Image Segmentation

Christopher Zanoli — czanoli@ethz.ch — Student Number: 23-942-394

1 Mean-Shift Algorithm

The Mean Shift algorithm is a non-parametric clustering technique that involves shifting data points toward the densest areas to identify prominent clusters. The following are the main steps of the algorithm.

1.1 Distance Function Implementation

This function computes the Euclidean distance between the reference point `x` and the set of all points `X`. It is implemented using `np.linalg.norm()` for efficiency reasons.

1.2 Gaussian Function Implementation

This function implements the Gaussian kernel. It results in a weight for each point, with values closer to the reference point receiving higher weights.

1.3 Update Point Function Implementation

This function computes a new point as the weighted average of the points in `X`, using the corresponding weights from the `weight` array. It results in a centroid-like point that reflects the weighted distribution of the points in `X`.

1.4 Mean-Shift Step Implementation

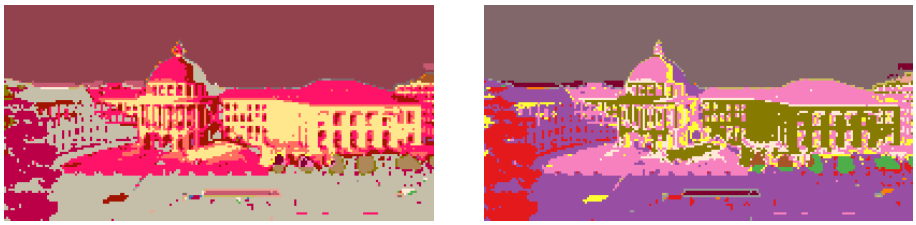
This function iteratively updates the set of data points `X` according to the previously described logic. After the last iteration, the function returns the modified set of points `X`, which are expected to have converged to the centroids of their respective clusters. The following figure shows the algorithm implementation.

```python
def distance(x, X):
    return np.linalg.norm(X - x, axis=1)

def gaussian(dist, bandwidth):
    return  (np.exp(-0.5 * (dist ** 2) / (bandwidth ** 2))) / (bandwidth * np.sqrt(2 * np.pi))

def update_point(weight, X):
    return (np.sum(weight[:, np.newaxis] * X, axis=0)) / np.sum(weight)

def meanshift_step(X, bandwidth):
    new_X = np.zeros_like(X)
    for i, x in enumerate(X):
        distances = distance(x, X)
        weights = gaussian(distances, bandwidth)
        new_X[i] = update_point(weights, X)
    return new_X
```

1.5 Experiment with Different Bandwidth and Interesting Remarks

As requested in the handout, the following explains why `bandwith = 1` makes the code not runnable. A low bandwidth (e.g. $= 1$) leads to greater sensitivity to fine details in the image. In practical terms, it means that the algorithm identifies more clusters, many of which may represent small color variations or noise in the image. This results in a larger number of unique centroids. In fact, with `bandwith = 1` as many as 282 centroids are obtained.
Conversely, a higher bandwidth (e.g., 2.5 or greater) makes the algorithm less sensitive to fine details, leading to a reduction in the number of clusters. In essence, the algorithm "smoothes out" minor variations and tends to merge similar regions into a single cluster. This reduces the total number of centroids, making it easier to match a limited number of colors. For `bandwith = 1`, the total number of clusters found (282) is significantly higher than the number of available colors (24), causing an error in the code.

The following figure shows the obtained results with working bandwidths. The results obtained agree with what has just been described regarding the impact of different bandwidths on clustering.



(a) bandwidth = 2.5

(b) bandwidth = 3

(c) bandwidth = 5

(d) bandwidth = 7

1.5.1 Possible solutions to the bandwidth problem

The first practical solution I considered to solve this issue was to limit the number of centroids to match the number of colors. If the number of clusters is greater than 24 (i.e., greater than the number of available colors), the centroids are sorted by frequency, and then the first 24 are selected. The selected centroids are likely to be those that capture the most distinctive and recurrent color variations in the image. However, this method may be sensitive to minor color variations, identifying numerous centroids even for small differences, resulting in a segmented image that overly emphasizes these minor variations.

The second practical solution I considered is to match each determined centroid to the closest available color. This approach computes the Euclidean distance between each centroid and the set of specified colors, then assigns each centroid to the color that is closest to it. Although this solution solves the problem, when each centroid is mapped to the nearest color in a limited palette, the subtle variations represented by different centroids might be assigned to the same color. This process inherently generalizes and simplifies the color representation, leading to a loss of detail.

I decided to implement such solutions to see how they worked on the input image. The following figures show the two implementations and the obtained results. These results agree with what has just been described.

```python
# Solution 1
unique_centroids, counts = np.unique(labels, return_counts=True)
sorted_centroids = unique_centroids[np.argsort(-counts)][:24]
mapped_labels = np.array([np.argmin(np.abs(sorted_centroids - label)) for label in labels])
result_image = colors[mapped_labels].reshape(shape)

# Solution 2
def find_nearest_color(centroid, colors):
    distances = np.linalg.norm(colors - centroid, axis=1)
    return np.argmin(distances)

mapped_labels = np.array([find_nearest_color(centroid, colors) for centroid in centroids])
result_image = colors[mapped_labels[labels]].reshape(shape)
```

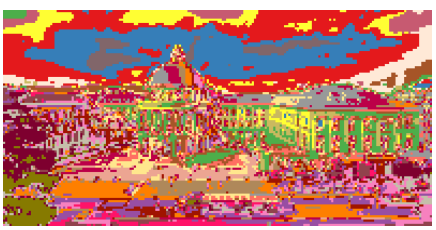


Figure 2: Solution 1 result

Figure 3: Solution 2 result