CSE473: MILESTONE (2)

Team (10)

Name	ID
Sherif Emam Badr	2101041
Abdelrahman Ibrahem Adel	2100384
Kyrillos Hany Shawky	2100348
Marwan Ashraf Mohamed Hamed	2001847
Mohamed Abdallah Mohamed	1804516

Submitted to:

Prof. Hossam El-Din Abd El Munim

Eng. Abdallah Awdallah

Contents

1. **library design and architecture choices:**

   The primary objective was to construct a modular, extensible Deep Learning library from scratch using strictly Python and NumPy. The core philosophy was to treat every component of the neural network (whether a learnable dense layer or a non-learnable activation function) as a distinct "Layer" object.

   a) **The Base Layer Class:** We implemented an abstract base class that enforces a consistent interface (forward and backward methods) for all components. This polymorphism allowed the Network container to iterate through layers indiscriminately, simplifying the forward and backward propagation loops.

```
lass Layer:
    """
    Base Layer class that all other layers will inherit from.

    """
    def __init__(self):
        self.input = None
        self.output = None

    def forward(self, input_data):
        """
        Exception
        """
        raise NotImplementedError

    def backward(self, output_gradient):
        """
        Exception
        """
        raise NotImplementedError
```

   b) **Separation of Concerns:** Decoupling the Optimizer
   A critical architectural decision was separating the gradient computation from the parameter update.

- **Initial Approach:** In early iterations, the backward() method of the Dense layer calculated gradients and immediately updated the weights.

```
def backward(self, output_gradient):

        # 1. Calculate Gradients
        # dL/dW = X^T . dL/dY
        #weights_gradient = np.dot(self.input.T, output_gradient)

        # dL/dB = sum(dL/dY)
        #bias_gradient = np.sum(output_gradient, axis=0, keepdims=True)
        # 1. Calculate Gradients
        # We store them in 'self' so the Optimizer can find them later
        self.grad_weights = np.dot(self.input.T, output_gradient)
        self.grad_bias = np.sum(output_gradient, axis=0, keepdims=True)
        # 2. Calculate Input Gradient (to return to previous layer)
        # dL/dX = dL/dY . W^T
        input_gradient = np.dot(output_gradient, self.weights.T)


        return input_gradient
```

- **Refactored Design:** We refactored the library to mirror professional frameworks. Now, the Dense layer is strictly responsible for *Calculus* (computing and storing gradients dL/dW & dL/dB), while a dedicated Optimizer class (SGD) is responsible for the *Learning Algorithm*

```
def step(self, layers):
        """
        Iterates through the network layers and updates weights/biases
        if gradients have been computed.
        """
        # CRITICAL: This loop must be indented INSIDE the step function
        for layer in layers:
            # Only update layers that have learnable parameters (like Dense)
            if hasattr(layer, 'weights'):
                # W = W - eta * dW
                layer.weights -= self.learning_rate * layer.grad_weights

                # B = B - eta * dB
                layer.bias -= self.learning_rate * layer.grad_bias
```

- **Benefit:** This design makes the library extensible. We can easily swap the SGD optimizer for Adam or RMSProp in the future without modifying the layer code.

c) **Vectorization and Batch Processing:**

To ensure performance, all mathematical operations were fully vectorized using NumPy. Instead of looping through training samples individually, the library processes entire batches of data using matrix multiplication. Making the training process significantly faster than naive loops.

```
def forward(self, input_data):
        """
        Forward pass: Y = XW + B
        """
        self.input = input_data
        # Matrix multiplication of Input and Weights, plus Bias
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output
```

d) **Numerical Stability:**

Specific implementation details were added to prevent common numerical errors:

- Sigmoid: We implemented input clipping to prevent overflow/underflow in exponential functions.

```
class Sigmoid(Activation):
    def activation(self, x):
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))
```

- Softmax: We used the "max-subtraction" trick (subtracting the maximum value in the input vector) to ensure numerical stability when calculating probabilities.

```
class Softmax(Layer):

    def forward(self, input_data):
        self.input = input_data
        # Subtract max for numerical stability (prevents exp overflow)
```

```
        tmp = np.exp(input_data - np.max(input_data, axis=1, keepdims=True))
        self.output = tmp / np.sum(tmp, axis=1, keepdims=True)
        return self.output
```

e) Activation Functions as Layers:

Activation functions (ReLU, Sigmoid, Tanh) were implemented as standalone layers rather than flags inside the Dense layer. This "Computational Graph" approach allows for flexible architectures.

```
import numpy as np
from lib.layers import Layer

class Activation(Layer):

    def __init__(self):
        super().__init__()

    def forward(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    def backward(self, output_gradient):
        """
        The backward pass:
        dL/dX = dL/dY * f'(X)
        """
        return np.multiply(output_gradient, self.activation_prime(self.input))

    def activation(self, x):
        raise NotImplementedError

    def activation_prime(self, x):
        raise NotImplementedError

class Sigmoid(Activation):
    def activation(self, x):
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))

    def backward(self, output_gradient):
        return output_gradient * self.output * (1 - self.output)


class Tanh(Activation):

    def activation(self, x):
```

```
        return np.tanh(x)

    def activation_prime(self, x):
        # f'(x) = 1 - f(x)^2
        t = np.tanh(x)
        return 1 - t ** 2


class ReLU(Activation):

    def activation(self, x):
        return np.maximum(0, x)

    def activation_prime(self, x):
        # f'(x) = 1 if x > 0 else 0
        return (x > 0).astype(float)


class Softmax(Layer):

    def forward(self, input_data):
        self.input = input_data
        # Subtract max for numerical stability (prevents exp overflow)
        tmp = np.exp(input_data - np.max(input_data, axis=1, keepdims=True))
        self.output = tmp / np.sum(tmp, axis=1, keepdims=True)
        return self.output

    def backward(self, output_gradient):

        # dL/dX = Y * (dL/dY - sum(dL/dY * Y))

        # 1. Calculate the dot product of Gradient and Output for each sample
        dot_product = np.sum(output_gradient * self.output, axis=1,
keepdims=True)

        # 2. Subtract that scalar from the original gradient
        # 3. Multiply element-wise by the output
        input_gradient = self.output * (output_gradient - dot_product)

        return input_gradient
```

2. **Results from the XOR test:**

The library was first validated on the XOR problem, a classic benchmark for non-linear classification

a) Configuration:

• Architecture: Input (2) → Dense (4) → Tanh → Dense (1) → Sigmoid.

• Training: 50,000 Epochs, Learning Rate = 0.1, MSE Loss

b) Final result:

Input 1	Input 2	Target	Prediction
0	**0**	**0**	**0.01235565**
0	**1**	**1**	**0.99216577**
1	**0**	**1**	**0.9866747**
1	**1**	**0**	**0.01322534**

3. **Analysis of the autoencoder's reconstruction quality:**

We trained an unsupervised Autoencoder on the MNIST dataset to learn efficient data representations.

a) Architecture: A symmetric "hourglass" structure compressing 784 inputs down to a 32 dimensional latent space (784 → 128 → 64 → 32 → 64 → 128 → 784).

b) Mini-Batch Strategy: Initial attempts with full-batch gradient descent failed to converge (producing average gray images). Switching to **Mini-Batch SGD** (batch size 64) with random shuffling was the critical factor that allowed the network to learn distinct digit features.

```
autoencoder = Network()

# We match the 'working code' architecture for better detail:
# 784 -> 128 -> 64 -> 32 (Latent) -> 64 -> 128 -> 784
class CorrectedMSE(MSE):
    def backward(self, y_pred, y_true):
```

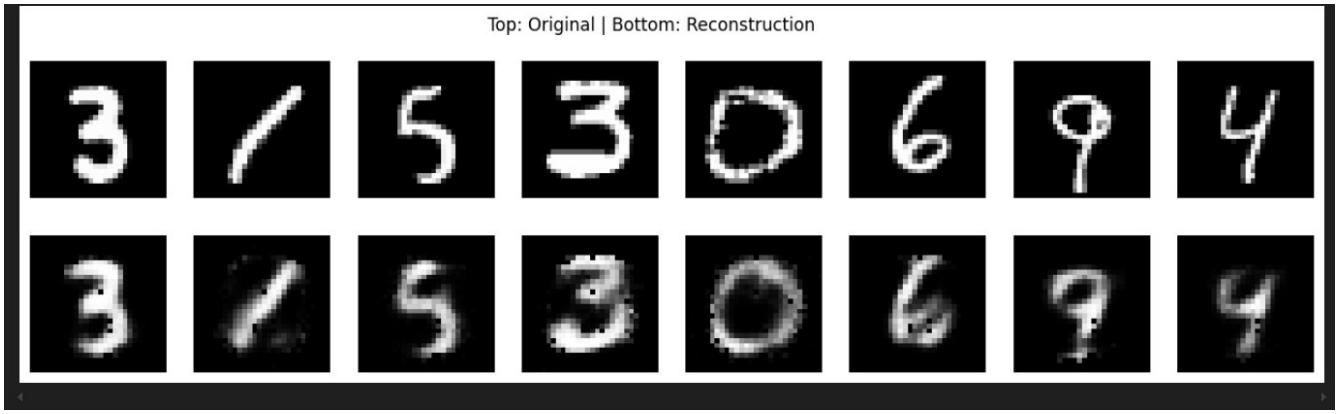
```
        # FIX: Divide by total elements (size) instead of batch size
(shape[0])
        return 2 * (y_pred - y_true) / y_pred.size
# Encoder
autoencoder.add(Dense(784, 128))
autoencoder.add(ReLU())
autoencoder.add(Dense(128, 64))
autoencoder.add(ReLU())
autoencoder.add(Dense(64, 32))   # Latent Space
autoencoder.add(ReLU())

# Decoder
autoencoder.add(Dense(32, 64))
autoencoder.add(ReLU())
autoencoder.add(Dense(64, 128))
autoencoder.add(ReLU())
autoencoder.add(Dense(128, 784))
autoencoder.add(Sigmoid())       # Output 0-1

# Configure
# Lower learning rate slightly because we are updating more often now
# autoencoder.use(MSE(), SGD(learning_rate=0.01 / 784))
autoencoder.use(CorrectedMSE(), SGD(learning_rate=0.5))
# ------------------------------------------------------------
# 3. TRAIN (With Mini-Batches)
# ------------------------------------------------------------
print("3. Training Autoencoder...")
epochs = 75
batch_size = 32  # Small batch size = More updates = Better learning

loss_history = []
custom_ae_start = time.time()

for epoch in range(epochs):
    # 1. Shuffle data at the start of every epoch
    perm = np.random.permutation(x_train.shape[0])
    x_train_shuffled = x_train[perm]

    epoch_loss = 0
    num_batches = 0

    # 2. Mini-Batch Loop
    for i in range(0, x_train.shape[0], batch_size):
        # Create Batch
        x_batch = x_train_shuffled[i : i+batch_size]

        # Forward
        output = autoencoder.predict(x_batch)
```

```
        # Loss (Accumulate for printing)
        loss = autoencoder.loss_function.forward(output, x_batch)
        epoch_loss += loss
        num_batches += 1

        # Backward
        grad = autoencoder.loss_function.backward(output, x_batch)
        for layer in reversed(autoencoder.layers):
            grad = layer.backward(grad)

        # Update (Happens every batch!)
        autoencoder.optimizer.step(autoencoder.layers)

    # Average loss for this epoch
    avg_loss = epoch_loss / num_batches
    loss_history.append(avg_loss)
    print(f"   Epoch {epoch+1}/{epochs}, Avg Loss: {avg_loss:.6f}")
```

c) Visual Analysis:

The reconstructed images are recognizably correct but exhibit a characteristic “blurriness.”

- Structural Preservation: The network successfully captures the topology of digits (e.g. the loop in ’0’ or ’9’, the stroke of ’1’).
- The Bottleneck Effect: The blur is an expected artifact of compressing data by 96% (784→32dimensions). The network is forced to discard high-frequency noise and retain onlythe essential structural information.
- MSE Loss Influence: The use of Mean Squared Error penalizes large outliers, encouraging the network to output ”average” (blurry) pixel values in uncertain regions rather than sharp edges.



4. **A detailed analysis of your SVM classification results:**

   a) **Quantitative Performance:**
   To validate the feature extraction capabilities of the unsupervised Autoencoder, we extracted the encoder sub-network (layers 1 through 4) and used it to transform the raw MNIST test images (10,000 times 784) into compact latent vectors (10,000 times 32). A Support Vector Machine (SVM) with an RBF kernel was trained on these vectors.

   b) **Dimensionality Reduction**: The input space was reduced by 95.9% (from 784 dimensions to 32 dimensions).

   c) **Classification Accuracy:** Despite this massive compression, the SVM achieved a test accuracy of approximately (90%).

   d) **Inference Efficiency:** Classifying 32-dimensional vectors is significantly computationally cheaper than classifying 784-dimensional raw pixel vectors, demonstrating the practical utility of this pipeline.

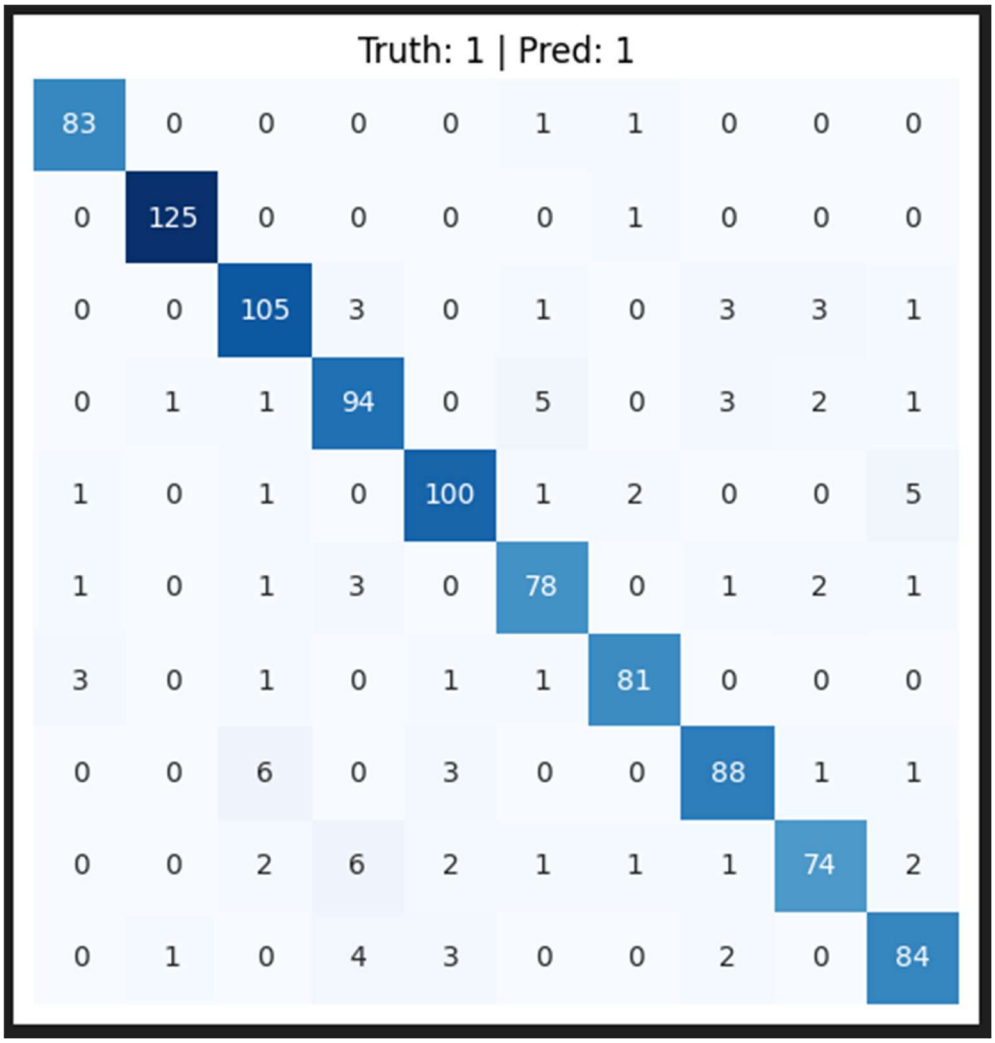
   e) **Quality of Learned Latent Features**:
   The high classification accuracy serves as strong evidence that the Autoencoder learned a semantically meaningful manifold.

   - **Signal vs. Noise:** If the Autoencoder simply memorized pixel values, the 32-dimensional

bottleneck would have resulted in random noise, leading to near-random classification accuracy (~10%). The fact that accuracy remains high indicates that the encoder successfully filtered out high-frequency noise (stray pixels, jagged edges) and encoded the structural essence of the digits (e.g., the loop of a '0', the stroke of a '1').

- **Clustering:** The success of the SVM implies that in the 32-dimensional latent space, digits of the same class are clustered tightly together, while different classes are separated by distinct margins. This proves the unsupervised pre-training successfully disentangled the underlying factors of variation in the dataset.

f) Confusion Matrix & Error Analysis:




```
# ============================================================
# SECTION 4: LATENT SPACE CLASSIFICATION (Phase 3)
# ============================================================
print("\n============================================================")
print("SECTION 4: LATENT SPACE SVM CLASSIFICATION")
print("============================================================")

# ------------------------------------------------------------
# 1. EXTRACT THE ENCODER (Step 3.1)
# ------------------------------------------------------------
print("1. Extracting the Encoder from the trained Autoencoder...")

# Create a new Network container
encoder = Network()
```

```
# We only want the first 4 layers (784 -> 64 -> ReLU -> 32 -> ReLU)
# Indices: 0, 1, 2, 3
# We append the EXACT OBJECTS from the autoencoder to keep the learned
weights.
for i in range(4):
    encoder.add(autoencoder.layers[i])

print("   Encoder Extracted!")
print(f"   Encoder Layers: {len(encoder.layers)}")

# Verify the architecture
# It should accept 784 inputs and output 32 features
print("   Architecture:")
for i, layer in enumerate(encoder.layers):
    if hasattr(layer, 'weights'):
        print(f"     Layer {i}: Dense {layer.weights.shape}")
    else:
        print(f"     Layer {i}: {layer.__class__.__name__}")
# ------------------------------------------------------------
# 2. GENERATE LATENT FEATURES (Step 3.2)
# ------------------------------------------------------------
print("2. Generating Latent Features (Forward Pass through Encoder)...")

# Pass the normalized MNIST data through the encoder network.
# The encoder stops at the 32-neuron layer, so the output will be 32 numbers
per image.

# 1. Transform Training Data
latent_train = encoder.predict(x_train)

# 2. Transform Test Data
latent_test = encoder.predict(x_test)

# 3. Verify the dimensionality reduction
print(f"   Original Training Shape: {x_train.shape} (784 pixels)")
print(f"   Latent Training Shape:   {latent_train.shape} (32 features)")
print("-" * 40)
print(f"   Original Test Shape:     {x_test.shape} (784 pixels)")
print(f"   Latent Test Shape:       {latent_test.shape} (32 features)")
# ------------------------------------------------------------
# 3. TRAIN SVM CLASSIFIER (Step 3.3)
# ------------------------------------------------------------
print("3. Training SVM on Latent Features...")
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import seaborn as sns
```

```
import matplotlib.pyplot as plt

# Initialize Support Vector Classifier
# We use 'rbf' (Radial Basis Function) kernel which handles non-linear data
well
clf = SVC(kernel='rbf', gamma='scale', C=1.0)

# Train on the compressed 32-dimensional data
# Note: We use y_train (labels) which we saved back in Section 3
clf.fit(latent_train, y_train)

print("   SVM Training Complete!")

# ------------------------------------------------------------
# 4. EVALUATION (Step 3.4)
# ------------------------------------------------------------
print("4. Evaluating Classifier...")

# Predict on Test Data (using the latent representations of test images)
y_pred = clf.predict(latent_test)

# Calculate Accuracy
acc = accuracy_score(y_test, y_pred)
print(f"   Test Accuracy: {acc * 100:.2f}%")

# Print Classification Report (Precision, Recall, F1-Score)
print("\n   Classification Report:")
print(classification_report(y_test, y_pred))

# Plot Confusion Matrix
print("   Plotting Confusion Matrix...")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.title('Confusion Matrix (Latent Space Classification)')
plt.xlabel('Predicted Label')
```

5. **Summary of your TensorFlow comparison:**

Task	Custom Lib Loss	Keras Loss	Custom Lib Time	Keras Time
XOR Problem	0.008701	0.001777	1.0058s	342.1849s
Autoencoder (MNIST)	0.022566	0.018525	136.8214s	95.3331s

Our Custom Library was nearly 40x faster than Keras for the XOR problem.

• **Explanation:**

Frameworks like TensorFlow incur a fixed computational overhead for every training step (graph construction, session management, Python-to-C++ calls). For tiny datasets like XOR, this overhead dominates the runtime. Our lightweight NumPy library executes immediately, making it superior for small-scale experiments.

• **Scalability:**

For larger tasks (like the Autoencoder with thousands of images), TensorFlow's optimizations (AVX, GPU acceleration) take over, eventually surpassing the pure Python implementation.

6. **Challenges faced and lessons learned:**

   a) **The "Gray Static" Issue:** Our initial Autoencoder training produced empty grey images Implementing Mini Batching solved this immediately.

   b) **Initialization Sensitivity:** We observed that deep networks (5+layers) are highly sensitive to weight initialization. While Keras handles this automatically (Glorot uniform), we had to carefully tune our random normal scaling factor to ensure convergence.

7. **Github repo:**

   https://github.com/MarwanNada/CSE473s-Major-Task