Lab-4: Single-Layer Neural Network Alarm System

R Abhijit Srivathsan - 2448044

In [1]:
```
import numpy as np
np.set_printoptions(precision=4, suppress=True)
np.random.seed(42)
```

Mathematical Foundation

This implementation demonstrates the fundamental concepts of neural network training:

Network Architecture

- **Input Layer**: 2 neurons (x_1, x_2) representing sensor inputs
- **Output Layer**: 1 neuron with sigmoid activation producing alarm signal
- **Parameters**: 2 weights (w_1, w_2) and 1 bias (b)

Forward Propagation

The network computes:

1. **Linear combination**: $z = w_1x_1 + w_2x_2 + b$
2. **Activation**: $\hat{y} = \sigma(z) = 1/(1 + e^{-z})$

Loss Function

Mean Squared Error (MSE): $E = (1/n)\Sigma(y - \hat{y})^2$

Learning Algorithm

Delta Rule with batch gradient descent:

- Weight update: $w \leftarrow w - \eta(\partial E/\partial w)$
- Bias update: $b \leftarrow b - \eta(\partial E/\partial b)$
- Learning rate: $\eta = 0.1$

1. Training data

For this scenario we assume that **any** sensor being active should raise the alarm (logical **OR**).

In [2]:
```
# Inputs: all 4 possible combinations
data_x = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]], dtype=float)

# Expected outputs according to OR logic
data_y = np.array([[0], [1], [1], [1]], dtype=float)
```

Truth Table for Alarm System

The training data represents all possible combinations of sensor inputs with their expected alarm outputs:

x_1 (Motion)	x_2 (Door)	y (Alarm)	Logic
0	0	0	No sensors active → No alarm
0	1	1	Door open → Alarm
1	0	1	Motion detected → Alarm
1	1	1	Both sensors → Alarm

This implements **logical OR** behavior: the alarm should sound if **any** sensor detects activity. This is a linearly separable problem that a single-layer perceptron can solve.

2. Helper functions

In [3]:
```python
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(a):
    return a * (1 - a)
```

Sigmoid Activation Function

The **sigmoid function** $\sigma(z) = 1/(1 + e^{-z})$ has several important properties:

- **Range**: (0, 1) - perfect for binary classification
- **Differentiable**: Smooth gradient for backpropagation
- **S-shaped curve**: Provides non-linear decision boundary
- **Derivative**: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ - computationally efficient

The sigmoid derivative is used during backpropagation to compute gradients for weight updates.

3. Initialize weights and bias (small random values)

In [4]:
```python
# two inputs -> one output
w = np.random.randn(2, 1) * 0.1  # (2x1)
b = np.random.randn(1) * 0.1      # scalar bias
print("Initial weights:\n", w)
print("Initial bias:\n", b)
```

```
Initial weights:
 [[ 0.0497]
 [-0.0138]]
Initial bias:
 [0.0648]
```

🗒️ Initial Weights and Bias

- **Initial Weights**: [W =

$$\begin{bmatrix} 0.0497 \\ -0.0138 \end{bmatrix}$$

]

- **Initial Bias**: [b = 0.0648]

Weight Initialization Strategy

Small Random Initialization is used here with the following rationale:

- **Random values**: Break symmetry - if all weights start identical, they remain identical
- **Small magnitude** (×0.1): Prevents saturation of sigmoid function
- **Normal distribution**: Provides balanced positive/negative initial values
- **Reproducible**: `np.random.seed(42)` ensures consistent results

The initial weights and bias will be small random values around zero, allowing the network to learn from gradient descent updates.

4. Forward pass – predictions and intermediate values

In [5]:
```
# Linear combination
z = np.dot(data_x, w) + b  # (4x1)
# Activation
y_hat = sigmoid(z)
print("z (linear output):\n", z)
print("ŷ (sigmoid output):\n", y_hat)

# Error (Mean Squared Error)
error = np.mean((data_y - y_hat) ** 2)
print("\nMean Squared Error before update =", error)
```

```
z (linear output):
 [[0.0648]
 [0.0509]
 [0.1144]
 [0.1006]]
ŷ (sigmoid output):
 [[0.5162]
 [0.5127]
 [0.5286]
 [0.5251]]

Mean Squared Error before update = 0.23790376766405488
```

⚙️ Forward Pass Output & Error (Before Update)

- **Linear Combination (z)**: [z = X \cdot W + b =

$$\begin{bmatrix} 0.0648 \\ 0.0509 \\ 0.1144 \\ 0.1006 \end{bmatrix}$$

  ]

- **Sigmoid Activation (ŷ)**: [\hat{y} = \sigma(z) =

$$\begin{bmatrix} 0.5162 \\ 0.5127 \\ 0.5286 \\ 0.5251 \end{bmatrix}$$

  ]

- **Mean Squared Error** (before update):
  [\text{MSE} = 0.2379]

Forward Pass Analysis

The forward pass computes predictions for all training samples simultaneously:

1. **Linear Transformation**: z = Xw + b
    - Matrix multiplication: (4×2) × (2×1) + (1×1) = (4×1)
    - Each row represents one training sample's linear output
2. **Activation**: ŷ = σ(z)
    - Applies sigmoid element-wise to convert linear outputs to probabilities
    - Values close to 0 indicate "no alarm", close to 1 indicate "alarm"
3. **Error Calculation**: MSE = mean((y - ŷ)²)
    - Measures how far predictions are from true labels
    - Higher MSE indicates poorer performance

The initial predictions will likely be poor since weights are random, but this establishes our baseline before learning.

5. Backward pass – update with delta rule

In [6]:
```
# Compute gradient of error w.r.t. y_hat
error_grad = -(data_y - y_hat)        # d(MSE)/dŷ  for each sample (4x1)
# Gradient w.r.t. z
z_grad = error_grad * sigmoid_derivative(y_hat)

# Gradients for weights and bias (batch = mean of sample grads)
w_grad = np.dot(data_x.T, z_grad) / len(data_x)
b_grad = np.mean(z_grad)

lr = 0.1
w_new = w - lr * w_grad
b_new = b - lr * b_grad

print("Gradient w.r.t weights:\n", w_grad)
```

```
print("Gradient w.r.t bias:\n", b_grad)
print("\nUpdated weights:\n", w_new)
print("Updated bias:\n", b_new)
```

```
Gradient w.r.t weights:
 [[-0.059]
 [-0.06 ]]
Gradient w.r.t bias:
 -0.05717840777138376

Updated weights:
 [[ 0.0556]
 [-0.0078]]
Updated bias:
 [0.0705]
```

🔁 Backpropagation & Weight Update

- **Gradient w.r.t. Weights**: [\frac{\partial \text{MSE}}{\partial W} =

$$\begin{bmatrix} -0.059 \\ -0.060 \end{bmatrix}$$

  ]

- **Gradient w.r.t. Bias**: [\frac{\partial \text{MSE}}{\partial b} = -0.0572]

- **Updated Weights** (after learning rate = 0.1): [W_{\text{new}} =

$$\begin{bmatrix} 0.0556 \\ -0.0078 \end{bmatrix}$$

  ]

- **Updated Bias**: [b_{\text{new}} = 0.0705]

Backpropagation and Weight Update

The **Delta Rule** implements gradient descent to minimize error:

Step 1: Error Gradient

- `error_grad = -(y - ŷ)` - Derivative of MSE w.r.t. predictions
- Negative sign because we want to minimize error

Step 2: Local Gradient

- `z_grad = error_grad × σ'(ŷ)` - Chain rule application
- Sigmoid derivative weights the error by activation sensitivity

Step 3: Parameter Gradients

- **Weight gradient**: `∂E/∂w = X^T × z_grad / n` (averaged over batch)
- **Bias gradient**: `∂E/∂b = mean(z_grad)` (averaged over batch)

Step 4: Parameter Update

- **New weights**: `w ← w - η × ∂E/∂w`
- **New bias**: `b ← b - η × ∂E/∂b`
- Learning rate η = 0.1 controls step size

This single update should move the network toward better OR gate behavior.

6. Verifying new error after one update (optional)

In [7]:
```
# Forward pass with updated parameters
z2 = np.dot(data_x, w_new) + b_new
y_hat2 = sigmoid(z2)
error2 = np.mean((data_y - y_hat2) ** 2)
print("New Mean Squared Error =", error2)
```

```
New Mean Squared Error = 0.23584379921143195
```

✅ Forward Pass After Update

- **New Mean Squared Error**: [\text{MSE}_{\text{after update}} = 0.2358]

> 🔻 The error decreased from **0.2379** to **0.2358**, showing that the network has started learning and adjusting weights in the right direction.

Learning Progress Analysis

After one epoch of training, we can observe:

1. **Error Reduction**: The new MSE should be lower than the initial MSE, indicating learning
2. **Weight Evolution**: Updated weights should better reflect the OR logic pattern
3. **Prediction Improvement**: New predictions should be closer to desired outputs

Expected Behavior:

- For input [0,0]: prediction should move toward 0 (no alarm)
- For inputs [0,1], [1,0], [1,1]: predictions should move toward 1 (alarm)

Convergence Notes:

- Single-layer networks can perfectly learn linearly separable patterns like OR
- Multiple epochs would be needed for complete convergence
- The delta rule guarantees convergence for linearly separable problems

🧠 Final Inference & Conclusion

After performing one forward and backward pass on a single-layer neural network with a sigmoid activation:

- The model learned to reduce its prediction error by adjusting weights and bias.
- A clear reduction in Mean Squared Error (MSE) confirms that backpropagation worked as expected.
- This demonstrates the basic principle of learning in neural networks using gradient descent and the delta rule.

🔄 Error Comparison Table

Step	MSE	Weights	Bias
Before Update	**0.2379**	[[0.0497], [-0.0138]]	0.0648
After One Update Step	**0.2358**	[[0.0556], [-0.0078]]	0.0705

✅ **Conclusion**:
The decrease in MSE indicates successful learning. This basic experiment confirms that a single-layer neural network can learn to map inputs to outputs by minimizing error using gradient descent and backpropagation.