NARUTO CHARACTER RECOGNITION AND ANALYSIS

Joshua Teichroeb Jacob Erickson

CSC420 Fall 2019, University of Toronto

E-mail: joshua.teichroeb@mail.utoronto.ca, jacob.erickson@mail.utoronto.ca

Github Repository: https://github.com/jaruserickson/naruto-cv

1 Introduction

Naruto is a popular Japanese animated show which holds a special place in many hearts around the world. Given this fact, along with the fact that my partner and I both happened to take interest in the show out of coincidence, he and I decided that it would be interesting to do a project based on some aspect of the show. Since anime (i.e. a type of Japanese animated film) character recognition seemed like a relatively untouched topic, we decided to make it the basis for our project. On top of this, we added a feature matching component, namely that of detecting *Naruto* village symbols, which appear throughout the show as icons on character's headbands. And thus the project was born!

The project was split into two parts, with Jacob Erickson working mostly on dataset creation and character recognition, and myself working on the feature matching component and a few extra components such as image processing and development of a UI. Originally, the extra component was going to be homography estimation of the direction in which the characters were facing, based on the position of the symbol on their headband, however, due to the unforeseen difficulty of the symbol detection process, we decided that any homography estimation would only be inaccurate, as well as mostly meaningless, since the symbol detection already covers rotation. The new image processing component involves the implementation of Canny edge detection, which was used in the symbol detection algorithm.

The Github repository for this project can be found at https://github.com/jaruserickson/naruto-cv. The code dealing specifically with what will be talked about in this paper is mostly located in the `detect_symbols` folder, while the image processing component (namely Canny) is located in the `imgproc` folder. Finally, the UI component is located in the `app` folder. All of these are components which were mainly or wholly written by myself. More details may be found on the Github page itself.

The symbol detection problem we have defined is a fairly unique problem. These symbols are hand-drawn and thus vary from one moment to the next, making for quite the dilemma, considering that feature matching techniques are usually designed to detect static, unchanging objects. Furthermore, we also wish to detect the rotation and scale of the symbols, and thus will need to consider methods which are invariant to these transformations. Finally, the speed of the algorithm will also play a factor in the choices we make during implementation, as a perfect implementation would, of course, run at the frame rate of an episode of *Naruto*.

My work makes significant mention of SIFT, by David G. Lowe [1], and a generalized version of the Hough Transform, by D. H. Ballard [2]. Apart from these works and the course slides, most of the design and implementation was completed by myself through experimentation.

2 Methods

2.1 SIFT Matching

One of the most obvious choices for detecting objects at different scales and rotations is to use SIFT features. Because we can treat SIFT descriptors as vectors, and match them using distance measures, they provide an easy way to match a template object with a scene. For this reason it was the first algorithm we chose to experiment with. However, before we look into how we implemented SIFT features into our algorithm, we will present an overview of the SIFT algorithm itself.

SIFT uses an approximation of the Laplacian of Gaussian (essentially a "blob" detector) at different scales to find potentially interesting points in an image. This is done using a difference of Gaussians technique which speeds up the detection along the different scales. The variances, or sigma values, of the Gaussians determine the size, or radius, of each potential keypoint, giving us a 3D representation of the score for each keypoint that includes scale along with location. Maxima on this 3D grid then give us scale-invariant keypoints. Finally, descriptors are created at each of the keypoints using a histogram of gradient directions, with the most dominant direction being assigned to the keypoint itself as its orientation; this makes the keypoint rotation invariant as well.

To take advantage of the predicting power of SIFT, I wrote a script to detect SIFT keypoints and descriptors in a selected folder of symbol images which I had set aside as our templates for matching. These keypoints and descriptors were then hand-chosen (using the script GUI) to remove outliers and saved to file. The goal was then to use these saved features to find the symbols in any given scene taken from the show *Naruto*. However, there may be many of any given symbol in a scene. Thus, a logical solution would include a map of scores indicating the potential of a symbol being at each location in the scene. Furthermore, this map might also include rotation and scale so as to include the information given by the SIFT keypoints. Keeping this in mind, I decided to track the scores of the positions of each symbol by using a Hough-style accumulator, whose parameters would be location (X and Y axis), scale, and orientation. Using this accumulator and taking the maxima, we can then determine the positions of the symbols in the given image.

What is left now is to determine how to obtain the predicted positon of a symbol, given that a keypoint in the image matched with a keypoint in the template symbol. That is, we wish to find the location, scale, and orientation predicted by a given keypoint match. Let's define the parameters of a SIFT keypoint as (x, y, s, ϕ) where the terms represent the 2D location, scale, and orientation respectively. Then we will define a symbol as the set of SIFT keypoints of the symbol, stored relative to a reference point chosen as the location of that symbol. Then for a symbol with reference point (cx, cy), its keypoints will store the distance to the symbol reference point (r), the angle from the keypoint to the reference point (α), and the scale and orientation of the keypoint itself, (s and ϕ). Figure 1 shows the relationship of a symbol keypoint to its reference point, and how to compute the required values from the original SIFT keypoint values.

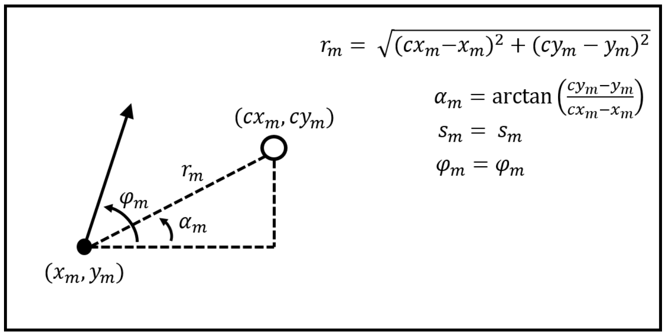
$r_m = \sqrt{(cx_m - x_m)^2 + (cy_m - y_m)^2}$
$\alpha_m = \arctan\left(\frac{cy_m - y_m}{cx_m - x_m}\right)$
$s_m = s_m$
$\varphi_m = \varphi_m$
(cx_m, cy_m)
r_m
φ_m
α_m
(x_m, y_m)

Figure 1: SIFT Keypoint

Finally, we can compute the predicted values of the symbol location, scale and orientation (cx, cy, s, and θ) given that a symbol keypoint (indexed m) matched with a scene keypoint (indexed i) by examining Figure 2.

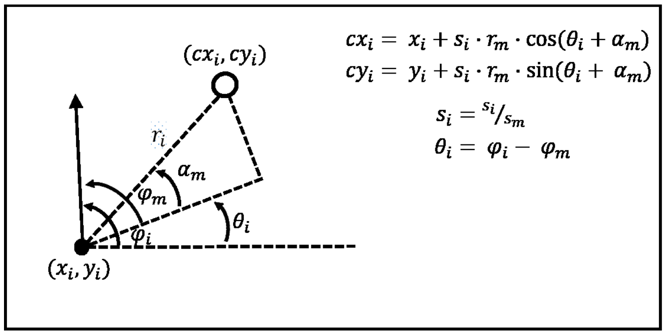
$cx_i = x_i + s_i \cdot r_m \cdot \cos(\theta_i + \alpha_m)$
$cy_i = y_i + s_i \cdot r_m \cdot \sin(\theta_i + \alpha_m)$
$s_i = {}^{s_i}/_{s_m}$
$\theta_i = \varphi_i - \varphi_m$
(cx_i, cy_i)
r_i
α_m
φ_m
φ_i
θ_i
(x_i, y_i)

Figure 2: SIFT-Based Prediction

The algorithm is described in detail as follows (see also Listing 1):

Step 1. Initialize symbol template keypoints and descriptors by hand-choosing SIFT inlier keypoints on a dataset of symbol images.

Step 2. Run SIFT on given input image and compute the squared distance between each keypoint descriptor with the descriptors of the saved template.

Step 3. Choose matches as those keypoint pairs whose computed distance is below some threshold.

Step 4. Compute the predicted location, scale, and orientation of the symbol of each keypoint match, assuming that the match is correct, and increment the accumulator at this position in the parameter space.

```
result = sift(image)

for kp, desc in result $(x_i, y_i, s_i, \phi_i)$, $(d_i)$:
    for kp, desc in table $(r_m, \alpha_m, s_m, \phi_m)$, $(d_m)$:
        dist = $\|d_i - d_m\|_2^2$

        if dist < thresh:
            $\theta = \phi_i - \phi_m$
            $s = s_i / s_m$
            $cx = x_i + s_i * r_m * \cos(\theta_i + \alpha_m)$
            $cy = y_i + s_i * r_m * \sin(\theta_i + \alpha_m)$
            acc[$cx, cy, s, \theta$] += 1
```

Listing 1: SIFT Matching Algorithm

2.2 Generalized Hough Transform

Our second approach was an optimized version of the Generalized Hough Transform, an algorithm proposed by D. H. Balard. The generalized Hough transform uses edge orientation to predict the position of an arbitrary shape in the same way that a Hough line detector uses the position of an edge pixel to predict the parameters of the line. In the general case, however, the parameters are the X and Y coordinates, and in our case, the scale and orientation as well. We will again first look at the general algorithm before explaining how we used it in our implementation and what modifications we made.

The input to the generalized Hough Transform is a grey, or binary, image of edges. The algorithm keeps a table for the shape that is to be detected, such that for each edge, or boundary, pixel in the referenced shape, there is a mapping from its orientation to the offset of that edge pixel to the shape's reference point. That is, each possible edge orientation has a mapping in the table to all the offsets in the shape from its reference point to a pixel of that edge orientation. This table is referred to as the R-table, and it can be visualized in Table 1, where ϕ is some edge orientation, and each (r, α) pair is an offset of a point with edge orientation ϕ to the shape reference point (where r is the offset distance, and α is the angle). Once this table is formed, the object location is predicted by considering all edges in the input image, looking up the possible pixel offsets using the orientation of the edges, then incrementing the accumulator at all these possible offsets. The maxima of the accumulator is guaranteed to be the location of the shape in the image, if it exists [2]. Furthermore, scale and rotation may be added to the accumulator's parameter space by considering that all the input edges could also be rotations/scales of the shapes edges. This is done by looping over all the scales and rotations and computing the new offset as the rotated and scaled version of the pre-computed table (See Listing 2).

R-Table	ϕ	(r, α)
	ϕ_1	(r_1^1, α_1^1), (r_2^1, α_2^1), ...
	ϕ_2	(r_1^2, α_1^2), (r_2^2, α_2^2), ...
	ϕ_3	(r_1^3, α_1^3), (r_2^3, α_2^3), ...
	$\dots$	$\dots$

Table 1: Original Hough Table

```
for each edge in image $(x, y, \phi)$:
    for each entry in table at $\phi$ $(r, \alpha)$:
        for each scale $(s)$:
            for each rotation $(\theta)$:
                $cx = x + s * \cos(\theta + \alpha)$
                $cy = y + s * \sin(\theta + \alpha)$
                acc[$cx, cy, s, \theta$] += 1
```

Listing 2: Generalized Hough Algorithm

We use the variant of the generalized Hough Transform with the scale and rotation parameters included. This makes the detection both scale and rotation invariant, but it also slows down the algorithm by a significant amount. However, I have made some optimizations which greatly increase the algorithms speed, without sacrificing any accuracy, along with a few which do sacrifice some of the accuracy which we do not need in order to solve our problem.

One thing to notice when examining the for loops in the algorithm in Listing 2 is that most of the work done by the two inner most loops can be done outside the loops. That is, we need only compute $s * \cos(\theta + \alpha)$ and $s * sin(\theta + \alpha)$ once for each s, θ, and α; and this can be done upon initialization of the R-Table. This means that instead of storing the (r, α) pairs in the R-table, we can compute all the necessary accumulator indexes (i.e. $dx = s*\cos(\theta+\alpha)$, $dy = s*\sin(\theta+\alpha)$, s, θ sets) and store them in the R-table instead, thus eliminating the need to compute these indexes every iteration of the loops (notice here that we store dx and dy which are offsets from x and y, rather than cx and cy themselves). Although this results in a larger R-table, and increases the necessary memory needed to store it if you don't initialize it every time on start up, as long as the number of possible values of s and θ are not too great, the increase will not be too costly. This optimization, although not actually decreasing the time complexity, will result in dramatic increases in speed.

The new R-table will look like Table 2, while the new algorithm will look like Listing 3. The steps of our matching algorithm in particular is as follows:

Step 1: Compute modified R-table for each of the template symbols and store them in a file.

Step 2: Compute Canny edges and edge orientations of input image.

Step 3: Loop over edges and increment accumulator at all indexes given by the R-table for that edge orientation.

R-Table	ϕ	(r, α)
	ϕ_1	$(dx_1^1, dy_1^1, s_1^1, \theta_1^1)$, $(dx_2^1, dy_2^1, s_2^1, \theta_2^1)$, ...
	ϕ_2	$(dx_1^2, dy_1^2, s_1^2, \theta_1^2)$, $(dx_2^2, dy_2^2, s_2^2, \theta_2^2)$, ...
	ϕ_3	$(dx_1^3, dy_1^3, s_1^3, \theta_1^3)$, $(dx_2^3, dy_2^3, s_2^3, \theta_2^3)$, ...
	$\dots$	$\dots$

Table 2: New Hough Table

```
for each edge in image $(x, y, \phi)$:
    for each entry in table at $\phi$ $(dx, dy, s, \theta)$:
        acc[$x + dx, y + dy, s, \theta$] += 1
```

Listing 3: New Hough Algorithm

2.3 Canny Edge Detector

I also implemented my own Canny edge detector from scratch, using only the Numpy library as a base. This was done because the OpenCV version did not output the edge orientations needed by the Hough algorithm, and so computing it alongside Canny was inefficient as Sobel filters would be passed along the image twice, and Canny also uses some measure for the edge gradient direction, which I wanted to take advantage of.

The Canny algorithm is summarized as follows. First, Sobel filter are passed along the image to compute the X and Y derivatives. Then, the gradient magnitude and direction is computed from these derivatives, where $\nabla = \|dx^2 + dy^2\|$ and the gradient orientation is $\arctan(dy/dx)$. After these values are computed, non-maxima suppression is used to thin edges by only keeping those edges which are stronger than their neighbours. Finally hysteresis is used by taking two thresholds, keeping all those strong edges above the high threshold, and only those edges above the low threshold which are connected to one of the stronger edges.

3 Results and Discussions

3.1 Detection Accuracy

3.1.1 SIFT Matching

The accuracy of our SIFT matching algorithm was quite poor. Although SIFT is very good at detecting interesting features in most images, its inability to detect enough interest points in the symbols we wanted to detect, and its reliance on feature points being distinct and constant caused its downfall.

During the initial phases of testing, SIFT performed well, however, as soon as we advanced to more realistic test images, it began to fail. The tests where SIFT performed well were when we used the template symbol as the test image, and only rotated, scaled or distorted it slightly. In these cases the symbol was still identical to the template, or at the very least, most of its features still were (in the case of the distorted image), and so SIFT was able to accurately detect the symbol. However, once new test images were brought in which contained different instances of the logo (e.g. from different frames or episodes of the show), then slight differences in the symbol caused SIFT to fail.

The number of interest points that SIFT would detect on our template symbols was usually between 10 and 20. Due to the simplicity of the symbols, and how generic the detected points were, this was not enough to find the symbol in images which contained many other objects. Because of this, false positives occurred more than correct predictions, and for this reason, we abandoned the SIFT approach.

3.1.2 Generalized Hough Transform

The generalized Hough transform performed much better than its counterpart. Hough was able to detect symbols consistently in images up to as large as a character's face. These images contained examples where the symbol was at slight angles, as well as some where the symbol was partially occluded, some where the viewpoint of the image was not perpendicular to the symbol's surface, and also in low resolutions. Failure cases included those where the symbol was largely occluded, or at an extreme angle, when other objects in the scene resembled the symbol too closely, when the symbol differed largely from its template, or in symbols that were too generic. See Figure 5 under section **Figures** for a sample of good as well as bad tests of our final algorithm.

The increased number of points used to predict the locations of the symbols allowed the Hough algorithm to outperform the SIFT based method. Using all the edges in the template gave the algorithm enough information to perform a meaningful search in the input images and find the symbols. However, problems still exist in many cases, and there is still much room for improvement. One of the weaknesses of the Hough based technique is that at large angles, the algorithm will completely miss (see figure: row 3, column 3). In these cases it is unlikely for the algorithm "as is" to ever detect the symbol correctly, for the offsets computed for each edge are likely placing the reference point of the symbol in a wide variety of places, leaving the score much to spread out over the accumulator to produce a high peak. Another common occurrence are false positives where, especially in the case of the leaf symbol, the symbol is detected on something that has a similar shape to the symbol (see figure: row 3, column 4). These failure cases are hard to examine, but a possible explanation might be that too much weight is being placed on generic features of the symbol, causing other generic objects to be detected in its place when the actual symbol has low confidence. These are areas that can still be explored for solutions.

Some of the failure cases are more indicative of the nature of the problem itself than of issues with the algorithm. Differences in a symbol from one image to the next and extremely simple and ununique symbols caused many missed detections and don't seem to be particular to the algorithm. Symbols like the mist symbol (Figure 3: column 5) make detection incredibly difficult, while symbols that are not constant immediately imply unpredictability. Nevertheless, we still get reasonable results on symbols which have quite a large amount of variance. To get a feel for the amount of variance in the symbols that our algorithm is able to handle, you may refer to Figure 4, which includes a number of symbols which were all detected with reasonable accuracy.

3.2 Timing

3.2.1 Generalized Hough Transform

Currently, it takes 5-10 seconds on average for our final algorithm to process an image and detect a symbol. This differs significantly from the initial version which took up to and above 1 1/2 minutes per image, making the final version approximately 10 times faster! The initial version had 3 for-loops, similar to the algorithm in Listing 2, except for one loop which was replaced with a Numpy operation. The final version now uses the modified R-table as well as my own Canny implementation, and it also cuts the possible range of angles allowed by the symbols to the range [-45,45] (degrees). The last optimization was made under the assumption that detecting a symbol at more than a 45 degree angle was unlikely, and not worth the extra processing time.

4 Main Challenges

First, I will note that in the original proposal, our goal was to have symbols be detected throughout the entire frame of an episode of the show, and that they were to be classified as well. However, the final state of our algorithm only detects given symbols on images that contain no more than a character's face. The reasons for this will hopefully become clear in the following discussions.

4.1 Varying Symbols

As was briefly mentioned earlier, unconstant symbols caused a variety of issues for detection. While designing the problems that we wanted to solve, we were under the impression that detecting symbols in an animated show, or more particularly, in *Naruto*, would be a similar task to finding logos in the real world, which always take the same form. However, because these shows are drawn images, rather than perfectly manufactured logos in the real world, they may differ from time to time, and in this case especially, it would seem that the variance over which they differed was quite high. This made the problem of detecting symbols a very difficult task, without even considering how to classify them, and so once we realized this, we decided that since the purpose of the final algorithm was to run on top of the character face recognition algorithm, it wasn't entirely necessary that we classify the symbols if we were given the cropped image of the characters face as input, along with which character it was, as this would determine the symbol that needed matching.

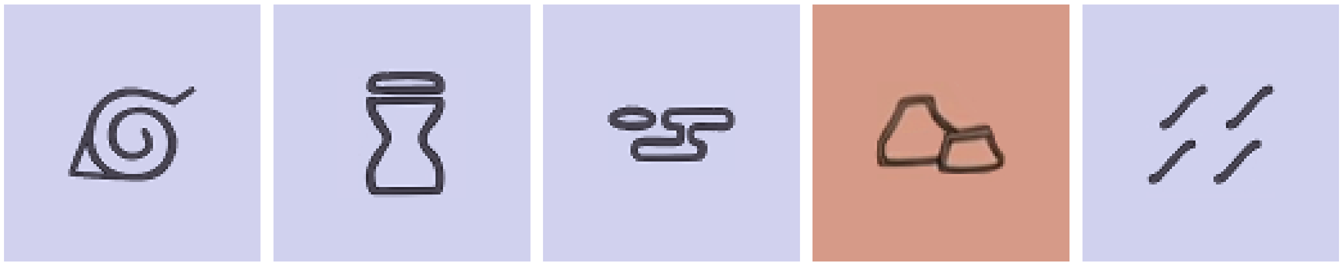

Figure 3: Symbol Templates

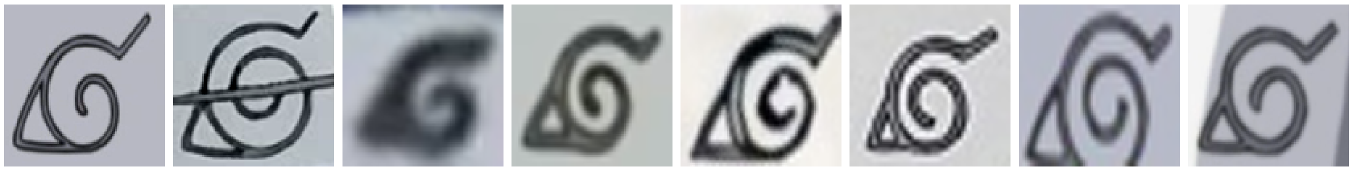

Figure 4: Detected Leaf Symbols

After focusing more on localization specifically, I was able to work with the algorithm to make it detect the symbols with reasonable consistency. Having a more specific goal allowed me to gain ground faster, and allowed me to produce better results.

5 Conclusion and Future Work

In conclusion, I believe the project was a success, considering the hurdles we had to overcome to get to the point where we are at right now. A few things that could be improved upon or continued: speed up detection even further, improve thresholding after detection is done and non-maxima suppression to be able to detect different numbers of symbols, and more. One thing I only just began to work on that might improve detection is a headband detector which locates the character's headband before sending the region of the headband to the symbol detector. If this idea works, the possibility of classifying symbols may even come back into the picture. Overall, much has been done, but there is still much that can be done.

6 Figures

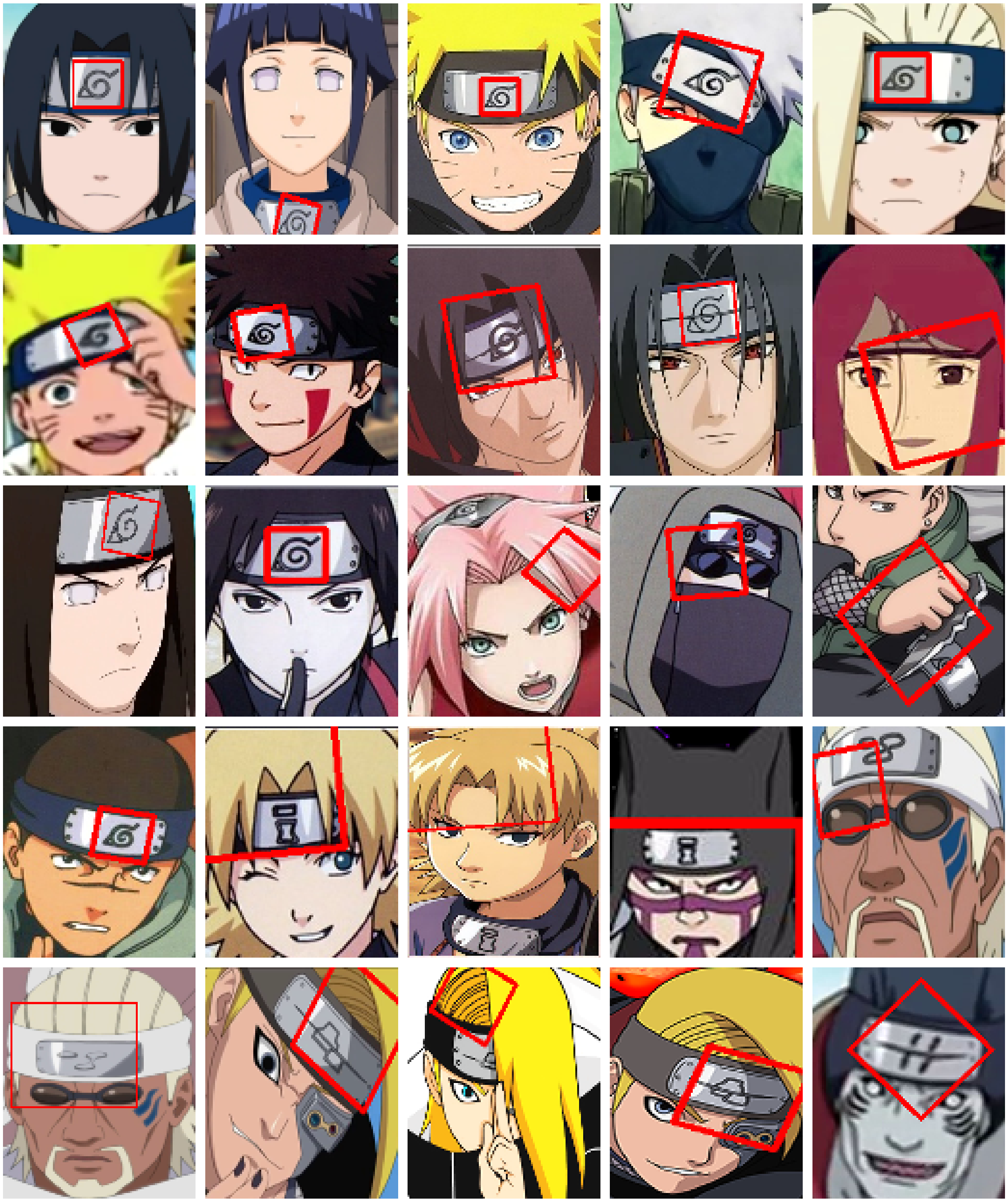

Figure 5: Results

References

[1] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” 2004.

[2] D. H. Ballard, “Generalizing the hough transform to detect arbitrary shapes,” 1980.

[3] J. Canny, “A computational approach to edge detection,” 1986.