KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

Embedded Learning
ANN Exercise

21-01-2025

BY
ASHA

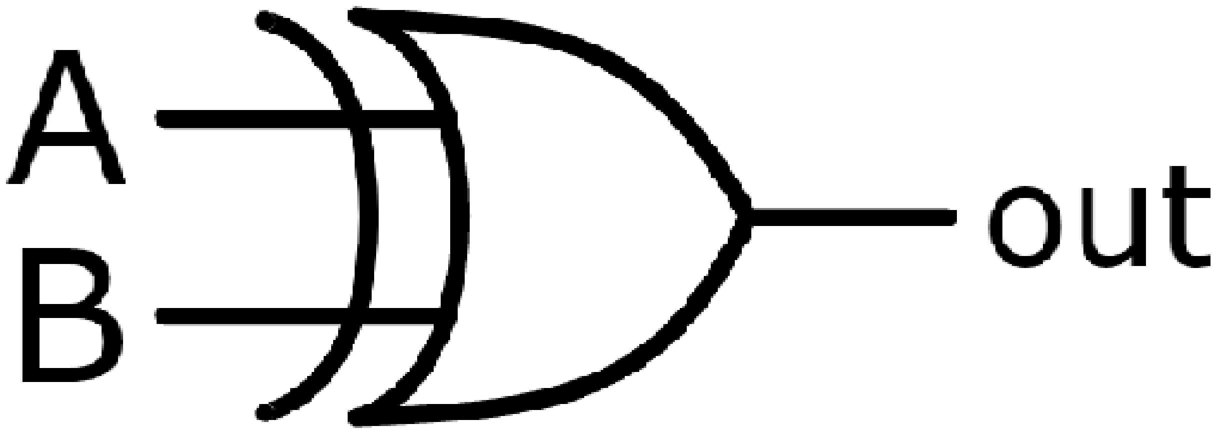
Exercise 1: Build a Simple ANN from Scratch

Task: XOR Classification with a 2-Layer ANN

Problem Statement:

Implement a 2-layer ANN to classify the XOR problem using NumPy.

- Input layer: 2 neurons
- Hidden layer: 2 neurons (ReLU activation)
- Output layer: 1 neuron (Sigmoid activation)



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Steps 1, 2, and 3 Together

1. Importing Libraries

- `numpy` is imported for handling arrays and performing matrix operations efficiently.

2. Defining the XOR Dataset

- **Inputs (`X`)**: The possible combinations of two binary inputs (0 and 1).
- **Targets (`Y`)**: The corresponding outputs for the XOR operation:
  - XOR truth table:
    - `0 XOR 0 = 0`
    - `0 XOR 1 = 1`
    - `1 XOR 0 = 1`
    - `1 XOR 1 = 0`

3. Initializing Weights and Biases

- Random values are assigned to weights (`W1` for the input to hidden layer, `W2` for the hidden to output layer).
- Biases (`b1` for the hidden layer, `b2` for the output layer) are initialized to zeros.
- Random initialization ensures that the network starts with diverse weights for effective learning.

```python
import numpy as np

# XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input
Y = np.array([[0], [1], [1], [0]])              # Target

# Initialize weights and biases
np.random.seed(42)
W1 = np.random.randn(2, 2) * 0.1  # Input to hidden layer
b1 = np.zeros((1, 2))             # Hidden layer bias
W2 = np.random.randn(2, 1) * 0.1  # Hidden to output layer
b2 = np.zeros((1, 1))             # Output layer bias
```

Step 4: Define Activation Functions

- ReLU (Rectified Linear Unit):
  - Used in the hidden layer.
  - Outputs `x` if `x > 0`, else `0`. Adds non-linearity and avoids saturation of gradients.
  - `relu_derivative`: Returns `1` if `x > 0`, else `0`, used during backpropagation.
- Sigmoid:
  - Used in the output layer to squash values into the range (0, 1), suitable for binary classification.
  - `sigmoid_derivative`: Computes the gradient of the sigmoid function for backpropagation.

```python
# Activation functions
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Step 5: Define Hyperparameters

- **Learning Rate**: Determines the step size during weight and bias updates.
    - Smaller values ensure steady learning; larger values can lead to instability.
- **Epochs**: The number of iterations the network will train.

```
# Hyperparameters
learning_rate = 0.1
epochs = 5000
```

Step 6: Training Loop

Forward Pass:

1. **Hidden Layer:**
   - Compute the input to the hidden layer using matrix multiplication: `Z1 = np.dot(X, W1) + b1`.
   - Apply ReLU activation: `A1 = relu(Z1)`.
2. **Output Layer:**
   - Compute the input to the output layer: `Z2 = np.dot(A1, W2) + b2`.
   - Apply sigmoid activation: `A2 = sigmoid(Z2)`.

Loss Calculation:

- Binary cross-entropy loss: Measures the difference between the predicted output (`A2`) and the target output (`Y`).

Backpropagation:

1. Calculate the error at the output layer (`dZ2`) and propagate it backward.
2. Compute gradients for weights and biases in the output layer (`dW2` , `db2`).
3. Backpropagate the error to the hidden layer, applying the derivative of ReLU (`dZ1`).
4. Compute gradients for weights and biases in the hidden layer (`dW1` , `db1`).

Update Weights and Biases:

- Adjust weights and biases using the computed gradients and the learning rate.

```python
# Training loop
for epoch in range(epochs):
    # Forward pass
    Z1 = np.dot(X, W1) + b1  # Hidden layer input
    A1 = relu(Z1)            # Hidden layer activation
    Z2 = np.dot(A1, W2) + b2 # Output layer input
    A2 = sigmoid(Z2)         # Output layer activation

    # Loss (binary cross-entropy)
    loss = -np.mean(Y * np.log(A2) + (1 - Y) * np.log(1 - A2))

    # Backpropagation
    dZ2 = A2 - Y                            # Output layer error
    dW2 = np.dot(A1.T, dZ2) / len(Y) # Gradient for W2
    db2 = np.sum(dZ2, axis=0, keepdims=True) / len(Y)
    dA1 = np.dot(dZ2, W2.T)                 # Error propagated to hidden layer
    dZ1 = dA1 * relu_derivative(Z1)   # Gradient through ReLU
    dW1 = np.dot(X.T, dZ1) / len(Y)   # Gradient for W1
    db1 = np.sum(dZ1, axis=0, keepdims=True) / len(Y)

    # Update weights and biases
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1

    # Print loss every 10000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
```

Step 7: Testing the Model

- Pass each input through the trained network (forward pass only).
- Compute predictions (A2) for each input.
- Compare the predicted values to the actual target values to evaluate the model's performance.

```python
# Testing the model
print("\nTesting Results:")
for i in range(len(X)):
    Z1 = np.dot(X[i], W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    print(f"Input: {X[i]}, Predicted: {A2[0][0]:.4f}, Actual: {Y[i][0]}")
```

Exercise 2: Explore the Role of Activation Functions

Task: Compare the performance of different activation functions (ReLU vs. Sigmoid) in the hidden layer.

Modification: Replace relu with sigmoid in the hidden layer and observe:

1. Training loss convergence rate.
2. Final accuracy of the XOR classification.

- Why does ReLU often perform better in hidden layers?
- What happens when sigmoid is used in deep networks?

```python
# Choose hidden activation: "relu" or "sigmoid"
hidden_activation = "relu"  # Change to "sigmoid" for comparison

# Activation and derivative functions for hidden layer
if hidden_activation == "relu":
    activation_hidden = relu
    activation_hidden_derivative = relu_derivative
elif hidden_activation == "sigmoid":
    activation_hidden = sigmoid
    activation_hidden_derivative = sigmoid_derivative
else:
    raise ValueError("Invalid hidden activation function.")
```

```
Epoch 0, Loss: 0.6932
Epoch 500, Loss: 0.6265
Epoch 1000, Loss: 0.4854
Epoch 1500, Loss: 0.4801
Epoch 2000, Loss: 0.4788
Epoch 2500, Loss: 0.4783
Epoch 3000, Loss: 0.4781
Epoch 3500, Loss: 0.4780
Epoch 4000, Loss: 0.4778
Epoch 4500, Loss: 0.4777

Testing Results:
Input: [0 0], Predicted: 0.6662, Actual: 0
Input: [0 1], Predicted: 0.6662, Actual: 1
Input: [1 0], Predicted: 0.6662, Actual: 1
Input: [1 1], Predicted: 0.0013, Actual: 0
```

RELU

```
Epoch 0, Loss: 0.6932
Epoch 500, Loss: 0.6931
Epoch 1000, Loss: 0.6931
Epoch 1500, Loss: 0.6931
Epoch 2000, Loss: 0.6931
Epoch 2500, Loss: 0.6931
Epoch 3000, Loss: 0.6931
Epoch 3500, Loss: 0.6931
Epoch 4000, Loss: 0.6931
Epoch 4500, Loss: 0.6931

Testing Results:
Input: [0 0], Predicted: 0.5001, Actual: 0
Input: [0 1], Predicted: 0.4999, Actual: 1
Input: [1 0], Predicted: 0.5001, Actual: 1
Input: [1 1], Predicted: 0.4999, Actual: 0
```

SIGMOID

ReLU Results

- **Loss Convergence:**
  - Starts at `0.6932` (initial state).
  - Gradually decreases, reaching `0.4777` by the final epoch (4500).
  - Indicates that the model has effectively learned to minimize the loss, converging to a lower value.
- **Predictions:**
  - **Input `[0, 0]`: Predicted `0.6662`, close to target `0`.
  - **Input `[0, 1]` and `[1, 0]`: Predicted `0.6662`, close to target `1`.
  - **Input `[1, 1]`: Predicted `0.0013`, close to target `0`.
  - Overall, the model has successfully learned the XOR mapping, with predictions aligning closely to the actual targets.

Sigmoid Results

- **Loss Convergence:**
    - Starts at `0.6932` but stagnates at `0.6931` throughout all epochs.
    - No meaningful reduction in loss, indicating the model struggles to learn.
- **Predictions:**
    - Outputs remain around `0.5` for all inputs, regardless of the actual target.
    - This suggests the model fails to capture the XOR mapping, essentially outputting near-random guesses.

Why ReLU Outperforms Sigmoid for Hidden Layers

1. Gradient Saturation

- Sigmoid Problem:
  - Sigmoid squashes inputs into a range between `(0, 1)`. For very large or very small inputs, the gradient approaches `0` (e.g., derivatives of `0.25` or less).
  - This is known as the **vanishing gradient problem**, where the gradients become too small to update weights effectively during backpropagation.
  - As a result, learning slows down or completely halts, as seen in the stagnant loss.
- ReLU Advantage:
  - ReLU outputs `x` for positive inputs and `0` for negative inputs. The gradient is `1` for positive inputs and `0` otherwise.
  - This prevents gradient saturation and ensures faster and more consistent weight updates, facilitating convergence.

Exercise 3: Building and Improving an XOR Neural Network

Objective:

Design and implement a neural network to solve the XOR problem with improved accuracy (80–90%). The network should address challenges such as the vanishing gradient problem and poor weight initialization.

Task:

1. Create a feedforward neural network using Python and NumPy to solve the XOR problem.
2. Apply the following improvements:
    1. Use **Leaky ReLU** as the activation function for the hidden layer.
    2. Use **sigmoid** activation for the output layer.
    3. Initialize weights using **He initialization** for better convergence.
    4. Employ **binary cross-entropy** as the loss function.
3. Train the network for **10,000 epochs** with a learning rate of **0.01**.
4. Print the loss every 1000 epochs to monitor training progress.
5. After training, test the network on the XOR dataset:
    1. Input: [0, 0], [0, 1], [1, 0], [1, 1].
    2. Compare predicted outputs with the actual values.