SIFT (Scale-Invariant Feature Transform):

SIFT method is widely used for image similarity search due to its robustness and ability to identify and match distinctive features in images, regardless of variations. SIFT methodology mainly involves keypoints and descriptors. Keypoints and descriptors are fundamental concepts used to identify and describe distinctive regions in an image for tasks like similarity search.

Keypoints identify where the distinctive features are located in the image. Each keypoint corresponds to a specific pixel in the image, typically representing corners, edges, or blobs that are rich in texture and less likely to change under transformations. The scale of the keypoint indicates the size of the region around it where the feature is detected. This helps make the keypoint invariant to changes in image size. An orientation is assigned to each keypoint based on the dominant gradient direction in its local neighborhood. This allows for rotation invariance.

Descriptors describe what those features look like. SIFT generates a 128-dimensional vector for each keypoint. This vector encodes gradient magnitudes and directions in a local region around the keypoint. Descriptors are normalized to ensure robustness against variations in illumination or contrast. Descriptors are used to compare and match keypoints between different images by calculating distances (e.g., Euclidean distance) between their vectors.

SIFT is not a machine learning algorithm but a mathematical based method to perform image similarity search. Usually similarity search in e-commerce and fashion platforms needs to be more precise and accurate and hence SIFT is used when high precision matching is important. It detects and describes features that are consistent across scale, rotation, and minor affine transformations. Hence, it's a very effective method while dealing with small and medium sized datasets.

However, it has a few limitations. SIFT is slower than modern alternatives (e.g., CNN-based methods), making it less suitable for real-time applications or large-scale datasets. The 128-dimensional descriptors increase memory and computational requirements for storing and processing features.

Implementation of SIFT:

1. `resize_image(image, maxD=1024)`

This function resizes an image while maintaining its aspect ratio. It is essential for ensuring uniformity in image dimensions, which simplifies processing.

- **Steps:**

- The function extracts the `height` and `width` of the input `image`.
- It calculates the `aspectRatio` as `width / height`.
- Depending on the aspect ratio:
  - If the image is taller (aspect ratio < 1), the height is scaled to `maxD`, and the width is scaled proportionally.
  - If the image is wider, the width is scaled to `maxD`, and the height is adjusted proportionally.
- The image is resized using OpenCV's `cv2.resize` function.
- `maxD=1024`: The maximum dimension for resizing. It ensures that the larger side of the image is scaled down to 1024 pixels while maintaining the aspect ratio.

2. `compute_SIFT(image)`

This function detects keypoints and computes descriptors for the input image using the SIFT algorithm.

- **Steps:**
  - A SIFT detector is created using `cv2.SIFT_create()`.
  - The `detectAndCompute` method is called on the input `image` to extract:
    - `Keypoints`: Distinctive points in the image (e.g., corners, blobs).
    - `Descriptors`: 128-dimensional feature vectors describing the local image region around each keypoint.
  - The function computes descriptors of 128 dimensions for each keypoint, a standard in the SIFT algorithm.

3. `match_descriptors(des1, des2)`

This function matches the SIFT descriptors of two images using the k-Nearest Neighbors (k-NN) algorithm and filters matches with the ratio test.

- **Steps:**
  - A brute-force matcher (`cv2.BFMatcher`) is initialized.
  - Descriptors from two images (`des1` and `des2`) are matched using k-NN (`k=2`).
  - For each pair of matches:
    - The first match (`m`) is compared with the second match (`n`).
    - If `m.distance < 0.7 * n.distance`, the match is considered "good" and added to the `good_matches` list.
  - `k=2`: Ensures that each descriptor is matched with its two nearest neighbors.
  - `0.7`: The ratio threshold filters out ambiguous matches, ensuring only distinctive matches are retained.

4. `similarity_score(matches, keypoints1, keypoints2)`

This function calculates a similarity score between two images based on the number of good matches and the total number of keypoints.

- **Steps:**
  - The function counts the number of `matches`.
  - It calculates the minimum number of keypoints in the two images: `min(len(keypoints1), len(keypoints2))`.
  - The similarity score is computed as: score=len(matches)min(len(keypoints1), len(keypoints2))×100\text{score} = \frac{\text{len(matches)}}{\text{min(len(keypoints1), len(keypoints2))}} \times 100score=min(len(keypoints1), len(keypoints2))len(matches)×100
  - The score is returned as a percentage.
  - The score is normalized to the smaller of the two sets of keypoints to avoid bias when one image has significantly more keypoints than the other.
  - The multiplication by 100 converts the score into a percentage.

5. `plot_similarity(image1, image2, keypoints1, keypoints2, matches)`

This function visualizes the matches between two images by drawing lines connecting matched keypoints.

- **Steps:**
  - The function uses `cv2.drawMatches` to create an image with lines connecting matched keypoints between `image1` and `image2`.
  - By default, unmatched keypoints are not displayed (`flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS`).
  - The resulting image is displayed using Matplotlib with labels and axis disabled.
  - By default, it plots up to 30 matches (when explicitly set in other parts of the code) for better readability.
  - `flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS`: Ensures the visualization focuses only on matched keypoints, avoiding clutter.

6. `compare_images(image1, image2)`

This function performs the full comparison workflow for two images, including computing SIFT features, matching descriptors, calculating similarity scores, and visualizing matches.

- **Steps:**

1. SIFT keypoints and descriptors are computed for both images using `compute_SIFT`.
2. The descriptors are matched using `match_descriptors`.
3. A similarity score is calculated using `similarity_score`.
4. The matches are visualized with `plot_similarity`, and the similarity score is printed.

7. `load_dataset()`

This function loads and prepares image pairs from the FashionMNIST dataset.

- **Steps:**
  - The dataset is loaded with a transformation to convert images into PyTorch tensors.
  - Images are extracted based on indices to create pairs of:
    - Same-class images.
    - Different-class images.
    - Identical images.
  - Each image is converted to an 8-bit unsigned integer (`uint8`) for compatibility with OpenCV.
  - Images are scaled from `[0, 1]` to `[0, 255]` and converted to `uint8` for SIFT processing.

8. `compare_class_with_random_samples(n_random_samples=10)`

This function compares a representative image from each class in FashionMNIST with a set of random samples.

- **Steps:**
  - A representative image is extracted for each class by finding the first image labeled with that class.
  - Random samples are selected from the dataset, and their labels are stored.
  - For each class representative:
    - The SIFT features are computed.
    - Each random sample is compared using SIFT matching and scored.
    - Matches are visualized with similarity scores displayed.
  - `n_random_samples=10`: Specifies the number of random samples to compare against each class representative.

Summary of SIFT-Based Similarity Search Analysis

The results of the SIFT-based similarity search highlight its strengths and limitations in handling various types of image comparisons:

1. Similar Images (Same Class):

- **Example:** Dress vs. T-shirt (Image pair 1-2).
- **Observation:** A moderate similarity score of 66.67% with only 2 matches indicates SIFT's ability to detect generic shared features (e.g., edges or folds) between similar-class items.

2. Completely Different Classes:

- **Examples:** Dress vs. T-shirt, Sneaker vs. T-shirt (Image pair 3-4 and Image pair 5-6).
- **Observation:** Both comparisons yielded very low similarity scores (0.00% and 20.00%), with minimal or no matches detected.

3. Identical Images:

- **Examples:** Identical dresses and near-identical dresses with slight variations (Image pair 7-8 and Image pair 9-10).
- **Observation:** Both comparisons achieved perfect similarity scores (100.00%) and aligned keypoints correctly, with the number of matches being 3 and 2, respectively.