Emoji Detection in Augmented Images Using SIFT-based Segmentation and Bhattacharyya Distance for Image Processing and Recognition (IPR)

Bhavanir Rai
Faculty of Engineering and Computer Science
University of Victoria
Victoria, Canada
brai@uvic.ca

Karen McDougall
Faculty of Engineering and Computer Science
University of Victoria
Victoria, Canada
karenmcdougall@uvic.ca

***Abstract*— We propose a novel approach for locating emojis in images that is resilient to noise-based image augmentations. We first segment the input image into an overlapping grid of equally sized sub-images, with each sub-image being the same dimension as the target emoji. We then apply the Scale-Invariant Feature Transform (SIFT) algorithm on each local feature set of the image, resulting in 1,521 keypoint comparisons between the current sub-image and the target emoji. This allows us to identify candidate keypoints that have a likely resemblance to the target emoji, forming a strong foundation for further filtering using the Bhattacharyya distance. This implemented approach achieved a final score of 195.92 with Cutout, CoarseSaltAndPepper. JpegCompression, GaussianBlur, MotionBlur and Rot90 augmentations enabled.**

Keywords— Segmentation, SIFT, Bhattacharyya

I. INTRODUCTION

This report outlines the development, implementation and outcomes of the Emoji Hunt project solution that was designed and executed by our group. The objective of the Emoji Hunt challenge was to employ computer vision concepts to identify all occurrences of a given emoji, provided from a pool of approximately 300, within an input image (see Fig. 1. and 2. for an example of an emoji and input image). Furthermore, an augmentation from a list of 24 augmentations could be applied to the emoji in the input image. Our goal was to implement a solution that would match with as many of the augmentations as possible.

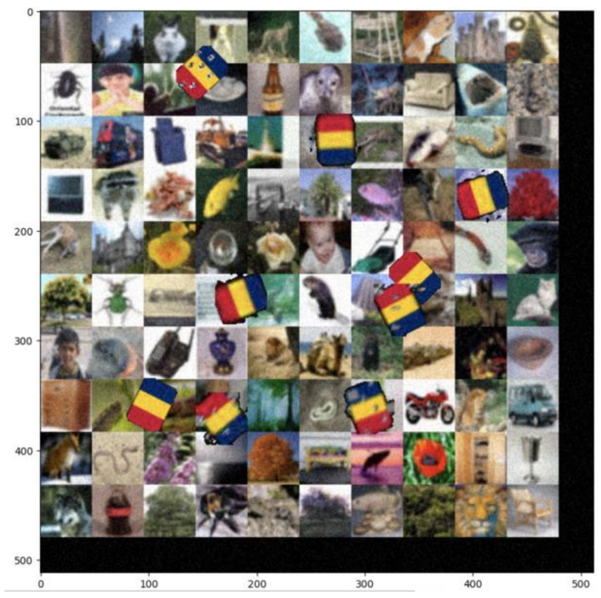


Fig. 1. Example of an input image

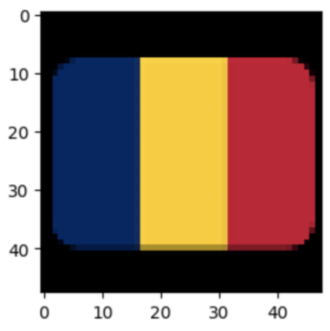


Fig. 2. Example of an emoji

Our solution utilized 2 computer vision concepts: histograms and the Scale-Invariant Feature Transform (SIFT) algorithm. Histograms are a way to represent images. Histograms are an array of values in which the value corresponds to the number of pixels in the image that are of a particular intensity. SIFT is an image detection algorithm that is invariant to scale and rotation changes and robust to illumination changes, noise and 3D perspective changes [1]. There are 3 main steps in SIFT: detection, description and matching. In the detection step, the SIFT algorithm produces keypoints by utilizing a scale space and Derivatives of Gradients (DoG) to identify points of interests. These points are then refined through thresholding to eliminate weak points of interest and produce the final set of keypoints. Next in the description step, histogram buckets are created and concatenated to produce a 128-feature vector for each keypoint. Finally, in the matching step, the keypoints from the sample image are compared to the keypoints the main image to find a match.

During the designing and implementation of our project, we encountered a few challenges that required us to modify our original approach. Initially we planned to rely solely on histograms to identify matched of emojis in the input image. While this method produced accurate matches for some of the input images, it also produced a significant number of false positives for others. Notably, we observed instances where the full image was mistakenly identified as a "match". Another considerable challenge we encountered during this project was the time constraint. While implementing our solution, we discovered that the emoji hunt problem was more complex than initially anticipated. Consequently, we invested more time implementing and testing the histogram and SIFT solution than initially planned and were left with limited time to explore other techniques to solve for additional augmentations.

II. Implemented Approach

Our solution to Emoji Hunt problem consists of first splitting the input `test_image` into equally sized sub-images, whose dimensions are identical to that of the `emoji_target` by defining a new function `segment_image()` which takes the image requiring segmentation as it's only parameter. The function starts by extracting the height and width of the input image (`image`) using the `shape` property, which returns the dimensions of the image as a tuple `(height, width)`. It also extracts the height and width of a target image (`emoji_target`) used for segmentation, using the same `shape` property. The function then defines a `stride` value, which determines the step size for moving the segmentation window across the input image. The `stride` value controls how much overlap there will be between adjacent sub-images. In this case, it is set to 12, which was determined experimentally, as it was found to offer the best balance between computational cost, by generating only 1,521 sub-images for a 512x512 input image, and performance on test inputs with an average per-unit runtime of 6s, and a total average test-suite runtime of 3m 19s. The function initializes an empty list `sub_images` to store the sub-images and their center points, utilizing nested loops to iterate over the height and width of the input image with the given stride. For each combination of `y` and `x` values, a sub-image is extracted from the input image using array slicing. The sub-image is obtained by specifying the region of interest (ROI) using the `y` and `x` values as starting coordinates and adding the height and width of the target image (`emoji_target`) as ending coordinates (see Fig. 3. for an example generated sub-image). The function calculates the center point of each sub-image by taking the average of `y` and `x` values of the ROI and stores the sub-image and its center point as a tuple in the `sub_images` list, where each element of the list is of form `(sub-image, (x, y))`.

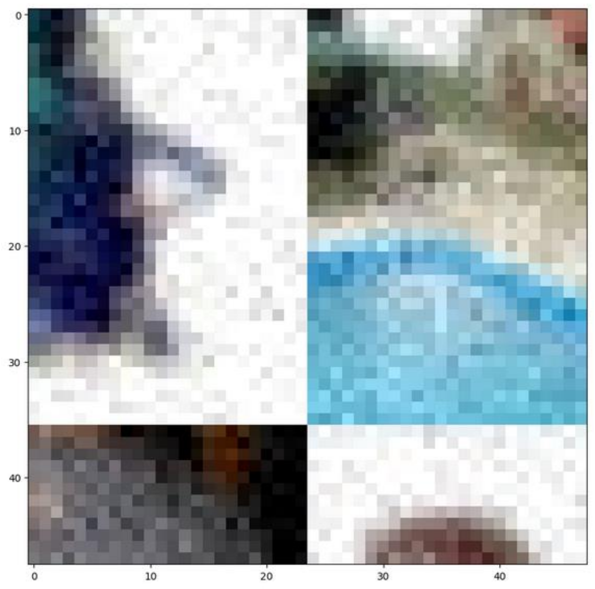


Fig. 3. Sample sub-image segmented from the test image, using a stride of 12.

Another function, `sift_filter()`, takes the `test_image` and `emoji_target` images as parameters. This function first creates a SIFT object using `cv2.SIFT_create()` from the OpenCV library. It then converts the `emoji` image to grayscale, in order to reduce complexity [2], using `cv2.cvtColor()` with the `cv2.COLOR_BGR2GRAY` colour conversion code and detects keypoints and computes descriptors for the grayscale image using `sift.detectAndCompute()`, storing them in `kp_ref` and `desc_ref` respectively. SIFT was chosen in the approach due to its effectiveness in identifying distinctice features in images that are robust to changes in scale, rotation, and illumination. Next, the test image is segmented using the previously defined `segment_image()` function, and the resulting filtered images are iterated over in a loop. For each filtered image, the colour image and its corresponding keypoints are extracted. The colour image is converted to grayscale, simliar to the emoji image. The keypoints and descriptors are computed using `sift.detectAndCompute()` and stored in `kp_ROI` and `desc_ROI` respectively. A brute-force matcher using `cv2.BFMatcher()` is then created, and the descriptors of the reference image and the filtered image [3] are matched using `bf.knnMatch()` with `k=2` to obtain a list of matches. The matches are filtered based on a distance ratio test, where only matches whose distance is less than a factor of the distance to the next best match are considered good matches. The colour image and the filtered keypoint coordinates are appended to a list, where each element is also of form `(sub-image, (x, y))`, in order to retain algorithmic consistency using a rigid data structure. Finally, non-maximum suppression is applied to the keypoints using a `non_max_suppression()` function with a radius of 48, and the resulting keypoints are returned as `suppressed`. The non-maximum suppression radius of 48 was chosen as the sub-image stride of 12 is a factor of the radius, and more importantly the radius represents the space requirement of each emoji, as each emoji has dimensions of 48x48.

A helper function which reduces the total number of redundant keypoints computed in the `sift_filter()` function, is `non_max_suppression()` [4], which takes two inputs: a list of `keypoints` and a `radius` value. The function first initializes an empty dictionary `non_duplicate` to keep track of keypoints without sub-image duplicates, as the SIFT process records identical sub-images, but with differing point coordinates. It then iterates over the input `keypoints` list, and for each keypoint, it extracts the coordinates (`point`) of the keypoint and uses it as the key in the `non_duplicate` dictionary. The value associated with each key is the entire keypoint item. Next, the `keypoints` list is updated with the values from the `non_duplicate` dictionary, effectively removing any duplicate keypoints. The function then initializes an empty list `selected_keypoints` to store the selected keypoints after non-maximum suppression. It enters a loop that continues until there are no keypoints left in the `keypoints` list. Inside the loop, the function selects the first keypoint from the `keypoints` list as the `selected_keypoint`, and appends it to the `selected_keypoints` list. It then calculates the distances between the `selected_keypoint` and all the remaining keypoints in the `keypoints` list using the `np.linalg.norm()` function, which calculates the Euclidean distance between two points. A boolean mask is created by comparing the distances with the `radius` value, resulting in a boolean array with `True` for distances less than the `radius` and `False` for distances greater than or equal to the `radius`. The keypoints in the `keypoints` list that correspond to `True` values in the mask are removed by using a list comprehension to filter out those keypoints. The remaining keypoints in the `keypoints` list are those that are far enough from the `selected_keypoint` based on the `radius` value. The loop continues until all keypoints in the `keypoints` list have been processed. Finally, the function returns the list of `selected_keypoints` after non-maximum suppression (see Fig. 4. for a comparison between not including the non-max suppression step and including it), which contains a subset of the original keypoints that are considered as the most salient or distinct keypoints based on the specified `radius` value.

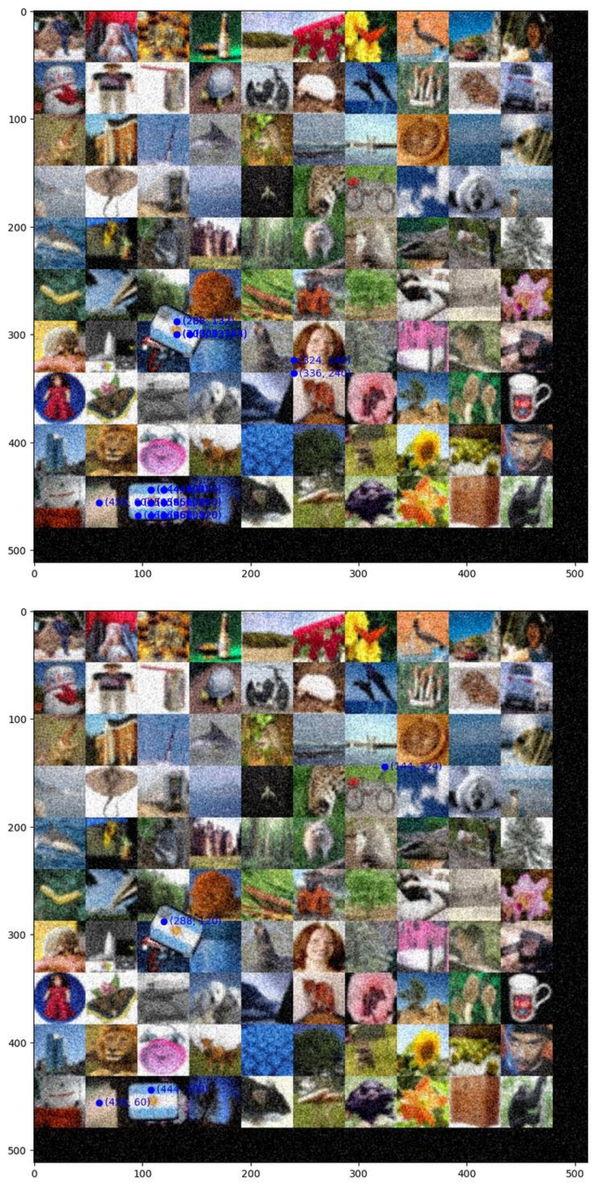


Fig. 4. Comparison between no non-max suppresion (top) and non-max suppresion applied (bottom) to the same image.

Finally, the `bhatt_filter()` function performs a filtering operation on an input image using the Bhattacharyya distance metric, which measures the similarity between two histograms. The function takes an input image (`image`) and a target image (`emoji`) as inputs. The function starts by computing the histogram of the target image (`emoji`) using a `compute_histogram()` function. The histogram is stored in the variable `emoji_hist`. Next, the function calls a `sift_filter()` function with the input image (`image`) and the target image (`emoji`) as arguments, which performs a SIFT-based filtering operation on the input image and returns a list of sub-images along with their corresponding points. The sub-images and their points are stored in the `sub_images` list. The function then iterates over each item in the `sub_images` list using a `for` loop. For each item, it extracts the sub-image and its corresponding point. It computes the histogram of the sub-image using the `compute_histogram()` function and stores it in the variable `hist`. It also calculates a similarity score between the histogram of the target image (`emoji_hist`) and the histogram of the sub-image (`hist`) using a `histogram_similarity()` function and stores the score in the variable `score`. The sub-image point along with the score is appended as a tuple to the `bhatts` list. The function then extracts the scores from the `bhatts` list using list comprehension and stores them in the `scores` list. It calculates the minimum score, mean score, and standard deviation of the scores using the NumPy library functions `np.amin()` and `np.mean()`, and `np.std()` respectively, and stores them in the variables `min_score`, `mean_score`, and `std_score`, respectively. The function then calculates a factor by multiplying the coefficient of variation (standard deviation divided by mean) of the scores by a fixed value of 1.275 and stores the result in the variable `ratio`. The function initializes an empty list `filtered` to store the filtered points. It then iterates over each item in the `bhatts` list using a `for` loop with an index variable `i`. For each item, it extracts the score and point. It compares the score with the minimum score, using the NumPy function `np.isclose()` with relative tolerance (`rtol`) set to the calculated `ratio`. If the score is close to the minimum score within the specified relative tolerance, the point is appended to the `filtered` list (see Fig.5. for a comparison between not including the Bhattacharyya step and including it). Finally, the function returns the `filtered` list, which contains the points of the sub-images that passed the filtering operation based on the Bhattacharyya distance metric.

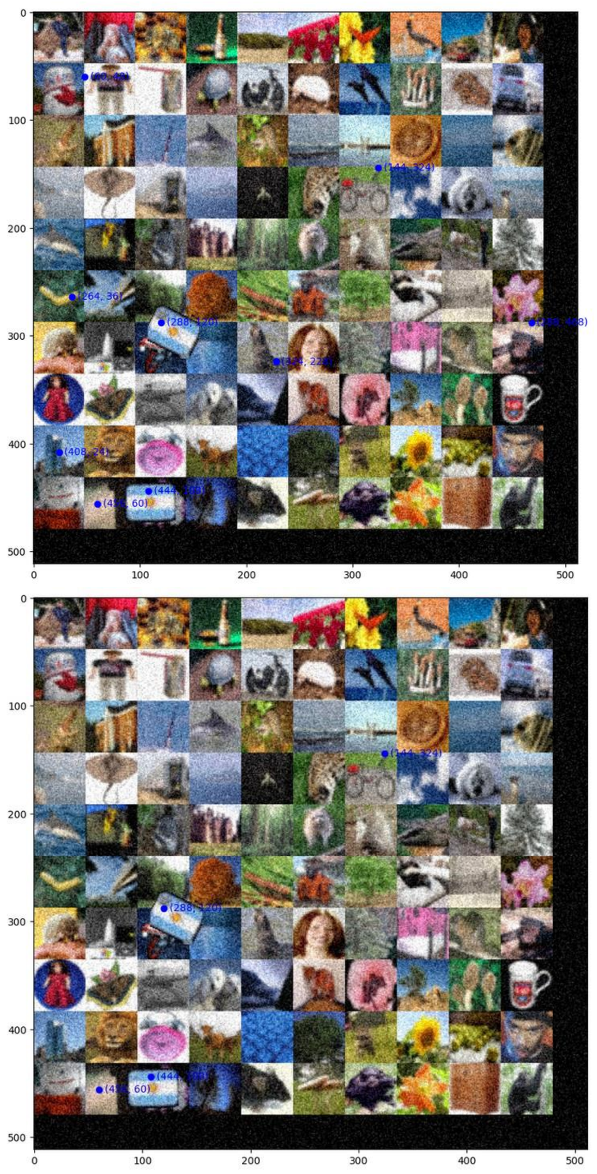


Fig. 5. Comparison between no Bhattacharyya (top) and Bhattacharyya thresholding applied (bottom) to the same image.

The `compute_histogram` helper function calculates and returns the histogram of an input image [5]. Histograms are used in the approach for emoji detection due to their ability to capture colour representation, robustness to image alterations, dimensionality reduction, compatibility with the proceeding techniques, and established usage in image processing applications. The input image is converted to grayscale using OpenCV's `cv2.cvtColor()` function with the conversion code `cv2.COLOR_BGR2GRAY`, which converts the image from the BGR color space to grayscale. The result is stored in a variable called `gray`. The function then defines the size of the histogram as 256 bins using the `hist_size` parameter and sets the range of intensity values in the histogram to be from 0 to 255 using the `hist_range` parameter. Next, the function calls OpenCV's `cv2.calcHist()` function with the following arguments:

- `gray`: the grayscale image

- `[0]`: a list specifying the channels to be used for histogram calculation (in this case, only the intensity channel, which is the first channel in grayscale image)
- `None`: no mask is used for histogram calculation
- `[hist_size]`: the number of bins in the histogram (256 in this case)
- `hist_range`: the range of intensity values in the histogram (0 to 255 in this case)
- `accumulate=False`: the histogram is not accumulated from multiple images

The calculated histogram is stored in the variable `hist`. After histogram calculation, the contribution of pixels with intensity 0 (i.e., the first bin in the histogram) is set to 0, as indicated by `hist[0] = 0`. This is done to remove the contribution of pixels with very low-intensity values, as the `segment_image()` function also includes the black bands which outline the right-most and bottom-most edges of the test image (see Fig. 6. for an example generated histogram); the inclusion of these 0-information sub-images has the potential to skew the results of our algorithm, depending on the target emoji.

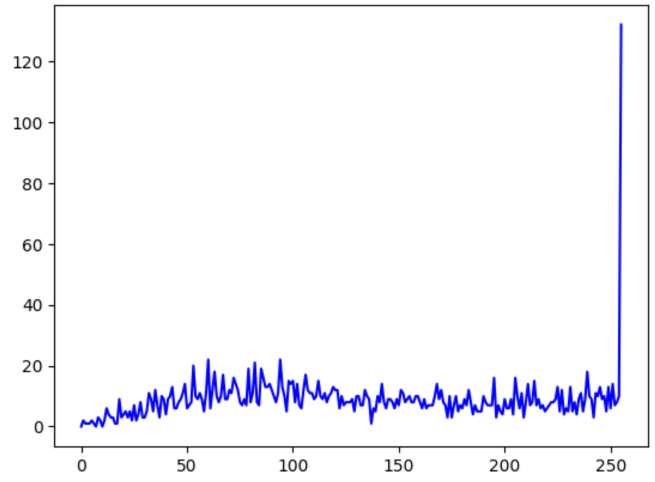
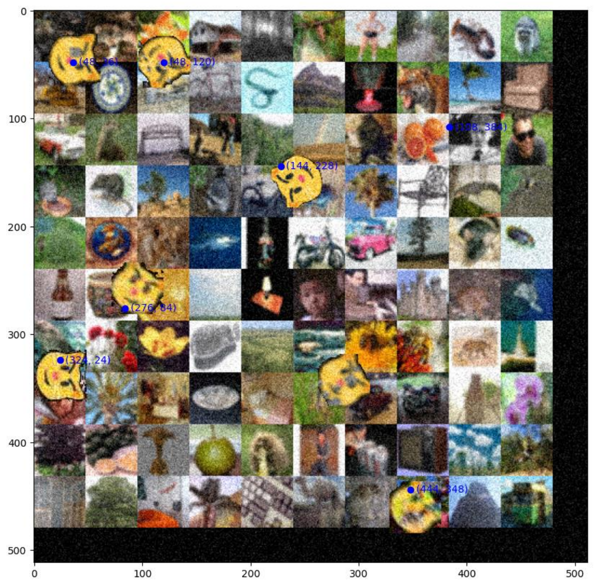


Fig. 6. Sample histogram of a sub-image, with the 0-intensity pixels accounted for.

The `histogram_similarity()` function calculates the similarity between two input histograms [6] using the Bhattacharyya distance metric and returns the computed similarity score. The function takes two input histograms as arguments: `histogram1` and `histogram2`, which are previously calculated histograms of two different image regions. The function first normalizes the input histograms using OpenCV's `cv2.normalize()` function. The normalization is performed using the Min-Max normalization method with `alpha=0` and `beta=1`, which scales the histograms to have values between 0 and 1. The Bhattacharyya distance was chosen as a metric in the approach, as it takes into consideration both the means and variances of the distributions, making it suitable for comparing histograms of different images that may have differences in pixel intensities and colour distributions. The output range of 0-1 also allows for easier thresholding, as the values are normalized and translate nicely when developing boundary expressions. The normalized histograms are stored in variables `hist1_norm` and `hist2_norm` respectively. Next, the function calculates the Bhattacharyya distance between the normalized histograms using OpenCV's `cv2.compareHist()` function with the `cv2.HISTCMP_BHATTACHARYYA` flag. The Bhattacharyya distance is a measure of similarity between two histograms, with values ranging from 0 (indicating identical histograms) to 1 (indicating completely dissimilar histograms). The calculated Bhattacharyya distance is stored in the variable `bhattacharyya`. Finally, the function returns the calculated Bhattacharyya distance as the similarity score between the two input histograms.

III. Experiments and Evaluation

To evaluate the effectiveness of our solution, we executed the `official_test()` function provided in the Emoji Hunt Project template to test our solution against 49 different emojis and input images. A low score on these tests indicates a higher accuracy. We conducted multiple trials enabling a single augmentation and observing whether the score improved or worsened with the enabled augmentation. Additionally, we experimented with changing the order of filtering to determine which sequence produced the best results. We found that first identifying matches using SIFT then further refining results using histograms resulted in a better score rather than using histograms first then refining matches using SIFT (see Fig. 7. for resultant image of our solution).



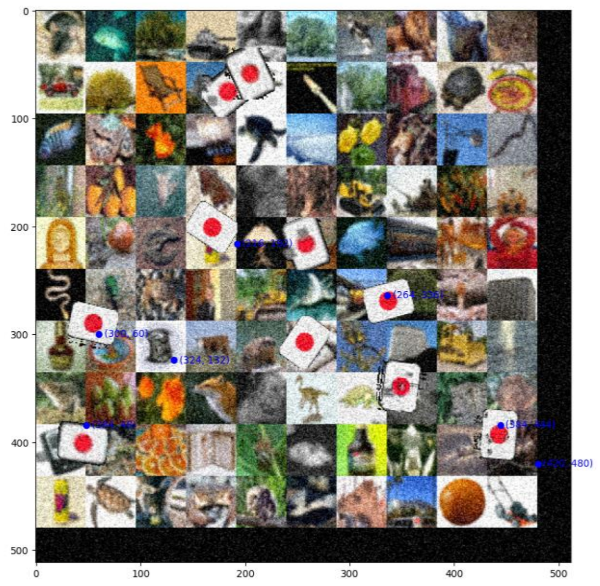
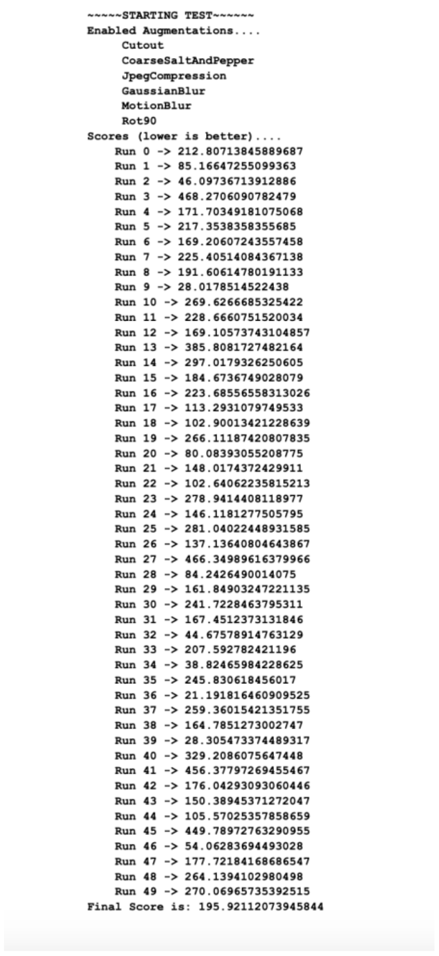


Fig. 7. Example of resultant images generated via our solution implementation (blue points indicate likely emoji occurrences).

Our solution achieved a final score of 195.92 with Cutout, CoarseSaltAndPepper. JpegCompression, GaussianBlur, MotionBlur and Rot90 augmentations enabled (see Fig. 8. for results for every run). Our best score was 21.19 while our worst score was 468.27.

Our solution demonstrated robustness to rotated and scaled occurrences of the emoji in the input image. Both techniques that we utilized in our solution, histogram comparisons and SIFT, are invariant to rotation, allowing our solution to identify Rot90 augmentations with high accuracy. Our solution also achieved a low score for JpegCompression augmentation which is likely due to our use of SIFT in our solution which is a technique that is invariant to scale changes due to its use of a scale space to calculate keypoints. Our solution was additionally able to identify emojis that were augmented using GaussianBlur and MotionBlur as we utilized SIFT as one of matching mechanisms and gaussian blurring is a crucial step in the SIFT when creating a scale space. The solution was also found to maintain a low score when noise such as CoarseSaltAndPepper and Cutout was applied. This is likely due to our thresholding of matches allowing for some dissimilarities between the emoji and the input image and our use of SIFT matching which is invariant to noise.



```
~~~~~STARTING TEST~~~~~~
Enabled Augmentations....
     Cutout
     CoarseSaltAndPepper
     JpegCompression
     GaussianBlur
     MotionBlur
     Rot90
Scores (lower is better)....
    Run 0 -> 212.80713845889687
    Run 1 -> 85.16647255099363
    Run 2 -> 46.09736713912886
    Run 3 -> 468.2706090782479
    Run 4 -> 171.70349181075068
    Run 5 -> 217.3538358355685
    Run 6 -> 169.20607243557458
    Run 7 -> 225.40514084367138
    Run 8 -> 191.60614780191133
    Run 9 -> 28.0178514522438
    Run 10 -> 269.6266685325422
    Run 11 -> 228.6660751520034
    Run 12 -> 169.10573743104857
    Run 13 -> 385.8081727482164
    Run 14 -> 297.0179326250605
    Run 15 -> 184.6736749028079
    Run 16 -> 223.68556558313026
    Run 17 -> 113.2931079749533
    Run 18 -> 102.90013421228639
    Run 19 -> 266.11187420807835
    Run 20 -> 80.08393055208775
    Run 21 -> 148.0174372429911
    Run 22 -> 102.64062235815213
    Run 23 -> 278.9414408118977
    Run 24 -> 146.1181277505795
    Run 25 -> 281.04022448931585
    Run 26 -> 137.13640804643867
    Run 27 -> 466.34989616379966
    Run 28 -> 84.2426490014075
    Run 29 -> 161.84903247221135
    Run 30 -> 241.7228463795311
    Run 31 -> 167.4512373131846
    Run 32 -> 44.67578914763129
    Run 33 -> 207.592782421196
    Run 34 -> 38.82465984228625
    Run 35 -> 245.830618456017
    Run 36 -> 21.191816460909525
    Run 37 -> 259.36015421351755
    Run 38 -> 164.7851273002747
    Run 39 -> 28.305473374489317
    Run 40 -> 329.2086075647448
    Run 41 -> 456.37797269455467
    Run 42 -> 176.04293093060446
    Run 43 -> 150.38945371272047
    Run 44 -> 105.57025357858659
    Run 45 -> 449.78972763290955
    Run 46 -> 54.06283694493028
    Run 47 -> 177.72184168686547
    Run 48 -> 264.1394102980498
    Run 49 -> 270.06965735392515
Final Score is: 195.92112073945844
```

Fig. 8. Full output of scores of the total 49 test-suite runs, along with the final score

However, our solution did not perform well for other noise augmentations such as CoarseDropout. We could improve our performance with this augmentation by applying noise-reducing smoothing to the input image prior to executing the histogram comparisons. Augmentations that can greatly alter an image's histogram such as Add and CLAHE produced high scores with our solution (see Fig. 9. for example of how a histogram is affected by a CLAHE augmentation). This can be attributed to our use of histogram comparisons when refining our matches produced by SIFT. Augmentations that we expected our solution to yield better results for were augmentations that changed the colour of the image but produced similar, gray-scaled images such as Add, Multiply and grayscale augmentations. In our solution we compute histograms and perform SIFT on the gray-scaled emoji and sub-images. As we use gray-scaled images in our comparisons, we expected that our solution would perform better on augmentations that produced similar, gray-scaled images to the original emoji image. One possible reason for our solution being ineffective with these augmentations is that the changes in colour decreased the contrast in the gray-scaled image resulting in SIFT producing dissimilar keypoints for the sub-image. To improve our solution, we could increase the contrast in both the emoji and input image so that edges are more distinct enabling the SIFT algorithm to identify keypoints in the image.

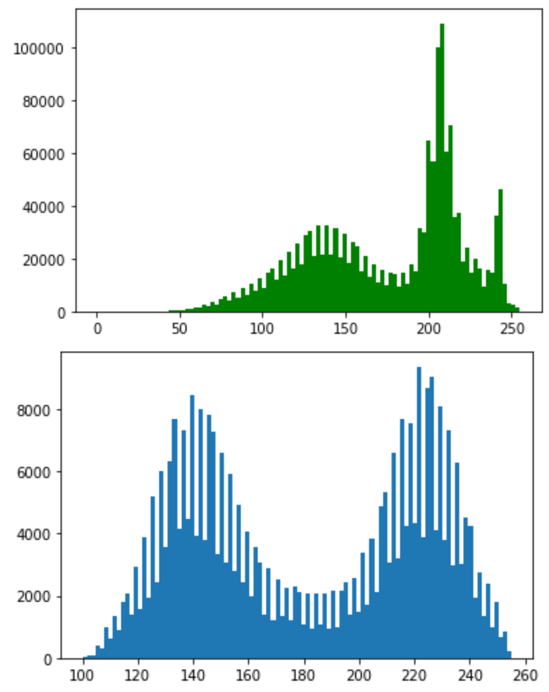


Fig. 9. Histogram of an unmodified image (top) and a histogram of an image after a CLAHE augmentation (bottom) [7]

IV. Conclusion

By first segmenting the input image into an overlapping grid of equally sized sub-images, with each sub-image being the same dimensions as the target emoji, we are able run the SIFT algorithm on each local feature set of the image with 1,521 keypoint comparisons made between the current sub-image and emoji target to find candidate keypoints that have a likely resemblance to the target emoji, which develops a strong foundation for further filtering using the Bhattacharyya. With the problem space being reduced using SIFT, we are then able to compute the Bhattacharyya distance between the histogram of each sub-image, ensuring that colour representation is considered in the final emoji location, as SIFT strictly considers grayscale image representations. This approach to finding all possible locations of an emoji is particularly resilient to image augmentations that alter the feature space of an image by means of noise addition, or removal and is particularly susceptible to any changes in pixel intensity distribution, due to the heavy integration of histogram comparisons. With additional pre-processing techniques, such as bi-lateral filtering to preserve edges and apply noise-reducing smoothing, we could better our performance and response to troubling augmentations currently not accounted for.

References

[1] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[2] Y. Zhao, Y. Zhai, E. Dubois, and S. Wang, “Image matching algorithm based on SIFT using color and exposure information,” *Journal of Systems Engineering and Electronics*, vol. 27, no. 3, pp. 691–699, 2016.

[3] “Feature matching,” *OpenCV*. [Online]. Available: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html. [Accessed: 2-Apr-2023].

[4] P. Chhikara, “Intuition and implementation of Non-Max suppression algorithm in object detection,” *Medium*, 01-Feb-2022. [Online]. Available: https://towardsdatascience.com/intuition-and-implementation-of-non-max-suppression-algorithm-in-object-detection-d68ba938b630. [Accessed: 02-Apr-2023].

[5] “Histograms,” *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d6/dc7/group__imgproc__hist.html. [Accessed: 08-Mar-2023].

[6] “Histogram comparison,” *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d8/dc8/tutorial_histogram_comparison.html. [Accessed: 2-Apr-2023].

[7] P. Marimuthu, “Image contrast enhancement using clahe,” *Analytics Vidhya*, 17-Aug-2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2022/08/image-contrast-enhancement-using-clahe/#:~:text=Contrast%20Limited%20AHE%20(CLAHE)%20is,high%20accuracy%20and%20contrast%20limiting. [Accessed: 06-Apr-2023].