School of Mechanical and Mechatronics Engineering
National University of Science and Technology

Machine Learning

RAI-832

Assignment 2

Submitted by:

Asfa Tahir 497470

Submitted to:

Dr. Muhammad Jawad Khan

Problem Statement

Write complete code to train a shallow neural network to detect a face. Use your own images dataset. Split dataset such that 60% for training, 20% for validation and 20% for testing. Libraries for machine learning can NOT be used. For example gradient descent has to be implemented yourself rather than using function from any library. Libraries for peripheral functions such as plotting, reading/writing of files etc can be used such as numpy, pillows.

Dataset Details

Two sets of images were collected, in .jpg format.

- Kim Jong Un images: 50 images.
- Not Kim Jong Un images: 50 images.

Images were converted to grayscale and resized to 64×64 pixels to maintain consistency. These images are also rescaled from 0-255 pixel intensity to [0, 1] pixel intensity. And then flattened from 64×64 size to 4096×1 size. Each image is labelled as

- 0 for Not Kim Jong Un and
- 1 for Kim Jong Un

Data splitting: The dataset is split in a stratified manner to maintain class balance across training, validation, and test sets. First, the indices of each class are identified and shuffled separately. Then, each class is divided so that 60% of the images are used for training, 20% for validation, and the remaining 20% for testing. Finally, the corresponding indices from both classes are combined and shuffled again to form the final splits, maintaining class balance while ensuring randomness within each set.

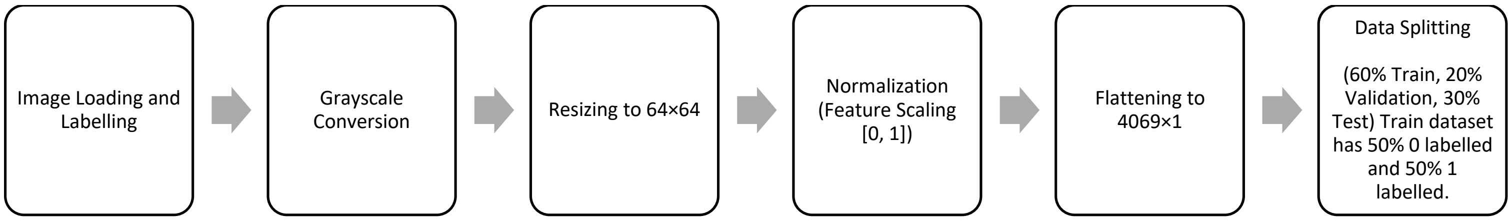


Image Loading and Labelling
Grayscale Conversion
Resizing to 64×64
Normalization (Feature Scaling [0, 1])
Flattening to 4069×1
Data Splitting
(60% Train, 20% Validation, 30% Test) Train dataset has 50% 0 labelled and 50% 1 labelled.

Figure 1 Image Loading, Preprocessing and Data splitting

Code and Methodology:

The dataset was loaded using PIL for image handling and numpy for array operations. The data was split into training, validation, and test sets in a stratified manner to maintain class balance. A shallow feedforward neural network with one hidden layer was implemented from scratch using only numpy. No external deep learning libraries were used.

The network structure:

- **Input layer**: **4096** neurons (flattened image)
- **Hidden layer**: **128** neurons with **sigmoid activation**
- **Output layer**: **1** neuron with **sigmoid activation** (binary classification)

Parameter Initialization: Weights W1 and W2 are initialized using random normal distribution, but because a fixed random seed was set, the initialization became deterministic. This ensures reproducibility, so every run produces the same ‘random’ weights. And biases b1 and b2 are initialized as a zero vector, where input_dim × hidden_dim is the shape of W1, 1 × hidden_dim is the shape of b1, hidden_dim × 1 is the shape of W2 and 1 × 1 is the shape of b2.

Forward Propagation: It computes the ouput of neural network. It is performed using the equations shown in the next section named as Mathematical Modelling, by putting the values of initialized weights and input images. It will provide a resultant value to the output neuron and

1. Hidden Layer
    - $z_1 = XW_1 + b_1$
    - $a_1 = \sigma(z_1)$
2. Output Layer
    - $z_2 = a_1W_2 + b_2$
    - $\hat{y} = \sigma(z_2)$

Where σ is the sigmoid activation function. The output $\hat{y}$ is a probability in [0, 1].
Decision rule:

- If output value ≥ 0.5, then label is 1 i.e. Kim Jong Un.
- If prediction value < 0.5, then label is 0 i.e. Not Kim Jong Un

Loss Computation: Binary Cross Entropy Computation is being computed in order to calculate how much change is required in the weights. The equation/ formula for BCE is shown in next section of Mathematical Modelling.

Hyperparameters Introduction: The following hyperparameters were selected based on initial experimentation:

- Hidden Layer Size =128
- Learning Rate (LR) = 0.1
- Regularization Strength (λ) = 0.001
- Epochs = 1500
- Batch Size= 16

These parameters significantly influence training stability and convergence. The learning rate controls step size during optimization, λ prevents overfitting via L2 regularization, and batch size affects gradient estimation.

Back Propagation: Backpropagation computes gradients of the loss with respect to weights and biases:

1. Compute gradient at output layer:

$$dz_2 = \frac{(\hat{y} - y)}{m}$$

2. Gradients:
    - $dW_2 = {a_1}^T dz_2 + \lambda W_2$
    - $db_2 = \sum dz_2$

3. Hidden layer:

- $da_1 = dz_2 {W_2}^T$
- $dz_1 = da_1 \cdot \sigma'(a_1)$ where $\sigma'(x) = \sigma(x) \times (1 - \sigma(x))$
- $dW_1 = X^T dz_1 + \lambda W_1$
- $db_1 = \sum dz_1$

L2 regularization reduces overfitting by penalizing large weights.

Weight Updating (Gradient Descent): Parameters are updated as:

$$W = W - \alpha \cdot dW$$
$$b = b - \alpha \cdot db$$

Where

- α = learning rate
- dW= computed gradient

Updates are applied to W1, b1, W2, and b2 after each batch.

After every weight updating, forward propagation is performed, loss is calculated till the epoch reaches 1500 times.

Error and Accuracy Calculation:

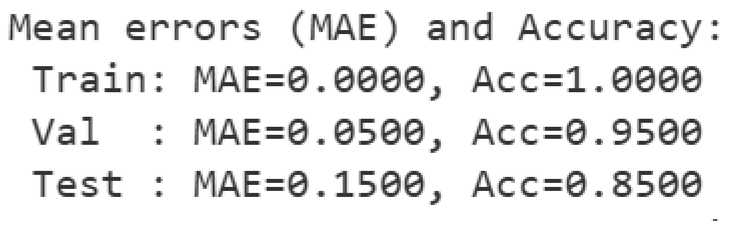
Mean Absolute Error is also calculated. It gives another interpretation of prediction errors:

$$MAE = \frac{1}{m} \sum | y - \hat{y}_{label} |$$

Lower MAE indicates better classification performance. Accuracy is calculated through given formula, the higher it is the better the model performs.

$$Accuracy = \frac{correct\ predictions}{total\ samples}$$

Figure 2 shows the MAE and accuracy for all three datasets i.e. train, validation and test at the end using the saved model.



```
Mean errors (MAE) and Accuracy:
 Train: MAE=0.0000, Acc=1.0000
 Val  : MAE=0.0500, Acc=0.9500
 Test : MAE=0.1500, Acc=0.8500
```

Figure 2 Computed MAE and Accuracy

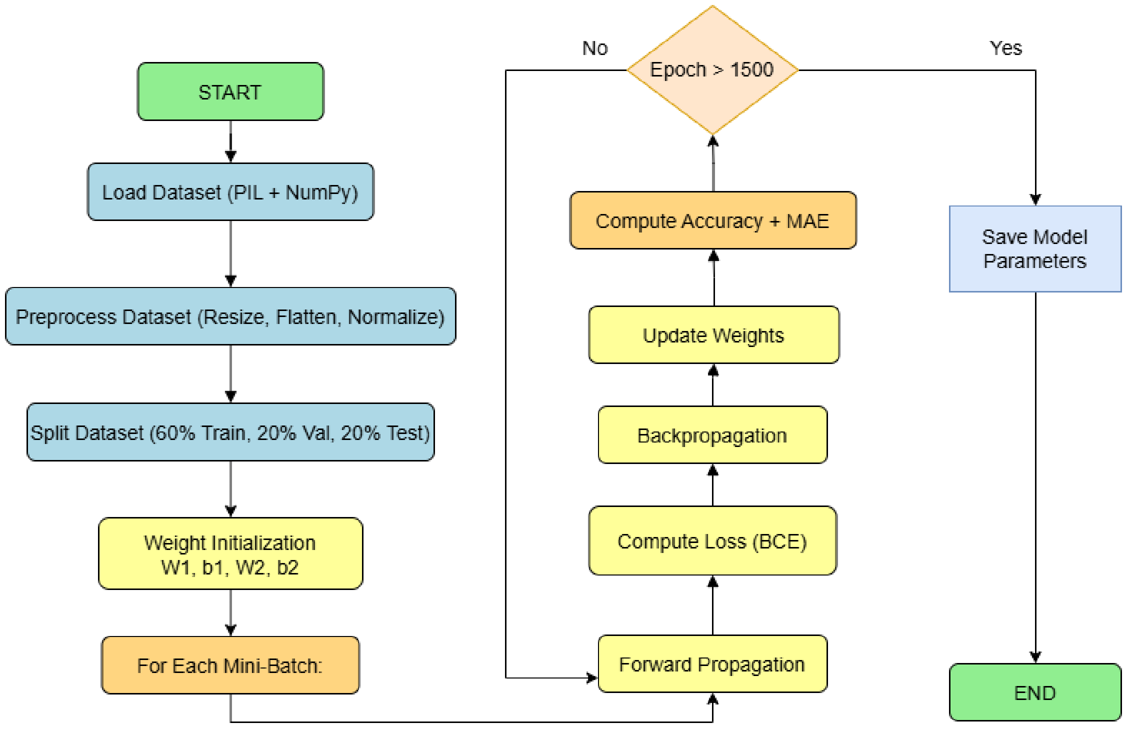


Figure 3 Model Training

- <u>Mathematical Modelling:</u>

The problem is formulated as a binary classification task.

- **Hypothesis**

For Hidden Layer: The model computes:

$$hidden\_layer\_input = X \cdot W1 + b1$$
$$hidden\_layer\_output = sigmoid(hidden\_layer\_input)$$

where X is the vector of size 4096×1 of the image from input layer W1 and b1 are the vectors of weights and bias of size 4096×128 and 1×128.

For Output Layer: The model computes:

$$output\_layer\_input = (hidden\_layer\_output) \cdot W2 + b2$$
$$prediction = sigmoid(output\_layer\_input)$$

Where W2 and b2 are the vectors of weights and bias of size 128×1 and 1×1. The final number i.e. prediction is always between 0 and 1.

- If prediction value ≥ 0.5, then label is 1 i.e. Kim Jong Un.
- If prediction value < 0.5, then label is 0 i.e. Not Kim Jong Un.

The sigmoid function compresses values into this 0–1 range.

- **Objective**

Binary Cross-Entropy (BCE) loss is used as objective function to be minimized.

$$L(y, \hat{y}) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Where L is the loss, m is the number of samples in the batch or dataset, y is the true label and $\hat{y}$ is the predicted label of the sample i. This has been verified through loss vs epoch plot as shown in figure 4, where at the end of the graph the line becomes smoothly horizontally straight showing no further minimization of the loss.

- **Parameter Optimization:**
  The best model with optimum values of weights and biases is saved based on minimum loss achievement with convergence in npz format.
- **Hyperparameter Optimization:**
  Multiple experiments were conducted by varying the learning rate (0.01, 0.05, 0.1), hidden units (64, 128, 256), regularization λ values (0, 0.0001, 0.001), and batch sizes (8, 16, 32). The architecture with 128 hidden neurons, learning rate 0.1, λ = 0.001, and batch size 16 resulted in the lowest validation loss and best generalization performance. The optimization was done manually by tracking validation loss curves.

➢ <u>Plots</u>

Figure 4 is a Loss vs Epoch and accuracy vs Epoch plot showing a significant decrease and increase respectively and also show convergence at the end.

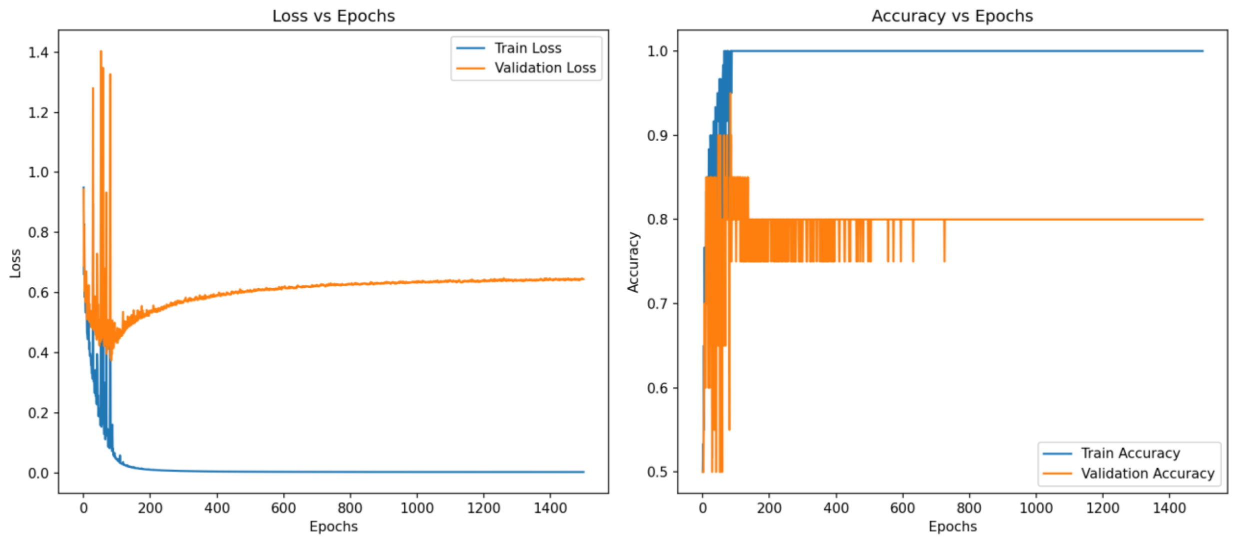


Figure 4 Loss and Accuracy vs Epoch

Figure 5 shows the confusion matrix showing that the model exhibits perfect performance on the Train set (100% accuracy). Performance drops slightly on the Validation set (95% accuracy, 1 False Negative). On the unseen Test set, the model shows its 85% accuracy, with 3 out of 10 True 1 cases incorrectly classified as 0 (3 FNs), demonstrating a weakness in identifying class 1.

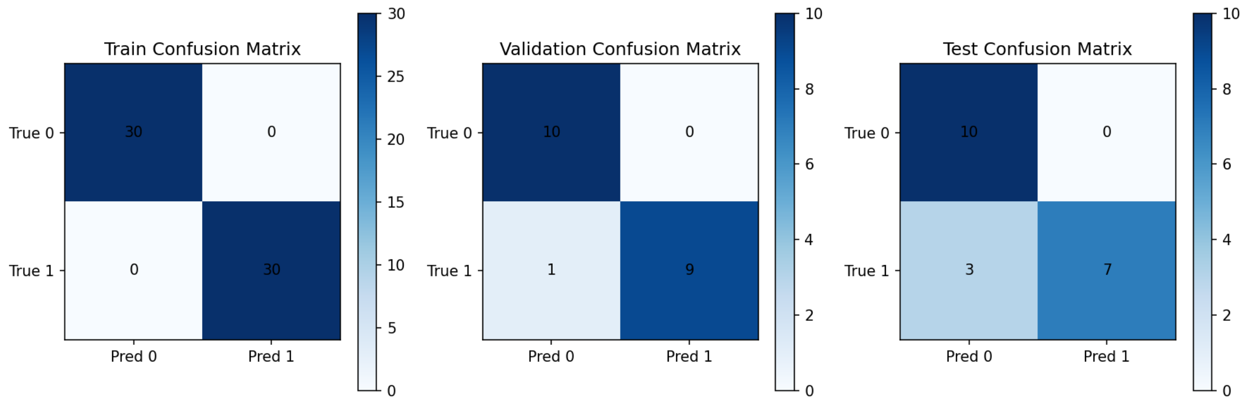


Figure 5 Confusion Matrix for Train, Validation and Test Set

Annex A : Instruction to run the code

1. Ensure Python 3.x installed with packages: numpy, pillow, matplotlib.
2. Open folder named as Face detection in Visual Studio Code, which will have train.py and predict.py along with another folder Dataset that contains face images of Kim Jong Un in one folder and other images in another folder.
3. Run training train.py. It will print epoch logs and show plots as well as train, validation and testing errors.
4. After completion, model.npz is saved. In order to read or find weights values run modelread.py.
5. Run predictions with predict.py, this will import and preprocess an image saved in the given folder, and as a result predict its label as 1 for Kim Jong Un and 0 for not Kim Jong Un.

Annex B: Training Code and Parameters

Training Code (train.py):

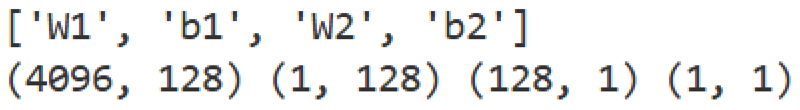
```
# train.py
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

kim_images = [f"Dataset/KimJongUn/K{i}.jpg" for i in range(1, 51)]
not_kim_images = [f"Dataset/NotKimJongUn/NK{i}.jpg" for i in range(1, 51)]

def load_dataset(image_size=(64, 64)):
    X = []
    y = []

    # label 1 = Kim Jong Un
    for path in kim_images:
```

```python
        img = Image.open(path).convert("L").resize(image_size)
        arr = np.asarray(img, dtype=np.float32)/255.0
        X.append(arr.flatten())
        y.append(1)

    # label 0 = Not Kim Jong Un
    for path in not_kim_images:
        img = Image.open(path).convert("L").resize(image_size)
        arr = np.asarray(img, dtype=np.float32)/255.0
        X.append(arr.flatten())
        y.append(0)

    X = np.array(X, dtype=np.float32)
    y = np.array(y, dtype=np.int32)
    return X, y

def train_val_test_split(X, y, seed=2):
    np.random.seed(seed)
    # get indices of each class
    idx_kim = np.where(y == 1)[0]
    idx_not = np.where(y == 0)[0]
    # shuffle each class separately
    np.random.shuffle(idx_kim)
    np.random.shuffle(idx_not)
    # helper function for splitting
    def split_indices(indices):
        n = len(indices)
        n_train = int(0.6 * n)
        n_val = int(0.2 * n)
        return (
            indices[:n_train],indices[n_train:n_train+n_val],indices[n_train+n_val:])

    # stratified splits
    kim_train, kim_val, kim_test = split_indices(idx_kim)
    not_train, not_val, not_test = split_indices(idx_not)

    # combine
    train_idx = np.concatenate([kim_train, not_train])
    val_idx   = np.concatenate([kim_val, not_val])
    test_idx  = np.concatenate([kim_test, not_test])

    # shuffle inside each split
    np.random.shuffle(train_idx)
    np.random.shuffle(val_idx)
    np.random.shuffle(test_idx)
    return (X[train_idx], y[train_idx],X[val_idx], y[val_idx],X[test_idx], y[test_idx])
def plot_all_confusion_matrices(train_cm, val_cm, test_cm):
    cms = [train_cm, val_cm, test_cm]
    titles = ["Train", "Validation", "Test"]

    fig, axes = plt.subplots(1, 3, figsize=(12,4))

    for ax, cm, title in zip(axes, cms, titles):
        im = ax.imshow(cm, interpolation='nearest', cmap='Blues')
        ax.set_title(f"{title} Confusion Matrix")
        ax.set_xticks([0,1])
        ax.set_yticks([0,1])
        ax.set_xticklabels(["Pred 0", "Pred 1"])
        ax.set_yticklabels(["True 0", "True 1"])

        # annotate values
        for i in range(2):
            for j in range(2):
                ax.text(j, i, str(cm[i, j]), ha="center", va="center", color="black")

        fig.colorbar(im, ax=ax)   # <-- IMPORTANT: colorbar per subplot

    plt.tight_layout()
    plt.show()
SEED = 2
np.random.seed(SEED)

# ---------- hyperparameters ----------
IMAGE_SIZE = (64, 64)
INPUT_DIM = IMAGE_SIZE[0] * IMAGE_SIZE[1]  # 4096
HIDDEN_DIM = 128
LR = 0.1
LAMBDA = 0.001
EPOCHS = 1500
BATCH_SIZE = 16
MODEL_PATH = "model.npz"
# -------------------------------------
# activation function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_deriv(a):
    return a * (1 - a)

# loss function
def binary_cross_entropy(y_true, y_pred, eps=1e-12):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

# metrics
def predict_labels(X, params):
    W1, b1, W2, b2 = params
    z1 = X.dot(W1) + b1
    a1 = sigmoid(z1)
    z2 = a1.dot(W2) + b2
    y_hat = sigmoid(z2).reshape(-1)
    labels = (y_hat >= 0.5).astype(np.int32)
    return labels, y_hat

def mean_absolute_error(y_true, y_pred_labels):
    return np.mean(np.abs(y_true - y_pred_labels))

def accuracy(y_true, y_pred_labels):
    return np.mean(y_true == y_pred_labels)

def confusion_matrix(y_true, y_pred_labels):
    tp = np.sum((y_true == 1) & (y_pred_labels == 1))
    tn = np.sum((y_true == 0) & (y_pred_labels == 0))
    fp = np.sum((y_true == 0) & (y_pred_labels == 1))
    fn = np.sum((y_true == 1) & (y_pred_labels == 0))
    return np.array([[tn, fp], [fn, tp]])

# ---------- load data ----------
X, y = load_dataset(image_size=IMAGE_SIZE)
print("Loaded dataset:", X.shape, y.shape)

X_train, y_train, X_val, y_val, X_test, y_test = train_val_test_split(X, y, seed=SEED)
print("Split sizes -> train:", len(X_train), "val:", len(X_val), "test:", len(X_test))

# ---------- initialize parameters ----------
def init_params(input_dim, hidden_dim):
    W1 = np.random.randn(input_dim, hidden_dim) * np.sqrt(2.0 / input_dim)
    b1 = np.zeros((1, hidden_dim), dtype=np.float32)
    W2 = np.random.randn(hidden_dim, 1) * np.sqrt(2.0 / hidden_dim)
    b2 = np.zeros((1, 1), dtype=np.float32)
    return W1.astype(np.float32), b1.astype(np.float32), W2.astype(np.float32),
b2.astype(np.float32)

W1, b1, W2, b2 = init_params(INPUT_DIM, HIDDEN_DIM)

# ---------- training loop ----------
best_val_loss = np.inf
```

```
best_params = None

train_loss_history = []
val_loss_history = []
train_acc_history = []
val_acc_history = []

n_train = X_train.shape[0]
for epoch in range(1, EPOCHS + 1):
    perm = np.random.permutation(n_train)
    X_train_shuf = X_train[perm]
    y_train_shuf = y_train[perm].reshape(-1, 1)

    for i in range(0, n_train, BATCH_SIZE):
        X_batch = X_train_shuf[i:i+BATCH_SIZE]
        y_batch = y_train_shuf[i:i+BATCH_SIZE]
        m = X_batch.shape[0]
        # forward
        z1 = X_batch.dot(W1) + b1
        a1 = sigmoid(z1)
        z2 = a1.dot(W2) + b2
        y_hat = sigmoid(z2)
        # backward
        dz2 = (y_hat - y_batch) / m
        dW2 = a1.T.dot(dz2)
        db2 = np.sum(dz2, axis=0, keepdims=True)

        da1 = dz2.dot(W2.T)
        dz1 = da1 * sigmoid_deriv(a1)
        dW1 = X_batch.T.dot(dz1)
        db1 = np.sum(dz1, axis=0, keepdims=True)
        # L2 regularization
        dW2 += LAMBDA * W2
        dW1 += LAMBDA * W1
        # gradient descent
        W2 -= LR * dW2
        b2 -= LR * db2
        W1 -= LR * dW1
        b1 -= LR * db1

    # --- end epoch: compute losses ---
    _, train_probs = predict_labels(X_train, (W1, b1, W2, b2))
    train_loss = binary_cross_entropy(y_train, train_probs)
    _, val_probs = predict_labels(X_val, (W1, b1, W2, b2))
    val_loss = binary_cross_entropy(y_val, val_probs)
    train_loss_history.append(train_loss)
    val_loss_history.append(val_loss)
    train_acc_history.append(accuracy(y_train, (train_probs >= 0.5).astype(int)))
    val_acc_history.append(accuracy(y_val, (val_probs >= 0.5).astype(int)))

    if epoch % 50 == 0 or epoch == 1:
        print(f"Epoch
{epoch:04d}  train_loss={train_loss:.4f}  val_loss={val_loss:.4f}")

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_params = (W1.copy(), b1.copy(), W2.copy(), b2.copy())

# ---------- save best model ----------
if best_params is None:
    best_params = (W1, b1, W2, b2)

np.savez(MODEL_PATH,
         W1=best_params[0], b1=best_params[1],
         W2=best_params[2], b2=best_params[3])
print("Saved best model to", MODEL_PATH)

# ---------- evaluation ----------
def evaluate_set(Xs, ys, params, name="set"):
    labels, probs = predict_labels(Xs, params)
    mae = mean_absolute_error(ys, labels)
    acc = accuracy(ys, labels)
    cm = confusion_matrix(ys, labels)
    loss = binary_cross_entropy(ys, probs)
    return {"loss": loss, "acc": acc, "mae": mae, "cm": cm}
print("\n--- Evaluation of BEST model on all splits ---")
train_metrics = evaluate_set(X_train, y_train, best_params, name="Train")
val_metrics   = evaluate_set(X_val, y_val, best_params, name="Validation")
test_metrics  = evaluate_set(X_test, y_test, best_params, name="Test")

# --- Plot confusion matrices ---
plot_all_confusion_matrices(train_metrics["cm"], val_metrics["cm"], test_metrics["cm"])

print("\nMean errors (MAE) and Accuracy:")
print(f" Train: MAE={train_metrics['mae']:.4f}, Acc={train_metrics['acc']:.4f}")
print(f" Val  : MAE={val_metrics['mae']:.4f}, Acc={val_metrics['acc']:.4f}")
print(f" Test : MAE={test_metrics['mae']:.4f}, Acc={test_metrics['acc']:.4f}")

# ---------- plot training curves ----------
fig, axes = plt.subplots(1, 2, figsize=(12,4))
```

```
# --- Loss subplot ---
axes[0].plot(train_loss_history, label="Train Loss")
axes[0].plot(val_loss_history, label="Validation Loss")
axes[0].set_title("Loss vs Epochs")
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Loss")
axes[0].legend()

# --- Accuracy subplot ---
axes[1].plot(train_acc_history, label="Train Accuracy")
axes[1].plot(val_acc_history, label="Validation Accuracy")
axes[1].set_title("Accuracy vs Epochs")
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Accuracy")
axes[1].legend()

plt.tight_layout()
plt.show()
```

Parameters: File modelread.py shows the size of all weights as well as the weights. Figure 6 shows the weights size shown after running the file.



```
['W1', 'b1', 'W2', 'b2']
(4096, 128) (1, 128) (128, 1) (1, 1)
```

Figure 6 Weights Size

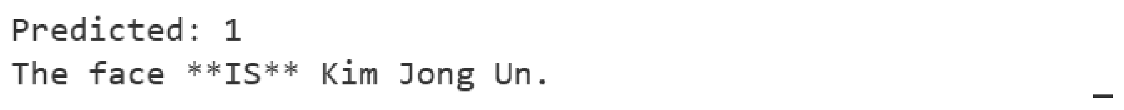
The b2 values is proposed to be [0.0601674] as it is 1*1 matrix.

Annex C: Prediction Code

```
# predict.py
import numpy as np

MODEL_PATH = "model.npz"

def load_model(path=MODEL_PATH):
    data = np.load(path)
    W1 = data["W1"]
    b1 = data["b1"]
    W2 = data["W2"]
    b2 = data["b2"]
    return W1, b1, W2, b2

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
```

```python
# The required function name: prediction
# Input: features -> 1D numpy array of length 4096 (already scaled between 0..1)
# Output: estimated count (we return predicted class 0/1). We also return probability.
def prediction(features):
    """
    features: 1D numpy array shape (4096,) values scaled 0..1
    Returns: integer 0 or 1 (predicted class)
    """
    W1, b1, W2, b2 = load_model(MODEL_PATH)
    x = features.reshape(1, -1)  # (1, 4096)
    z1 = x.dot(W1) + b1
    a1 = sigmoid(z1)
    z2 = a1.dot(W2) + b2
    prob = sigmoid(z2)[0, 0]
    label = int(prob >= 0.5)
    return label  # "estimated count" as requested

# helper: predict from image file (optional convenience)
def predict_from_image(path, image_size=(64,64)):
    from PIL import Image
    import numpy as _np
    img = Image.open(path).convert("L").resize(image_size)
    arr = _np.asarray(img, dtype=_np.float32) / 255.0
    feats = arr.flatten()
    return prediction(feats)


# if you have an image file
label = predict_from_image("KimJongUn.jpg")
print("Predicted:", label)
```

This code will preprocess the image and predicts is label using the saved model in model.npz. In the given code a random image saved in the Face Detection Folder is uploaded and the model predicts its label to be 1 which is a true label as shown in image.



```
Predicted: 1
The face **IS** Kim Jong Un.
```

Figure 7 Model Prediction on Uploaded Image.