Deep Learning for Computer Vision: HW 2

Computer Science: COMS W 4995 005

Due: October 10, 2024

Problem

You are given the noisy XOR data generated for you below. Your task is to implement a multi-layer perceptron binary classifier with one hidden layer. For the activiation function of the hidden units use ReLu. For the loss function use a softplus on a linear output layer as we did in class. Randomly initialize the weight parameters for your network.

a) Implement each layer of the network as a separate function with both forward propagation and backpropagation.

b) Train the network using stochastic gradient descent with mini-batches.

c) Show the decision regions of the trained classifier by densely generating points in the plane and color coding these points according to the label your classifier would assign them. For instance, if a sample point x is classified as class = 1, then color the point blue, otherwise color the point orange.

d) Repeat (b) and (c) varying the number of hidden units: 3, 16, 512. Discuss how the number of hidden units effects your solution.

e) Try at least two different learning schedules. For instance, you can start with a constant learning rate and see how that converges. Then, you can repeat everything by using a learning schedule that decays with time.

f) Try choosing your own loss function (**without asking me or the TAs what you should choose**), repeating (d).

g) Now try with three input features, generating your own training and testing data. (For this XOR the output should be a 1 if and only if exactly one of the inputs is 1. But make the training data noisy as before.) Use softplus loss. Do not try to show the decision regions, instead generate a test set in the same manner as the training set, classify the samples, and compute the classification accuracy.

h) Using your data from HW1 or any new data you curate if you don't think your HW1 data is appropriate for this assighnment, train your MLP using your training set (80%). Compute the error rate on your test set (20%). It's up to you how many hidden units to use.

If you are struggling to get the network to converge, experiment with different learning rates.

Grading: a-d = 50%, e=10%, f=10%, g=10%, h=20%.

NOTE: Do not to use keras, tensorflow, pytorch, sklearn, etc. to do this. You must build the machine learning components from scratch.

Let's start by importing some libraries.

In [31]:
```
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt
```

Let's make up some noisy XOR data.

In [32]:
```python
data = pd.DataFrame(np.zeros((5000, 3)), columns=['x1', 'x2', 'y'])

# Let's make up some noisy XOR data to use to build our binary classifier
for i in range(len(data.index)):
    x1 = 1.0 * random.randint(0,1)
    x2 = 1.0 * random.randint(0,1)
    y = 1.0 * np.logical_xor(x1==1,x2==1)
    x1 = x1 + 0.15 * np.random.normal()
    x2 = x2 + 0.15 * np.random.normal()
    data.iloc[i,0] = x1
    data.iloc[i,1] = x2
    data.iloc[i,2] = y

data.head()
```

Out[32]:

	x1	x2	y
0	0.060706	-0.056250	0.0
1	-0.262028	0.286595	0.0
2	0.274969	-0.138655	0.0
3	1.170938	0.105167	1.0
4	0.800837	0.008939	1.0

Let's message this data into a numpy format.

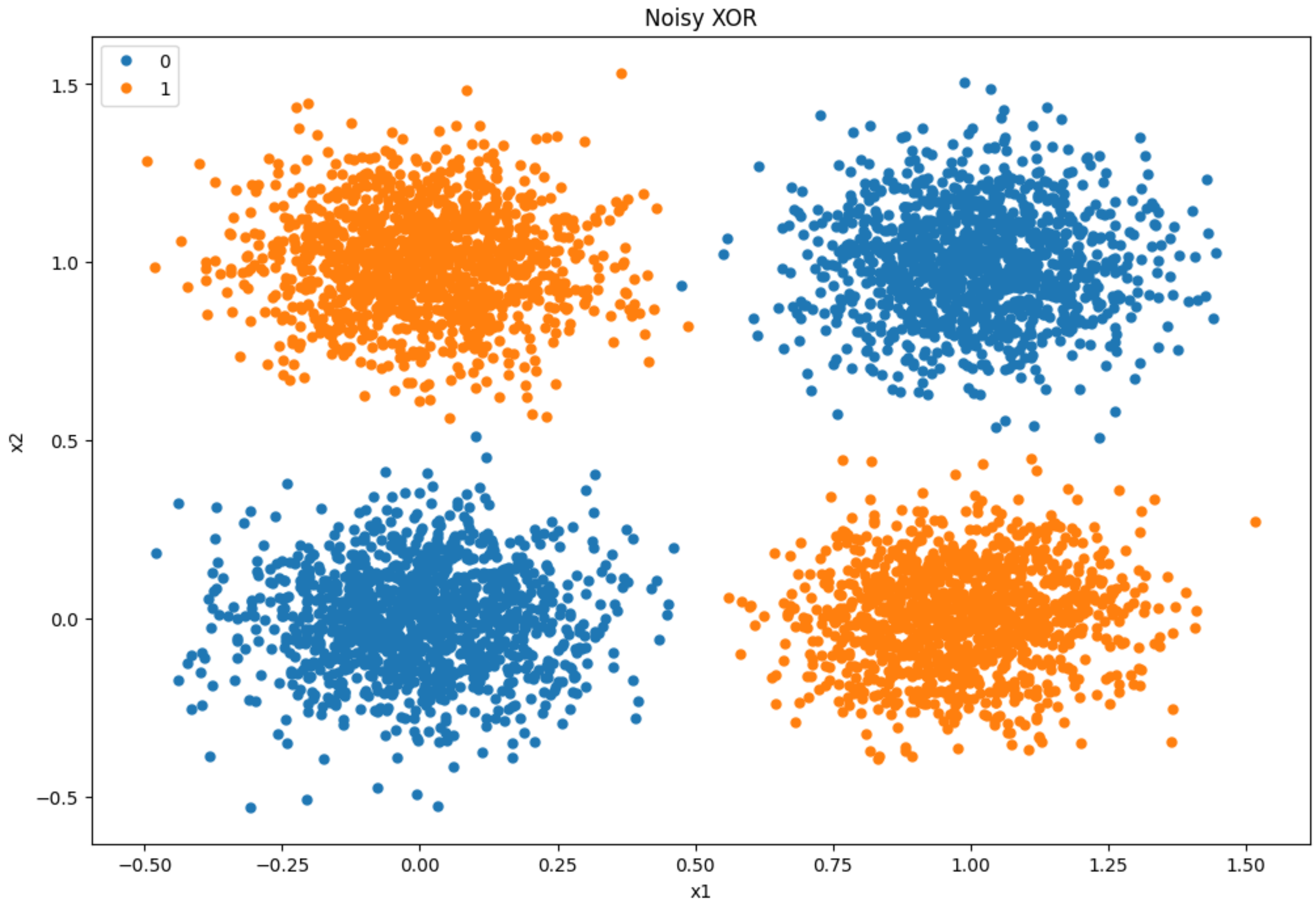
In [33]:
```python
# set X (training data) and y (target variable)
cols = data.shape[1]
X = data.iloc[:,0:cols-1]
y = data.iloc[:,cols-1:cols]

# The cost function is expecting numpy matrices so we need to convert X and y before we can use them.
X = np.matrix(X.values)
y = np.matrix(y.values)
```

Let's make a sloppy plotting function for our binary data.

In [34]:
```
# Sloppy function for plotting our data
def plot_data(X, y_prob):

    fig, ax = plt.subplots(figsize=(12,8))
    ax.margins(0.05) # Optional, just adds 5% padding to the autoscaling

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    ax.plot(X[indices_0, 0], X[indices_0,1], marker='o', linestyle='', ms=5, label='0')
    ax.plot(X[indices_1, 0], X[indices_1,1], marker='o', linestyle='', ms=5, label='1')

    ax.legend()
    ax.legend(loc=2)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title('Noisy XOR')
    plt.show()
```

Now let's plot it.

In [35]:
```
plot_data(X, y)
```



Now let's create functions for forward and backward prop through the layers and we are off...

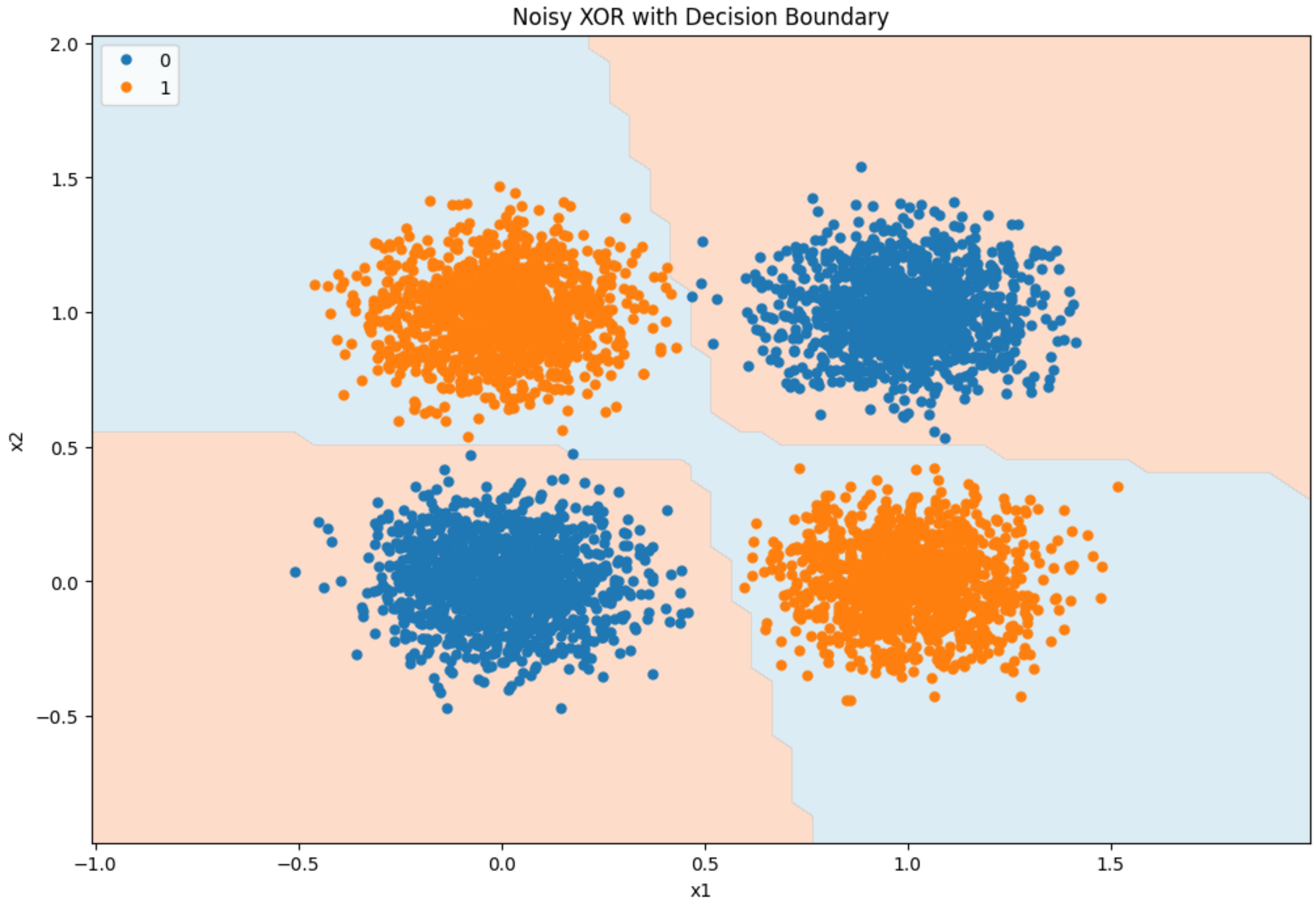
Mini batches + soft plus

In [37]:
```
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

# Data generation code (same as before)
data = pd.DataFrame(np.zeros((5000, 3)), columns=['x1', 'x2', 'y'])
for i in range(len(data.index)):
    x1 = 1.0 * random.randint(0, 1)
    x2 = 1.0 * random.randint(0, 1)
    y = 1.0 * np.logical_xor(x1 == 1, x2 == 1)
    x1 = x1 + 0.15 * np.random.normal()
    x2 = x2 + 0.15 * np.random.normal()
    data.iloc[i, 0] = x1
    data.iloc[i, 1] = x2
    data.iloc[i, 2] = y


cols = data.shape[1]
X = data.iloc[:, 0:cols - 1].values
y = data.iloc[:, cols - 1:cols].values

# Sloppy function for plotting our data
def plot_data(X, y_prob, W1, b1, W2, b2):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.margins(0.05)  # Optional, just adds 5% padding to the autoscaling

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    # Plot the data points
    ax.plot(X[indices_0, 0], X[indices_0, 1], marker='o', linestyle='', ms=5, label='0')
    ax.plot(X[indices_1, 0], X[indices_1, 1], marker='o', linestyle='', ms=5, label='1')

    # Create a grid to plot decision boundary
    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, 0.05), np.arange(y_min, y_max, 0.05))

    # Forward propagate on the grid points to get predictions
    grid_points = np.c_[xx.ravel(), yy.ravel()]
    _, _, Z = forward_propagation(grid_points, W1, b1, W2, b2)
    Z = Z > 0.5  # Use threshold of 0.5 to classify

    # Reshape the predictions to match the grid shape
    Z = Z.reshape(xx.shape)

    # Plot decision boundary by coloring regions
    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)

    # Set labels and title
    ax.legend(loc=2)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title('Noisy XOR with Decision Boundary')

    plt.show()
```

```python
# Neural network code
input_size = 2
hidden_size = 6
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

def relu(z):
    return np.maximum(0, z)

def softplus(z):
    return np.log(1 + np.exp(z))

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softplus(Z2)
    return Z1, A1, A2

def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

# Training with mini-batch gradient descent
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    m = X.shape[0]
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)
```

```python
            # Calculate the Softplus loss
            loss = np.mean(softplus(A2 - y_batch))

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%')

    return W1, b1, W2, b2



# Set hyperparameters
learning_rate = 0.1
epochs = 1000
batch_size = 32  # You can adjust the batch size as needed

# Train the network
W1, b1, W2, b2 = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size)

# Make predictions
_, _, y_prob = forward_propagation(X, W1, b1, W2, b2)

# Plot the data and predictions
# plot_data(X, y_prob)
plot_data(X, y_prob, W1, b1, W2, b2)
```

```
Epoch 0, Loss: 0.7189, Accuracy: 100.00%
Epoch 100, Loss: 0.6696, Accuracy: 100.00%
Epoch 200, Loss: 0.6596, Accuracy: 100.00%
Epoch 300, Loss: 0.6941, Accuracy: 100.00%
Epoch 400, Loss: 0.6934, Accuracy: 100.00%
Epoch 500, Loss: 0.5977, Accuracy: 87.50%
Epoch 600, Loss: 0.6809, Accuracy: 100.00%
Epoch 700, Loss: 0.6968, Accuracy: 100.00%
Epoch 800, Loss: 0.6935, Accuracy: 100.00%
Epoch 900, Loss: 0.6874, Accuracy: 100.00%
```



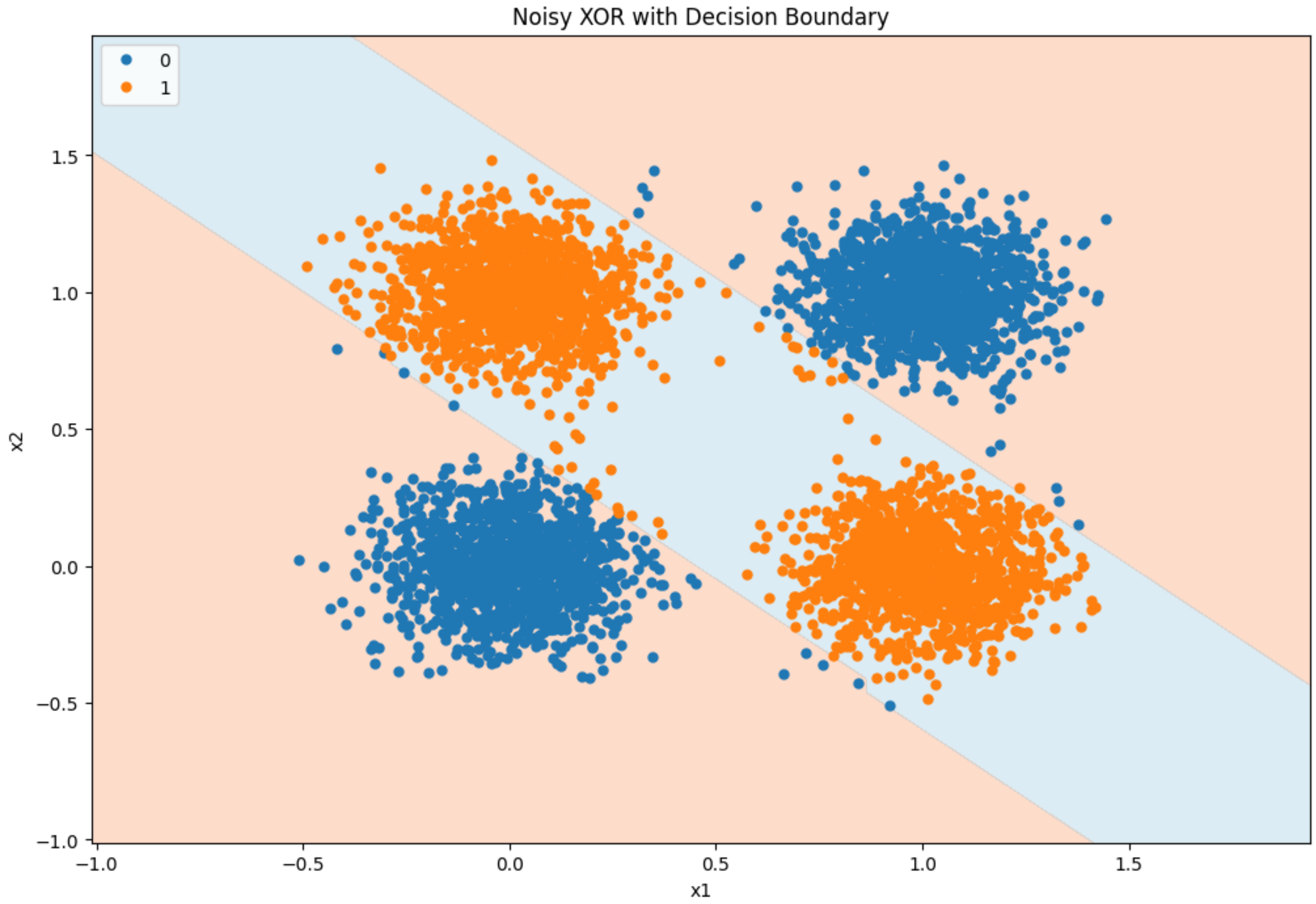
Mini batches + cross entropy loss

In [38]:
```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

# Data generation code (same as before)
data = pd.DataFrame(np.zeros((5000, 3)), columns=['x1', 'x2', 'y'])
for i in range(len(data.index)):
    x1 = 1.0 * random.randint(0, 1)
    x2 = 1.0 * random.randint(0, 1)
    y = 1.0 * np.logical_xor(x1 == 1, x2 == 1)
    x1 = x1 + 0.15 * np.random.normal()
    x2 = x2 + 0.15 * np.random.normal()
    data.iloc[i, 0] = x1
    data.iloc[i, 1] = x2
    data.iloc[i, 2] = y


cols = data.shape[1]
X = data.iloc[:, 0:cols - 1].values
y = data.iloc[:, cols - 1:cols].values

# Sloppy function for plotting our data
def plot_data(X, y_prob, W1, b1, W2, b2):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.margins(0.05)  # Optional, just adds 5% padding to the autoscaling

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    # Plot the data points
    ax.plot(X[indices_0, 0], X[indices_0, 1], marker='o', linestyle='', ms=5, label='0')
    ax.plot(X[indices_1, 0], X[indices_1, 1], marker='o', linestyle='', ms=5, label='1')

    # Create a grid to plot decision boundary
    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, 0.05), np.arange(y_min, y_max, 0.05))

    # Forward propagate on the grid points to get predictions
    grid_points = np.c_[xx.ravel(), yy.ravel()]
    _, _, Z = forward_propagation(grid_points, W1, b1, W2, b2)
    Z = Z > 0.5  # Use threshold of 0.5 to classify

    # Reshape the predictions to match the grid shape
    Z = Z.reshape(xx.shape)

    # Plot decision boundary by coloring regions
    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)

    # Set labels and title
    ax.legend(loc=2)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title('Noisy XOR with Decision Boundary')

    plt.show()
```

```python
# Neural network code
input_size = 2
hidden_size = 6
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

# Replace softplus with sigmoid for output layer
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Modify the forward_propagation function to use sigmoid for A2
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)  # Use sigmoid for binary classification
    return Z1, A1, A2

# Add binary cross-entropy loss function
def binary_crossentropy_loss(A2, y):
    m = y.shape[0]
    loss = -(1/m) * np.sum(y * np.log(A2) + (1 - y) * np.log(1 - A2))
    return loss

# Update backward propagation to match the new activation and loss
def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y  # No change here since cross-entropy loss with sigmoid leads to this
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

# Modify the training loop to calculate cross-entropy loss
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    m = X.shape[0]
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)
```

```python
            # Calculate the binary cross-entropy loss
            loss = binary_crossentropy_loss(A2, y_batch)

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%')

    return W1, b1, W2, b2

# Set hyperparameters
learning_rate = 0.1
epochs = 1000
batch_size = 32

# Train the network
W1, b1, W2, b2 = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size)

# Make predictions
_, _, y_prob = forward_propagation(X, W1, b1, W2, b2)

# Plot the data and predictions
plot_data(X, y_prob, W1, b1, W2, b2)
```

```
Epoch 0, Loss: 0.3381, Accuracy: 100.00%
Epoch 100, Loss: 0.0010, Accuracy: 100.00%
Epoch 200, Loss: 0.0178, Accuracy: 100.00%
Epoch 300, Loss: 0.0832, Accuracy: 100.00%
Epoch 400, Loss: 0.0039, Accuracy: 100.00%
Epoch 500, Loss: 0.0001, Accuracy: 100.00%
Epoch 600, Loss: 0.2894, Accuracy: 87.50%
Epoch 700, Loss: 0.3640, Accuracy: 87.50%
Epoch 800, Loss: 0.0100, Accuracy: 100.00%
Epoch 900, Loss: 0.0239, Accuracy: 100.00%
```



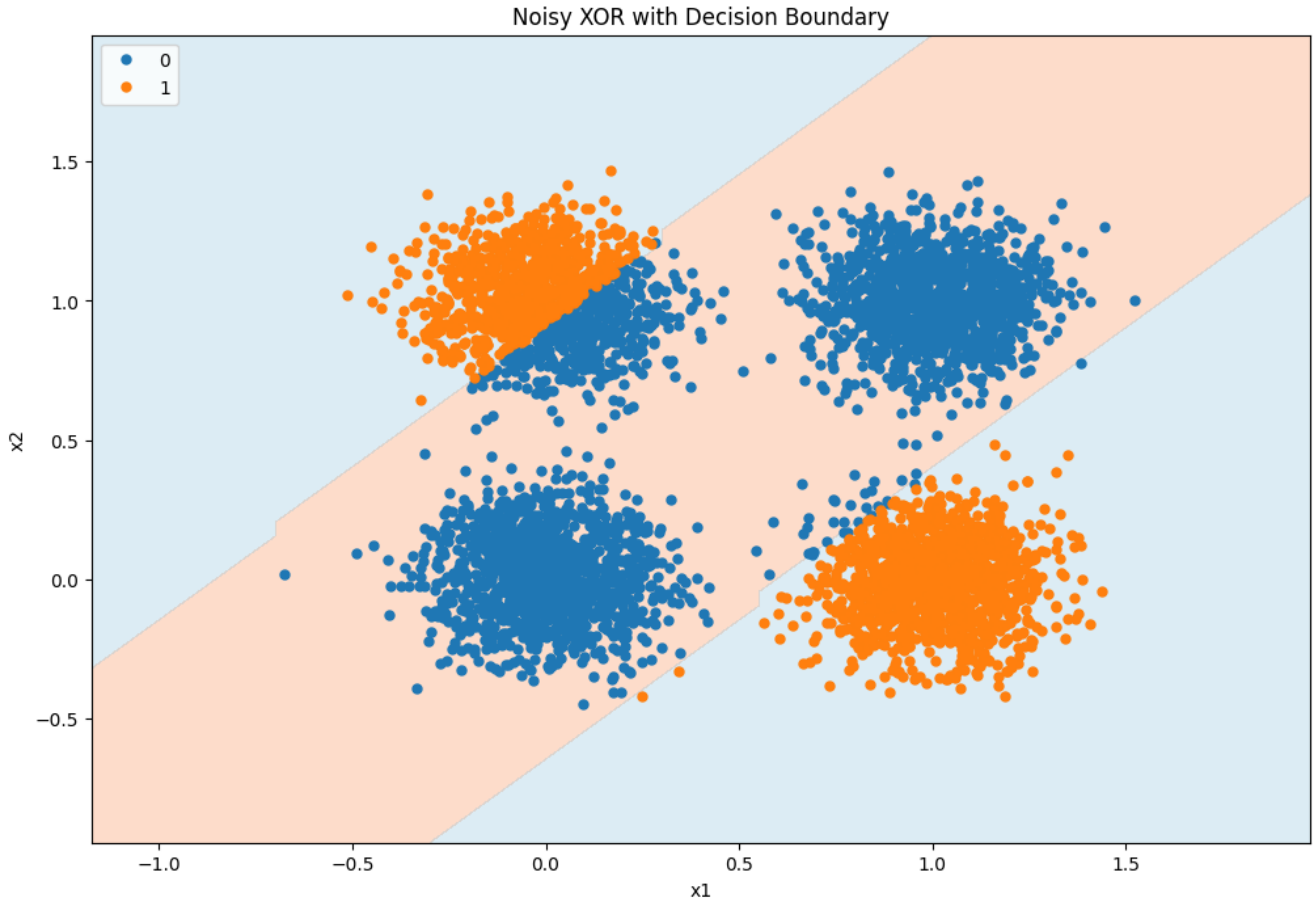
Mini batches + Softmax + hidden units: 3

In [39]:
```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

# Data generation code (same as before)
data = pd.DataFrame(np.zeros((5000, 3)), columns=['x1', 'x2', 'y'])
for i in range(len(data.index)):
    x1 = 1.0 * random.randint(0, 1)
    x2 = 1.0 * random.randint(0, 1)
    y = 1.0 * np.logical_xor(x1 == 1, x2 == 1)
    x1 = x1 + 0.15 * np.random.normal()
    x2 = x2 + 0.15 * np.random.normal()
    data.iloc[i, 0] = x1
    data.iloc[i, 1] = x2
    data.iloc[i, 2] = y


cols = data.shape[1]
X = data.iloc[:, 0:cols - 1].values
y = data.iloc[:, cols - 1:cols].values

# Sloppy function for plotting our data
def plot_data(X, y_prob, W1, b1, W2, b2):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.margins(0.05)  # Optional, just adds 5% padding to the autoscaling

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    # Plot the data points
    ax.plot(X[indices_0, 0], X[indices_0, 1], marker='o', linestyle='', ms=5, label='0')
    ax.plot(X[indices_1, 0], X[indices_1, 1], marker='o', linestyle='', ms=5, label='1')

    # Create a grid to plot decision boundary
    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, 0.05), np.arange(y_min, y_max, 0.05))

    # Forward propagate on the grid points to get predictions
    grid_points = np.c_[xx.ravel(), yy.ravel()]
    _, _, Z = forward_propagation(grid_points, W1, b1, W2, b2)
    Z = Z > 0.5  # Use threshold of 0.5 to classify

    # Reshape the predictions to match the grid shape
    Z = Z.reshape(xx.shape)

    # Plot decision boundary by coloring regions
    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)

    # Set labels and title
    ax.legend(loc=2)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title('Noisy XOR with Decision Boundary')

    plt.show()
```

```python
# Neural network code
input_size = 2
hidden_size = 3
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

def relu(z):
    return np.maximum(0, z)

def softplus(z):
    return np.log(1 + np.exp(z))

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softplus(Z2)
    return Z1, A1, A2

def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

# Training with mini-batch gradient descent
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    m = X.shape[0]
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)
```

```
            # Calculate the Softplus loss
            loss = np.mean(softplus(A2 - y_batch))

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%')

    return W1, b1, W2, b2



# Set hyperparameters
learning_rate = 0.1
epochs = 1000
batch_size = 32  # You can adjust the batch size as needed

# Train the network
W1, b1, W2, b2 = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size)

# Make predictions
_, _, y_prob = forward_propagation(X, W1, b1, W2, b2)

# Plot the data and predictions
# plot_data(X, y_prob)
plot_data(X, y_prob, W1, b1, W2, b2)
```

```
Epoch 0, Loss: 0.7449, Accuracy: 62.50%
Epoch 100, Loss: 0.6805, Accuracy: 100.00%
Epoch 200, Loss: 0.7708, Accuracy: 100.00%
Epoch 300, Loss: 0.7137, Accuracy: 100.00%
Epoch 400, Loss: 0.7076, Accuracy: 100.00%
Epoch 500, Loss: 0.7092, Accuracy: 100.00%
Epoch 600, Loss: 0.6527, Accuracy: 87.50%
Epoch 700, Loss: 0.6638, Accuracy: 100.00%
Epoch 800, Loss: 0.7010, Accuracy: 100.00%
Epoch 900, Loss: 0.7003, Accuracy: 100.00%
```



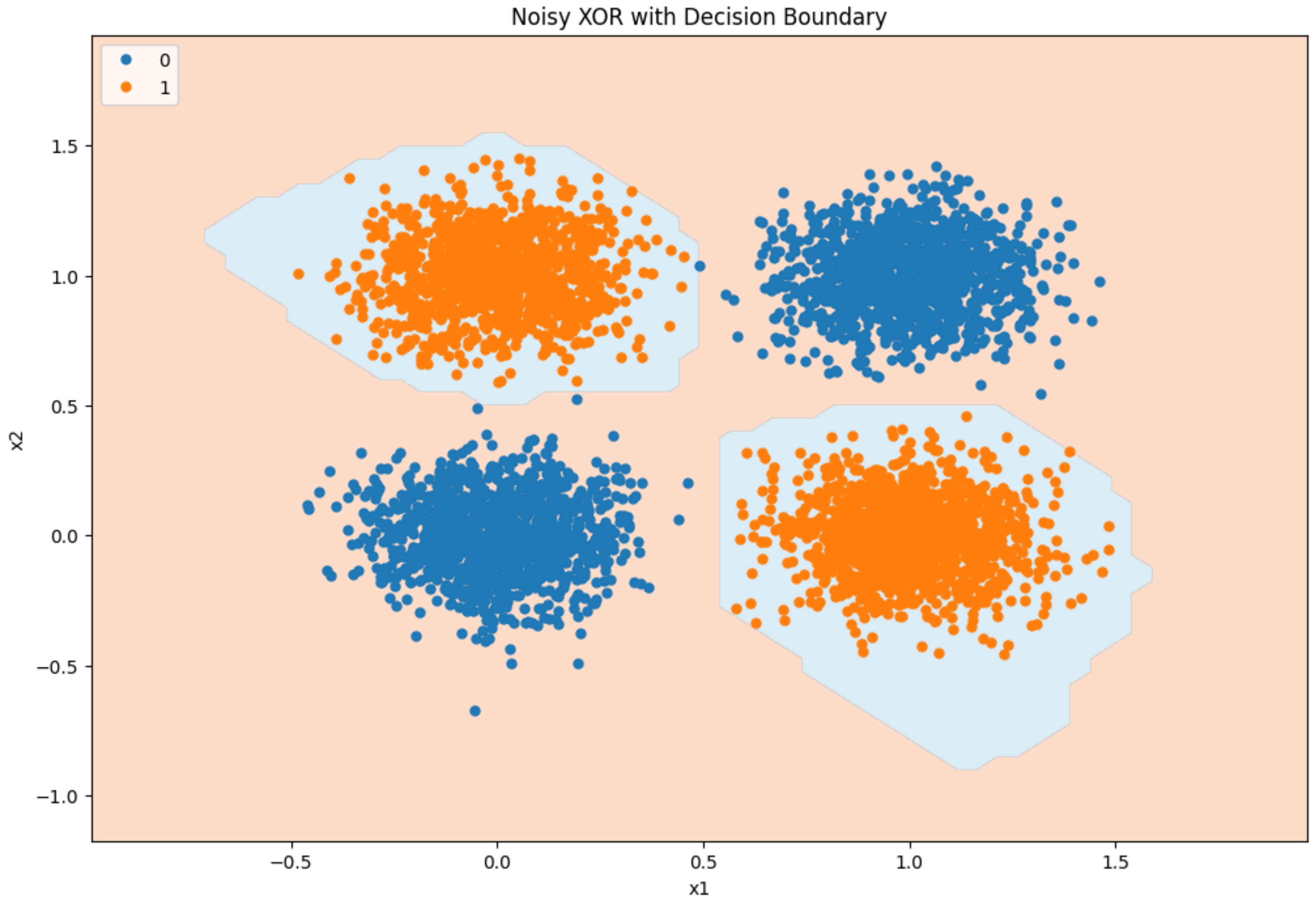
Mini batches + Softmax + hidden units: 512

In [40]:
```
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

# Data generation code (same as before)
data = pd.DataFrame(np.zeros((5000, 3)), columns=['x1', 'x2', 'y'])
for i in range(len(data.index)):
    x1 = 1.0 * random.randint(0, 1)
    x2 = 1.0 * random.randint(0, 1)
    y = 1.0 * np.logical_xor(x1 == 1, x2 == 1)
    x1 = x1 + 0.15 * np.random.normal()
    x2 = x2 + 0.15 * np.random.normal()
    data.iloc[i, 0] = x1
    data.iloc[i, 1] = x2
    data.iloc[i, 2] = y


cols = data.shape[1]
X = data.iloc[:, 0:cols - 1].values
y = data.iloc[:, cols - 1:cols].values

# Sloppy function for plotting our data
def plot_data(X, y_prob, W1, b1, W2, b2):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.margins(0.05)  # Optional, just adds 5% padding to the autoscaling

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    # Plot the data points
    ax.plot(X[indices_0, 0], X[indices_0, 1], marker='o', linestyle='', ms=5, label='0')
    ax.plot(X[indices_1, 0], X[indices_1, 1], marker='o', linestyle='', ms=5, label='1')

    # Create a grid to plot decision boundary
    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, 0.05), np.arange(y_min, y_max, 0.05))

    # Forward propagate on the grid points to get predictions
    grid_points = np.c_[xx.ravel(), yy.ravel()]
    _, _, Z = forward_propagation(grid_points, W1, b1, W2, b2)
    Z = Z > 0.5  # Use threshold of 0.5 to classify

    # Reshape the predictions to match the grid shape
    Z = Z.reshape(xx.shape)

    # Plot decision boundary by coloring regions
    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)

    # Set labels and title
    ax.legend(loc=2)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title('Noisy XOR with Decision Boundary')

    plt.show()
```

```python
# Neural network code
input_size = 2
hidden_size = 512
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

def relu(z):
    return np.maximum(0, z)

def softplus(z):
    return np.log(1 + np.exp(z))

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softplus(Z2)
    return Z1, A1, A2

def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

# Training with mini-batch gradient descent
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    m = X.shape[0]
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)
```

```python
            # Calculate the Softplus loss
            loss = np.mean(softplus(A2 - y_batch))

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%')

    return W1, b1, W2, b2



# Set hyperparameters
learning_rate = 0.1
epochs = 1000
batch_size = 32  # You can adjust the batch size as needed

# Train the network
W1, b1, W2, b2 = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size)

# Make predictions
_, _, y_prob = forward_propagation(X, W1, b1, W2, b2)

# Plot the data and predictions
# plot_data(X, y_prob)
plot_data(X, y_prob, W1, b1, W2, b2)
```

```
Epoch 0, Loss: 0.8223, Accuracy: 100.00%
Epoch 100, Loss: 0.6891, Accuracy: 100.00%
Epoch 200, Loss: 0.7227, Accuracy: 100.00%
Epoch 300, Loss: 0.7079, Accuracy: 100.00%
Epoch 400, Loss: 0.6904, Accuracy: 100.00%
Epoch 500, Loss: 0.6905, Accuracy: 100.00%
Epoch 600, Loss: 0.6787, Accuracy: 100.00%
Epoch 700, Loss: 0.6941, Accuracy: 100.00%
Epoch 800, Loss: 0.7017, Accuracy: 100.00%
Epoch 900, Loss: 0.6950, Accuracy: 100.00%
```



Disucssion about hidden units

- We can see that with the increase in hidden units, while there's a more detailed boundary, it's also much more fitting, possibly overfitting in the case of hidden units: 512.

Different learning schedules - constant learning rate and slow decay

In [41]:
```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

# Data generation code (same as before)
data = pd.DataFrame(np.zeros((5000, 3)), columns=['x1', 'x2', 'y'])
for i in range(len(data.index)):
    x1 = 1.0 * random.randint(0, 1)
    x2 = 1.0 * random.randint(0, 1)
    y = 1.0 * np.logical_xor(x1 == 1, x2 == 1)
    x1 = x1 + 0.15 * np.random.normal()
    x2 = x2 + 0.15 * np.random.normal()
    data.iloc[i, 0] = x1
    data.iloc[i, 1] = x2
    data.iloc[i, 2] = y

cols = data.shape[1]
X = data.iloc[:, 0:cols - 1].values
y = data.iloc[:, cols - 1:cols].values

# Sloppy function for plotting our data
def plot_data(X, y_prob, W1, b1, W2, b2):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.margins(0.05)

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    ax.plot(X[indices_0, 0], X[indices_0, 1], marker='o', linestyle='', ms=5, label='0')
    ax.plot(X[indices_1, 0], X[indices_1, 1], marker='o', linestyle='', ms=5, label='1')

    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, 0.05), np.arange(y_min, y_max, 0.05))

    grid_points = np.c_[xx.ravel(), yy.ravel()]
    _, _, Z = forward_propagation(grid_points, W1, b1, W2, b2)
    Z = Z > 0.5
    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)

    ax.legend(loc=2)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title('Noisy XOR with Decision Boundary')

    plt.show()

# Neural network code
input_size = 2
hidden_size = 6
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
```

```python
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

def relu(z):
    return np.maximum(0, z)

def softplus(z):
    return np.log(1 + np.exp(z))

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softplus(Z2)
    return Z1, A1, A2

def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

# Training with mini-batch gradient descent
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size, decay=False):
    m = X.shape[0]
    decay_rate = 0.96  # Rate for learning rate decay
    decay_steps = 100  # How often to apply decay
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Update learning rate if using decay
        if decay:
            lr = learning_rate * (decay_rate ** (i // decay_steps))
        else:
            lr = learning_rate

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)
```

```python
            # Calculate the Softplus loss
            loss = np.mean(softplus(A2 - y_batch))

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, lr)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%, Learning Rate: {lr:.6f}')

    return W1, b1, W2, b2

# Set hyperparameters
learning_rate = 0.1
epochs = 1000
batch_size = 32  # You can adjust the batch size as needed

# Train the network with a constant learning rate
print("Training with Constant Learning Rate:")
W1_const, b1_const, W2_const, b2_const = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size, decay=False)

# Make predictions with constant learning rate
_, _, y_prob_const = forward_propagation(X, W1_const, b1_const, W2_const, b2_const)

# Plot the data and predictions for constant learning rate
# plot_data(X, y_prob_const, W1_const, b1_const, W2_const, b2_const)

# Reset weights for the second training
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

# Train the network with learning rate decay
print("Training with Learning Rate Decay:")
W1_decay, b1_decay, W2_decay, b2_decay = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size, decay=True)

# Make predictions with learning rate decay
_, _, y_prob_decay = forward_propagation(X, W1_decay, b1_decay, W2_decay, b2_decay)

# Plot the data and predictions for learning rate decay
# plot_data(X, y_prob_decay, W1_decay, b1_decay, W2_decay, b2_decay)
```

```
Training with Constant Learning Rate:
Epoch 0, Loss: 0.6741, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 100, Loss: 0.6761, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 200, Loss: 0.6893, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 300, Loss: 0.6869, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 400, Loss: 0.6942, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 500, Loss: 0.6909, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 600, Loss: 0.6920, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 700, Loss: 0.6939, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 800, Loss: 0.6864, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 900, Loss: 0.6929, Accuracy: 100.00%, Learning Rate: 0.100000
Training with Learning Rate Decay:
Epoch 0, Loss: 0.6957, Accuracy: 100.00%, Learning Rate: 0.100000
Epoch 100, Loss: 0.6530, Accuracy: 87.50%, Learning Rate: 0.096000
Epoch 200, Loss: 0.7034, Accuracy: 100.00%, Learning Rate: 0.092160
Epoch 300, Loss: 0.6931, Accuracy: 100.00%, Learning Rate: 0.088474
Epoch 400, Loss: 0.6933, Accuracy: 100.00%, Learning Rate: 0.084935
Epoch 500, Loss: 0.7039, Accuracy: 100.00%, Learning Rate: 0.081537
Epoch 600, Loss: 0.6935, Accuracy: 100.00%, Learning Rate: 0.078276
Epoch 700, Loss: 0.6932, Accuracy: 100.00%, Learning Rate: 0.075145
Epoch 800, Loss: 0.6937, Accuracy: 100.00%, Learning Rate: 0.072139
Epoch 900, Loss: 0.6931, Accuracy: 100.00%, Learning Rate: 0.069253
```

3 input features

In [2]:
```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

# Data generation code for 3 input features
def generate_data(num_samples):
    data = pd.DataFrame(np.zeros((num_samples, 4)), columns=['x1', 'x2', 'x3', 'y'])
    for i in range(len(data.index)):
        x1 = 1.0 * random.randint(0, 1)
        x2 = 1.0 * random.randint(0, 1)
        x3 = 1.0 * random.randint(0, 1)
        y = 1.0 * ((x1 + x2 + x3) == 1)  # Output is 1 if exactly one input is 1
        # Adding noise
        x1 += 0.15 * np.random.normal()
        x2 += 0.15 * np.random.normal()
        x3 += 0.15 * np.random.normal()
        data.iloc[i, 0] = x1
        data.iloc[i, 1] = x2
        data.iloc[i, 2] = x3
        data.iloc[i, 3] = y

    return data

# Generate training and test data
train_data = generate_data(5000)
test_data = generate_data(1000)

# Prepare input features and target variable
X_train = train_data.iloc[:, 0:3].values
y_train = train_data.iloc[:, 3].values.reshape(-1, 1)

X_test = test_data.iloc[:, 0:3].values
y_test = test_data.iloc[:, 3].values.reshape(-1, 1)

# Neural network code
input_size = 3  # Update input size to 3 for 3 features
hidden_size = 6
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

def relu(z):
    return np.maximum(0, z)

def softplus(z):
    return np.log(1 + np.exp(z))

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softplus(Z2)
    return Z1, A1, A2
```

```python
def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

# Training with mini-batch gradient descent
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    m = X.shape[0]
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)

            # Calculate the Softplus loss
            loss = np.mean(softplus(A2 - y_batch))

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%')

    return W1, b1, W2, b2

# Set hyperparameters
learning_rate = 0.1
```

```python
epochs = 1000
batch_size = 32

# Train the network
W1, b1, W2, b2 = train(X_train, y_train, W1, b1, W2, b2, learning_rate, epochs, batch_size)

# Make predictions on the test set
_, _, y_prob_test = forward_propagation(X_test, W1, b1, W2, b2)

# Calculate test set accuracy
test_predictions = (y_prob_test > 0.5).astype(int)
test_accuracy = np.mean(test_predictions == y_test) * 100

print(f'Test Accuracy: {test_accuracy:.2f}%')
```

```
Epoch 0, Loss: 0.6304, Accuracy: 75.00%
Epoch 100, Loss: 0.6940, Accuracy: 100.00%
Epoch 200, Loss: 0.7139, Accuracy: 100.00%
Epoch 300, Loss: 0.6846, Accuracy: 100.00%
Epoch 400, Loss: 0.7545, Accuracy: 87.50%
Epoch 500, Loss: 0.6924, Accuracy: 100.00%
Epoch 600, Loss: 0.6931, Accuracy: 100.00%
Epoch 700, Loss: 0.6779, Accuracy: 100.00%
Epoch 800, Loss: 0.6912, Accuracy: 100.00%
Epoch 900, Loss: 0.6725, Accuracy: 100.00%
Test Accuracy: 98.90%
```

Training on HW1 data

data link: https://drive.google.com/drive/folders/1Y_IUvrpNJi5gdBnyQbpYRqoC3YB4PO6y?usp=drive_link (https://drive.google.com/drive/folders/1Y_IUvrpNJi5gdBnyQbpYRqoC3YB4PO6y?usp=drive_link)

In [7]:
```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

df = pd.read_csv(r'C:\Users\Rosh\Desktop\all\Columbia\classes\Y4S1\DL_for_CV\hw1.ipynb\raw_data\weather\training-data.csv', encoding='ISO-8859-1')

df_selected = df[['Normalized water pressure', 'normalized humidity']]

df['sunny?'] = (df['Sunny or not'] == 'Sunny').astype(int) # Sunny = 1, not sunny = 0

print(df_selected.head())

# Prepare the features and labels for training
X = df_selected.values  # Features
y = df['sunny?'].values.reshape(-1, 1)  # Labels

df_test = pd.read_csv(r'C:\Users\Rosh\Desktop\all\Columbia\classes\Y4S1\DL_for_CV\hw1.ipynb\raw_data\weather\test-data.csv', encoding='ISO-8859-1')

df_test_selected = df_test[['Normalized water pressure', 'normalized humidity']]

df_test['sunny?'] = (df_test['Sunny or not'] == 'Sunny').astype(int) # Sunny = 1, not sunny = 0 into "sunny?" column

X_test = df_test[['Normalized water pressure', 'normalized humidity']]
y_test = (df_test['Sunny or not'] == 'Sunny').astype(int)  # Assuming 'Sunny' is your target column
```

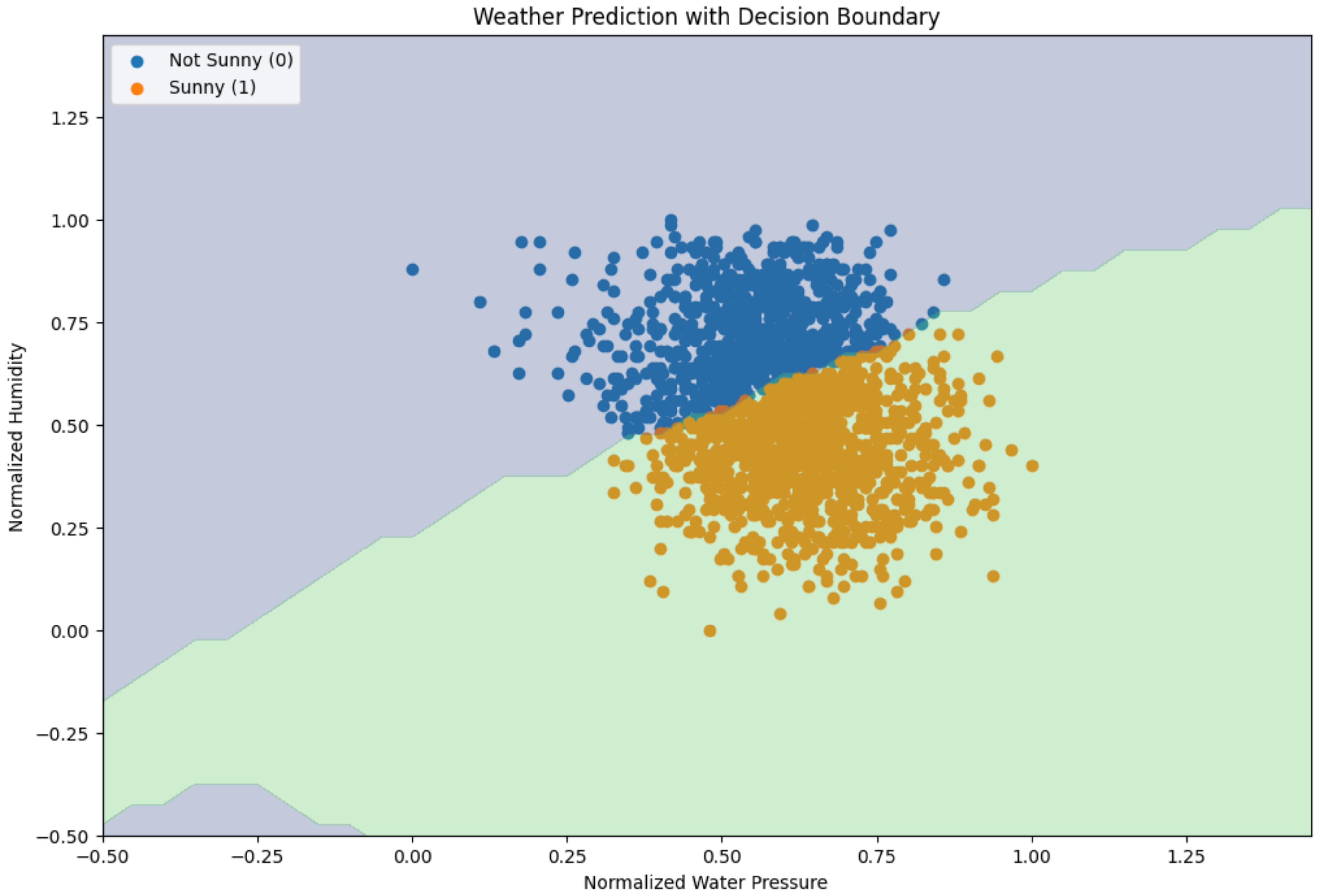
```
   Normalized water pressure  normalized humidity
0                   0.417143             0.413333
1                   0.428571             0.413333
2                   0.428571             0.400000
3                   0.508571             0.493333
4                   0.617143             0.306667
```

In [9]:
```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

# Load the data from the CSV file
df = pd.read_csv(r'C:\Users\Rosh\Desktop\all\Columbia\classes\Y4S1\DL_for_CV\hw1.ipynb\raw_data\weather\training-data.csv', encoding='ISO-8859-1')

# Select relevant features
df_selected = df[['Normalized water pressure', 'normalized humidity']]

# Convert target variable to binary (1 for sunny, 0 for not sunny)
df['sunny?'] = (df['Sunny or not'] == 'Sunny').astype(int)

# Prepare the features and labels for training
X = df_selected.values  # Features
y = df['sunny?'].values.reshape(-1, 1)  # Labels

# Neural network parameters
input_size = X.shape[1]  # Number of features
hidden_size = 6
output_size = 1

np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

def relu(z):
    return np.maximum(0, z)

def softplus(z):
    return np.log(1 + np.exp(z))

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softplus(Z2)
    return Z1, A1, A2

def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.reshape(np.sum(dZ2, axis=0), (1, -1))
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, (Z1 > 0).astype(int))
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.reshape(np.sum(dZ1, axis=0), (1, -1))
    return dW1, db1, dW2, db2

def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
```

```
    return W1, b1, W2, b2

# Training with mini-batch gradient descent
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    m = X.shape[0]
    for i in range(epochs):
        # Shuffle the data at the start of each epoch
        indices = np.arange(m)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Mini-batch training
        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward propagation
            Z1, A1, A2 = forward_propagation(X_batch, W1, b1, W2, b2)

            # Calculate the Softplus loss
            loss = np.mean(softplus(A2 - y_batch))

            # Binary predictions
            predictions = (A2 > 0.5).astype(int)

            # Calculate accuracy
            accuracy = np.mean(predictions == y_batch) * 100

            # Backward propagation
            dW1, db1, dW2, db2 = backward_propagation(X_batch, y_batch, Z1, A1, A2, W2)

            # Update parameters
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        # Print the loss and accuracy every 100 epochs
        if i % 100 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%')

    return W1, b1, W2, b2

# Set hyperparameters
learning_rate = 0.1
epochs = 1000
batch_size = 32  # You can adjust the batch size as needed

# Train the network
W1, b1, W2, b2 = train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size)

# Make predictions
_, _, y_prob = forward_propagation(X, W1, b1, W2, b2)

# Prepare the test features and labels
X_test = df_test_selected.values  # Use the selected features
y_test = df_test['sunny?'].values.reshape(-1, 1)  # Use the binary labels

# Make predictions on the test set
_, _, y_test_prob = forward_propagation(X_test, W1, b1, W2, b2)
```

```python
# Convert probabilities to binary predictions
y_test_predictions = (y_test_prob > 0.5).astype(int)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_test_predictions) * 100
print(f'Test Accuracy: {accuracy:.2f}%')


# Function for plotting
def plot_data(X, y_prob, W1, b1, W2, b2):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.margins(0.05)  # Optional, just adds 5% padding to the autoscaling

    y_predict = y_prob > 0.5
    indices_0 = [k for k in range(0, X.shape[0]) if not y_predict[k]]
    indices_1 = [k for k in range(0, X.shape[0]) if y_predict[k]]

    # Plot the data points
    ax.scatter(X[indices_0, 0], X[indices_0, 1], marker='o', label='Not Sunny (0)')
    ax.scatter(X[indices_1, 0], X[indices_1, 1], marker='o', label='Sunny (1)')

    # Create a grid to plot decision boundary
    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, 0.05), np.arange(y_min, y_max, 0.05))

    # Forward propagate on the grid points to get predictions
    grid_points = np.c_[xx.ravel(), yy.ravel()]
    _, _, Z = forward_propagation(grid_points, W1, b1, W2, b2)
    Z = Z > 0.5  # Use threshold of 0.5 to classify

    # Reshape the predictions to match the grid shape
    Z = Z.reshape(xx.shape)

    # Plot decision boundary by coloring regions
    ax.contourf(xx, yy, Z, alpha=0.3)

    # Set labels and title
    ax.legend(loc=2)
    ax.set_xlabel('Normalized Water Pressure')
    ax.set_ylabel('Normalized Humidity')
    ax.set_title('Weather Prediction with Decision Boundary')

    plt.show()

# Plot the data and predictions
plot_data(X, y_prob, W1, b1, W2, b2)
```

```
Epoch 0, Loss: 0.6075, Accuracy: 100.00%
Epoch 100, Loss: 0.7383, Accuracy: 100.00%
Epoch 200, Loss: 0.6877, Accuracy: 50.00%
Epoch 300, Loss: 0.8104, Accuracy: 83.33%
Epoch 400, Loss: 0.6501, Accuracy: 83.33%
Epoch 500, Loss: 0.7372, Accuracy: 100.00%
Epoch 600, Loss: 0.5984, Accuracy: 83.33%
Epoch 700, Loss: 0.6571, Accuracy: 83.33%
Epoch 800, Loss: 0.7766, Accuracy: 16.67%
Epoch 900, Loss: 0.5875, Accuracy: 83.33%
Test Accuracy: 76.08%
```


Weather Prediction with Decision Boundary
Not Sunny (0)
Sunny (1)
Normalized Humidity
Normalized Water Pressure
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25