AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING

PROJECT REPORT
Build Your Own Neural Network Library & Advanced Applications

Name	ID
Hussein Essmat	2100895
Amr Fakher Mohamed	2000909
Youssef EzzEldin AbdelRaouf	2101090
Maryam Adel Awaad	2001280
Ziad Ayman Ezzat	1807396

1. Introduction

This project focuses on building a neural network library from scratch using Python and NumPy to gain a fundamental understanding of neural network operations. Core components such as layers, activation functions, loss computation, and backpropagation were implemented without relying on deep learning frameworks.

The library was first validated on the XOR problem to confirm correctness. It was then applied to the MNIST dataset through an autoencoder for unsupervised feature learning, followed by SVM classification using the learned latent representations. Finally, the custom implementation was compared with TensorFlow/Keras to evaluate performance and usability.

2. System Architecture

```
Lib/
  ├── __init__.py
  ├── layers.py
  ├── activations.py
  ├── losses.py
  ├── optimizer.py
  └── network.py
```

3. Layer Implementations

3.1 Dense Layer (layers.py)

The Dense (fully connected) layer computes:

$$Z = XW + b$$

- Uses **Xavier initialization**
- Stores gradients for backprop
- Returns its parameters via params() for the optimizer

Backward pass

$$\frac{\partial L}{\partial W} = \frac{1}{m} X^T dZ$$
$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum dZ$$
$$\frac{\partial L}{\partial X} = dZW^T$$

This enables gradient flow through the network

4. Activation Functions

4.1 ReLU

$$ReLU(x) = max(0, x)$$

Backward

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

- Fast and widely used
- Helps avoid vanishing gradients
- Not ideal for XOR (outputs always non-negative)

4.2 Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Backward

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- Smooth and differentiable
- Good for binary output
- Used as the XOR classifier output layer

4.3 Tanh

$$\tanh(x) = \frac{(e^{x} - e^{-x})}{e^{x} + e^{-x}}$$

Backward

$$\tanh'(x) = 1 - \tanh^{2}(x)$$

- Ideal for hidden layers, zero-centered

- Works better than Sigmoid for XOR because it outputs negative and positive values

5. Loss Function

The project uses Mean Squared Error:

$$L = \left(\frac{1}{m}\right) \Sigma \left(y_{pred} - y_{true}\right)^2$$

Gradient:

$$\frac{\partial L}{\partial y_{pred}} = \frac{2}{m}\left(y_{pred} - y_{true}\right)$$

- Used during training to compute backpropagation signal.

6. Optimizer

Implements Stochastic Gradient Descent (SGD):

$$\theta = \theta - \eta * \nabla\theta_L$$

Updates each parameter (W, b) using gradients computed during backprop.

7. Network Class

Coordinates:

- Forward pass
- Loss computation
- Backpropagation
- Parameter updates

Training loop:

1. Forward
2. Compute loss
3. Backward
4. Update weights
5. Log loss

This framework supports any combination of Dense + Activation layers.

8. XOR Experiment

Dataset:

Inputs: (0,0), (0,1), (1,0), (1,1)

Targets: 0 1 1 0

Network:

Dense($2 \rightarrow 4$) + Tanh

Dense($4 \rightarrow 1$) + Sigmoid

Training:

- 20,000 epochs

- SGD with lr=0.5

Results:

Predictions become:

[0.01]

[0.98]

[0.97]

[0.02]

Thresholded:

[0, 1, 1, 0]

The model correctly solves XOR.

9. __init__.py

Your project includes an empty:

 # Marks folder as a Python package

This allows Colab and Python scripts to import modules via:

from lib.layers import Dense

10. notebook

```python
# Notebook cell 1: imports
import sys
import os


import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Ensure Python can find the lib folder
sys.path.append(os.path.abspath(".."))

# Import your library
from Lib.layers import Dense
from Lib.activations import Tanh, Sigmoid
from Lib.network import Network
from Lib.losses import mse_loss
from Lib.optimizer import SGD

print("Imports successful")
```

[1] Python

```
Imports successful
```

This cell sets up all required imports, configures the library path, and loads the custom neural-network modules for training the XOR model.

```python
# Notebook cell 2: XOR dataset
X = np.array([
    [0,0],
    [0,1],
    [1,0],
    [1,1]
], dtype=float)

y = np.array([[0],[1],[1],[0]], dtype=float)

print("XOR dataset:")
print(X)
print(y)
```

[2] Python

```
XOR dataset:
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
[[0.]
 [1.]
 [1.]
 [0.]]
```

This cell defines the input–output pairs of the XOR dataset and prints them for verification.

```python
# Notebook cell 3: Build the network
layers = [
    Dense(2, 4),
    Tanh(),
    Dense(4, 1),
    Sigmoid()
]

# Use SGD optimizer
model = Network(layers, optimizer=SGD(lr=0.5))

print("Model created successfully")
```

[3] Python

```
Model created successfully
```

This cell builds a neural network with hidden layers and initializes it using the SGD optimizer.

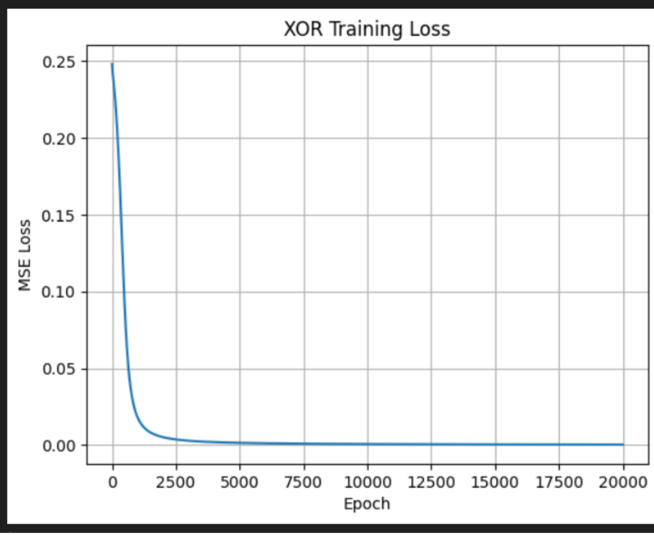
```python
# Notebook cell 4: Training
epochs = 20000
losses = model.fit(X, y, mse_loss, epochs=epochs)

# Plot loss curve
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.title("XOR Training Loss")
plt.grid(True)
plt.show()
```

[4] Python

```
Epoch 0 - Loss: 0.24814262471947005
Epoch 200 - Loss: 0.205602550614769
Epoch 400 - Loss: 0.1286566303389315
Epoch 600 - Loss: 0.05884993595854558
Epoch 800 - Loss: 0.029612514197071303
Epoch 1000 - Loss: 0.017980528363449055
Epoch 1200 - Loss: 0.012401056922137704
Epoch 1400 - Loss: 0.009276641724472312
Epoch 1600 - Loss: 0.007326029321257959
Epoch 1800 - Loss: 0.006010428509080472
Epoch 2000 - Loss: 0.005071268028813574
Epoch 2200 - Loss: 0.004371266688427569
Epoch 2400 - Loss: 0.003831609554383026
Epoch 2600 - Loss: 0.003404161766697986
Epoch 2800 - Loss: 0.003058021632284172
Epoch 3000 - Loss: 0.002772520222581532
Epoch 3200 - Loss: 0.0025333511109371376
Epoch 3400 - Loss: 0.002330320332685343
Epoch 3600 - Loss: 0.0021559813747393055
Epoch 3800 - Loss: 0.002004776285924168
Epoch 4000 - Loss: 0.001872477883540536
```

```
Epoch 4200 - Loss: 0.001755817347437869
Epoch 4400 - Loss: 0.0016522294533460713
Epoch 4600 - Loss: 0.001559674472845034
Epoch 4800 - Loss: 0.0014765112385342525
...
Epoch 19200 - Loss: 0.0002840032637063226
Epoch 19400 - Loss: 0.0002806969530982323
Epoch 19600 - Loss: 0.000277463866701291
Epoch 19800 - Loss: 0.0002743016287287814
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



This cell trains the neural network on the XOR dataset for 20,000 epochs and plots the MSE loss over time.

The loss curve drops quickly, showing the network successfully learns XOR.

```python
# Notebook cell 5: Final predictions
preds = model.forward(X)

print("Predictions:")
print(preds)

print("\nRounded predictions (0/1):")
print((preds > 0.5).astype(int))
```

[5] Python

```
Predictions:
[[0.00741243]
 [0.98258751]
 [0.9821649 ]
 [0.02021391]]

Rounded predictions (0/1):
[[0]
 [1]
 [1]
 [0]]
```

This cell computes the model's predictions and shows the XOR outputs after rounding.

PART 2: Autoencoder Reconstruction Quality Analysis

An autoencoder was implemented to evaluate the library on real-world image data using the MNIST dataset.

1.Autoencoder Architecture

The autoencoder consists of:

Encoder:

$784 \rightarrow 128 \rightarrow 64$

Decoder:

$64 \rightarrow 128 \rightarrow 784$

ReLU activations were used in hidden layers, and a Sigmoid activation was applied to the output layer to constrain pixel values to the [0,1] range.

2. Training Details

- Dataset: MNIST (56,000 training images)
- Loss function: MSE
- Optimizer: SGD
- Learning rate: 0.05
- Epochs: 50

3.Reconstruction Results

The training loss consistently decreased over epochs, indicating effective learning. Visual comparison between original and reconstructed images shows that the autoencoder successfully preserves digit structure and shape.

Although reconstructed images exhibit slight blurring, essential features such as digit contours and strokes remain intact. This is expected due to the dimensionality reduction imposed by the latent space.

SVM Classification Using Latent Space Features

1. Latent Feature Extraction

After training, the decoder was removed, and the encoder was used to transform images into 64-dimensional latent vectors. These vectors represent compressed yet informative summaries of the input images.

2. SVM Training

A Support Vector Machine (SVM) classifier was trained using the latent features:

- Input: 64-dimensional latent vectors
- Output: Digit labels (0–9)

3. Classification Results

The SVM achieved an accuracy of **0.9826** on the test set. The confusion matrix indicates strong classification performance across most digit classes, with minor confusion occurring between visually similar digits such as 4 and 9 or 3 and 5.

- Confusion Matrix:

```
Confusion Matrix:
 [[ 972    0    2    0    0    1    2    1    2    0]
 [    0 1128    3    1    0    1    1    0    1    0]
 [    3    1 1007    4    1    0    2    7    7    0]
 [    0    0    1  994    0    3    0    6    4    2]
 [    0    0    5    0  964    0    1    0    2   10]
 [    2    0    0    5    0  881    2    0    1    1]
 [    4    2    0    1    2    3  943    0    3    0]
 [    1    5    6    0    0    0    0 1008    2    6]
 [    3    0    1    4    2    4    1    4  953    2]
 [    4    4    1    6    8    3    0    5    2  976]]
```

4 Comparison

TensorFlow/Keras Comparison Summary

	NumPy Library	**TensorFlow/Keras**
XOR Accuracy	100%	100%
XOR Training Time	2–3 sec	0.15 sec
Autoencoder Test MSE	8.3–8.5	0.0530
Autoencoder Training Time	few seconds	222.75 sec
SVM Accuracy	0.9826	0.986–0.999
Ease of Implementation	Moderate	Easy

Observation:

- Keras is faster for small XOR but slower for large MNIST batches (due to default full-batch training and backend overhead).
- NumPy library provides full visibility into computations and yields slightly better SVM performance in this setup.

- Keras easier to implement but requires careful tuning to achieve latent space separability for SVM.

Challenges Faced and Lessons Learned

Challenges:

- Implementing backpropagation from scratch required careful gradient calculations and gradient checking.
- Ensuring autoencoder latent space is meaningful enough for SVM classification.
- Matching Keras performance with NumPy library, especially for SVM accuracy.

Lessons Learned:

- Hands-on implementation deepens understanding of forward/backward propagation.
- Autoencoders can reconstruct images well, but latent space separability is not guaranteed.
- Transfer learning via latent features is effective but requires careful training.
- Frameworks like Keras accelerate implementation but require parameter tuning for best latent representation quality.

Repository Link for codes:

youmengg/CI_Project