 0.3298
Iteration 6000: Log Loss = 0.3296
Iteration 7000: Log Loss = 0.3295
```

```
Iteration 8000: Log Loss = 0.3295
Iteration 9000: Log Loss = 0.3295
Final Accuracy: 0.7951
```



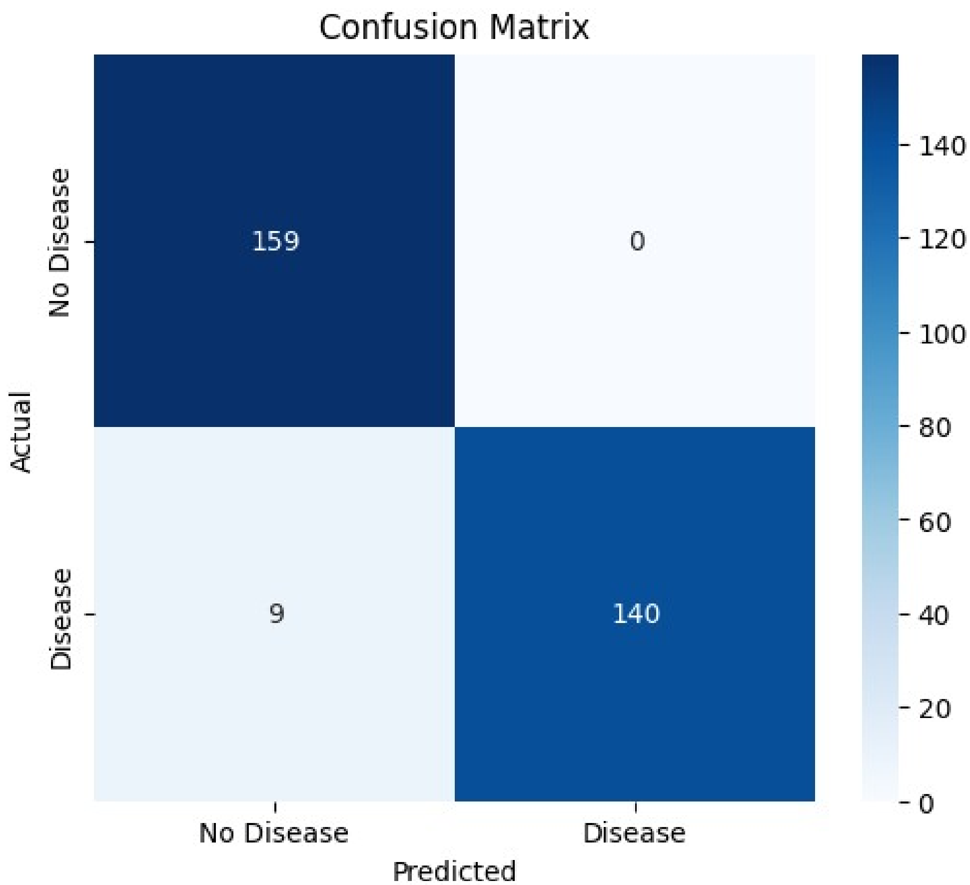
ANN using sklearn

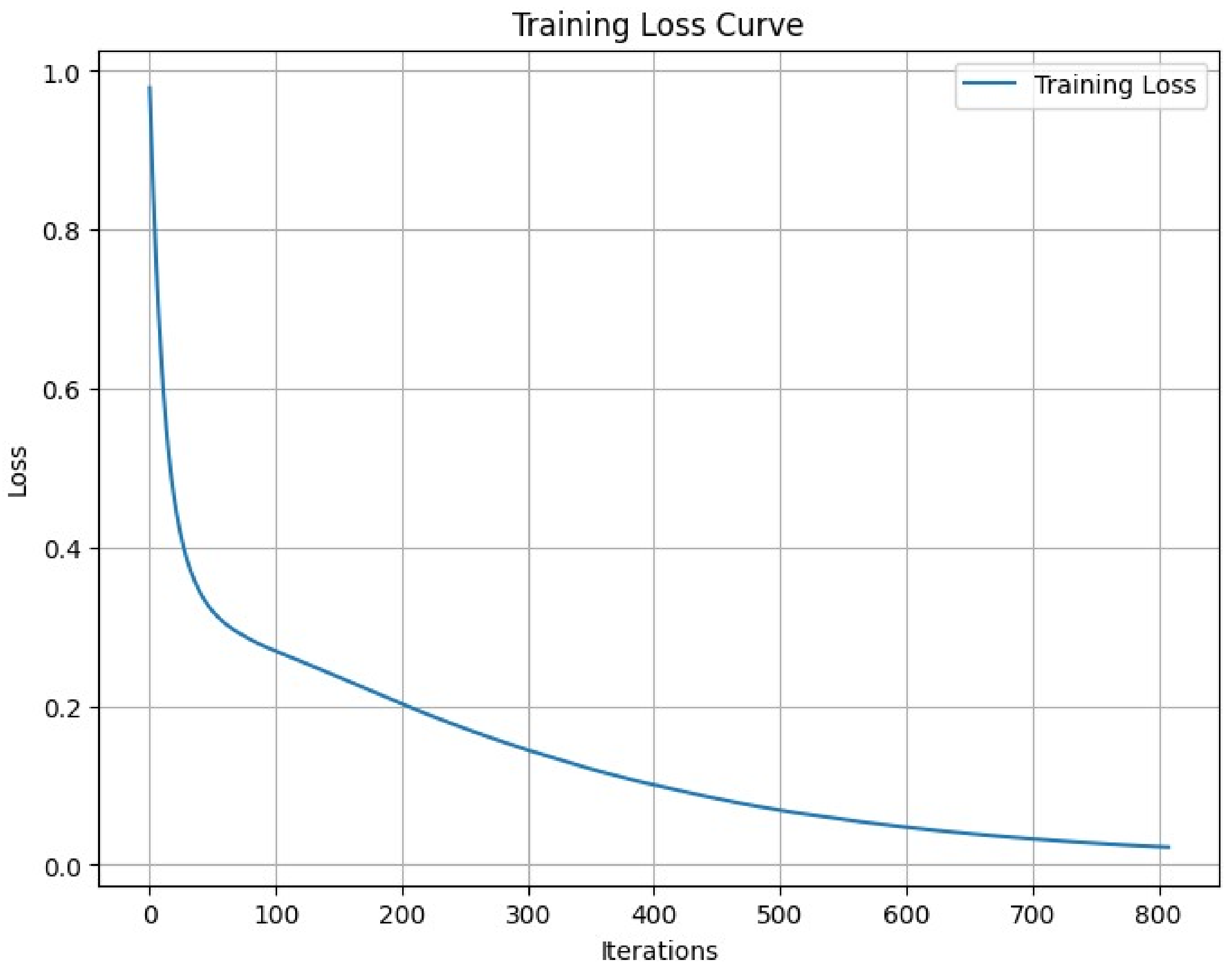
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import seaborn as sns

# Load the Heart dataset
df = pd.read_csv("heart1.csv")  # Replace with actual file path or URL

# Preprocess the data (e.g., fill missing values, select features,
etc.)
df.fillna(df.median(), inplace=True)  # Fill missing values with
median (for simplicity)

# Select features and target (assuming 'target' is the column with
labels)
```

```python
X = df.drop('target', axis=1).values  # Features (adjust column name)
y = df['target'].values  # Target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Standardize the features (important for neural networks)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize and train the MLPClassifier (ANN)
mlp = MLPClassifier(hidden_layer_sizes=(30,), max_iter=1000,
activation='relu', random_state=42)
mlp.fit(X_train_scaled, y_train)

# Make predictions
y_pred = mlp.predict(X_test_scaled)

# Accuracy and classification report
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
print("Classification Report:\n", classification_report(y_test,
y_pred))

# Confusion Matrix plot
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=['No Disease', 'Disease'], yticklabels=['No Disease',
'Disease'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Plot the loss curve during training
plt.figure(figsize=(8, 6))
plt.plot(mlp.loss_curve_, label='Training Loss')
plt.title('Training Loss Curve')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
Accuracy: 97.08%
Classification Report:
               precision    recall  f1-score   support
```

```
           0       0.95      1.00      0.97       159
           1       1.00      0.94      0.97       149

    accuracy                           0.97       308
   macro avg       0.97      0.97      0.97       308
weighted avg       0.97      0.97      0.97       308
```





KNN using keras

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from keras.models import Sequential
from keras.layers import Dense
import seaborn as sns

# Load the Heart dataset
df = pd.read_csv("heart1.csv")  # Replace with actual file path or URL

# Preprocess the data (e.g., fill missing values, select features,
etc.)
df.fillna(df.median(), inplace=True)  # Fill missing values with
median (for simplicity)
```

```python
# Select features and target (assuming 'target' is the column with
labels)
X = df.drop('target', axis=1).values  # Features (adjust column name)
y = df['target'].values  # Target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Standardize the features (important for neural networks)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Create the ANN model using Keras
model = Sequential()

# Add input layer (with 30 units) and first hidden layer (with 64
units)
model.add(Dense(64, input_dim=X_train_scaled.shape[1],
activation='relu'))

# Add second hidden layer (with 32 units)
model.add(Dense(32, activation='relu'))

# Add output layer (binary classification with 1 unit and sigmoid
activation)
model.add(Dense(1, activation='sigmoid'))

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

# Train the model
history = model.fit(X_train_scaled, y_train, epochs=100,
batch_size=10, validation_data=(X_test_scaled, y_test), verbose=1)

# Evaluate the model on the test set
y_pred = (model.predict(X_test_scaled) > 0.5).astype(int)

# Accuracy and classification report
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
print("Classification Report:\n", classification_report(y_test,
y_pred))

# Confusion Matrix plot
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
```

```
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=['No Disease', 'Disease'], yticklabels=['No Disease',
'Disease'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Plot the training loss and accuracy
plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()

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

```
Epoch 1/100
```

```
/usr/local/lib/python3.11/dist-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)
```

```
72/72 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6130 - loss:
0.6482 - val_accuracy: 0.7760 - val_loss: 0.4436
Epoch 2/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8207 - loss:
0.3873 - val_accuracy: 0.8084 - val_loss: 0.3863
Epoch 3/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8926 - loss:
0.2891 - val_accuracy: 0.8084 - val_loss: 0.3750
Epoch 4/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8813 - loss:
0.3116 - val_accuracy: 0.8279 - val_loss: 0.3553
Epoch 5/100
```

```
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9099 - loss:
0.2570 - val_accuracy: 0.8377 - val_loss: 0.3421
Epoch 6/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9066 - loss:
0.2532 - val_accuracy: 0.8506 - val_loss: 0.3230
Epoch 7/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8911 - loss:
0.2640 - val_accuracy: 0.8766 - val_loss: 0.3063
Epoch 8/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9258 - loss:
0.2057 - val_accuracy: 0.8701 - val_loss: 0.2905
Epoch 9/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9202 - loss:
0.2118 - val_accuracy: 0.8864 - val_loss: 0.2706
Epoch 10/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9268 - loss:
0.1814 - val_accuracy: 0.8766 - val_loss: 0.2517
Epoch 11/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9268 - loss:
0.1858 - val_accuracy: 0.8961 - val_loss: 0.2353
Epoch 12/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9445 - loss:
0.1518 - val_accuracy: 0.9026 - val_loss: 0.2146
Epoch 13/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9411 - loss:
0.1473 - val_accuracy: 0.9253 - val_loss: 0.2021
Epoch 14/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9551 - loss:
0.1324 - val_accuracy: 0.9221 - val_loss: 0.1888
Epoch 15/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9647 - loss:
0.1098 - val_accuracy: 0.9383 - val_loss: 0.1675
Epoch 16/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9715 - loss:
0.1077 - val_accuracy: 0.9351 - val_loss: 0.1541
Epoch 17/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9690 - loss:
0.0981 - val_accuracy: 0.9416 - val_loss: 0.1431
Epoch 18/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9751 - loss:
0.0849 - val_accuracy: 0.9448 - val_loss: 0.1338
Epoch 19/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9801 - loss:
0.0872 - val_accuracy: 0.9481 - val_loss: 0.1275
Epoch 20/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9770 - loss:
0.0786 - val_accuracy: 0.9578 - val_loss: 0.1176
Epoch 21/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9827 - loss:
```

```
0.0598 - val_accuracy: 0.9675 - val_loss: 0.1106
Epoch 22/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9934 - loss:
0.0446 - val_accuracy: 0.9740 - val_loss: 0.1097
Epoch 23/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9975 - loss:
0.0482 - val_accuracy: 0.9740 - val_loss: 0.1027
Epoch 24/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9975 - loss:
0.0422 - val_accuracy: 0.9805 - val_loss: 0.0985
Epoch 25/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
0.0434 - val_accuracy: 0.9805 - val_loss: 0.0945
Epoch 26/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9998 - loss:
0.0346 - val_accuracy: 0.9805 - val_loss: 0.0909
Epoch 27/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
0.0300 - val_accuracy: 0.9805 - val_loss: 0.0882
Epoch 28/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0230 - val_accuracy: 0.9805 - val_loss: 0.0859
Epoch 29/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
0.0240 - val_accuracy: 0.9805 - val_loss: 0.0849
Epoch 30/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
0.0165 - val_accuracy: 0.9805 - val_loss: 0.0918
Epoch 31/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0147 - val_accuracy: 0.9805 - val_loss: 0.0877
Epoch 32/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0136 - val_accuracy: 0.9805 - val_loss: 0.0936
Epoch 33/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
0.0114 - val_accuracy: 0.9805 - val_loss: 0.0946
Epoch 34/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
0.0121 - val_accuracy: 0.9805 - val_loss: 0.0932
Epoch 35/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
0.0106 - val_accuracy: 0.9805 - val_loss: 0.0955
Epoch 36/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0101 - val_accuracy: 0.9805 - val_loss: 0.0930
Epoch 37/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
0.0080 - val_accuracy: 0.9805 - val_loss: 0.0987
```

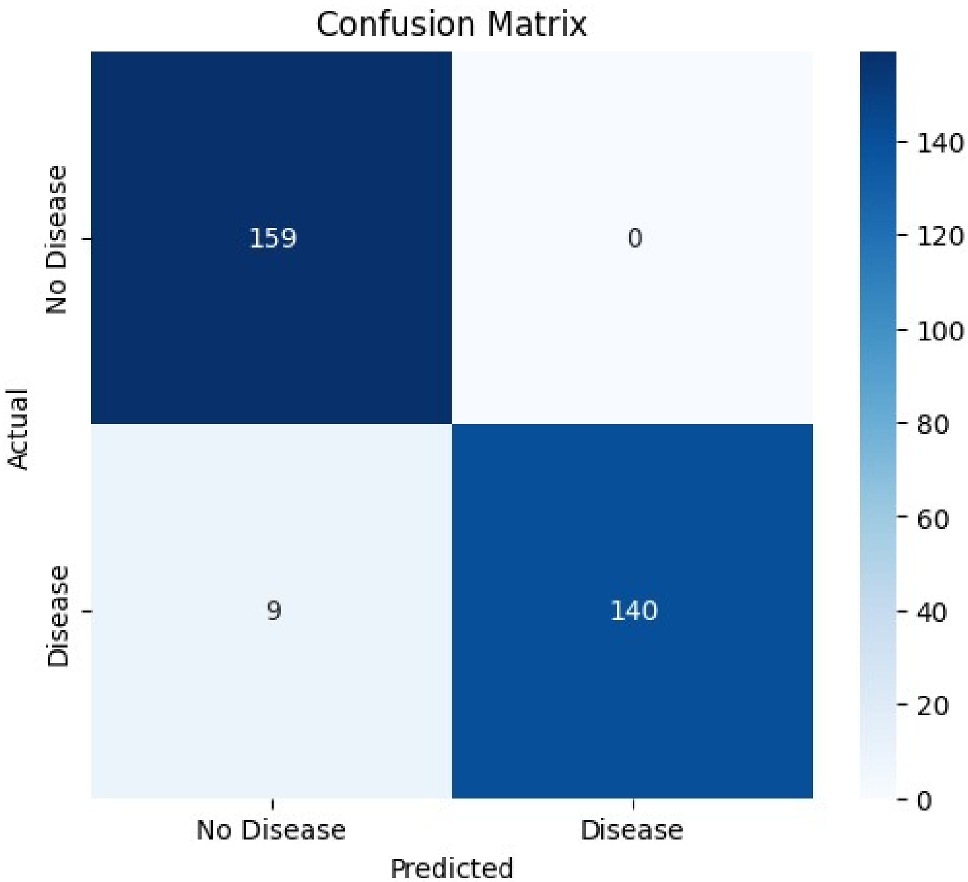
```
Epoch 38/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0076 - val_accuracy: 0.9805 - val_loss: 0.0997
Epoch 39/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
0.0065 - val_accuracy: 0.9805 - val_loss: 0.1003
Epoch 40/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0063 - val_accuracy: 0.9805 - val_loss: 0.0995
Epoch 41/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
0.0058 - val_accuracy: 0.9805 - val_loss: 0.1001
Epoch 42/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0051 - val_accuracy: 0.9805 - val_loss: 0.1004
Epoch 43/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 1.0000 - loss:
0.0052 - val_accuracy: 0.9805 - val_loss: 0.1019
Epoch 44/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0043 - val_accuracy: 0.9805 - val_loss: 0.1034
Epoch 45/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
0.0043 - val_accuracy: 0.9805 - val_loss: 0.1071
Epoch 46/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
0.0038 - val_accuracy: 0.9805 - val_loss: 0.1099
Epoch 47/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 1.0000 - loss:
0.0034 - val_accuracy: 0.9805 - val_loss: 0.1079
Epoch 48/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
0.0035 - val_accuracy: 0.9805 - val_loss: 0.1080
Epoch 49/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
0.0026 - val_accuracy: 0.9805 - val_loss: 0.1117
Epoch 50/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0026 - val_accuracy: 0.9805 - val_loss: 0.1116
Epoch 51/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
0.0026 - val_accuracy: 0.9805 - val_loss: 0.1217
Epoch 52/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0024 - val_accuracy: 0.9805 - val_loss: 0.1201
Epoch 53/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0025 - val_accuracy: 0.9805 - val_loss: 0.1136
Epoch 54/100
```

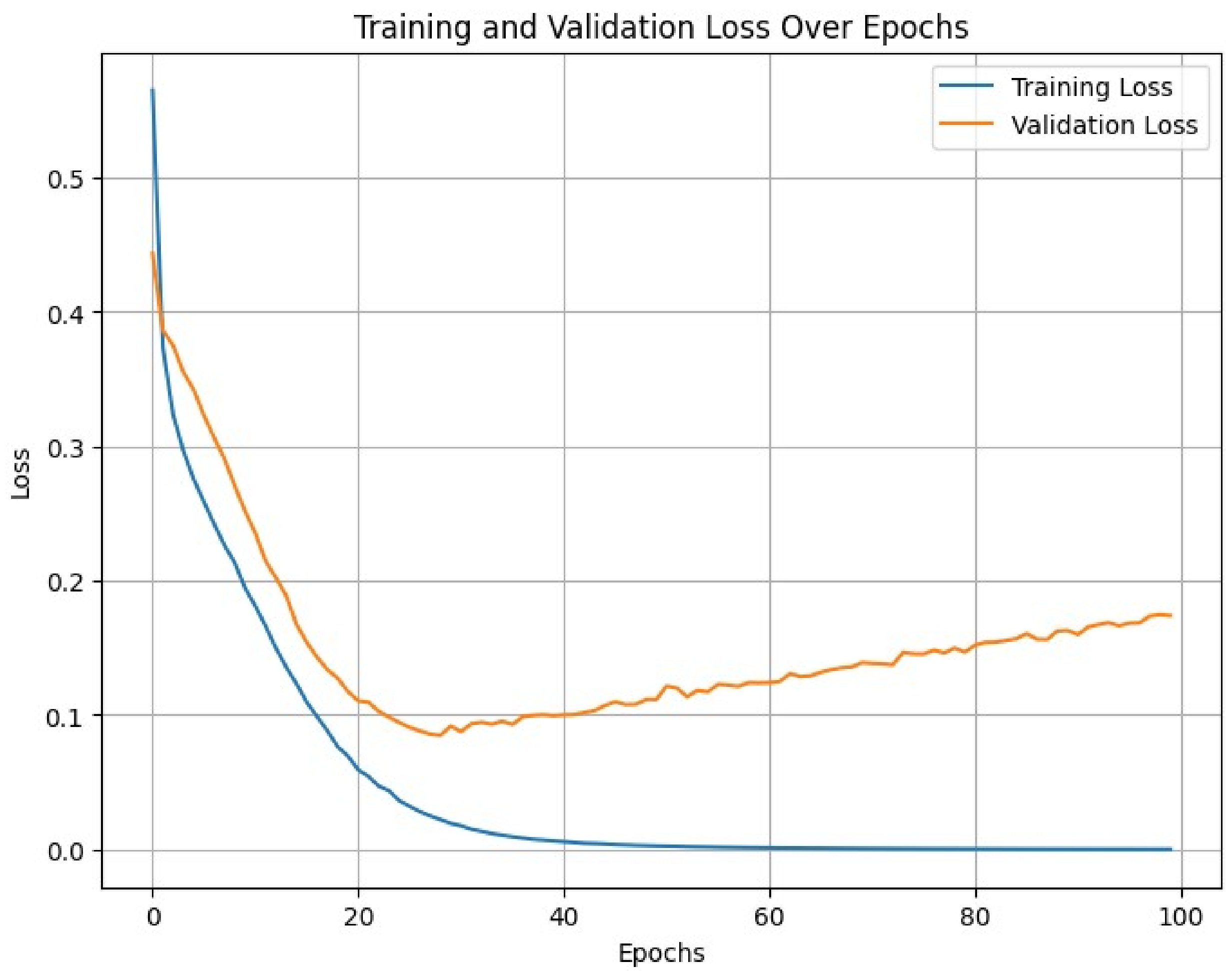
```
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
0.0018 - val_accuracy: 0.9805 - val_loss: 0.1184
Epoch 55/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
0.0021 - val_accuracy: 0.9805 - val_loss: 0.1173
Epoch 56/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
0.0016 - val_accuracy: 0.9805 - val_loss: 0.1229
Epoch 57/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0017 - val_accuracy: 0.9805 - val_loss: 0.1223
Epoch 58/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
0.0014 - val_accuracy: 0.9805 - val_loss: 0.1214
Epoch 59/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0014 - val_accuracy: 0.9805 - val_loss: 0.1242
Epoch 60/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0013 - val_accuracy: 0.9805 - val_loss: 0.1240
Epoch 61/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0012 - val_accuracy: 0.9805 - val_loss: 0.1243
Epoch 62/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
0.0012 - val_accuracy: 0.9805 - val_loss: 0.1250
Epoch 63/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0010 - val_accuracy: 0.9805 - val_loss: 0.1309
Epoch 64/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 1.0000 - loss:
0.0011 - val_accuracy: 0.9805 - val_loss: 0.1287
Epoch 65/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
8.7202e-04 - val_accuracy: 0.9805 - val_loss: 0.1292
Epoch 66/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
9.9630e-04 - val_accuracy: 0.9805 - val_loss: 0.1319
Epoch 67/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
9.2012e-04 - val_accuracy: 0.9805 - val_loss: 0.1338
Epoch 68/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss:
8.0455e-04 - val_accuracy: 0.9805 - val_loss: 0.1352
Epoch 69/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
7.1822e-04 - val_accuracy: 0.9708 - val_loss: 0.1358
Epoch 70/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
```

```
7.4026e-04 - val_accuracy: 0.9708 - val_loss: 0.1391
Epoch 71/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss:
8.2236e-04 - val_accuracy: 0.9708 - val_loss: 0.1385
Epoch 72/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
5.6767e-04 - val_accuracy: 0.9708 - val_loss: 0.1383
Epoch 73/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
6.4872e-04 - val_accuracy: 0.9805 - val_loss: 0.1376
Epoch 74/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
5.4796e-04 - val_accuracy: 0.9708 - val_loss: 0.1466
Epoch 75/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
5.1401e-04 - val_accuracy: 0.9708 - val_loss: 0.1456
Epoch 76/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
4.8681e-04 - val_accuracy: 0.9708 - val_loss: 0.1455
Epoch 77/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
5.0165e-04 - val_accuracy: 0.9708 - val_loss: 0.1484
Epoch 78/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 1.0000 - loss:
4.0563e-04 - val_accuracy: 0.9708 - val_loss: 0.1462
Epoch 79/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
4.8004e-04 - val_accuracy: 0.9708 - val_loss: 0.1499
Epoch 80/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
4.3011e-04 - val_accuracy: 0.9708 - val_loss: 0.1469
Epoch 81/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 1.0000 - loss:
3.7066e-04 - val_accuracy: 0.9708 - val_loss: 0.1522
Epoch 82/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
3.3713e-04 - val_accuracy: 0.9708 - val_loss: 0.1541
Epoch 83/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
3.0205e-04 - val_accuracy: 0.9708 - val_loss: 0.1543
Epoch 84/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
3.7259e-04 - val_accuracy: 0.9708 - val_loss: 0.1555
Epoch 85/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
3.4884e-04 - val_accuracy: 0.9708 - val_loss: 0.1568
Epoch 86/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss:
2.8636e-04 - val_accuracy: 0.9708 - val_loss: 0.1606
```

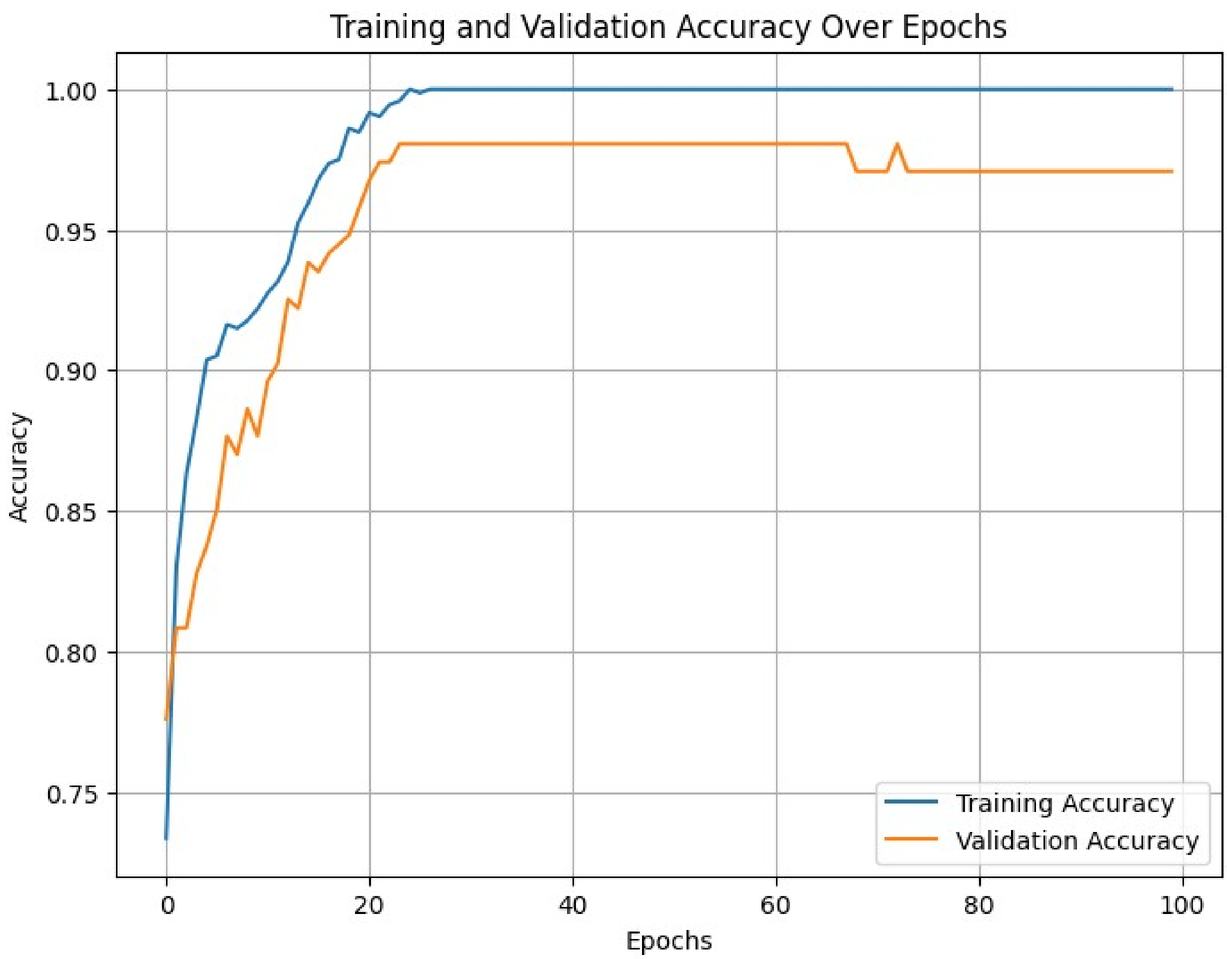
```
Epoch 87/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 
2.9946e-04 - val_accuracy: 0.9708 - val_loss: 0.1565
Epoch 88/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 
2.9384e-04 - val_accuracy: 0.9708 - val_loss: 0.1563
Epoch 89/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss: 
2.6534e-04 - val_accuracy: 0.9708 - val_loss: 0.1626
Epoch 90/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss: 
2.5785e-04 - val_accuracy: 0.9708 - val_loss: 0.1630
Epoch 91/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 
2.1753e-04 - val_accuracy: 0.9708 - val_loss: 0.1599
Epoch 92/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss: 
2.3266e-04 - val_accuracy: 0.9708 - val_loss: 0.1657
Epoch 93/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss: 
2.0749e-04 - val_accuracy: 0.9708 - val_loss: 0.1674
Epoch 94/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 
2.1731e-04 - val_accuracy: 0.9708 - val_loss: 0.1688
Epoch 95/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 
1.8531e-04 - val_accuracy: 0.9708 - val_loss: 0.1665
Epoch 96/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss: 
1.8726e-04 - val_accuracy: 0.9708 - val_loss: 0.1685
Epoch 97/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 
1.8131e-04 - val_accuracy: 0.9708 - val_loss: 0.1687
Epoch 98/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss: 
1.5776e-04 - val_accuracy: 0.9708 - val_loss: 0.1739
Epoch 99/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 
1.4636e-04 - val_accuracy: 0.9708 - val_loss: 0.1748
Epoch 100/100
72/72 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 
1.4356e-04 - val_accuracy: 0.9708 - val_loss: 0.1741
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
Accuracy: 97.08%
Classification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97       159
           1       1.00      0.94      0.97       149
```

```
    accuracy                           0.97       308
   macro avg       0.97      0.97      0.97       308
weighted avg       0.97      0.97      0.97       308
```



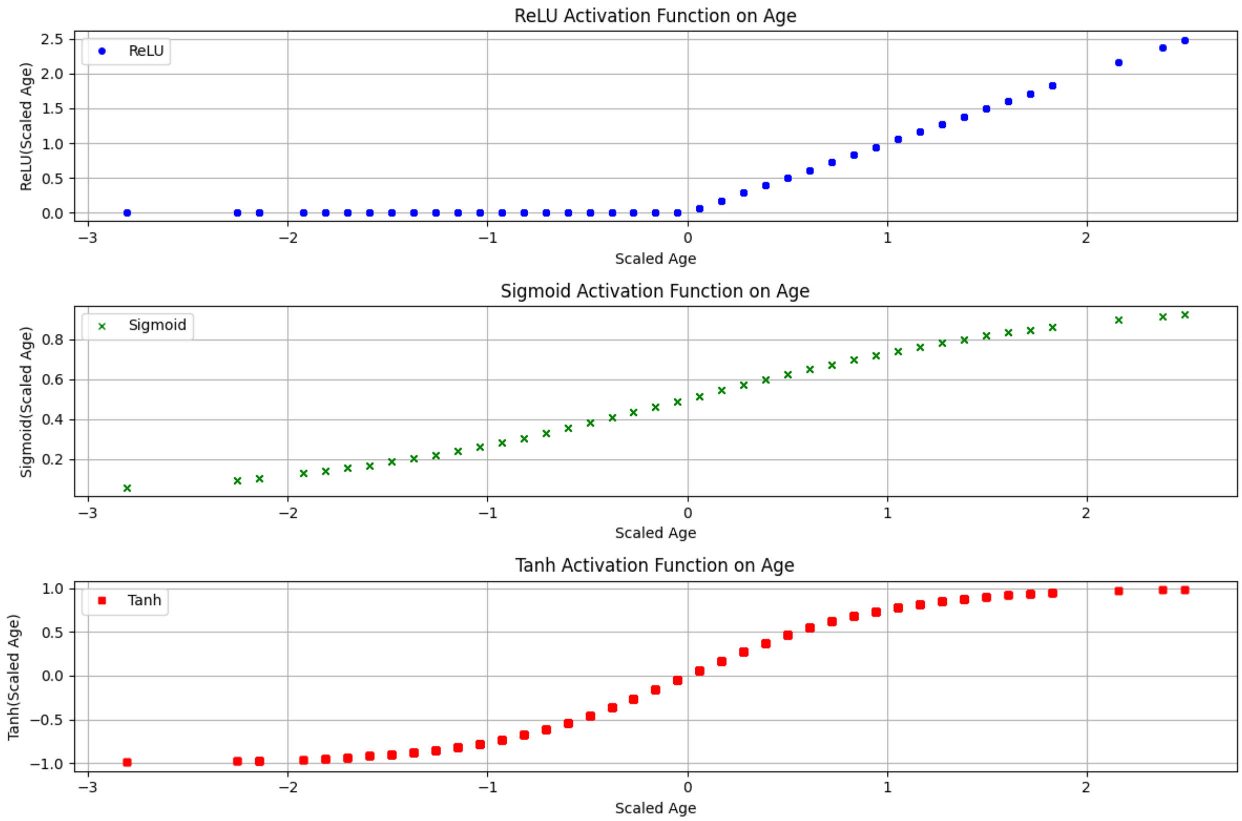


Training and Validation Loss Over Epochs
Training Loss
Validation Loss
Loss
Epochs
0.0
0.1
0.2
0.3
0.4
0.5
0
20
40
60
80
100




```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# Load the Heart dataset
df = pd.read_csv("heart1.csv")  # Replace with actual file path or URL

# Fill missing values (if any) in a column of interest, e.g., 'Age' or
'Cholesterol'
df['age'] = df['age'].fillna(df['age'].median())  # Example: Replace
missing values in 'Age'

# Select the 'Age' column (or any other numeric column of interest) as
our feature (X)
X = df['age'].values

# Standardize the feature (important for activation functions)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X.reshape(-1, 1)).flatten()  # Reshape
```

```python
and standardize

# Apply activation functions on the feature (ReLU, Sigmoid, Tanh)
y_relu = np.maximum(0, X_scaled)  # ReLU
y_sigmoid = 1 / (1 + np.exp(-X_scaled))  # Sigmoid
y_tanh = np.tanh(X_scaled)  # Tanh

# Create subplots to display the activation functions
plt.figure(figsize=(12, 8))

# ReLU plot
plt.subplot(3, 1, 1)
plt.plot(X_scaled, y_relu, label="ReLU", color='blue', marker='o',
linestyle='none', markersize=4)
plt.title("ReLU Activation Function on Age")
plt.xlabel("Scaled Age")
plt.ylabel("ReLU(Scaled Age)")
plt.grid(True)
plt.legend()

# Sigmoid plot
plt.subplot(3, 1, 2)
plt.plot(X_scaled, y_sigmoid, label="Sigmoid", color='green',
marker='x', linestyle='none', markersize=4)
plt.title("Sigmoid Activation Function on Age")
plt.xlabel("Scaled Age")
plt.ylabel("Sigmoid(Scaled Age)")
plt.grid(True)
plt.legend()

# Tanh plot
plt.subplot(3, 1, 3)
plt.plot(X_scaled, y_tanh, label="Tanh", color='red', marker='s',
linestyle='none', markersize=4)
plt.title("Tanh Activation Function on Age")
plt.xlabel("Scaled Age")
plt.ylabel("Tanh(Scaled Age)")
plt.grid(True)
plt.legend()

# Show the plots
plt.tight_layout()
plt.show()
```


ReLU Activation Function on Age
ReLU
ReLU(Scaled Age)
Scaled Age
Sigmoid Activation Function on Age
Sigmoid
Sigmoid(Scaled Age)
Scaled Age
Tanh Activation Function on Age
Tanh
Tanh(Scaled Age)
Scaled Age