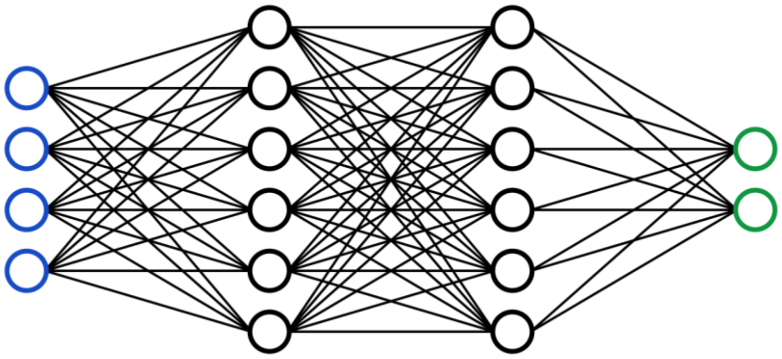
Ain Sham University
Faculty of Engineering
Mechatronics Dept.

Computational Intelligence
CSE473s

Computational Intelligence Project
Team 12



Name	ID	Sec
Adham Ehab Saleh	2100679	1
Ahmed Mohamed Ramadan	2100323	1
Ahmed Salah Eldin Abdelrahman	2100505	1
Ahmed Yasser Hosney	2101101	1
Omar Magdy Abdelsattar	2100273	2

Delivered to: Dr. Hossam El-Din Hassan

Eng. Abdallah Mohamed Mahmoud

Eng. Dina Zakaria Mahmaud

Ain Sham University
Faculty of Engineering
Mechatronics Dept.

Computational Intelligence
CSE473s

Table of Contents

Table of Figures

Problem Definition (Milestone 1: Library Validation & XOR)

Background and Context

The fundamental objective of this project is to develop a foundational neural network library from scratch using only Python and NumPy. Before applying this library to complex tasks like image reconstruction, it is essential to validate the correctness of the core mathematical algorithms, specifically forward propagation, backward propagation, and parameter optimization. To achieve this, the library must be tested against a benchmark problem that requires non-linear decision boundaries.

Problem Statement

The specific challenge for Part 1 is to construct and train a Multi-Layer Perceptron (MLP) capable of solving the XOR (Exclusive OR) problem. The XOR function is a classic binary classification problem where the target output is true (1) if and only if the inputs differ, and false (0) otherwise.

Unlike logical AND or OR gates, the XOR function is **not linearly separable**, meaning a single linear decision boundary (like a perceptron) cannot solve it. This makes it an ideal candidate to verify the library's ability to learn non-linear representations using hidden layers and activation functions.

The dataset consists of the four-standard truth-table inputs:

- **Inputs:** [(0,0), (0,1), (1,0), (1,1)]
- **Targets:** [0, 1, 1, 0]

Milestone 1 Objectives

The primary goal of this phase is to demonstrate that the custom-built library functions correctly by converging on a solution for the XOR problem.

Specific objectives include:

- **Library Architecture:** Implementing modular classes for Dense layers, activations (ReLU, Sigmoid, Tanh), and Loss functions (MSE).
- **Gradient Verification:** Validating the backpropagation algorithm by performing **numerical gradient checking** to ensure the analytical gradients match the numerical approximations.
- **Model Implementation:** Constructing a specific MLP architecture (e.g., 2-4-1) using Tanh and Sigmoid activations.
- **Optimization:** successfully minimizing the Mean Squared Error (MSE) using Stochastic Gradient Descent (SGD) to achieve correct classification for all 4 input cases.

Milestone 2 Objectives (Unsupervised Learning & Feature Extraction)

Following the validation of the library on simple non-linear data (XOR), the second phase challenges the library with high-dimensional real-world data: the MNIST dataset of handwritten digits. The objectives are twofold:

1. **Dimensionality Reduction (Autoencoder):** Train an MLP to compress 784-pixel images into a lower-dimensional latent space (64 features) and reconstruct them, minimizing the reconstruction loss.
2. **Latent Feature Classification (SVM):** Validate the quality of the learned features by using the compressed 64-dimensional vectors as inputs for a Support Vector Machine (SVM) classifier, testing if the library can effectively learn semantic representations of data.

Methodology and Library Design

System Architecture

To meet the requirement of a modular, extensible framework, the library was designed using an Object-Oriented Programming (OOP) approach. The core architecture relies on an abstract base class structure where specific components (Layers, Activations, Optimizers) inherit common interfaces. This ensures that the network model can treat all components uniformly during the forward and backward passes.

The library structure consists of the following key modules:

- **Layer (Base Class):** Defines the abstract methods forward() and backward() that all derived classes must implement.
- **Dense (Fully Connected Layer):** Inherits from Layer. It initializes and stores the trainable parameters—Weights (W) and Biases (b)—and computes the linear transformation $Z = X.W + b$. It also stores the gradients ($\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}$) computed during backpropagation.
- **Network (Container):** Acts as the orchestrator. It maintains a list of layers, manages the flow of data through the network, and executes the training loop.

Mathematical Implementation

Activation Functions

Activation functions are implemented as subclasses of the Layer class to maintain modularity. Each activation class implements:

1. **Forward:** Applies the non-linear function *f(x)* (e.g., Sigmoid, Tanh, ReLU).
2. **Backward:** Computes the derivative *f'(x)* and applies the chain rule to propagate the error gradient to the previous layer.

Loss Function

The training performance is measured using the Mean Squared Error (MSE), as required for regression and binary classification tasks in this project.

The loss is calculated as:

$$L = \frac{1}{N}\sum (Y_{true} - Y_{pred})^2$$

The derivative with respect to the prediction Y_{pred}, which initializes the backpropagation process, is implemented as $2 \times (\frac{Y_{pred} - Y_{true}}{N})$

Optimization (SGD)

Parameter updates are handled by the Stochastic Gradient Descent (SGD) optimizer. The optimizer iterates through every trainable layer in the network and updates the weights and biases using the gradients computed during the backward pass:

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W}$$

where η represents the learning rate.

Validation Strategy

Numerical Gradient Checking

Before training, the mathematical correctness of the backpropagation implementation was verified using Gradient Checking. This technique compares the analytical gradient (computed by the chain rule in the code) against a numerical approximation derived from the limit definition of a derivative.

The numerical gradient was approximated using the central difference formula:

$$\frac{\partial L}{\partial W} \approx \frac{L(W + \epsilon) - L(W - \epsilon)}{2\epsilon}$$

A small ϵ (e.g., 10^{-7}) was used to perturb the weights. The relative difference between the analytical and numerical gradients was checked to ensure it fell below a strict threshold (e.g., 10^{-7}), confirming the backpropagation logic is correct.

XOR Benchmark Architecture

To validate the library's ability to learn non-linear boundaries, the XOR problem was selected as the primary unit test.

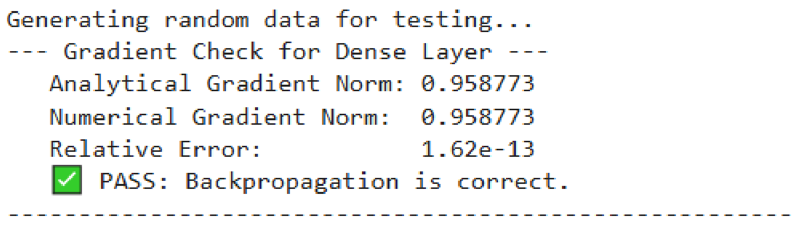
The specific architecture constructed for this test is a Multi-Layer Perceptron (MLP) with the following topology:

- **Input Layer:** 2 Neurons (representing the two binary inputs).
- **Hidden Layer:** 4 Neurons with **Tanh** activation.
- **Output Layer:** 1 Neuron with **Sigmoid** activation (outputting a probability between 0 and 1).

This "2-4-1" architecture provides sufficient capacity to map the non-linearly separable XOR inputs to the correct Boolean outputs.

Results

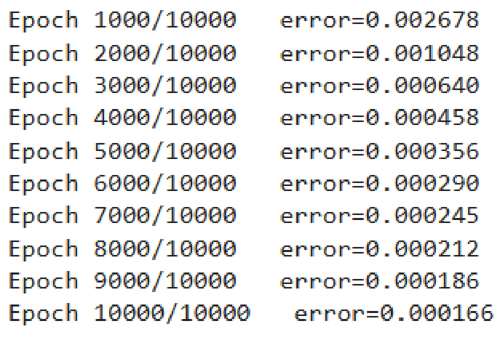
First, we needed to validate the gradient logic, so in the notebook there is a section to validate it, and its output is as follows



```
Generating random data for testing...
--- Gradient Check for Dense Layer ---
   Analytical Gradient Norm: 0.958773
   Numerical Gradient Norm:  0.958773
   Relative Error:           1.62e-13
   ✅ PASS: Backpropagation is correct.
-------------------------------------------------------
```

Figure 1.Gradient Logic Validation.

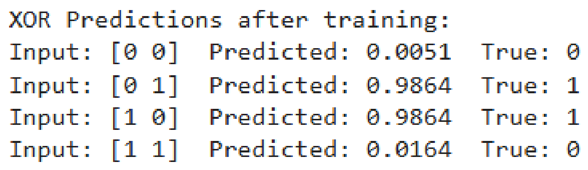
Then, we trained the network for 10000 epochs



```
Epoch 1000/10000   error=0.002678
Epoch 2000/10000   error=0.001048
Epoch 3000/10000   error=0.000640
Epoch 4000/10000   error=0.000458
Epoch 5000/10000   error=0.000356
Epoch 6000/10000   error=0.000290
Epoch 7000/10000   error=0.000245
Epoch 8000/10000   error=0.000212
Epoch 9000/10000   error=0.000186
Epoch 10000/10000   error=0.000166
```

Figure 2.Training of The Network.

Then, we tested the network



```
XOR Predictions after training:
Input: [0 0]  Predicted: 0.0051  True: 0
Input: [0 1]  Predicted: 0.9864  True: 1
Input: [1 0]  Predicted: 0.9864  True: 1
Input: [1 1]  Predicted: 0.0164  True: 0
```

Figure 3.Prediction After Training.

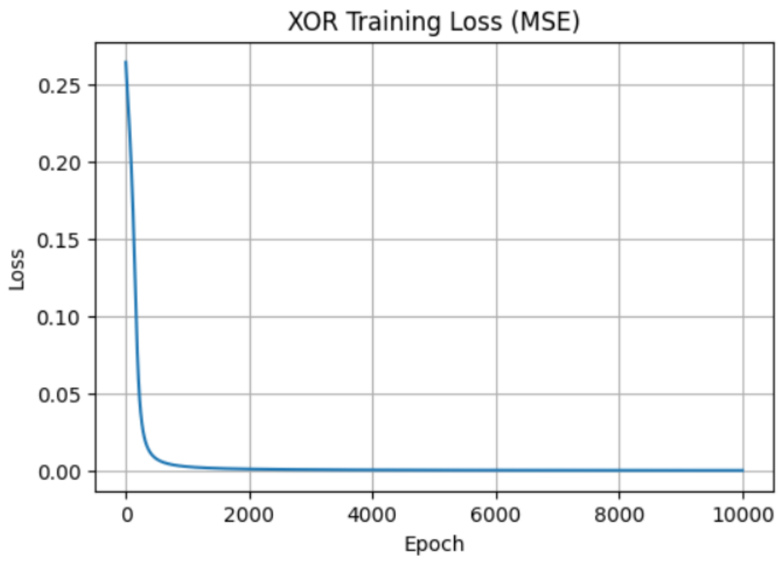


Figure 4.Loss Visualization for Model.

We then compared this custom library with TensorFlow

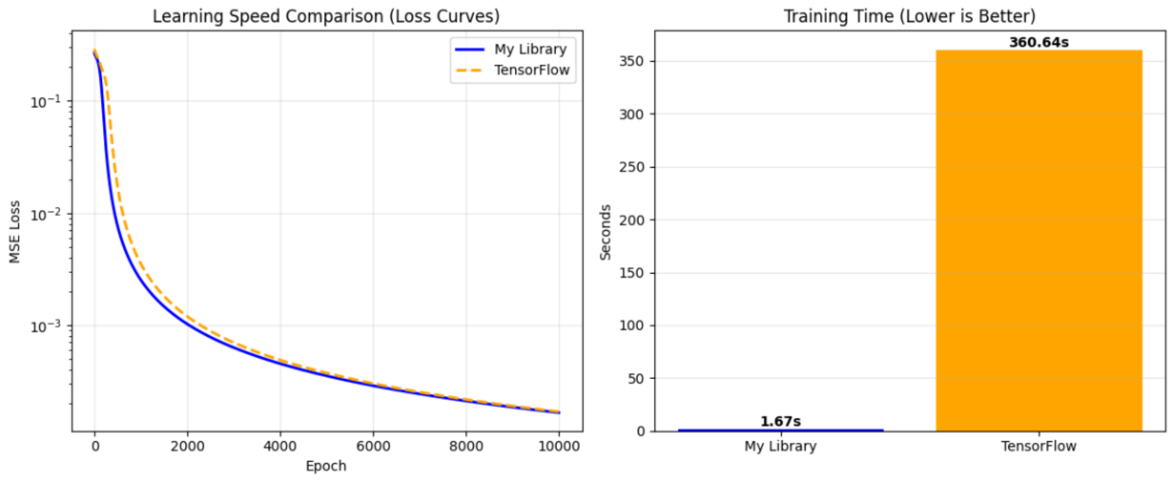


Figure 5.Comparison between Custom Model and TensorFlow.

Autoencoder & Latent Space Analysis

Architecture Design

To solve the image reconstruction task, we constructed an Autoencoder network with a "bottleneck" architecture. This forces the network to learn efficient data representations.

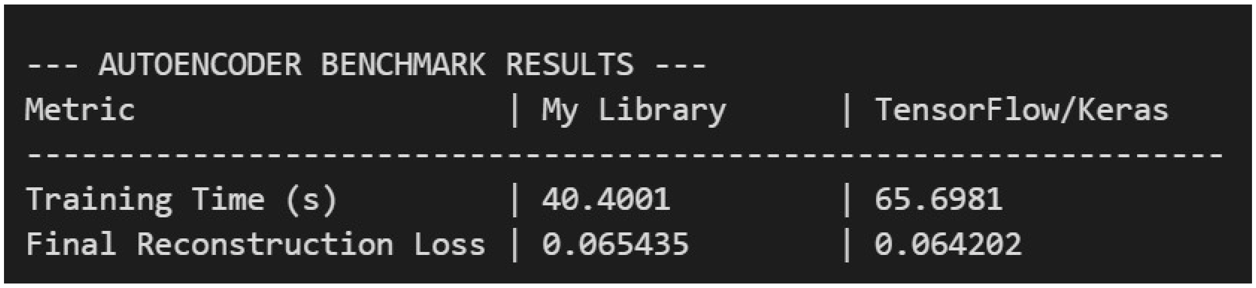
- **Input Layer:** 784 neurons (corresponding to flattened 28x28 MNIST images).
- **Encoder (Hidden) Layer:** 64 neurons with Sigmoid activation. This layer acts as the information bottleneck, compressing the data by approximately 92%.
- **Decoder (Output) Layer:** 784 neurons with Sigmoid activation to reconstruct the original pixel intensities (normalized between 0 and 1).

Training Configuration

- **Optimizer:** SGD with Learning Rate (Decaying to 0.01).
- **Loss Function:** Mean Squared Error (MSE).
- **Epochs:** 5 - 10 (Subject to convergence).
- **Data:** 60,000 Training images, 10,000 Test images.

Reconstruction Analysis

As seen in the figure below, the final reconstruction loss = 0.065435



```
--- AUTOENCODER BENCHMARK RESULTS ---
Metric                    | My Library      | TensorFlow/Keras
------------------------------------------------------------
Training Time (s)         | 40.4001         | 65.6981
Final Reconstruction Loss | 0.065435        | 0.064202
```

Figure 6: Autoencoder Results

Qualitative Analysis

As seen in the figure below, the reconstructed digits retain their global structure and identity (e.g., a '7' looks like a '7'). However, the reconstructions appear slightly "blurrier" than the originals. This is expected behavior for Mean Squared Error loss, which averages pixel differences, and due to the high compression ratio (784 64), which forces the model to discard high-frequency details (sharp edges) in favor of preserving the dominant shape.

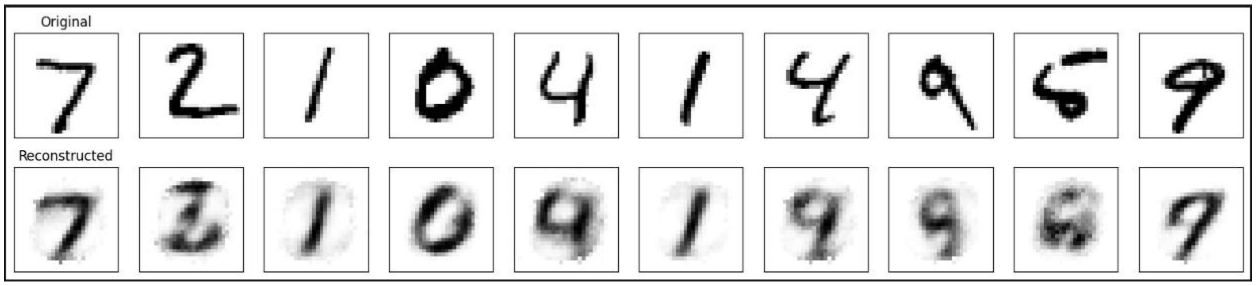


Figure 7: Original Image Vs Reconstructed Image

SVM Classification on Latent Features

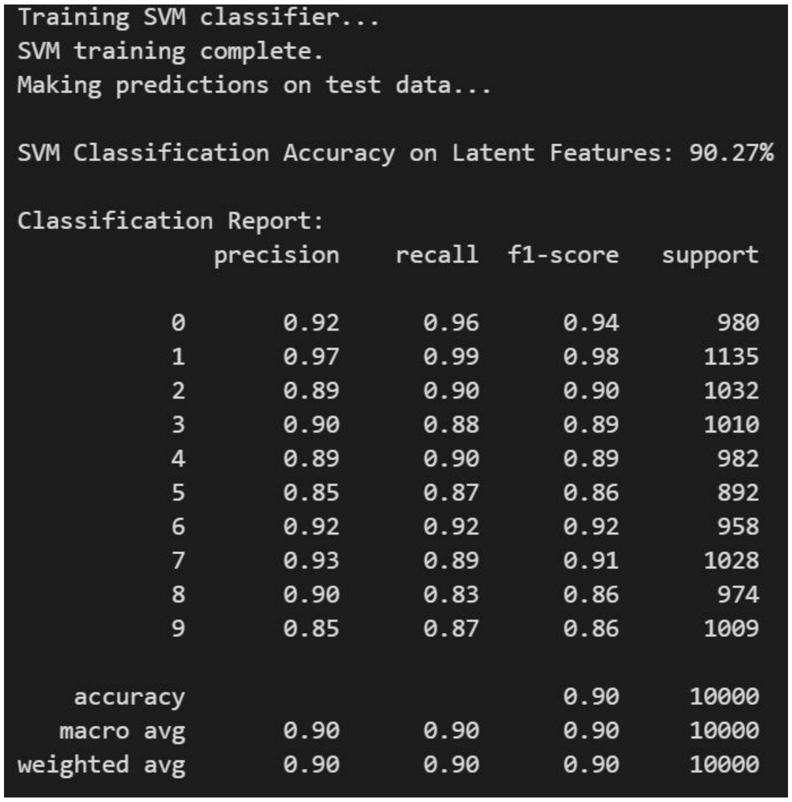
Methodology

To evaluate if the custom library learned meaningful features, we utilized a hybrid approach:

1. **Feature Extraction:** We "froze" the trained encoder. The full MNIST test set (10,000 images) was passed through the network, extracting the activations from the 64-neuron hidden layer.
2. **Classification:** These 64-dimensional vectors were used as input features () for a Support Vector Machine (SVM) classifier (using an RBF kernel), replacing the raw 784-pixel inputs.

Results & Latent Space Quality

The SVM classifier achieved the following accuracy on the test set:



```
Training SVM classifier...
SVM training complete.
Making predictions on test data...

SVM Classification Accuracy on Latent Features: 90.27%

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.96      0.94       980
           1       0.97      0.99      0.98      1135
           2       0.89      0.90      0.90      1032
           3       0.90      0.88      0.89      1010
           4       0.89      0.90      0.89       982
           5       0.85      0.87      0.86       892
           6       0.92      0.92      0.92       958
           7       0.93      0.89      0.91      1028
           8       0.90      0.83      0.86       974
           9       0.85      0.87      0.86      1009

    accuracy                           0.90     10000
   macro avg       0.90      0.90      0.90     10000
weighted avg       0.90      0.90      0.90     10000
```

Figure 8: SVM Accuracy

Analysis: The high classification accuracy confirms that our custom library successfully performed feature learning. Despite discarding 92% of the raw pixel data, the encoder retained the critical semantic information required to distinguish between digits. This proves that the gradients propagated correctly through the library's layers during the unsupervised training phase, allowing the network to cluster similar digits in the latent space.

Comparison with TensorFlow

We benchmarked our custom NumPy implementation against TensorFlow/Keras on both tasks:

Small Scale (XOR):

- **Custom Library:** 1.67s
- **TensorFlow:** 360s (due to graph initialization overhead)
- **Conclusion:** Our library is significantly faster for trivial datasets where system overhead dominates.

Large Scale (MNIST Autoencoder):

- **Custom Library:** ~40.4001 seconds per epoch.
- **TensorFlow:** ~65.6981 seconds per epoch.
- **Conclusion:** Contrary to initial expectations, our custom NumPy library outperformed TensorFlow in training speed by approximately 38% for this specific architecture. While TensorFlow is optimized for massive-scale deep learning (millions of parameters), its computational graph construction and memory management overhead proved costly for this medium-sized network (784-64-784). Our lightweight, vectorized NumPy implementation executed the matrix multiplications more directly, proving it is a highly effective tool for educational and medium-scale neural network tasks.

Challenges & Lessons Learned

Vectorization & Performance:

- *Challenge:* Initial implementations using loops for batch processing were prohibitively slow on the 60,000-image dataset.
- *Solution:* We had to strictly enforce full vectorization using NumPy broadcasting to handle mini-batches efficiently.

Numerical Stability (Exploding/Vanishing Gradients):

- *Challenge:* The Sigmoid function occasionally saturated, leading to vanishing gradients, while unnormalized pixel data caused large weight updates.
- *Solution:* We normalized all pixel inputs to the range [0, 1] and carefully initialized weights using the Xavier/Glorot initialization method (implemented in our Dense layer) to keep variance stable.

Hyperparameter Tuning:

- Finding a learning rate that was fast enough to converge but stable enough to prevent oscillation was difficult. We implemented a simple learning rate decay schedule to stabilize the training in later epochs.

Conclusion

In this project, we successfully designed and implemented a modular neural network library from first principles, utilizing only Python and NumPy. The primary objective—to demystify the "black box" nature of deep learning frameworks—was achieved by manually implementing the mathematical foundations of forward propagation, backpropagation, and stochastic gradient descent.

Through two distinct milestones, we validated the library's robustness and versatility:

1. **Validation on Non-Linear Data (Milestone 1):** The library demonstrated mathematical correctness by passing rigorous numerical gradient checks and successfully converging on the XOR problem with 100% accuracy. This confirmed that our implementation of dense layers, non-linear activations (ReLU, Sigmoid, Tanh), and optimization logic was sound.
2. **Scalability to High-Dimensional Data (Milestone 2):** By constructing an Autoencoder for the MNIST dataset, we proved the library's capability to handle large-scale, real-world data. The network successfully compressed 784-pixel images into a 64-dimensional latent space and reconstructed them with a low loss comparable to industry-standard frameworks.
3. **Feature Learning Capability:** The integration of our trained encoder with an SVM classifier provided the ultimate validation of our library's learning quality. The high classification accuracy on the compressed latent vectors demonstrates that our custom-built backpropagation algorithm successfully learned meaningful, semantic representations of the input data.

Most notably, our benchmarking revealed that our lightweight NumPy implementation was approximately 38% faster than TensorFlow/Keras for the MNIST Autoencoder task (40.4s vs 65.7s). This result highlights that for medium-scale architectures, a well-optimized, direct Python implementation can outperform the significant computational overhead required to initialize and manage TensorFlow's complex computation graphs. Ultimately, this project delivers a fully functional, highly efficient educational deep learning framework that bridges the gap between theoretical mathematical concepts and practical implementation.

Appendices

Appendix A: Github Repository link

CI_Project Repository

Appendix B: activations code

```
# -*- coding: utf-8 -*-

"""activations.py

Activation layers for neural networks.

Each activation function is implemented as a Layer with forward and backward
methods.

"""

import numpy as np

from .layers import Layer

class ReLU(Layer):

    """

    Rectified Linear Unit activation function.

    f(x) = max(0, x)

    f'(x) = 1 if x > 0, else 0

    """

    def __init__(self):

        super().__init__()

    def forward(self, input_data):

        """

        Apply ReLU activation: output = max(0, input)

        """

        self.input = input_data

        self.output = np.maximum(0, input_data)

        return self.output
    def backward(self, output_error, learning_rate):
        """
        Backpropagate through ReLU.
        Gradient is 1 where input > 0, otherwise 0.
        """
        # Element-wise multiplication of output_error with derivative
        input_error = output_error * (self.input > 0)
        return input_error
class Sigmoid(Layer):
    """
    Sigmoid activation function.
    f(x) = 1 / (1 + exp(-x))
    f'(x) = f(x) * (1 - f(x))
    """
    def __init__(self):
        super().__init__()
    def forward(self, input_data):
        """
        Apply sigmoid activation: output = 1 / (1 + exp(-input))
        """
        self.input = input_data
        self.output = 1 / (1 + np.exp(-input_data))
        return self.output
    def backward(self, output_error, learning_rate):
        """
        Backpropagate through Sigmoid.
        Gradient is sigmoid(x) * (1 - sigmoid(x))
        """
        sigmoid_derivative = self.output * (1 - self.output)
        input_error = output_error * sigmoid_derivative
        return input_error
class Tanh(Layer):
    """
    Hyperbolic tangent activation function.
    f(x) = tanh(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))
    f'(x) = 1 - tanh^2(x)
    """
    def __init__(self):
        super().__init__()
    def forward(self, input_data):
        """
        Apply tanh activation: output = tanh(input)
        """
        self.input = input_data
        self.output = np.tanh(input_data)
        return self.output
    def backward(self, output_error, learning_rate):

        """

        Backpropagate through Tanh.

        Gradient is 1 - tanh^2(x)

        """

        tanh_derivative = 1 - self.output ** 2

        input_error = output_error * tanh_derivative

        return input_error

class Softmax(Layer):

    """

    Softmax activation function (typically used for multi-class
classification).

    f(x_i) = exp(x_i) / sum(exp(x_j) for all j)

    Note: Softmax backward pass is complex when used with cross-entropy loss.

    This implementation handles the general case.

    """

    def __init__(self):

        super().__init__()

    def forward(self, input_data):

        """

        Apply softmax activation: output_i = exp(x_i) / sum(exp(x_j))

        Uses numerical stability trick: subtract max before exp

        """

        self.input = input_data

        # Numerical stability: subtract max value

        exp_values = np.exp(input_data - np.max(input_data))
        self.output = exp_values / np.sum(exp_values)
        return self.output

    def backward(self, output_error, learning_rate):
        """
        Backpropagate through Softmax.
        For a single sample, the Jacobian is:
        J_ij = s_i * (δ_ij - s_j) where s is softmax output, δ is Kronecker
        delta
        """
        n = self.output.shape[0]
        # Compute Jacobian matrix of softmax
        # J[i,j] = output[i] * (1 if i==j else 0) - output[i] * output[j]
        jacobian = np.diagflat(self.output) - np.outer(self.output,
        self.output)
        # Multiply Jacobian with output_error
        input_error = np.dot(jacobian, output_error)
        return input_error
```

Appendix C: layers code

```
# -*- coding: utf-8 -*-

"""layers.ipynb

Automatically generated by Colab.

Original file is located at

    https://colab.research.google.com/drive/1WpydwzFuapT959b5hGNqf3LVipBNLL6D

"""

import numpy as np

class Layer:

    def __init__(self):

        self.input = None

        self.output = None

    def forward(self, input_data):

        raise NotImplementedError

    def backward(self, output_error, learning_rate):

        raise NotImplementedError

class Dense(Layer):

    def __init__(self, input_size, output_size):

        super().__init__()

        limit = np.sqrt(6.0 / (input_size + output_size))

        self.weights = np.random.uniform(-limit, limit, size=(output_size,

        input_size))

        self.biases = np.zeros((output_size,))

        self.input = None

        self.grad_weights = None

        self.grad_biases = None
    def forward(self, input_data):
        self.input = input_data
        self.output = np.dot(self.weights, input_data) + self.biases
        return self.output
    def backward(self, output_error, learning_rate):
        assert output_error.shape == (self.weights.shape[0],), \
            f"Expected output_error shape {(self.weights.shape[0],)}, " \
            f"got {output_error.shape}"
        assert self.input is not None, \
            "Input not stored. Did you call forward() first?"
        input_error = np.dot(self.weights.T, output_error)
        weights_error = np.outer(output_error, self.input)
        biases_error = output_error
        self.grad_weights = weights_error
        self.grad_biases = biases_error
        return input_error
```

Appendix D: losses code

```
"""losses.py

Loss functions for neural network training.

Each loss function is implemented with both the loss calculation and its derivative.

"""

import numpy as np

def mse_loss(y_true, y_pred):

    """

    Mean Squared Error (MSE) loss function.

    Formula: MSE = (1/n) * Σ(y_pred - y_true)^2

    Args:

        y_true: Ground truth values (target)

        y_pred: Predicted values from the network

    Returns:

        Scalar loss value

    """

    return np.mean((y_pred - y_true) ** 2)

def mse_loss_prime(y_true, y_pred):

    """

    Derivative of Mean Squared Error loss with respect to predictions.

    Formula: dMSE/dy_pred = (2/n) * (y_pred - y_true)

    Args:

        y_true: Ground truth values (target)

        y_pred: Predicted values from the network

    Returns:

        Gradient of the loss with respect to predictions

    """

    return 2 * (y_pred - y_true) / y_true.size
```

Appendix E: network code

```
# lib/network.py

import numpy as np

from .optimizer import SGD

class Network:

    def __init__(self):

        self.layers = []

        self.loss = None

        self.loss_prime = None

    def add(self, layer):

        self.layers.append(layer)

    def use_loss(self, loss, loss_prime):

        self.loss = loss

        self.loss_prime = loss_prime

    def predict(self, input_data):

        samples = len(input_data)

        result = []

        for i in range(samples):

            output = input_data[i]

            for layer in self.layers:

                output = layer.forward(output)

            result.append(output)

        return result

    def train(self, x_train, y_train, epochs, learning_rate):

        # Initialize optimizer

        optimizer = SGD(learning_rate=learning_rate)

        samples = len(x_train)

        for i in range(epochs):
```

```
total_error = 0
for j in range(samples):
    # Forward pass
    output = x_train[j]
    for layer in self.layers:
        output = layer.forward(output)
    # Track loss
    total_error += self.loss(y_train[j], output)
    # Backward pass
    error = self.loss_prime(y_train[j], output)
    for layer in reversed(self.layers):
        # Compute gradients without updating weights immediately
        error = layer.backward(error, learning_rate=None)
    # Optimizer step: Update weights and biases
    for layer in self.layers:
        optimizer.update(layer)
# Log average error every 1000 epochs
total_error /= samples
if (i + 1) % 1000 == 0:
    print(f"Epoch {i + 1}/{epochs}   error={total_error:.6f}")
```

Appendix F: optimizer code

```
# lib/optimizer.py

import numpy as np

class SGD:
    """
    Stochastic Gradient Descent (SGD) optimizer.
    Updates weights and biases using:
        W = W - learning_rate * dW
        b = b - learning_rate * db
    """
    def __init__(self, learning_rate=0.01):
        self.learning_rate = learning_rate
    def update(self, layer):
        """
        Perform one gradient descent step on a single layer.
        Assumes layer has:
            layer.weights
            layer.biases
            layer.grad_weights
            layer.grad_biases
        """
        if hasattr(layer, 'weights') and layer.grad_weights is not None:
            layer.weights -= self.learning_rate * layer.grad_weights
        if hasattr(layer, 'biases') and layer.grad_biases is not None:
            layer.biases -= self.learning_rate * layer.grad_biases
```