Experiment 9 : Implementing a Neural Network and Backpropagation from Scratch

Total Marks: 100

1. Learning Objectives

Upon successful completion of this assignment, students will be able to:

Understand and articulate the mathematical foundations of a feedforward neural network.

Implement the core components of an ANN, including parameter initialization, activation functions (ReLU, Sigmoid), and their derivatives.

Implement the **Forward Propagation** algorithm to generate predictions from network inputs.

Implement the **Backpropagation** algorithm from scratch to calculate gradients for all network parameters.

Implement various **loss functions** (Binary Cross-Entropy, Mean Squared Error) and their derivatives.

Implement the **Gradient Descent** algorithm to update network weights and biases.

Build a complete, modular `MyANNClassifier` class using only **NumPy**.

Train the "from scratch" classifier on a real-world dataset and evaluate its performance.

Compare the custom-built classifier's performance and behavior against `sklearn.neural_network.MLPClassifier` .

Analyze the impact of different loss functions and network architectures on model training and final performance.

2. Introduction

This assignment is designed to demystify the "black box" of neural networks. You will move beyond high-level libraries and implement the core engine of a simple, fully-connected neural network using only NumPy. Your primary task is to build a classifier by implementing the two most critical components: **Forward Propagation** (for making predictions) and **Backpropagation** (for learning from errors).

You will use the well-known **Wisconsin Breast Cancer dataset** for a binary classification task. After building your network, you will experiment with different loss functions (BCE vs. MSE) and architectures. Finally, you will compare your "from scratch" model to scikit-learn's `MLPClassifier` to benchmark your work and appreciate the optimizations provided by modern libraries.

3. Prerequisites

Ensure your Python environment has the following libraries installed:

```
pip install numpy pandas scikit-learn matplotlib seaborn
```

4. Experiment Tasks

You are required to build a complete neural network pipeline. Follow the structured tasks below.

Task 1: Data Loading and Preprocessing (15 Marks)

1. **Load Data:** Load the **Breast Cancer Wisconsin dataset** directly from scikit-learn.

```
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X = data.data
y = data.target
```

2. **Inspect Data:** Print the shapes of `X` and `y` and the feature names to understand the data. This is a binary classification problem.

3. **Create Hold-Out Set:** Perform a single **70/30 split** on the data.

   `X_train` , `y_train` (70% of the data)

   `X_val` , `y_val` (30% of the data)

   Use `train_test_split` with `random_state=42` for reproducibility.

4. **Standardize Features:** This is **critical** for neural networks.

   Fit a `StandardScaler` from `sklearn.preprocessing` on `X_train` **only**.

   Transform both `X_train` and `X_val` using the *fitted* scaler.

   `X_train_scaled` will be used for training, and `X_val_scaled` for all final evaluations.

Task 2: 'From Scratch' Utilities (NumPy) (20 Marks)

Implement the following helper functions using only NumPy.

1. **Activation Functions:**

   `sigmoid(Z)` : Computes the sigmoid.

   `relu(Z)` : Computes the Rectified Linear Unit (`np.maximum(0, Z)`).

2. **Activation Derivatives:** These are crucial for backpropagation.

   `sigmoid_derivative(A)` : Where `A = sigmoid(Z)` . The derivative is `A * (1 - A)` .

   `relu_derivative(Z)` : The derivative is `1` if `Z > 0` , and `0` otherwise.

3. **Loss Functions:**

   `compute_bce_loss(Y, Y_hat)` : Computes the **Binary Cross-Entropy (BCE)** loss. (Add a small `epsilon=1e-15` for numerical stability to avoid `log(0)`).

   `compute_mse_loss(Y, Y_hat)` : Computes the **Mean Squared Error (MSE)** loss.

Task 3: 'From Scratch' ANN Classifier (40 Marks)

Implement a `MyANNClassifier` class. This class will orchestrate the entire learning process. 1. **Class Structure (`__init__`):**

`__init__(self, layer_dims, learning_rate=0.01, n_iterations=1000, loss='bce')` :

`layer_dims` : A list specifying the number of units in each layer. e.g., `[n_x, 10, 5, 1]` , where `n_x` is the number of input features (30 for the breast cancer dataset).

Store `learning_rate` , `n_iterations` , and `loss` (either 'bce' or 'mse').

`self.parameters_` : A dictionary to store weights (`W1` , `W2` , ...) and biases (`b1` , `b2` , ...).

`self.costs_` : A list to store the loss at each iteration (for plotting).

2. **Parameter Initialization (`_initialize_parameters`):**

   Create a helper method that iterates through `layer_dims`.

   Initialize weights `W` with small random values (`np.random.randn(...) * 0.01`) to break symmetry.

   Initialize biases `b` as zeros (`np.zeros(...)`).

   Store them in `self.parameters_` (e.g., `self.parameters_['W1']`, `self.parameters_['b1']`).

3. **Forward Propagation (`_forward_propagation`):**

   Create a method `_forward_propagation(self, X)`.

   `A_prev = X`.

   Loop from layer 1 to L:

   The **hidden layers (1 to L-1)** must use the **ReLU** activation.

   The **output layer (L)** must use the **Sigmoid** activation (for binary classification).

   Calculate `Z = W @ A_prev + b`.

   Calculate `A = activation(Z)`.

   Store all `A` (activations) and `Z` (linear results) in a `cache` (e.g., a list of tuples `(A, Z)`). This `cache` is essential for backpropagation.

   Return the final activation `A_L` (which is `Y_hat`) and the `cache`.

4. **Backward Propagation (`_backward_propagation`):**

   Create a method `_backward_propagation(self, Y, Y_hat, cache)`. This is the most complex task.

   `Y` is the true labels, `Y_hat` is the prediction (`A_L`) from the forward pass.

   Initialize Backprop:

   Calculate `dA_L` (the derivative of the loss function w.r.t. `Y_hat`).

   If `self.loss == 'bce'`: `dA_L = -(np.divide(Y, Y_hat) - np.divide(1 - Y, 1 - Y_hat))`

   If `self.loss == 'mse'`: `dA_L = 2 * (Y_hat - Y)`

   Output Layer (Sigmoid):

   Get `A_L` and `Z_L` from the `cache`.

`dZ_L = dA_L * sigmoid_derivative(A_L)`

Calculate `dW_L` and `db_L` using `dZ_L` and the corresponding `A_prev` from the cache.

Loop Backwards (Hidden Layers - ReLU):

Iterate from layer L-1 down to 1.

Calculate `dA_prev = W.T @ dZ` (using `W` and `dZ` from the *current* layer).

`dZ_prev = dA_prev * relu_derivative(Z_prev)` (using `Z_prev` from the cache).

Calculate `dW` and `db` for this layer.

Store all gradients (`dW1`, `db1`, `dW2`, `db2`, ...) in a `grads` dictionary.

5. **Parameter Update (`_update_parameters`)**:

   Create a method `_update_parameters(self, grads)`.

   Iterate through all parameters in `self.parameters_`.

   Update them using gradient descent:

   `W = W - self.learning_rate * dW`

   `b = b - self.learning_rate * db`

6. **Fit and Predict Methods**:

   `fit(self, X, y)`:

   Reshape `y` to be `(1, n_samples)`.

   Reshape `X` to be `(n_features, n_samples)`.

   Call `_initialize_parameters`.

   Loop for `n_iterations`:

   1. `Y_hat, cache = _forward_propagation(X)`
   2. `loss = compute_bce_loss(y, Y_hat)` (or `mse` based on `self.loss`)
   3. `grads = _backward_propagation(y, Y_hat, cache)`
   4. `_update_parameters(grads)`
   5. Store the `loss` in `self.costs_`.

   `predict(self, X)`:

Reshape `X` to `(n_features, n_samples)` .

Run `_forward_propagation(X)` to get `Y_hat` .

Convert probabilities to binary predictions: `predictions = (Y_hat > 0.5).astype(int)` .

Return the flattened 1D array of predictions.

Task 4: Training and Experimentation (15 Marks)

Use your **scaled** training and validation sets (`X_train_scaled` , `y_train` , `X_val_scaled` , `y_val`). 1. **Model 1 (BCE Loss):**

Define your `layer_dims` . Start with one hidden layer (e.g., `[30, 10, 1]`).

Instantiate `MyANNClassifier` with `loss='bce'` , `learning_rate=0.001` , and `n_iterations=5000` .

`fit` the model on `X_train_scaled` and `y_train` .

`predict` on `X_val_scaled` .

Print the `classification_report` for this model.

2. **Model 2 (MSE Loss):**

Instantiate a new model with the *exact same parameters* as Model 1, but set `loss='mse'` .

`fit` and `predict` as before.

Print the `classification_report` for this model.

3. **Model 3 (Deeper Architecture):**

Instantiate a new model with `loss='bce'` but a *deeper* architecture (e.g., `[30, 10, 5, 1]`).

`fit` and `predict` .

Print the `classification_report` for this model.

Task 5: Comparison with scikit-learn (10 Marks)

1. **Train `MLPClassifier` :**

Import `from sklearn.neural_network import MLPClassifier` .

Instantiate `MLPClassifier` with parameters that roughly match your best "from scratch" model.

Example: `MLPClassifier(hidden_layer_sizes=(10,), activation='relu', solver='adam', max_iter=1000, learning_rate_init=0.001, random_state=42)` .

`fit` the `MLPClassifier` on `X_train_scaled` and `y_train` .

2. **Evaluate `MLPClassifier` :**

`predict` on `X_val_scaled` .

Print the `classification_report` for the `sklearn` model.

5. Submission Guidelines

Submit a single `.zip` archive containing:

1. **Source Code:** A single Jupyter Notebook (`.ipynb`) or multiple `.py` files containing all your code, clearly separated by task.
2. **PDF Report:** A formal report (`StudentID_Report.pdf`) that includes:

"From Scratch" Code: Include the code snippets for your `MyANNClassifier` class (specifically the `_forward_propagation` , `_backward_propagation` , and `_update_parameters` methods).

Experiment Results: Present a **comparison table** showing the key metrics (Precision, Recall, F1-Score for class 1) from the `classification_report` for all four models:

1. `MyANN (BCE, 1 hidden layer)`
2. `MyANN (MSE, 1 hidden layer)`
3. `MyANN (BCE, 2 hidden layers)`
4. `sklearn.MLPClassifier`

Loss Curve: Include a plot (using `matplotlib`) of `self.costs_` vs. `iterations` for Model 1 (`BCE`) and Model 2 (`MSE`) on the same graph to visualize convergence.

Analysis & Conclusion:

Discuss the difference in performance between the **BCE** and **MSE** loss functions for this classification task. Why is one better than the other?

Compare your best "from scratch" model to the `sklearn.MLPClassifier` . Why is the `sklearn` model likely different (e.g., `adam` optimizer vs. batch gradient descent)?

What was the most challenging part of implementing the network from scratch?