Multi-Biometric Iris Recognition System Based on a Deep Learning Approach

Course Name: SIL775: Biometric Security

Assignment 2

Submitted by:

Name: Anand Sharma
Entry No: 2024JCS2049

1 Iris Localization Implementation

Iris localization is the process of detecting and identifying the iris and its surrounding structures (such as the pupil and eyelids) in an image. This step is crucial in iris recognition systems because the quality of localization directly affects recognition accuracy. The goal is to accurately extract the iris region (including the pupil and iris boundaries) while excluding extraneous regions (such as the sclera and parts of the eyelids).

1.1 Raw and Localized Images

Figures 1 and 2 show examples of raw iris images (1 and 2) and their corresponding localized versions (3 and 4), respectively.

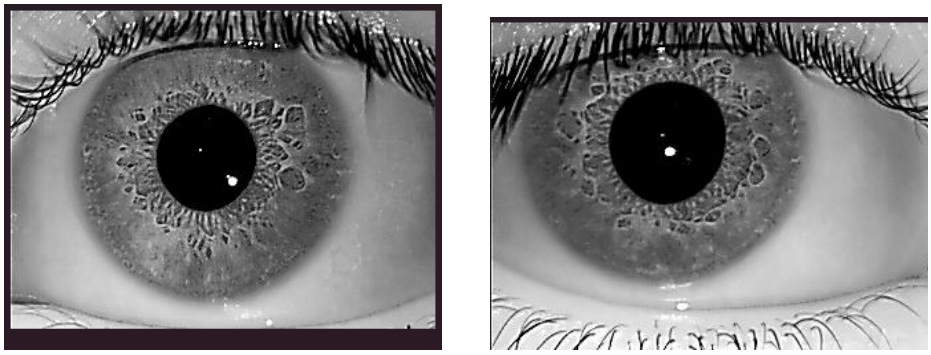

Figure 1: (Left) Raw iris image 1; (Right) Raw iris image 2.

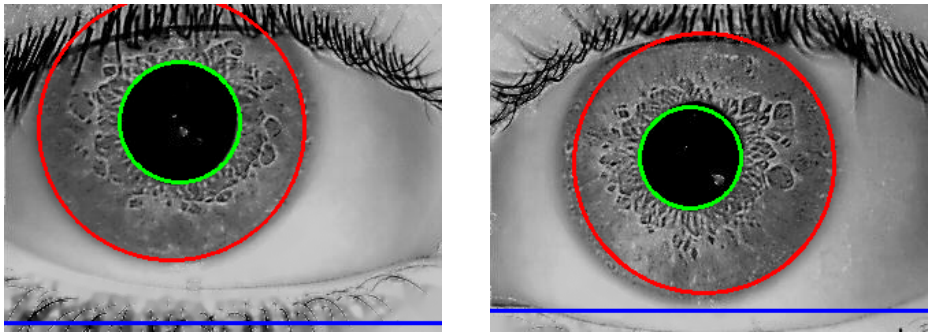

Figure 2: (Left) Localized iris image (pupil in green, iris boundary in red); (Right) Another localized iris image.

The localization process involves several steps:

- **Reflection Removal:** Specular reflections are removed so that the pupil and iris boundaries can be detected without interference.
- **Pupil Detection:** The pupil is detected as the darkest circular region within the eye.
- **Iris Detection:** The iris is detected as the circular region surrounding the pupil using image enhancement techniques and edge detection.

- **Eyelid Detection:** The upper and lower eyelid boundaries are detected using edge detection and the Radon transform to help define the vertical limits of the iris.

These functions work together in a pipeline:

1. `remove_reflections(img)`
   Why needed: Removes specular reflections that can obscure important boundaries.

   Working (elaborated): The image is first converted to grayscale. A binary threshold is applied to detect bright regions (reflections), and then OpenCV's inpainting is used to fill these areas based on surrounding pixel values. This produces a cleaner image that enhances the subsequent detection of the pupil.

   Relation: The output is passed to `detect_pupil`.

2. `detect_pupil(img, debug=False)`
   Why needed: Accurately locating the pupil is critical as it defines the inner boundary of the iris.

   Working (elaborated): The function converts the image to grayscale and applies inverse thresholding to highlight the dark pupil region. Morphological operations refine the candidate regions, and the Circular Hough Transform is used to detect the circular shape that best fits the pupil. The resulting pupil center and radius are returned.

   Relation: These parameters are used by `detect_iris`.

3. `detect_iris(img, pupil_center, pupil_radius, debug=False)`
   Why needed: With the pupil located, the outer boundary of the iris must be determined to isolate the iris region for normalization.

   Working (elaborated): The function enhances the image using Gaussian blur and CLAHE to boost contrast, then applies Canny edge detection to delineate boundaries. A mask based on the pupil's position restricts the search area, and the Circular Hough Transform is applied to detect the iris boundary. The function outputs the iris center and radius.

   Relation: The iris information is further used in eyelid detection and normalization.

4. `detect_eyelids(img, iris_center, iris_radius, niter=10, kappa=50, gamma=0.1)`
   Why needed: Determines the vertical limits of the iris by detecting the eyelids, which is vital for accurate cropping.

   Working (elaborated): The image is converted to grayscale and smoothed using anisotropic diffusion to reduce noise while preserving edges. The Radon transform is then applied to the upper and lower regions of the iris to detect significant intensity changes corresponding to eyelid boundaries. The detected y-coordinates are returned.

   Relation: These boundaries are used by `localize_iris` to annotate the image.

The `localize_iris(img)` function integrates these steps by sequentially calling:

- `remove_reflections` to clean the image,
- `detect_pupil` to find the pupil,

- `detect_iris` to locate the iris boundary,
- `detect_eyelids` to determine the eyelid limits.

It returns an annotated image and a dictionary of detected features.

2 Iris Normalization Implementation

Figures 3 shows examples of normalized iris images (5 and 6). These appear as rectangular strips, each representing the unwrapped iris region.

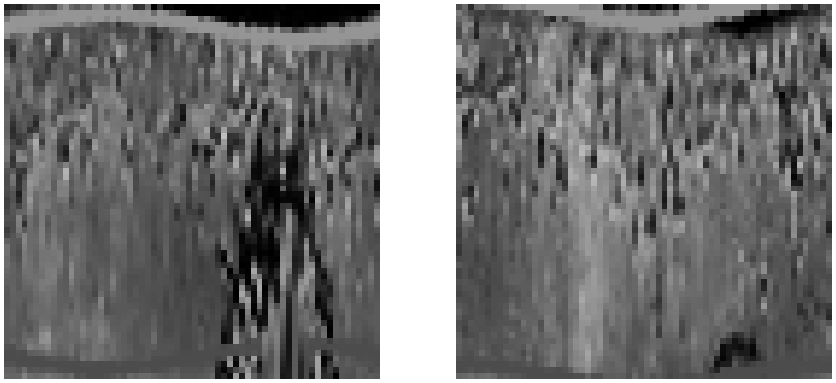

Figure 3: Examples of normalized iris images (rectangular strips).

Iris normalization transforms the circular iris region into a fixed-size rectangular image (commonly 64x512 or 64x64) for consistent feature extraction and easier comparison.

The normalization process involves:

1. **Localization Pre-requisite:** The pupil and iris centers, along with their radii (obtained from `localize_iris`), define the region of interest.
2. **Daugman's Rubber-Sheet Mapping:** Converts the iris region from Cartesian to polar coordinates, effectively unwrapping the circular iris into a rectangular format.
3. **Resizing:** The normalized image is resized (using bilinear interpolation) to a fixed resolution (e.g., 64x64) for further processing.

Key functions involved:

1. `normalize_iris_daugman(img, pupil_center, pupil_radius, iris_center, iris_radius, radial_res=64, angular_res=512)`
   Why needed: Standardizes the iris region into a fixed rectangular format while compensating for variations in size and position.

   Working (elaborated): This function implements Daugman's rubber-sheet model. For each angular sample (dividing the circle into equal segments) and each radial sample (from

the pupil to the iris boundary), it calculates the corresponding Cartesian coordinates using linear interpolation. It then calls `bilinear_interpolate` to retrieve the pixel value at these coordinates from the original image. This constructs a normalized image with dimensions defined by `radial_res` and `angular_res`.

Code snippet:

```
def normalize_iris_daugman(img, pupil_center, pupil_radius, iris_center, iris_radius, radial_res=64, angular_
    normalized = np.zeros((radial_res, angular_res), dtype=img.dtype)
    for j in range(angular_res):
        theta = 2 * np.pi * j / angular_res
        xp = pupil_center[0] + pupil_radius * np.cos(theta)
        yp = pupil_center[1] + pupil_radius * np.sin(theta)
        xi = iris_center[0] + iris_radius * np.cos(theta)
        yi = iris_center[1] + iris_radius * np.sin(theta)
        for i in range(radial_res):
            r = i / (radial_res - 1)
            x = (1 - r) * xp + r * xi
            y = (1 - r) * yp + r * yi
            normalized[i, j] = bilinear_interpolate(img, x, y)
    return normalized
```

2. `bilinear_interpolate_image(image)`
   Why needed: Resizes the normalized image to a fixed resolution (e.g., 64x64) using bilinear interpolation, ensuring consistency for feature extraction.

   Working (elaborated): This function calls OpenCV's `cv2.resize` with bilinear interpolation to produce a smooth, fixed-size image, maintaining transitions between pixel values.

Helper Function: `bilinear_interpolate`

Why needed: Essential for sampling pixel values during normalization. It calculates a weighted average of the four nearest pixels to approximate the value at non-integer coordinates.

Working (elaborated): For given non-integer coordinates (x, y), the function computes the integer boundaries, determines the distances, calculates weights, and returns the weighted sum of the surrounding pixels. This function is repeatedly called by `normalize_iris_daugman`.

```
def bilinear_interpolate(img, x, y):
    epsilon = 1e-6
    x = np.clip(x, 0, img.shape[1] - 1 - epsilon)
    y = np.clip(y, 0, img.shape[0] - 1 - epsilon)
    x0 = int(np.floor(x))
    x1 = min(x0 + 1, img.shape[1] - 1)
    y0 = int(np.floor(y))
    y1 = min(y0 + 1, img.shape[0] - 1)
    Ia = img[y0, x0]
    Ib = img[y1, x0]
    Ic = img[y0, x1]
    Id = img[y1, x1]
    wa = (x1 - x) * (y1 - y)
    wb = (x1 - x) * (y - y0)
```

```
    wc = (x - x0) * (y1 - y)
    wd = (x - x0) * (y - y0)
    return wa * Ia + wb * Ib + wc * Ic + wd * Id
```

3 Query Image Processing and Matching Implementation

Before matching, the query image undergoes localization and normalization. The following function handles this process:

`process_query_image_temp(query_file, temp_query_file="temp_query.png")`
Why needed: Processes a raw query iris image by applying localization and normalization, then splits the normalized image into patches and selects one patch for matching.

Working (elaborated):

1. The query image is loaded in grayscale.
2. For localization, the image is converted to BGR and passed to `localize_iris`, which detects the pupil, iris, and eyelid boundaries.
3. Using these detected features, the image is normalized with `normalize_iris_daugman` (producing a 64x512 image).
4. The normalized image is split into eight 64x64 patches by `split_image_into_patches`.
5. One patch (currently the first) is selected and saved as a temporary image.

```
def process_query_image_temp(query_file, temp_query_file="temp_query.png"):
    img = cv2.imread(query_file, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error: Could not load query image.")
        return None
    annotated, features = localize_iris(cv2.cvtColor(img, cv2.COLOR_GRAY2BGR))
    if features is None:
        print("Error: Iris localization failed for the query image.")
        return None
    pupil_center = features["pupil_center"]
    pupil_radius = features["pupil_radius"]
    iris_center = features["iris_center"]
    iris_radius = features["iris_radius"]
    normalized_img = normalize_iris_daugman(img, pupil_center, pupil_radius, iris_center, iris_radius,
                                            radial_res=64, angular_res=512)
    if normalized_img is None:
        print("Error: Iris normalization failed.")
        return None
    patches = split_image_into_patches(normalized_img, patch_width=64)
    if not patches:
        print("Error: No patches were extracted from the normalized image.")
        return None
    chosen_patch = patches[0]
    patch_img = Image.fromarray(chosen_patch)
```

```
    patch_img.save(temp_query_file)
    print(f"Randomly selected query patch saved as {temp_query_file}")
    return temp_query_file
```

In addition, the matching process uses similarity comparisons between the query feature vector and stored template features. Key helper functions include:

Function: `get_ranking(query_feature, db_features, metric='cosine')`

Working (elaborated): This function computes similarity scores between the query feature and all stored template features. If the metric is set to `'cosine'`, it normalizes the features and calculates the cosine similarity; otherwise, it computes the Euclidean distance. It returns the sorted ranking and the similarity scores.

```
def get_ranking(query_feature, db_features, metric='cosine'):
    if metric == 'cosine':
        scores = compute_cosine_similarity(query_feature, db_features)
        ranking = torch.argsort(scores, descending=True)
    elif metric == 'euclidean':
        scores = compute_euclidean_distance(query_feature, db_features)
        ranking = torch.argsort(scores, descending=False)
    return ranking, scores
```

Function: `compute_cosine_similarity(query_feature, db_features)`

Working (elaborated): Normalizes both the query feature and each stored feature, then computes the cosine similarity via matrix multiplication.

```
def compute_cosine_similarity(query_feature, db_features):
    query_norm = F.normalize(query_feature, p=2, dim=0)
    db_norm = F.normalize(db_features, p=2, dim=1)
    similarity = torch.mm(query_norm.unsqueeze(0), db_norm.t()).squeeze(0)
    return similarity
```

Function: `compute_euclidean_distance(query_feature, db_features)`

Working (elaborated): Calculates the Euclidean distance between the query feature and each stored template feature.

```
def compute_euclidean_distance(query_feature, db_features):
    distances = torch.norm(db_features - query_feature.unsqueeze(0), dim=1)
    return distances
```

4 Fusion Methods and Final Matching Process

After obtaining the top-5 matches from the similarity comparisons, the system applies fusion methods to combine rankings from different sources for a robust final matching decision.

4.1 Highest Rank Fusion (`fuse_highest_rank`)

Goal: To select the best (lowest) rank for each identity by comparing two ranking lists, using similarity scores to resolve ties.

Working (elaborated):

1. Build dictionaries for the `left` and `right` ranking lists that map each identity to its rank and similarity.

```
left_dict = {entry[1]: (entry[0], entry[2]) for entry in left}
right_dict = {entry[1]: (entry[0], entry[2]) for entry in right}
```

2. For each identity (from the union of both rankings), compare the ranks:
   - If one system provides a lower rank, that rank is selected.
   - If the ranks are equal, the higher similarity score is chosen.
3. Collect and sort the results by best rank (ascending) and similarity (descending).

Code snippet:

```
def fuse_highest_rank(left, right):
    left_dict = {entry[1]: (entry[0], entry[2]) for entry in left}
    right_dict = {entry[1]: (entry[0], entry[2]) for entry in right}
    identities = set(left_dict.keys()).union(right_dict.keys())
    results = []
    for ident in identities:
        l_rank, l_sim = left_dict.get(ident, (6, 0.0))
        r_rank, r_sim = right_dict.get(ident, (6, 0.0))
        if l_rank < r_rank:
            best_rank = l_rank
            best_sim = l_sim
        elif r_rank < l_rank:
            best_rank = r_rank
            best_sim = r_sim
        else:
            best_rank = l_rank
            best_sim = l_sim if l_sim >= r_sim else r_sim
        results.append((ident, best_rank, best_sim))
    results.sort(key=lambda item: (item[1], -item[2]))
    return results
```

4.2 Borda Count Fusion (`fuse_borda`)

Goal: Combine rankings by summing the ranks from two systems and use the average similarity to resolve ties.

Working (elaborated):

1. For each identity, retrieve its rank and similarity from both ranking lists (using default values if absent).
2. Compute the **rank sum** (the sum of the two ranks) and the **average similarity**.
3. Sort the identities by the rank sum (ascending) and then by average similarity (descending).

Code snippet:

```
def fuse_borda(left, right):
    left_dict = {entry[1]: (entry[0], entry[2]) for entry in left}
    right_dict = {entry[1]: (entry[0], entry[2]) for entry in right}
    identities = set(left_dict.keys()).union(right_dict.keys())
    results = []
    for ident in identities:
        l_rank, l_sim = left_dict.get(ident, (6, 0.0))
        r_rank, r_sim = right_dict.get(ident, (6, 0.0))
        rank_sum = l_rank + r_rank
        avg_sim = (l_sim + r_sim) / 2.0
        results.append((ident, rank_sum, avg_sim, l_rank))
    results.sort(key=lambda x: (x[1], -x[2], x[3]))
    return [(ident, rank_sum, avg_sim) for ident, rank_sum, avg_sim, l_rank in results]
```

4.3 Logistic Regression Fusion (`fuse_logistic`)

Goal: To combine the rankings using a weighted sum approach, where the influence of each ranking system is adjusted via weights. Ties are resolved using the average similarity.

Working (elaborated):

1. For each identity, retrieve its rank and similarity from both ranking lists.
2. Compute the **fused rank** using a weighted sum:

$$\text{fused\_rank} = \text{weight\_left} \times l\_rank + \text{weight\_right} \times r\_rank$$

3. Calculate the **average similarity** between the two systems.
4. Sort the identities by the fused rank (ascending) and then by the average similarity (descending).

Code snippet:

```
def fuse_logistic(left, right, weight_left=0.5, weight_right=0.5):
    left_dict = {entry[1]: (entry[0], entry[2]) for entry in left}
    right_dict = {entry[1]: (entry[0], entry[2]) for entry in right}
    identities = set(left_dict.keys()).union(right_dict.keys())
    results = []
    for ident in identities:
        l_rank, l_sim = left_dict.get(ident, (6, 0.0))
        r_rank, r_sim = right_dict.get(ident, (6, 0.0))
        fused_rank = weight_left * l_rank + weight_right * r_rank
        avg_sim = (l_sim + r_sim) / 2.0
        results.append((ident, fused_rank, avg_sim, l_rank))
    results.sort(key=lambda x: (x[1], -x[2], x[3]))
    return [(ident, fused_rank, avg_sim) for ident, fused_rank, avg_sim, l_rank in results]
```

4.4 Final Matching Process

After obtaining the top-5 matches from the similarity comparisons, each fusion method produces a sorted list of identities:

- Each fusion method's top-ranked identity (the one with the lowest rank) is considered the best candidate match.
- The final matching decision is made by comparing these top candidates from each fusion method.

This multi-fusion approach enhances overall matching accuracy by leveraging the strengths of different ranking systems and effectively resolving ties through similarity scores.

5 Cumulative Challenges Faced

The key challenges encountered across the entire system are:

- **Accurate Tie-Breaking:** Resolving ties effectively using similarity scores is critical to ensuring the most accurate match.
- **Rank Normalization and Weight Tuning:** Ensuring that the ranks from different systems are comparable and optimally tuning the weights in the logistic regression fusion method.
- **Robustness to Variability:** Handling variations in iris size, illumination, and partial occlusions consistently across localization, normalization, and matching steps.