Question 1: XOR Gate Classification

XOR dataset

```
In [52]: import pandas as pd
         import numpy as np
         from sklearn.model_selection import train_test_split

         x = np.array([[0,0],[0,1],[1,0],[1,1]]) #XOR Inputs
         y = np.array([0,1,1,0]) #XOR Outputs
```

single layer Perceptron using MCP Neuron

```
In [53]: from sklearn.linear_model import Perceptron

         perceptron = Perceptron(max_iter=1000, tol=1e-3)
         perceptron.fit(x,y)
```

```
Out[53]: ▾ Perceptron
         Perceptron()
```

predict and evaluate

```
In [54]: prediction = perceptron.predict(x)
         print("Perceptron prediction: ", prediction)
         print("Actual Outputs: ", y)
```

```
Perceptron prediction:  [0 0 0 0]
Actual Outputs:  [0 1 1 0]
```

Performance observence

```
In [55]: accuracy = np.mean(prediction == y)
         print("Accuracy: ", accuracy*100)
```

```
Accuracy:  50.0
```

Multi-Layer Perceptron (MLP) to solve XOR

```
In [56]: from tensorflow.keras.models import Sequential
         from tensorflow.keras.layers import Dense

         mlp = Sequential()
         mlp.add(Dense(2, input_dim=2, activation='relu'))  # 2 neurons in the hidder
         mlp.add(Dense(1, activation='sigmoid'))  # 1 output neuron

         mlp.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy

         mlp.fit(x, y, epochs=100, verbose=0)
```

Out[56]:
```
<keras.src.callbacks.History at 0x7981fc6d2bf0>
```

Evaluating the model

In [57]:
```
loss, accuracy = mlp.evaluate(x, y)
predictions_mlp = (mlp.predict(x) > 0.5).astype("int32")
print("MLP Predictions:", predictions_mlp.flatten())
print("MLP Accuracy", accuracy*100)
```

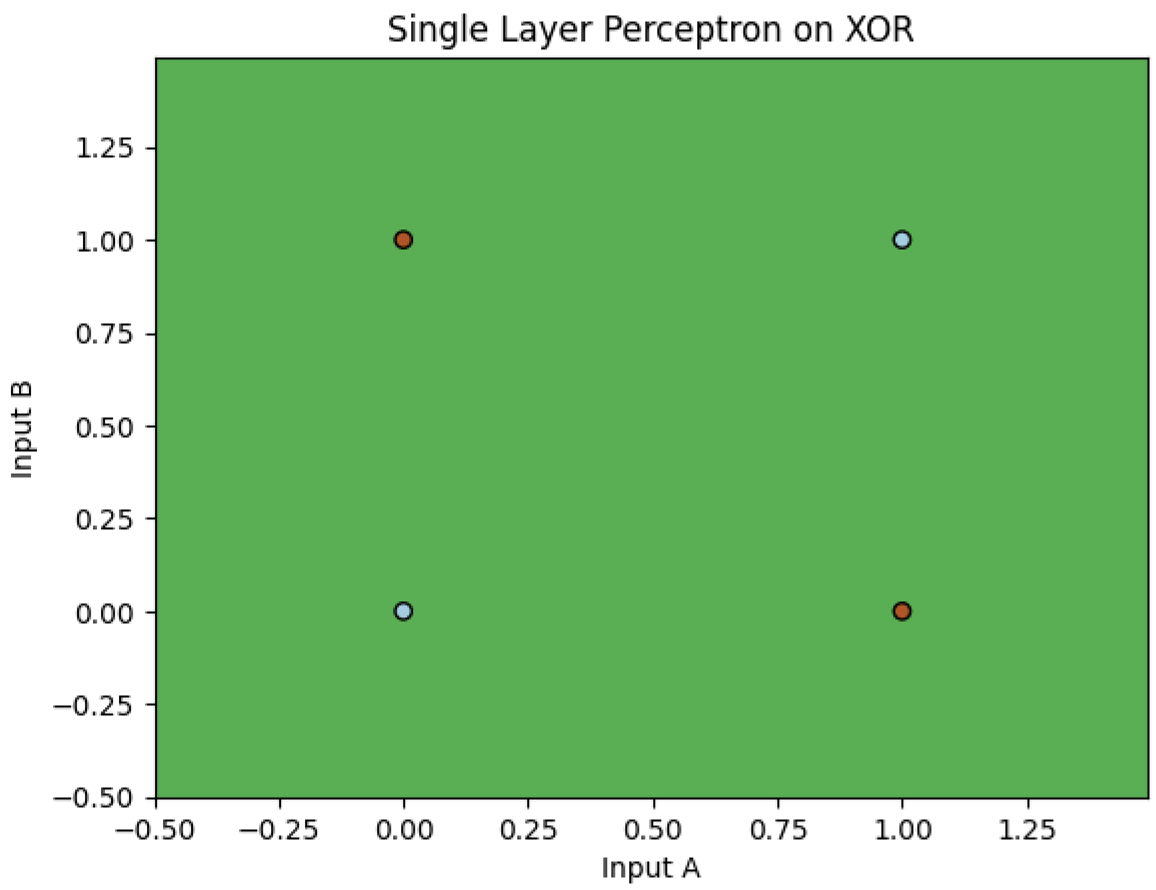
```
1/1 [==============================] - 0s 110ms/step - loss: 0.7117 - accura
cy: 0.5000
1/1 [==============================] - 0s 51ms/step
MLP Predictions: [1 1 0 0]
MLP Accuracy 50.0
```

Method to help for visualisation

In [58]:
```
import matplotlib.pyplot as plt

def plot_decision_boundary(X, y, model, title):
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))

    # Get model predictions for the grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    if hasattr(Z, 'reshape'):
        Z = Z.reshape(xx.shape)
    else:
        Z = (Z > 0.5).astype(int).reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o', cmap=plt.
    plt.title(title)
    plt.xlabel('Input A')
    plt.ylabel('Input B')
    plt.show()
```

Plotting for perceptron

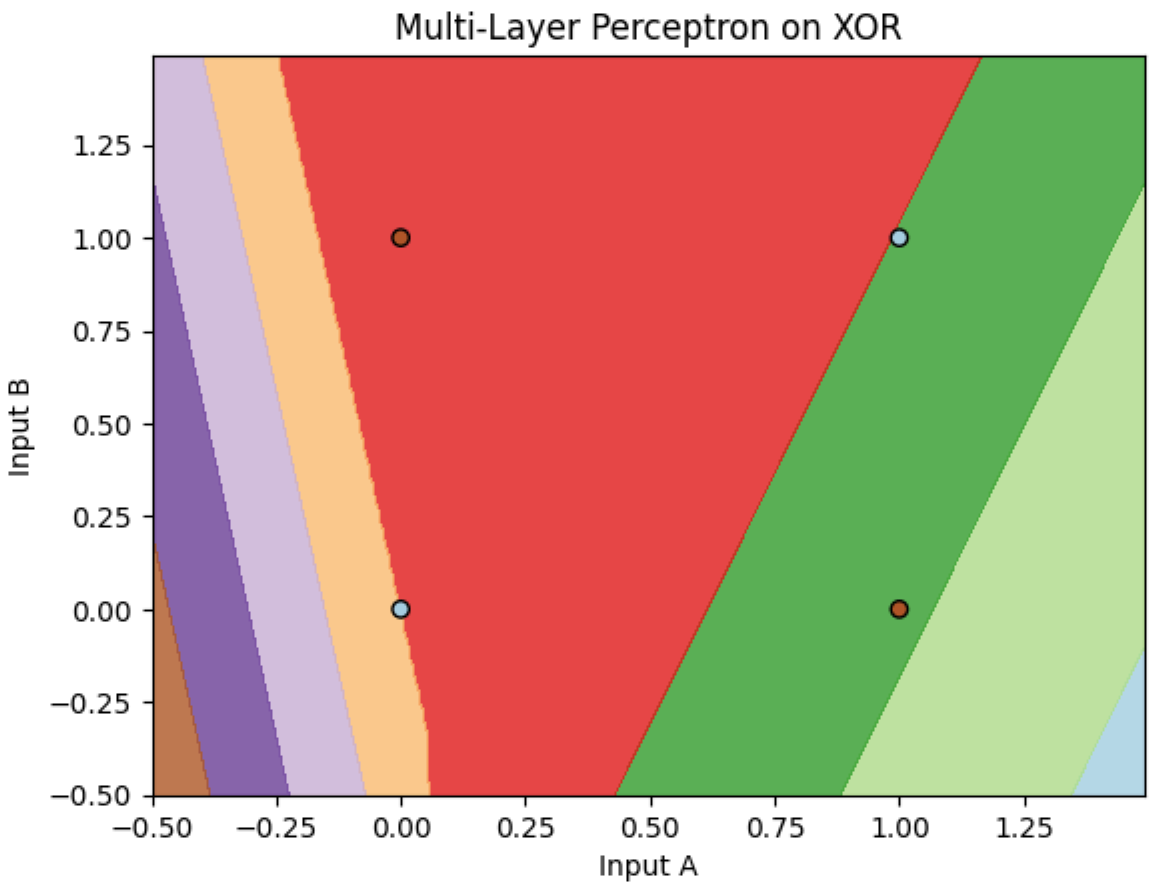
In [59]:
```
plot_decision_boundary(x, y, perceptron, "Single Layer Perceptron on XOR")
```



plotting for MLP

```
In [60]: plot_decision_boundary(x, y, mlp, "Multi-Layer Perceptron on XOR")
```

```
1250/1250 [==============================] - 1s 1ms/step
```



Question 2: A. Sentiment Analysis Twitter Airline

Architecture

- **Input Layer**: Receives the vectorized input.
- **Hidden Layer**: Applies different activation functions.
- **Output Layer**: Outputs a probability between 0 and 1, suitable for binary classification.

Loadinig Dataset

In [61]:
```
import pandas as pd

data = pd.read_csv('Tweets.csv')

data.head()
```

Out[61]:

	tweet_id	airline_sentiment	airline_sentiment_confidence	negat
0	570306133677760513	neutral	1.0000	
1	570301130888122368	positive	0.3486	
2	570301083672813571	neutral	0.6837	
3	570301031407624196	negative	1.0000	
4	570300817074462722	negative	1.0000	

Data Preprocessing The dataset is preprocessed by converting the text into numerical form using CountVectorizer, and labels are transformed into binary form (0 for negative, 1 for positive).

In [63]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(max_features=input_size)
X = vectorizer.fit_transform(data['text']).toarray()

y = (data['airline_sentiment'] == 'positive').astype(int).values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar


X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)
```

Building the neural network model with sigmoid activation function for hidden layers

In [64]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
class SentimentNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, activation_func
        super(SentimentNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.activation_func = activation_func
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.activation_func(x)
        x = self.fc2(x)
        return torch.sigmoid(x)

activations = {
    'sigmoid': nn.Sigmoid(),
    'relu': nn.ReLU(),
    'tanh': nn.Tanh()
}

input_size = 1000
hidden_size = 128
output_size = 1
S_model = SentimentNet(input_size, hidden_size, output_size, activations['si
R_model = SentimentNet(input_size, hidden_size, output_size, activations['re
T_model = SentimentNet(input_size, hidden_size, output_size, activations['ta

S_optimizer = optim.Adam(S_model.parameters(), lr=0.001)
R_optimizer = optim.Adam(R_model.parameters(), lr=0.001)
T_optimizer = optim.Adam(T_model.parameters(), lr=0.001)
```

Activation Functions

Sigmoid Activation: Often used for binary classification but can suffer from vanishing gradients.

ReLU Activation: Helps avoid vanishing gradients but may lead to "dead neurons."

Tanh Activation: Similar to sigmoid but output ranges from -1 to 1, potentially making optimization easier.

1. Sigmoid Activation

- Maps the input to a range between 0 and 1.
- Often used for binary classification, but can suffer from vanishing gradient issues.
- Formula is: 1/(1+e^(-x))

2. Tanh Activation

- Introduces sparsity, meaning only some neurons activate, making learning faster.

- Solves vanishing gradient problems but can suffer from dead neurons where gradients are zero for large negative inputs.
- formula is: max(0,x)

3. Relu Activation

- Maps inputs to the range -1 to 1.
- Unlike Sigmoid, Tanh is centered around 0, making optimization easier. Still, it can suffer from vanishing gradient issues like Sigmoid.
- formula is: (e^x)-(e^-x)/((e^x)+(e^-x))

Training Function

In [65]:
```
from torch.utils.data import DataLoader, TensorDataset

train_data = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)

def train_model(model, optimizer, criterion, epochs=100):
    model.train()
    losses = []

    for epoch in range(epochs):
        epoch_loss = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(train_loader)
        losses.append(avg_loss)
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}')

    return losses
```

training and comparing the models

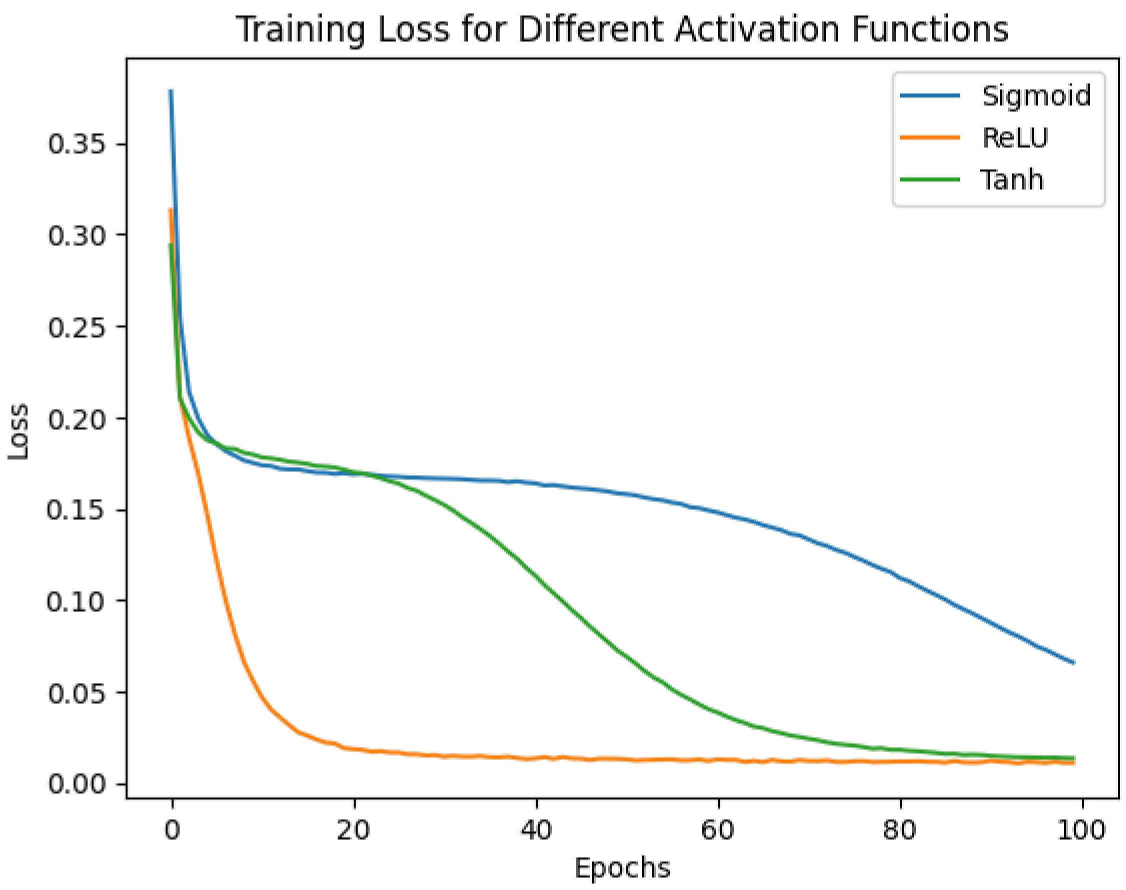
In [66]:
```
print("Training model with Sigmoid activation...")
sigmoid_losses = train_model(S_model, S_optimizer, criterion)

print("\nTraining model with ReLU activation...")
relu_losses = train_model(R_model, R_optimizer, criterion)

print("\nTraining model with Tanh activation...")
tanh_losses = train_model(T_model, T_optimizer, criterion)
```

```
Training model with Sigmoid activation...
Epoch [1/100], Loss: 0.3783
Epoch [2/100], Loss: 0.2543
Epoch [3/100], Loss: 0.2139
Epoch [4/100], Loss: 0.1994
Epoch [5/100], Loss: 0.1903
Epoch [6/100], Loss: 0.1853
Epoch [7/100], Loss: 0.1815
Epoch [8/100], Loss: 0.1790
Epoch [9/100], Loss: 0.1764
Epoch [10/100], Loss: 0.1750
Epoch [11/100], Loss: 0.1737
Epoch [12/100], Loss: 0.1733
Epoch [13/100], Loss: 0.1718
Epoch [14/100], Loss: 0.1714
Epoch [15/100], Loss: 0.1714
Epoch [16/100], Loss: 0.1705
Epoch [17/100], Loss: 0.1697
Epoch [18/100], Loss: 0.1696
Epoch [19/100], Loss: 0.1689
Epoch [20/100], Loss: 0.1694
Epoch [21/100], Loss: 0.1688
Epoch [22/100], Loss: 0.1690
Epoch [23/100], Loss: 0.1686
Epoch [24/100], Loss: 0.1680
Epoch [25/100], Loss: 0.1676
Epoch [26/100], Loss: 0.1674
Epoch [27/100], Loss: 0.1670
Epoch [28/100], Loss: 0.1669
Epoch [29/100], Loss: 0.1667
Epoch [30/100], Loss: 0.1665
Epoch [31/100], Loss: 0.1664
Epoch [32/100], Loss: 0.1663
Epoch [33/100], Loss: 0.1661
Epoch [34/100], Loss: 0.1657
Epoch [35/100], Loss: 0.1654
Epoch [36/100], Loss: 0.1654
Epoch [37/100], Loss: 0.1652
Epoch [38/100], Loss: 0.1645
Epoch [39/100], Loss: 0.1650
Epoch [40/100], Loss: 0.1642
Epoch [41/100], Loss: 0.1638
Epoch [42/100], Loss: 0.1627
Epoch [43/100], Loss: 0.1629
Epoch [44/100], Loss: 0.1622
Epoch [45/100], Loss: 0.1615
Epoch [46/100], Loss: 0.1611
Epoch [47/100], Loss: 0.1607
Epoch [48/100], Loss: 0.1600
Epoch [49/100], Loss: 0.1593
Epoch [50/100], Loss: 0.1584
Epoch [51/100], Loss: 0.1579
Epoch [52/100], Loss: 0.1572
Epoch [53/100], Loss: 0.1561
Epoch [54/100], Loss: 0.1552
Epoch [55/100], Loss: 0.1545
Epoch [56/100], Loss: 0.1533
Epoch [57/100], Loss: 0.1526
Epoch [58/100], Loss: 0.1508
Epoch [59/100], Loss: 0.1502
Epoch [60/100], Loss: 0.1490
Epoch [61/100], Loss: 0.1479
Epoch [62/100], Loss: 0.1465
Epoch [63/100], Loss: 0.1451
Epoch [64/100], Loss: 0.1440
Epoch [65/100], Loss: 0.1427
Epoch [66/100], Loss: 0.1410
Epoch [67/100], Loss: 0.1396
Epoch [68/100], Loss: 0.1382
Epoch [69/100], Loss: 0.1362
Epoch [70/100], Loss: 0.1353
Epoch [71/100], Loss: 0.1331
Epoch [72/100], Loss: 0.1310
Epoch [73/100], Loss: 0.1295
Epoch [74/100], Loss: 0.1274
Epoch [75/100], Loss: 0.1259
Epoch [76/100], Loss: 0.1236
Epoch [77/100], Loss: 0.1215
Epoch [78/100], Loss: 0.1191
Epoch [79/100], Loss: 0.1171
Epoch [80/100], Loss: 0.1152
Epoch [81/100], Loss: 0.1120
Epoch [82/100], Loss: 0.1102
Epoch [83/100], Loss: 0.1075
Epoch [84/100], Loss: 0.1050
Epoch [85/100], Loss: 0.1027
Epoch [86/100], Loss: 0.1002
Epoch [87/100], Loss: 0.0973
Epoch [88/100], Loss: 0.0949
Epoch [89/100], Loss: 0.0926
Epoch [90/100], Loss: 0.0900
Epoch [91/100], Loss: 0.0874
Epoch [92/100], Loss: 0.0848
Epoch [93/100], Loss: 0.0823
Epoch [94/100], Loss: 0.0801
Epoch [95/100], Loss: 0.0776
Epoch [96/100], Loss: 0.0747
Epoch [97/100], Loss: 0.0727
Epoch [98/100], Loss: 0.0703
Epoch [99/100], Loss: 0.0679
Epoch [100/100], Loss: 0.0658

Training model with ReLU activation...
Epoch [1/100], Loss: 0.3132
Epoch [2/100], Loss: 0.2113
Epoch [3/100], Loss: 0.1886
Epoch [4/100], Loss: 0.1692
Epoch [5/100], Loss: 0.1465
Epoch [6/100], Loss: 0.1218
Epoch [7/100], Loss: 0.1000
Epoch [8/100], Loss: 0.0817
Epoch [9/100], Loss: 0.0663
Epoch [10/100], Loss: 0.0559
Epoch [11/100], Loss: 0.0467
Epoch [12/100], Loss: 0.0401
Epoch [13/100], Loss: 0.0357
Epoch [14/100], Loss: 0.0315
Epoch [15/100], Loss: 0.0276
Epoch [16/100], Loss: 0.0257
Epoch [17/100], Loss: 0.0236
Epoch [18/100], Loss: 0.0220
Epoch [19/100], Loss: 0.0214
Epoch [20/100], Loss: 0.0189
Epoch [21/100], Loss: 0.0184
Epoch [22/100], Loss: 0.0180
Epoch [23/100], Loss: 0.0170
Epoch [24/100], Loss: 0.0173
Epoch [25/100], Loss: 0.0166
Epoch [26/100], Loss: 0.0166
Epoch [27/100], Loss: 0.0156
Epoch [28/100], Loss: 0.0156
Epoch [29/100], Loss: 0.0149
Epoch [30/100], Loss: 0.0151
Epoch [31/100], Loss: 0.0143
Epoch [32/100], Loss: 0.0147
Epoch [33/100], Loss: 0.0143
Epoch [34/100], Loss: 0.0143
Epoch [35/100], Loss: 0.0146
Epoch [36/100], Loss: 0.0140
Epoch [37/100], Loss: 0.0139
Epoch [38/100], Loss: 0.0143
Epoch [39/100], Loss: 0.0135
Epoch [40/100], Loss: 0.0129
Epoch [41/100], Loss: 0.0133
Epoch [42/100], Loss: 0.0140
Epoch [43/100], Loss: 0.0131
Epoch [44/100], Loss: 0.0141
Epoch [45/100], Loss: 0.0133
Epoch [46/100], Loss: 0.0132
Epoch [47/100], Loss: 0.0125
Epoch [48/100], Loss: 0.0132
Epoch [49/100], Loss: 0.0132
Epoch [50/100], Loss: 0.0130
Epoch [51/100], Loss: 0.0128
Epoch [52/100], Loss: 0.0122
Epoch [53/100], Loss: 0.0124
Epoch [54/100], Loss: 0.0126
Epoch [55/100], Loss: 0.0127
Epoch [56/100], Loss: 0.0128
Epoch [57/100], Loss: 0.0122
Epoch [58/100], Loss: 0.0123
Epoch [59/100], Loss: 0.0127
Epoch [60/100], Loss: 0.0119
Epoch [61/100], Loss: 0.0128
Epoch [62/100], Loss: 0.0125
Epoch [63/100], Loss: 0.0125
Epoch [64/100], Loss: 0.0114
Epoch [65/100], Loss: 0.0120
Epoch [66/100], Loss: 0.0113
Epoch [67/100], Loss: 0.0125
Epoch [68/100], Loss: 0.0117
Epoch [69/100], Loss: 0.0115
Epoch [70/100], Loss: 0.0125
Epoch [71/100], Loss: 0.0119
Epoch [72/100], Loss: 0.0118
Epoch [73/100], Loss: 0.0122
Epoch [74/100], Loss: 0.0114
Epoch [75/100], Loss: 0.0113
Epoch [76/100], Loss: 0.0118
Epoch [77/100], Loss: 0.0118
Epoch [78/100], Loss: 0.0113
Epoch [79/100], Loss: 0.0114
Epoch [80/100], Loss: 0.0115
Epoch [81/100], Loss: 0.0116
Epoch [82/100], Loss: 0.0116
Epoch [83/100], Loss: 0.0118
Epoch [84/100], Loss: 0.0114
Epoch [85/100], Loss: 0.0114
Epoch [86/100], Loss: 0.0110
Epoch [87/100], Loss: 0.0119
Epoch [88/100], Loss: 0.0111
Epoch [89/100], Loss: 0.0111
Epoch [90/100], Loss: 0.0112
Epoch [91/100], Loss: 0.0121
Epoch [92/100], Loss: 0.0115
Epoch [93/100], Loss: 0.0113
Epoch [94/100], Loss: 0.0106
Epoch [95/100], Loss: 0.0115
Epoch [96/100], Loss: 0.0112
Epoch [97/100], Loss: 0.0108
Epoch [98/100], Loss: 0.0115
Epoch [99/100], Loss: 0.0109
Epoch [100/100], Loss: 0.0109

Training model with Tanh activation...
Epoch [1/100], Loss: 0.2941
Epoch [2/100], Loss: 0.2103
Epoch [3/100], Loss: 0.1994
Epoch [4/100], Loss: 0.1917
Epoch [5/100], Loss: 0.1874
Epoch [6/100], Loss: 0.1857
Epoch [7/100], Loss: 0.1829
Epoch [8/100], Loss: 0.1827
Epoch [9/100], Loss: 0.1806
Epoch [10/100], Loss: 0.1796
Epoch [11/100], Loss: 0.1781
Epoch [12/100], Loss: 0.1775
Epoch [13/100], Loss: 0.1767
Epoch [14/100], Loss: 0.1757
Epoch [15/100], Loss: 0.1752
Epoch [16/100], Loss: 0.1744
Epoch [17/100], Loss: 0.1733
Epoch [18/100], Loss: 0.1730
Epoch [19/100], Loss: 0.1725
Epoch [20/100], Loss: 0.1713
Epoch [21/100], Loss: 0.1699
Epoch [22/100], Loss: 0.1694
Epoch [23/100], Loss: 0.1681
Epoch [24/100], Loss: 0.1667
Epoch [25/100], Loss: 0.1650
Epoch [26/100], Loss: 0.1637
Epoch [27/100], Loss: 0.1613
Epoch [28/100], Loss: 0.1597
Epoch [29/100], Loss: 0.1570
Epoch [30/100], Loss: 0.1546
Epoch [31/100], Loss: 0.1520
Epoch [32/100], Loss: 0.1489
Epoch [33/100], Loss: 0.1452
Epoch [34/100], Loss: 0.1420
Epoch [35/100], Loss: 0.1386
Epoch [36/100], Loss: 0.1350
Epoch [37/100], Loss: 0.1309
Epoch [38/100], Loss: 0.1265
Epoch [39/100], Loss: 0.1225
Epoch [40/100], Loss: 0.1175
Epoch [41/100], Loss: 0.1133
Epoch [42/100], Loss: 0.1081
Epoch [43/100], Loss: 0.1036
Epoch [44/100], Loss: 0.0992
Epoch [45/100], Loss: 0.0944
Epoch [46/100], Loss: 0.0901
Epoch [47/100], Loss: 0.0854
Epoch [48/100], Loss: 0.0811
Epoch [49/100], Loss: 0.0768
Epoch [50/100], Loss: 0.0723
Epoch [51/100], Loss: 0.0689
Epoch [52/100], Loss: 0.0652
Epoch [53/100], Loss: 0.0610
Epoch [54/100], Loss: 0.0575
Epoch [55/100], Loss: 0.0549
Epoch [56/100], Loss: 0.0510
Epoch [57/100], Loss: 0.0481
Epoch [58/100], Loss: 0.0455
Epoch [59/100], Loss: 0.0428
Epoch [60/100], Loss: 0.0402
Epoch [61/100], Loss: 0.0385
Epoch [62/100], Loss: 0.0362
Epoch [63/100], Loss: 0.0343
Epoch [64/100], Loss: 0.0327
Epoch [65/100], Loss: 0.0307
Epoch [66/100], Loss: 0.0299
Epoch [67/100], Loss: 0.0282
Epoch [68/100], Loss: 0.0272
Epoch [69/100], Loss: 0.0259
Epoch [70/100], Loss: 0.0251
Epoch [71/100], Loss: 0.0241
Epoch [72/100], Loss: 0.0233
Epoch [73/100], Loss: 0.0221
Epoch [74/100], Loss: 0.0213
Epoch [75/100], Loss: 0.0208
Epoch [76/100], Loss: 0.0203
Epoch [77/100], Loss: 0.0196
Epoch [78/100], Loss: 0.0186
Epoch [79/100], Loss: 0.0189
Epoch [80/100], Loss: 0.0180
Epoch [81/100], Loss: 0.0181
Epoch [82/100], Loss: 0.0175
Epoch [83/100], Loss: 0.0172
Epoch [84/100], Loss: 0.0168
Epoch [85/100], Loss: 0.0163
Epoch [86/100], Loss: 0.0158
Epoch [87/100], Loss: 0.0159
Epoch [88/100], Loss: 0.0152
Epoch [89/100], Loss: 0.0154
Epoch [90/100], Loss: 0.0152
Epoch [91/100], Loss: 0.0147
Epoch [92/100], Loss: 0.0145
Epoch [93/100], Loss: 0.0142
Epoch [94/100], Loss: 0.0140
Epoch [95/100], Loss: 0.0139
Epoch [96/100], Loss: 0.0138
Epoch [97/100], Loss: 0.0137
Epoch [98/100], Loss: 0.0138
Epoch [99/100], Loss: 0.0135
Epoch [100/100], Loss: 0.0134
```

Plotting the losses

In [67]:

```python
import matplotlib.pyplot as plt

plt.plot(sigmoid_losses, label='Sigmoid')
plt.plot(relu_losses, label='ReLU')
plt.plot(tanh_losses, label='Tanh')
plt.title('Training Loss for Different Activation Functions')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



- The ReLU function achieves the lowest training loss among the three activations.
- Tanh initially performs similarly to ReLU, but after about 20 epochs, its convergence slows down.
- Sigmoid has the slowest convergence and the highest final training loss.
- ReLU is the best activation function for this scenario, offering faster convergence and lower training loss.
- Tanh is also a good choice but slightly less efficient.
- Sigmoid performs poorly in comparison, suffering from slower convergence and higher training loss due to issues like vanishing gradients.

Model Evaluation on Test Set

In [70]:
```
def evaluate_model(model, X_test_tensor, y_test_tensor):
    model.eval()
    with torch.no_grad():
        outputs = model(X_test_tensor)
        predictions = (outputs >= 0.5).float()
        accuracy = (predictions == y_test_tensor).float().mean().item()
    return accuracy

sigmoid_acc = evaluate_model(S_model, X_test_tensor, y_test_tensor)
relu_acc = evaluate_model(R_model, X_test_tensor, y_test_tensor)
tanh_acc = evaluate_model(T_model, X_test_tensor, y_test_tensor)

print(f"ReLU Accuracy: {relu_acc*100:.2f}%")
print(f"Sigmoid Accuracy: {sigmoid_acc*100:.2f}%")
print(f"Tanh Accuracy: {tanh_acc*100:.2f}%")
```

```
ReLU Accuracy: 89.14%
Sigmoid Accuracy: 89.52%
Tanh Accuracy: 88.97%
```

Despite the differences in training loss curves, the final accuracies of the models are quite similar:

- Sigmoid Accuracy: 89.52%
- ReLU Accuracy: 89.14%
- Tanh Accuracy: 88.97%