Ain Shams University
Faculty of Engineering
Mechatronics Dept.

CSE473s-Computational Intelligence

Major Task-Part (1)

Sec 3/Team 20

Name	ID
Omar Mohsen Mohamed	2100881
Youssf Mostafa Kamel	2101019
Ahmed Mohamed Abdallah	2101037

Submitted to: Eng. Abdallah Awdallah

Table of Contents

Introduction:

In this project, we developed a modular neural network library from scratch using only Python and NumPy. The goal from part (1) is to implement all core components of neural network-layers, activations, losses, optimizers and validate correct functionality by successfully solving the XOR classification problem. This serves as proof that both forward propagation and backward propagation are correctly implemented before moving to the more complex tasks in part (2).

Objective:

This phase focuses on understanding and building the low-level mechanics that drive modern neural networks.

More specifically, the objectives are:

1. **Design a Modular Neural Network Library**
    - Implement reusable components including layers, activation functions, loss functions, and optimizer.
    - Ensure the library follows clean and extensible architecture suitable for later tasks (autoencoder + SVM).
2. **Implement Forward and Backward Propagation**
    - Code the mathematical operations behind dense layers and nonlinear activations.
    - Compute gradients analytically for trainable parameters.
3. **Verify Correctness through Gradient Checking**
    - Compare analytical gradients with numerical approximations
    - Ensure the backpropagation implementation is accurate to a high degree of precision.
4. **Train and validate a Model on the XOR problem**
    - Construct a small MLP capable of learning the XOR function.
    - Demonstrate correct predictions for all four XOR inputs after training.

Library design and Architecture Choices

The neural network library implemented in this project was designed with two main goals:

1. **Modularity and extensibility**
2. **Transparency of the underlying mathematical operations.**

1. Modular file structure

The library organized into five main modules, each responsible for a sperate functional component of the network:

- **Layers.py:** Implements trainable layers (mainly the dense layer).
- **Activations.py:** Implements nonlinear activation layers as standalone classes.
- **Losses.py:** Handles loss computation and the intial guess gradient passed into backpropagation.
- **Optimizer.py:** Defines the update rules for weights and biases.
- **Network.py:** A sequential-like model class that orchestrates the forward pass, backward pass, and training loop.

---

2. Layer Abstraction

A base layer interface was used to enforce a consistent design across all components.

Every layer implements:

- **Forward(x):** computes output of the layer.
- **Backward(out):** propagates gradients to the previous layer.
- **For trainable layers:** stores parameter gradients for optimizer updates.

---

3. Activation layers as independent Modules

Activation functions were implemented as separate classes (e.g. **Sigmoid, tanh, ReLU, Softmax).**

Each activation implements its own forward and backward formulas; we made it like this for many reasons:

- It makes the architecture more flexible; any dense layer can be followed by any activation.
- Allows developing new activations easily without modifying core logic.
- This design becomes essential for part (2) when autoencoder uses mixed activations.

---

4. Loss Function as a Standalone Component

The library implements **Mean Squared Error (MSE)** as required:

$$L = \frac{1}{N}\Sigma(y_{true} - y_{pred})\ \hat{}\ 2$$

Design choice:

- The loss function is responsible only for
  1) **Computing the forward loss**
  2) **Providing the gradient w.r.t the model output.**

By keeping the loss independent, the same network can be trained using different loss functions without changing the model architecture.

---

5. SGD Optimizer Design

The optimizer is intentionally simple: **Stochastic Gradient Descent**

Design decisions:

- **The optimizer receives a list of trainable layers and their gradients.**
- **It directly updates weights and biases after backward pass.**

---

6. Sequential Network Architecture

A key component of the library is the Sequential Model (in network.py). It manages the entire training pipeline.

The model class handles:

- Forward pass through all layers.
- Loss computation.
- Backward pass starting from the loss.
- Optimizer updates.
- Training loop logic.

Results for XOR Test

Network Architecture

```
# Create the model
model = Sequential()

# Add the first hidden layer with 4 units
model.add(Dense(2,4,activation="relu"))

# Add the second hidden layer with 2 units
model.add(Dense(4,2,activation=""))

# Add the output layer with only 1 unit
model.add(Dense(2,1,activation="sigmoid"))


print("Network Architecture:")
model.summary()
print()
```

Figure 1: Network Architecture

Training the Network

```
optimizer = SGD(learning_rate=0.1)
optimizer.register_layers(model.layers)
history = model.fit(X, Y, epochs=10000, optimizer=optimizer, print_every=100)
```

Figure 2: Network Parameters

Completed Training

- Final Loss: 0.000313

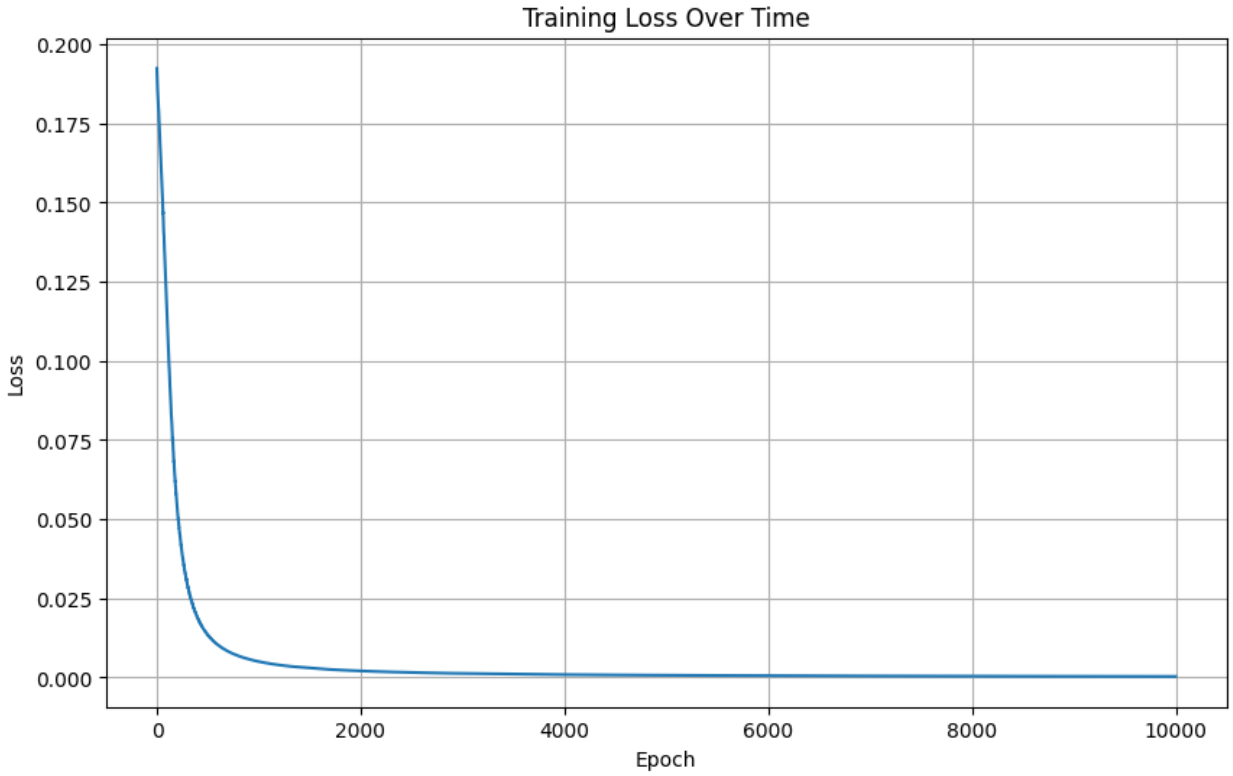

Figure 3 : Training Loss over time