PROJECT MILESTONE 1

CSE473S_COMPUTER INTELLIGENCE
BUILD YOUR OWN NEURAL NETWORK LIBRARY & ADVANCED APPLICATIONS

NAME	ID	SEC.
ZEYAD SAMER LOTFY	2100820	3
MOHAMED ISLAM SALAH	2101439	3
MOAZ GAMAL ALSAYED	2101231	1
MOHAMED MONTASSER	2100543	1
FATMA SAMY AHMED	2100660	2

1. Overview of the Custom Neural Network Library

The custom neural network library developed in this project is a minimal yet complete framework that replicates the internal mechanisms of deep-learning systems such as TensorFlow and PyTorch.

It uses only NumPy as the numerical backend, meaning everything :forward propagation, backward propagation, gradient calculation, and weight updates was fully implemented manually.

Purpose of the library:
• To implement all essential components without relying on high-level frameworks.
• To create a foundation capable of training simple networks (XOR, autoencoders, classifiers).

Main components of the library:
• layers.py – defines core building blocks (Layer base class and Dense layer).
• activations.py – ReLU, Sigmoid, Tanh, Softmax activations (each with forward & backward).
• losses.py – Mean Squared Error (MSE) loss function.
• optimizer.py – Stochastic Gradient Descent (SGD) optimizer.
• network.py – Sequential model that handles forward(), backward(), and parameter update.
• __init__.py – groups everything to behave like an importable package.

The main purpose is to recreate the workflow inside TensorFlow:
Input → Layers → Activations → Loss Calculation → Backpropagation → Weight Updates.

2. Gradient Checking and Why It Is Important

Gradient checking is a method used to verify the correctness of the backward() function in each layer especially Dense layers which contain trainable parameters.

Neural networks learn using gradient descent:

$$W_{new} = W_{old} - learning\_rate \times (dL/dW)$$

If the gradient (dL/dW) is incorrect:

- Training fails or becomes unstable.
- Loss may increase instead of decreasing.
- The model will not converge regardless of the number of epochs.

How gradient checking works:

1. Compute numerical gradient using finite-difference approximation:

$$dL/dW \approx (L(W + \varepsilon) - L(W - \varepsilon)) / (2\varepsilon)$$

2. Compute analytic gradient using backward() in your implementation.
3. Compare both. If the difference is extremely small → gradient is correct.

Why Dense layer is the most important to check:

- It contains both weights and biases (the actual learnable parameters).
- Any mistake in matrix multiplication propagates errors across the whole network.
- Gradient shape and direction matter greatly in weight updates.

Activation layers are simpler mathematically, so gradient errors there are less likely, while Dense layer gradients are the foundation of the whole training process.

3. XOR Implementation and Validation

The XOR (exclusive OR) problem is a classic benchmark for testing neural networks. It is non-linearly separable, meaning a single-layer perceptron cannot solve it. A neural network with at least one hidden layer MUST be used.

The goal:
XOR(0,0) = 0
XOR(0,1) = 1
XOR(1,0) = 1
XOR(1,1) = 0

Network architecture used:
Input → Dense(2 → 4) → Tanh → Dense(4 → 1) → Sigmoid

Why Tanh in hidden layer?
• Tanh outputs both negative and positive values (range: –1 to 1).
• This helps the hidden layer separate XOR patterns better than Sigmoid.

Why Sigmoid in output layer?
• XOR output must be 0 or 1.
• Sigmoid gives a probability between 0 and 1.

Training process steps:
1. Forward pass – compute predictions.
2. Loss calculation – using MSE.
3. Backward pass – compute gradients using chain rule.
4. SGD update – adjust the weights.

Training validation:
• Loss curve decreases steadily (proving learning is happening).
• Predictions approach correct values.
• After training, outputs are close to ideal XOR values.

The successful learning of XOR proves that:
• forward() and backward() are implemented correctly
• gradients match numerical expectations
• SGD updates work properly
• the model can learn non-linear patterns