One-layer Neural Network

Pseudocode

High-Level Pseudocode for a One-Layer Neural Network

This pseudocode outlines the steps for a one-layer neural network with backpropagation.

Given

- **A one-layer neural network (NN)** with two inputs and two outputs.
- **Four data groups** to be classified, each with a set of points in a 2D space:
  - **Group 1:** (0.1, 1.2), (0.7, 1.8), (0.8, 1.6)
  - **Group 2:** (0.8, 0.6), (1.0, 0.8)
  - **Group 3:** (0.3, 0.5), (0.0, 0.2), (-0.3, 0.8)
  - **Group 4:** (-0.5, -1.5), (-1.5, -1.3)
- **Binary codes** representing the four groups:
  - Group 1: (1, 0)
  - Group 2: (0, 0)
  - Group 3: (1, 1)
  - Group 4: (0, 1)
- **An input matrix,** X:

  X=[0.11.2 0.71.8 0.81.6 0.80.6 1.00.8 0.30.5 0.00.2 −0.30.8 −0.5−1.5 −1.5−1.3]

- **A desired target matrix,** Y:

  Y=[10 10 10 00 00 11 11 11 01 01]

- **Activation function:** Sigmoid.
- **Training algorithm:** Backpropagation.

Needed

- **Design and train** the neural network.
- **Plot the following figures:**

    - Training error versus epoch number.
    - Decision boundary and data points after 3, 10, 100, and 1000 epochs.
- **Provide a relative assessment** of the following activation functions for this problem:
    - Tanh
    - Logsig
    - Radialbasis
    - Relu

1. Data Definition and Initialization

- **Define Input and Target Data:**
    - Create an input matrix X with 2 features (x1, x2) and 10 samples.
    - Create a target matrix Y with 2 outputs (y1, y2) and 10 samples, representing different groups.
    - Determine the number of training examples (M), input size (N_{IN}), and output size (N_{OUT}).
- **Initialize Model Parameters:**
    - Define a function initialize_parameters(input_size, output_size).
    - Initialize the weight matrix W with small random values.
    - Initialize the bias vector b with zeros.
    - Return W and b.

2. Activation Functions

- **Sigmoid Function:**
    - Define a function sigmoid(Z) that takes a weighted sum Z.
    - Calculate the standard logistic sigmoid function: $A = 1 / (1 + e^{-Z})$.
    - Return the activated output A.
- **Sigmoid Derivative:**
    - Define a function sigmoid_derivative(A) that takes the activated output A.
    - Calculate the derivative of the sigmoid function as $A * (1 - A)$.
    - Return the derivative.

3. Forward Propagation

- **Define forward_propagation(X, W, b):**
    - Calculate the weighted sum of inputs: $Z = WX + b$.
    - Apply the sigmoid activation function to Z to get the activated output A.
    - Return Z and A.

4. Backpropagation and Parameter Update

- **Backpropagation:**
    - Define a function back_propagation(X, Y, A, Z) to calculate gradients for weights and bias.
    - Calculate the error at the output layer, $dE\_dA = A - Y$.
    - Calculate the gradient of the loss with respect to Z: $dE\_dZ = dE\_dA * sigmoid\_derivative(A)$.
    - Calculate the gradient for weights: $dW = (1 / M) * dE\_dZ * X^T$.
    - Calculate the gradient for the bias: $db = (1 / M) * sum(dE\_dZ)$ over all samples.
    - Return dW and db.
- **Update Parameters:**
    - Define a function update_parameters(W, b, dW, db, learning_rate).
    - Update weights: $W = W - learning\_rate * dW$.
    - Update bias: $b = b - learning\_rate * db$.
    - Return the updated W and b.

5. Training Loop

- **Define a main training function train_nn(X, Y, epochs, learning_rate):**
    - Initialize parameters W and b.
    - Create a list to store errors.
    - Loop for a specified number of epochs.
    - Inside the loop, perform **forward propagation** to get Z and A.
    - Calculate and store the **Mean Squared Error (MSE)** loss for tracking.
    - Perform **backpropagation** to get gradients dW and db.
    - **Update parameters** using the learning rate.

    - Optionally, save parameter snapshots at specific epochs to visualize the decision boundary later.
    - Print the loss at each epoch.
    - After the loop, return the final parameters, error history, and snapshots.

6. Classification

- **Define classify (X, W, b):**
    - Perform forward propagation to get the activated output A.
    - Round the output A to the nearest binary code (0 or 1) for classification.
    - Return the final predictions.

7. Visualization

- Plot the Mean Squared Error (MSE) loss against the number of epochs to visualize the training progress.
- Decision Boundary Evolution:
    - Decision Boundary and Data Points (Epoch 3)
    - Decision Boundary and Data Points (Epoch 10)
    - Decision Boundary and Data Points (Epoch 100)
    - Decision Boundary and Data Points (Epoch 1000)

8. Relative Assessment of Activation Functions

1. **logsig (Sigmoid) (Used Here):**

* **Pros:** Outputs are bounded between (0, 1), making it directly suitable for probability and binary classification outputs, matching the target codes (0, 1).

* **Cons:** Not zero-centered (output mean is positive), which can slow down gradient descent convergence. Suffers from the "vanishing gradient" problem when inputs are very large or very small (saturation), causing the network to learn very slowly.

2. **tanh (Hyperbolic Tangent):**

* **Pros:** Outputs are bounded between (-1, 1). It is zero-centered, which typically makes optimization faster than the sigmoid function as gradients are, on average, closer to zero.

* **Cons:** Still suffers from the vanishing gradient problem when saturated (inputs are very large). To use this effectively, the target values Y would ideally need to be re-coded from $\{0, 1\}$ to $\{-1, 1\}$.

3. **tansig (Tangent Sigmoid):**

* This is often used as a synonym for **tanh** in neural network literature (e.g., MATLAB's Neural Network Toolbox). The assessment is the same as for **tanh**.

4. **radialbasis (Radial Basis Function - RBF):**

* **Pros:** Excellent for non-linear, localized classification boundaries. Its activation is based on the distance from a center point (local responsiveness), making it very powerful for certain types of non-linear classification problems.

* **Cons:** Not commonly used as a general activation in deep feed-forward networks (like ReLU or Tanh). Requires different training methodologies (determining the center and radius). Highly non-local in effect, making pure backpropagation in a standard architecture less common.

5. **relu (Rectified Linear Unit):**

* **Pros:** Computationally efficient ($f(z) = \max(0, z)$). Most importantly, it **solves the vanishing gradient problem** for positive inputs, leading to much faster convergence, especially in deep networks.

* **Cons:** Not zero-centered. Can suffer from the "dying ReLU" problem where a large negative gradient can push the pre-activation Z into a state where the output is always zero, effectively "killing" the neuron. Since the target is $\{0, 1\}$, a modified ReLU (like Leaky ReLU) might be needed, but standard ReLU's $[0, \infty)$ range does not perfectly match the bounded output requirement, though it can still work well in classification tasks before a final sigmoid/softmax layer.

Overall for this problem:

* **Logsig (Sigmoid) or Tanh** are theoretically the most appropriate because they are bounded and match the binary nature of the target output codes (0 or 1, or -1 and 1).

* **ReLU** would be a fast learner but less ideal as the final output layer activation due to its unbounded nature. In a real application, you'd typically use ReLU in a hidden layer, followed by a Sigmoid/Softmax in the output layer.

"""