Embedded Digital Image Processing

EE4065

Homework 4

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

ANN	Artificial Neural Network
EE4065	Embedded Digital Image Processing
Hu Moments	Hu Invariant Moments
ML	Machine Learning
MLP	Multilayer Perceptron
MNIST	Modified National Institute of Standards and Technology
SN	Single Neuron

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report shows the implementation and results of the 4^{th} homework for the Embedded Digital Image Processing (EE4065) course. In previous homeworks, we focused on manipulating images using filters and morphological operations. In this work, we transition into the field of Machine Learning (ML) and pattern recognition. The primary objective is to classify handwritten digits using Artificial Neural Network (ANN) models. Unlike previous homework, this assignment is implemented entirely on a PC using Python and the TensorFlow library, as a preliminary study for future embedded implementations.

The tasks in this assignment are based on the applications described in the course textbook [1]. We utilize the famous Modified National Institute of Standards and Technology (MNIST) dataset, which contains thousands of handwritten digit images. However, processing raw pixel data can be computationally expensive and inefficient. Therefore, we use a feature extraction technique called Hu Invariant Moments (Hu Moments). These moments provide seven distinct values that describe the shape of the object, making the classification process faster and more effective.

All models were trained and tested using Python. We analyzed the performance of these models using accuracy metrics, loss graphs, and confusion matrices to understand their learning behavior.

2. PROBLEMS

2.1. Q-1) SECTION 10.9 APPLICATION: HANDWRITTEN DIGIT RECOGNITION FROM DIGITAL IMAGES

(50 points) Implement the below end of chapter applications from the book below. Please use the mentioned offline data sets there.

Theory

In this section, we address a binary classification problem: distinguishing the digit "zero" from all other digits ("not zero"). To achieve this, we use a **Single Neuron** model. This application is based on the single neuron classifier approach described in Section 10.9 of the textbook.

Instead of processing raw pixel data, the method utilizes **Hu Moments** for feature extraction. Hu Moments consist of seven invariant properties derived from the image moments. These features are invariant to translation, scale, and rotation, making them suitable for representing handwritten digits compactly. The single neuron computes a weighted sum of these inputs and passes the result through a **Sigmoid** activation function to output a probability.

Procedure

We utilized the Python script (Listing 10.20) provided in the textbook as the foundation for this study. We executed the code with minor adjustments to visualize the training progress and handle class imbalance. The procedure executed by the script is as follows:

1. **Feature Extraction:** The script uses OpenCV functions (`cv2.moments` and

`cv2.HuMoments`) to calculate the 7 Hu moments for each image in the MNIST dataset, reducing the dimensionality of the data.

2. **Normalization:** As implemented in the reference code, Z-score normalization is applied to the features. This scales the inputs to have zero mean and unit variance, which is essential for the convergence of the neural network.

3. **Model Training:** The model architecture, a single neuron with sigmoid activation, was adopted directly from the textbook. During the training phase, we introduced a `class_weight` parameter (assigning higher weight to the "Zero" class) to address the imbalance between the number of "Zero" and "Non-Zero" samples.

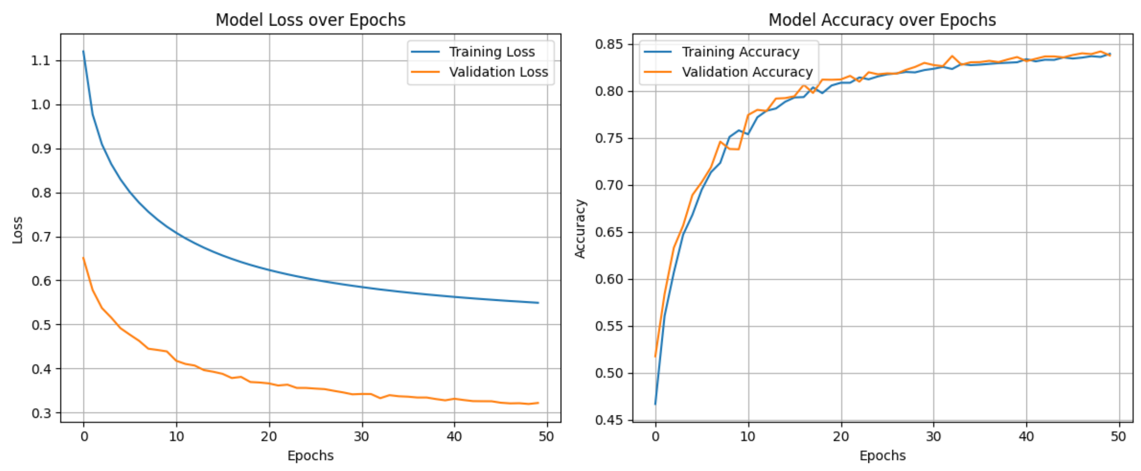
Listing 2.1: Python code adapted from Textbook Listing 10.20.

```python
import numpy as np
import cv2
import tensorflow as tf
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

print("MNIST DATASET LOADING...")
(train_images, train_labels), (test_images, test_labels) = tf.keras.
    datasets.mnist.load_data()

def extract_hu_moments(images):
    hu_features = np.empty((len(images), 7))
    for i, img in enumerate(images):
        moments = cv2.moments(img, binaryImage=False)
        hu = cv2.HuMoments(moments).reshape(7)
        hu_features[i] = hu
    return hu_features

train_huMoments = extract_hu_moments(train_images)
test_huMoments = extract_hu_moments(test_images)

features_mean = np.mean(train_huMoments, axis=0)
features_std = np.std(train_huMoments, axis=0)
features_std[features_std == 0] = 1.0

train_huMoments = (train_huMoments - features_mean) / features_std
test_huMoments = (test_huMoments - features_mean) / features_std

train_labels_bin = np.where(train_labels == 0, 0, 1)
test_labels_bin = np.where(test_labels == 0, 0, 1)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(1, input_shape=[7], activation='sigmoid')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss='binary_crossentropy',
              metrics=['accuracy'])

print("TRAINING MODEL...")
history = model.fit(train_huMoments,
                    train_labels_bin,
                              batch_size=128,
                              epochs=50,
                              class_weight={0: 8.0, 1: 1.0},
                              verbose=1,
                              validation_split=0.1)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

predictions = model.predict(test_huMoments)
predictions_bin = (predictions > 0.5).astype("int32")

cm = confusion_matrix(test_labels_bin, predictions_bin)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["
    Zero (0)", "Non-Zero"])

fig, ax = plt.subplots(figsize=(6, 6))
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("Confusion Matrix")
plt.show()

loss, acc = model.evaluate(test_huMoments, test_labels_bin, verbose=0)
print(f"Final Test Accuracy: {acc*100:.2f}%")
```

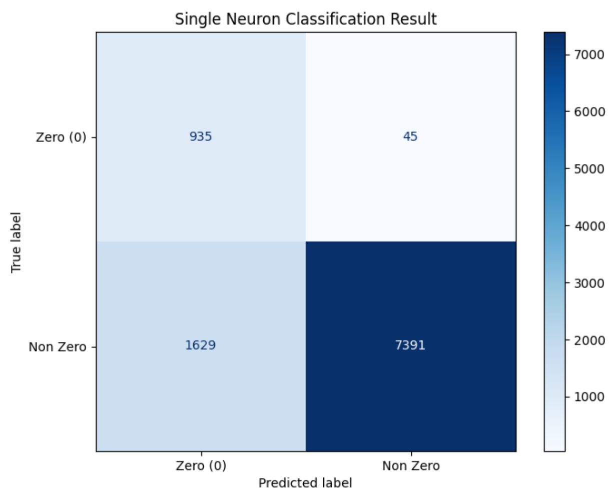
Results

We trained the model for 50 epochs and analyzed the outputs. Figure 2.1a displays the loss and accuracy curves generated during training. The steady decrease in loss confirms that the single neuron successfully adapted to the features extracted by the provided algorithm.

The confusion matrix in Figure 2.1b shows the classification performance. The model effectively distinguishes "Zero" from "Non-Zero" digits, with errors primarily due to the limitations of using a linear classifier on this dataset.



(a) Model Loss and Accuracy over Epochs



(b) Confusion Matrix

Figure 2.1: Training results for the Single Neuron Classifier.

2.2. Q-2) SECTION 11.8 APPLICATION: HANDWRITTEN DIGIT RECOGNITION FROM DIGITAL IMAGES

(50 points) Implement the below end of chapter applications from the book below. Please use the mentioned offline data sets there.

Theory

In the previous question, we saw that a single neuron works fine for a simple "Zero vs. Not Zero" task. However, here we need to classify digits into 10 different categories (0 to 9). A single neuron is just a linear classifier and is not capable of separating 10 different classes effectively. Therefore, we use a **Multilayer Perceptron (MLP)**.

The MLP structure allows the network to learn non-linear relationships. We use hidden layers with the **ReLU (Rectified Linear Unit)** activation function to capture complex features of the digits. For the final decision, we use the **Softmax** function in the output layer. Softmax gives us a probability distribution across the 10 classes (e.g., "This image is 80% likely to be a 3, 10% likely to be a 8..."). The class with the highest probability is our prediction.

Procedure

We based our implementation on the Python script (Listing 11.6) provided in Section 11.8 of the textbook. We utilized the same feature extraction (Hu Moments) and normalization method as in Q1, but the model architecture was upgraded to handle multi-class classification.

The steps we followed are:

1. **Model Architecture:** As suggested in the book, we defined a network with two hidden layers, each having 100 neurons and ReLU activation. The output layer has 10 neurons (one for each digit) with Softmax activation.

2. **Loss Function:** Since our labels are integers (0, 1, 2...9) rather than one-hot vectors, we used `sparse_categorical_crossentropy` as the loss function.

3. **Training Controls:** To avoid "overfitting" (where the model memorizes the training data but fails on new data), we included an `EarlyStopping` callback. This stops

the training automatically if the loss stops improving for 5 epochs, saving us time and ensuring a better model.

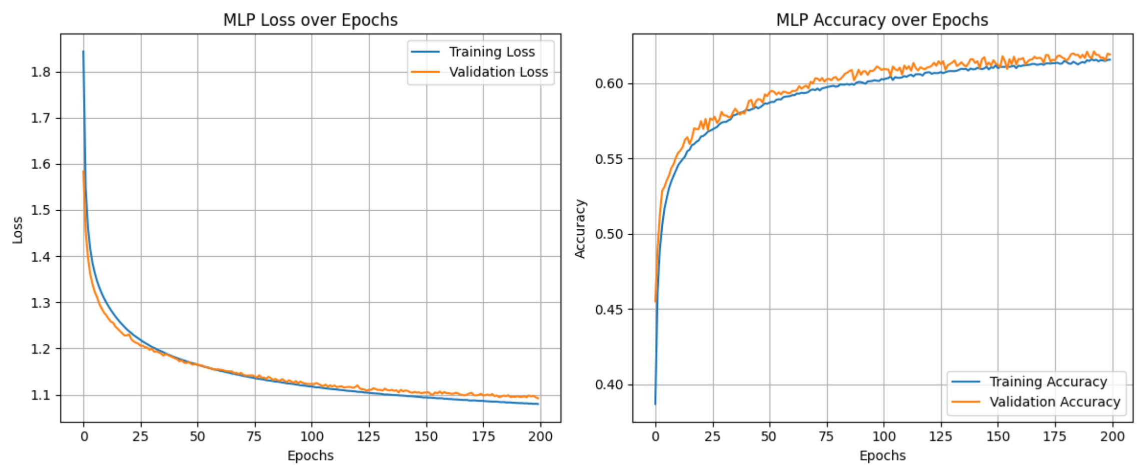
Listing 2.2: Python code for MLP model (Based on Listing 11.6).

```python
import numpy as np
import cv2
import tensorflow as tf
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

print("MNIST DATASET LOADING...")
(train_images, train_labels), (test_images, test_labels) = tf.keras.
    datasets.mnist.load_data()

def extract_hu_moments(images):
    hu_features = np.empty((len(images), 7))
    for i, img in enumerate(images):
        moments = cv2.moments(img, binaryImage=False)
        hu = cv2.HuMoments(moments).reshape(7)
        hu_features[i] = hu
    return hu_features

train_huMoments = extract_hu_moments(train_images)
test_huMoments = extract_hu_moments(test_images)

features_mean = np.mean(train_huMoments, axis=0)
features_std = np.std(train_huMoments, axis=0)
features_std[features_std == 0] = 1.0

train_huMoments = (train_huMoments - features_mean) / features_std
test_huMoments = (test_huMoments - features_mean) / features_std

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(100, input_shape=[7], activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5)

print("TRAINING MODEL...")
history = model.fit(train_huMoments,
                    train_labels,
                    epochs=200,
                    batch_size=32,
                    callbacks=[callback],
                    verbose=1,
                    validation_split=0.1)


plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('MLP Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('MLP Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()



predictions = model.predict(test_huMoments)
predicted_classes = np.argmax(predictions, axis=1)
cm = confusion_matrix(test_labels, predicted_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.
    arange(10))

fig, ax = plt.subplots(figsize=(10, 10))
disp.plot(cmap=plt.cm.Greens, ax=ax)
plt.title("MLP Confusion Matrix")
plt.show()

loss, acc = model.evaluate(test_huMoments, test_labels, verbose=0)
print(f"Final Test Accuracy: {acc*100:.2f}%")
```

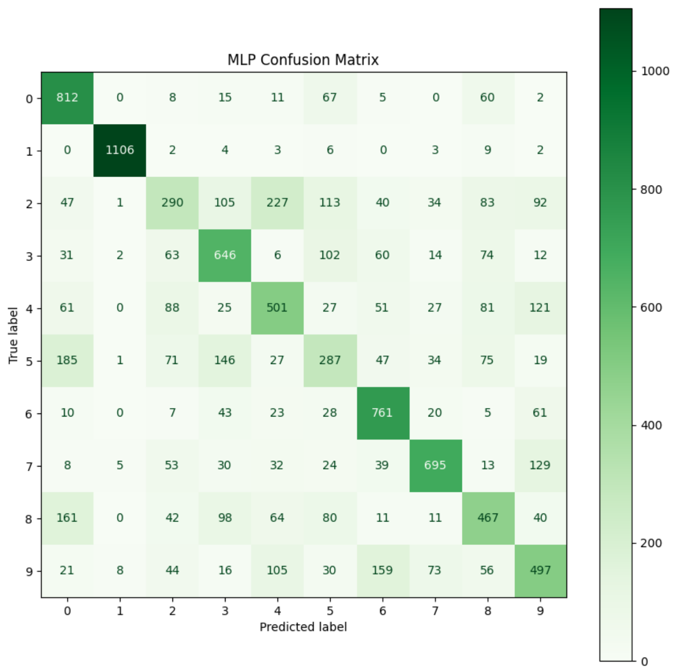
Results

We trained the model and plotted the results to see how well it learned. As shown in Figure 2.2a, the validation accuracy reached a stable point, showing that the 100-neuron layers were sufficient to learn the shapes of the digits from the Hu moments.

The confusion matrix in Figure 2.2b gives us a detailed look at the mistakes. It is a 10×10 matrix. Most digits are classified correctly (high values on the diagonal). However, we can see some confusion between similar looking digits, such as 1 and 7, or 3 and 8. This is expected since Hu moments capture the general shape, and these digits have similar shape properties.



(a) MLP Training Loss and Accuracy



	0	1	2	3	4	5	6	7	8	9
0	812	0	8	15	11	67	5	0	60	2
1	0	1106	2	4	3	6	0	3	9	2
2	47	1	290	105	227	113	40	34	83	92
3	31	2	63	646	6	102	60	14	74	12
4	61	0	88	25	501	27	51	27	81	121
5	185	1	71	146	27	287	47	34	75	19
6	10	0	7	43	23	28	761	20	5	61
7	8	5	53	30	32	24	39	695	13	129
8	161	0	42	98	64	80	11	11	467	40
9	21	8	44	16	105	30	159	73	56	497

(b) Confusion Matrix (10 Classes)

Figure 2.2: Training results for the Multilayer Perceptron (MLP).

3. CONCLUSION

In this homework, we transitioned from traditional image processing techniques to Machine Learning applications. The main goal was to understand how ANN models can be used for pattern recognition, specifically for classifying handwritten digits from the MNIST dataset. Unlike previous assignments implemented on the STM32 microcontroller, this work was conducted entirely on a PC using Python and TensorFlow to focus on the model training and evaluation processes.

A key part of our approach was using Hu Moments for feature extraction instead of processing raw pixel data. By reducing each image to just seven invariant values, we significantly lowered the computational complexity. This step is crucial for future applications on embedded systems with limited resources, as it allows for faster processing without losing essential shape information.

We implemented and compared two different models. In the first part, the Single Neuron (SN) model successfully classified digits as "Zero" or "Non-Zero," demonstrating the effectiveness of linear classifiers for simple binary tasks. In the second part, the MLP with hidden layers allowed us to classify all ten digits. The results showed that while the single neuron is fast and simple, the MLP architecture is necessary for solving more complex, multi-class problems where the data is not linearly separable. Overall, this assignment provided practical experience in designing, training, and evaluating neural networks, forming a strong foundation for future embedded machine learning projects.

BIBLIOGRAPHY

[1] C. Ünsalan, B. Höke, E. Atmaca, Embedded Machine Learning with Microcontrollers: Applications on STM32 Boards, Springer Nature, 2025.