Computer Vision (CM30080) Report - Filtering and Object Recognition

Thomas Vanner, Manvir Ubhi

1 Introduction

For this project, we chose to implement the tasks with Python. This was chosen over languages such as MATLAB and C++ due to the availability of libraries that provided functions similar to those of MATLAB, as well as Python being very robust. With available libraries such as OpenCV for image manipulation and Numpy for advanced mathematical functions that involve array manipulations, Python was suitable for the tasks at hand.

2 Image Convolutions

Image convolutions are a ubiquitous process in computer graphics. They involve taking a convolution matrix, or kernel, and applying it to an image of arbitrary dimensions as to produce some visual effect, be it a blur, sharpen, or edge detection.

An image convolution can be described mathematically with the following double summation formula

$$I' = (I * f)(x, y) = \sum_k \sum_l I(k, l) f(x - k, y - l) \tag{1}$$

where:

I = an arbitrary image $\in \mathbb{R}^{nxm}$
f = a 2D filter $\in \mathbb{R}^2$

Depending on the specific implementation, this process is performed by firstly padding the image with zeros, which is done in order to ensure that the image edge pixels can be filtered, since by default, no values are assigned to the space outside the image bounds. Then, the filter is inverted both horizontally and vertically, and is slid over the image, with the value of the target pixel being the center of the filter (kernel). The value of the target pixel is calculated by multiplying together the filter value with the corresponding pixel value in the image for each cell in the filter and summing all of these values. This is depicted in figure 1 .

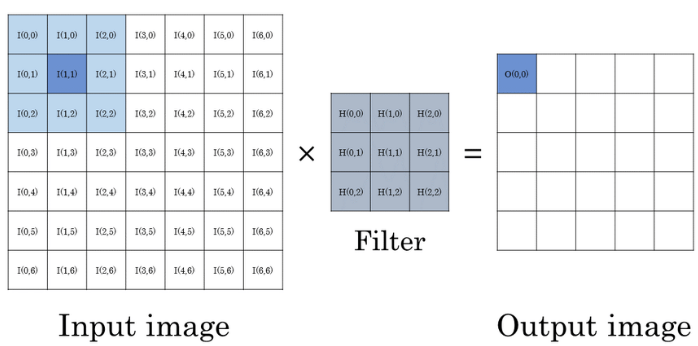

Figure 1: Image convolution being performed using double summation. Note the 0 values for space outside the image dimensions. (Baskin et al., 2017)

2.1 Implementation

We created a script which took in an RGB image of arbitrary dimensions, a filter/kernel, and outputted the image with the filter applied to it. This was achieved through the use of convolutions, more specifically double summation (as defined in Equation 1). To ensure that the image size remained the same after filtering, we padded space outside the image boundaries with zeros. Although we set this value to zero, it is ultimately down to the discretion of the calling code to set this. The code for this implementation can be found in Appendix B.

To test that the function produced the correct convolved image, we created a list of filters to test it with, namely: Gaussian blur, edge detectors and box blur, with different kernel sizes.

We checked our results by comparing the output of "filter2D" from OpenCV and our own function. The original image was included to check that our convolved image was visibly different. Appendix C contains the list of filters used. Figures 2 and 3 shows some of the visual results of our convolutions with one of the test images we used:

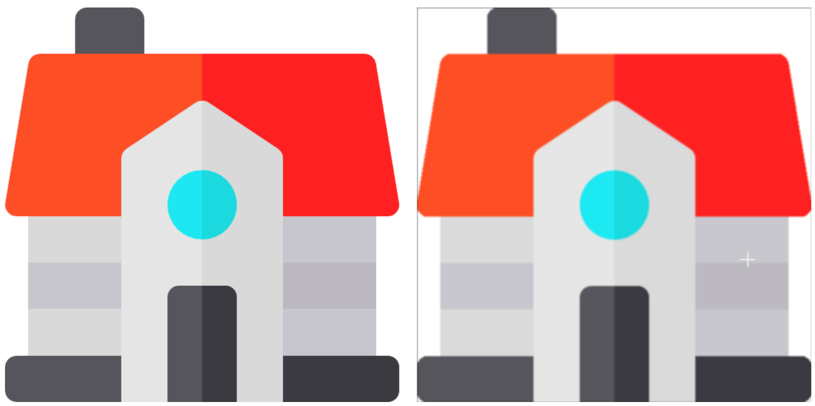

Figure 2: A test image that was used and the image after convolution with a box blur

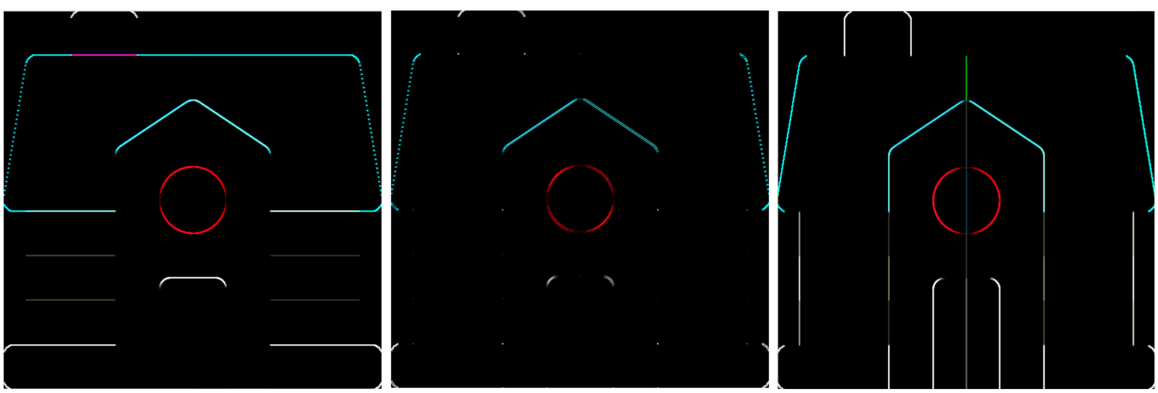

Figure 3: The test image being convolved with the following edge detectors from left to right: horizontal, diagonal, vertical

2.2 Further testing and evaluation

To further explore the effectiveness of our convolution algorithm, we performed tests with the same filters of differing dimensions. We conducted tests with a 3×3 and a 5×5 Gaussian blur, looking for differences in the amount of blurring and in runtime between the two filters. Figure 4 shows the differences in blur between the two convolved images, using the same test image as the previous tests.

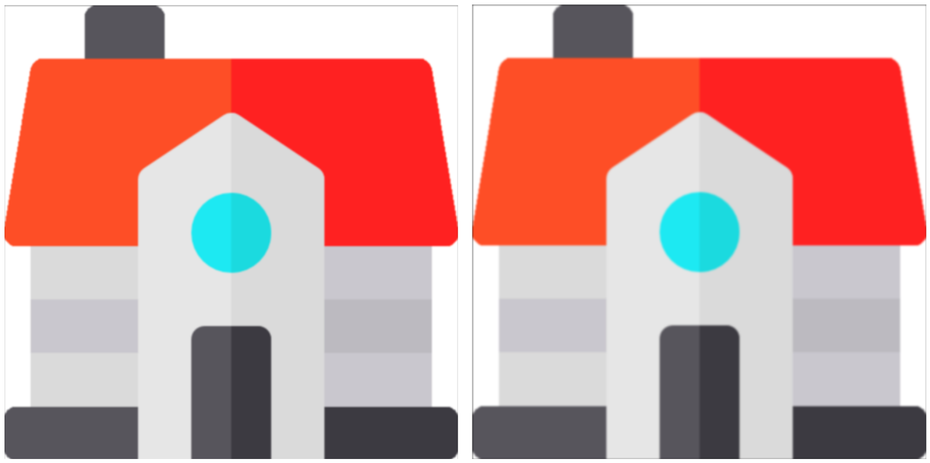

Figure 4: The test image after being convolved with 3x3 and 5x5 Gaussian blurs respectively. Note the stronger blur after the 5x5 convolution

The computational complexity of our implementation was $O(n^2k^2)$ per image, where n corresponds to the dimension sizes of the image and k corresponds to those of the kernel. This is slightly more efficient compared to the standard complexity of $O(n^4)$ for 2D convolution (Bing and Babu, 2019). Indeed, the runtimes reflected this complexity, with the 3×3 filter running in 8.45 seconds, and the 5×5 filter in 22.51 seconds. We also tested images of half dimensions (256×256), and found that the 3×3 filter had a runtime of 2.62 seconds. All of these tests were done over 100 iterations, and then averaged as to achieve a more accurate result.

3 Intensity-based Template Matching

Template matching is the process of matching some template image to parts of another image (i.e., an origin image) in order to identify any resemblances. The overall process of template matching comprises of the following steps: selecting a template to detect, sliding it through the origin image, and for every pixel in the origin image calculating the best template based on some similarity score (see section 3.3.1). The template matching generation and testing code can be found in Appendix G and Appendix H respectively.

3.1 Pre-processing

Before analysing images in the training data set, we first had to apply pre-processing steps to ensure appropriate levels of detail could be extracted from each image. This involved removing the white background from each image. This was done by loading each image with its alpha channel (RGBA), and replacing transparent transparent pixels with white ones. We then converted the image to greyscale, and applied a binary threshold function to the image as to find its general outline, and performed an erode function to remove any white noise surrounding the image. This process trumped the naive approach of simply replacing all white pixels with black ones, as it ensured that any pixels inside the actual image were minimally affected. This allowed us to clearly identify the object background and remove it, before converting the image back to RGB (see Appendix D).

3.2 Scaling and Rotating Templates

3.2.1 Gaussian Pyramids

In order for templates to be matched to ones in the test image data set, we had to ensure that we had an appropriate number of image scales, since template matching is a scale variant process. To do this, we generated a Gaussian Pyramid for each image in the training data set, which is the process of continually smoothing and subsampling an image in order to have multiple representations of a given image. This is to ensure enough scales are present when performing template matching, since it is a scale-variant process.

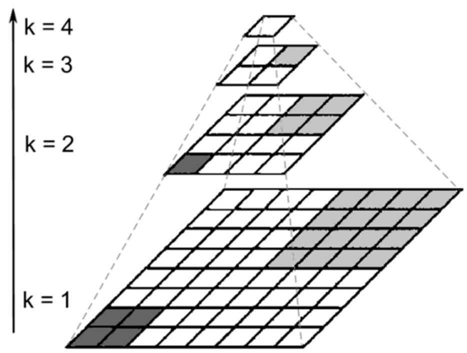

Figure 5: A visual representation of a Gaussian pyramid (Y. Wang et al., 2019)

To scale the images, we created a function that recursively scaled the image down by sampling every other pixel. As per the Gaussian pyramid, a low pass filter was applied prior to the subsampling process in order to prevent aliasing in the output. Appendix E shows the implementation of the Gaussian pyramid and subsampling.

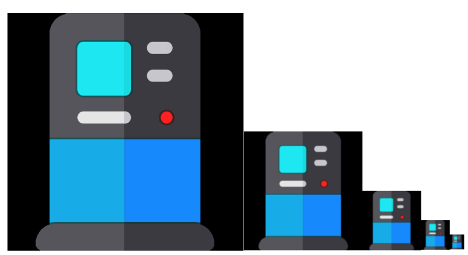

Figure 6: Example of Gaussian pyramid for the ATM class with 5 levels

3.2.2 Template Rotation

Since the testing data set contained objects in the training data set that were not oriented at their original angle, we had to take each template generated in the previous stage and rotate them by a specified number of increments. These rotations ranged from 0° to 330° with increments of 30°. The appropriateness of these values will be justified in section 3.3.4.

In order to rotate the templates, we first calculated the rotation matrix for it using the image's center of rotation and its rotation value. With the computed rotation matrix, we calculated the new image bounds by inspecting the rotation matrix as to find its bounding box, i.e. the new scale factors. Recomputing the image bounds was a vital step in the success of the template matching, since simply rotating an image without adjusting its dimensions would likely result in clipping, resulting in loss of image detail.

In order to compute the final rotated image, we applied an affine transformation to the image using OpenCV's "warpAffine" function, as to map the rotation matrix computed in the previous step to the original image, with it occupying its newly computed dimensions (see Appendix F).

3.3 Matching Templates

3.3.1 Intensity-based Template Matching

After training the template matching, we needed to test it to ensure that for each image, the best matching template was returned by the process described at the beginning of this section. This was done by examining each classes templates and finding the one with the highest confidence score with respect to the test image. Matched templates which had a confidence score below a specified threshold were then removed, with the remaining templates having their bounding boxes calculated and a NMS strategy applied. The returned boxes were then drawn around each object, and their corresponding class labelled.

3.3.2 Non-maxima Suppression Strategy

Due to the nature of template matching, the issue of false positives pose a real issue. In order to mitigate this issue, a non-maxima suppression (NMS) strategy was employed to ensure that only one object class per detection was identified, i.e. only one rectangle was drawn around each object. Given that, as discussed in the previous section, the template matching algorithm returns the top left corner of the matched object, along with the matched template dimensions, it is trivial to compute its bounding box.

With the set of bounding boxes for each test image, we were able to define a generic NMS algorithm accepting a set of inputs:

$B = \{b_1, \ldots, b_n\}$ where each b_i is the bounding box of each detected object
$S = \{s_1, \ldots, s_n\}$ where each s_i is the confidence score of each box
β = overlap confidence threshold such that $0 \leq \beta \leq 1$, $\beta \in \mathbb{R}$

The first step in the algorithm is to sort the input bounding boxes by their respective confidence scores, S, which allow us to identify the boxes which carry the highest probability of being the correct match. Once each box in B is sorted into a list ordered by its confidence score descending, B', is then iterated over. On each iteration, a bounding box b_j is selected and added to an initially empty list of candidate boxes C. With this selected bounding box, the Intersection Over Union (IOU) between it and every other $b_i \in B$ is computed as can be seen in figure 7. Let $g(x, y)$ denote the IOU of two boxes, and suppose $y = g(b_j, b_i)$. If $y > \beta$ then the box b_i is removed from B'. It's important to note that since B' is sorted by confidence score, the inequality, $S_{b_i} \leq S_{b_j}$, is satisfied, meaning that only boxes with high confidence scores remaining in B'. This whole process is repeated until there are no remaining entries in B'. The code for its implementation can be seen in Appendix I.

The NMS strategy significantly reduced the number of false positives being detected in the test images (figure 8). The false positives were likely due to the variety of different template scales and rotations that were constructed in the training phase, in particular templates of small scale. Due to their small size, they are more likely to match to part of an object in the test image.

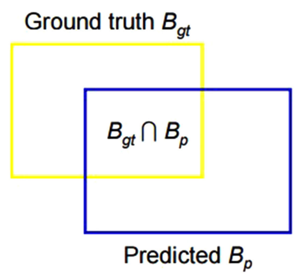

Figure 7: The intersection over union of two bounding boxes (Shao et al., 2018)

An important part of NMS is defining an appropriate threshold value β. Research tends to indicate that the range of values for the overlap threshold is $0.2 \leq \beta \leq 0.5$ (Rothe, Guillaumin, and Van Gool, 2015; D. Wang et al., 2019). We trialed this range of values, finding that $\beta = 0.2$ yielded the best results. This was likely due to the fact that small templates produced many false positives by matching with small parts of larger objects.

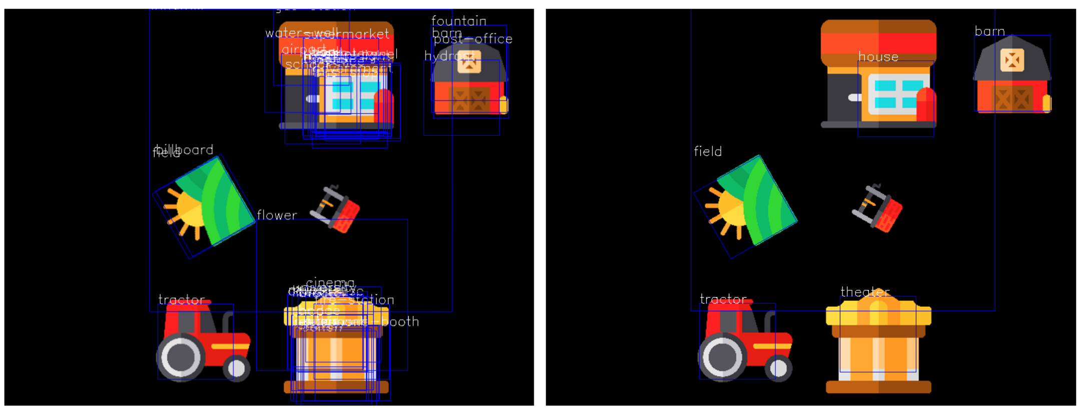

Figure 8: Before NMS (left) and after NMS (right).

3.3.3 Assembling Scaled and Rotated Boxes

After filtering the bounding boxes using our NMS strategy, we had to draw these boxes around each detected object. Since the computed templates consisted of many different scales and rotations, the drawn box had to match these variables, e.g. if the detected object had a rotation of 30°, a scale of 50%, and belonged to the class *car*, the box should reflect these variables.

Finding the correct scale for the box was trivial, since we already knew the information about the template being dealt with, and by extension, its bounding box, as discussed in section 3.3.2. Given the detected object's bounding box, the next task was to rotate it in accordance with its detected angle. In order to do this, we created a function (Appendix J) that took in the angle of rotation, the box side length, and outputted a rotated box ensuring it fit within the original bounding box, i.e. had the same vertical and horizontal dimensions. Figure 9 depicts the pointwise translations that need to occur to get the coordinates of the newly rotated square. Performing basic trigonometry gives us the following system of equations:

$$y = \frac{h}{1 + \tan(\theta)}$$

$$x = h - y$$

Solving for x then gives us the translation variable which to apply to each point. This, of course, assumes that the square is centered around $(\frac{h}{2}, \frac{h}{2})$ to begin with, and indeed, the output points are translated relative to this. However, it is trivial to offset these points to occupy any arbitrary 2D space by simply adding an offset to each original point.

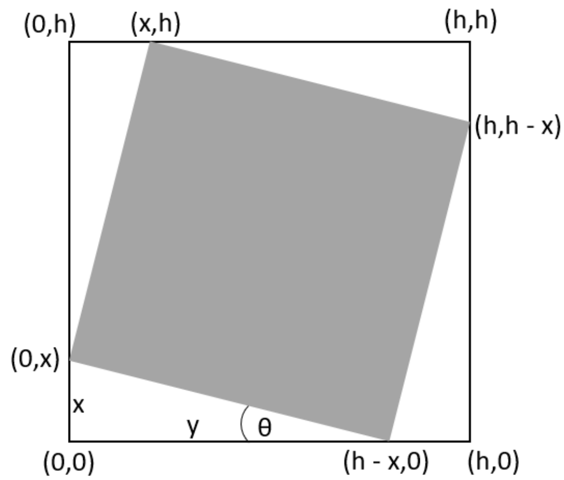

Figure 9: Square rotation within its bounding box

It's important to note that the angle of rotation θ always satisfies the inequality $0° \leq \theta \leq 90°$ since a square a rotated square resets its orientation every $90°$. Our implementation of this function adjusted the input angle accordingly.

The final step was to indicate which image class was detected for each object in each test image. As stated previously, we already knew which template was the best match, and we were able to simply write the name of the class above of the detected object (see Appendix K).

3.3.4 Justification of Hyperparameters and Evaluation

With the overall template matching process outlined, the next step was to find the optimal implementation parameters that resulted in the highest accuracy, i.e. true positives among the test images. Since template matching is just a means to detecting objects, its accuracy is largely dependent on the set of hyperparameters that surround it, namely the Gaussian kernel; Gaussian pyramid level; rotation increments. We found all of these parameters using the code in Appendix P

For our Gaussian kernel parameter settings we experimented with many different values. Prior to determining via empirical means, we first considered the theoretical implications of different parameter values. Due to the potential level of scaling and subsampling that was to occur during the Gaussian pyramid phase, it was important that we minimise the number of visual artefacts in the templates. An obvious way to achieve this was to select an appropriately large value for σ, as well as an appropriate dimension values. We tested a range of values, including both 3×3 and 5×5

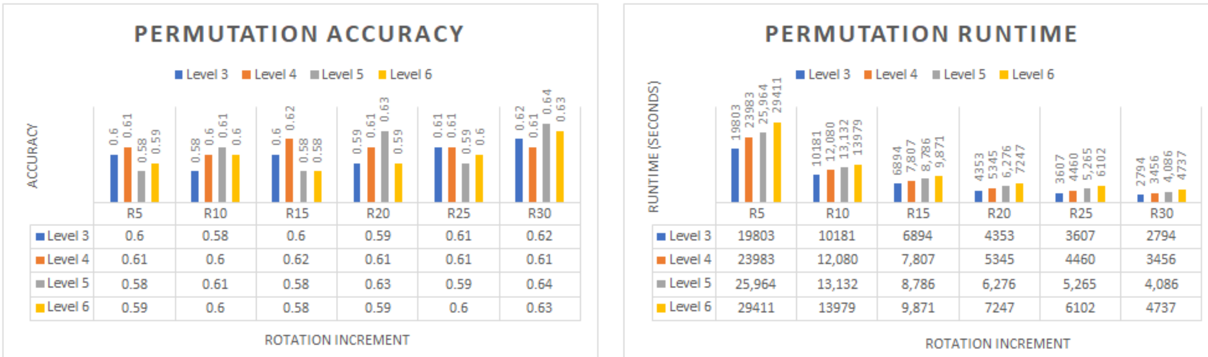

Figure 10: The accuracy and runtime of each template rotation-scale (level) permutation, where RX is the rotation increment.

kernel dimensions, as well as deviations of 5, 10, and 15 using code in Appendix M. We found that the 5×5 Gaussian with $\sigma = 15$ yielded the best results, i.e. number of true positives.

Deciding the number range of Gaussian pyramid levels to scale the templates by was a slightly easier task, since the test images only contain objects between certain sizes. Since template matching is scale variant, we started testing pyramids with levels ranging from 3-6, as these would ensure that each object size was accounted for. We found that a pyramid of size 5 gave the most accurate results in the given runtime. This was likely due to the fact that no images in the test set were smaller than a scale of 6, meaning that it did not increase the accuracy of the model, a sentiment which was echoed in our empirical findings. Using levels of 5 and 6 raised an interesting issue with the template matching process, namely the fact that many false positive matches were detected at these levels. This was likely due to the templates being small enough to match well with parts of larger objects. As such, we set a bias against template matches of levels 4 and above, effectively reducing their confidence score (appendix Appendix L), which greatly reduced the number of false positives being detected.

Computing the number of rotation increments to apply to each template was a comprehensive task, as there were many increments available. Since it would be infeasible to test every possible rotation-scale permutation, we tested rotations from 5° - 30° as to ensure enough angles were covered, whilst also taking into account the time taken to test each variable. We found that templates with rotation increments of 30 °gave the best results. This decision was ultimately a tradeoff between accuracy and runtime. The computational complexity of our template generation algorithm was O($g\ r\ i^2$) for each image, where g is the number of levels in the Gaussian pyramid, r is the number of rotations, and i is the image size. The complexity of our template matching algorithm is approximately O($d\ t$ O(m)) where d is the number of images in the testing data set, t are the total number of templates, and O(m) is the complexity of the given template matching implementation.

For the given data set with our parameters, 72,000 comparisons were made (50 template classes $\times$ 72 templates $\times$ 20 testing images). This meant that templates with rotation increments of 5° made 432,000 comparisons, which greatly increased the runtime of the algorithm with little change in accuracy. This further justifies our parameter choice, since they gave a good tradeoff between runtime and accuracy. This algorithm is clearly not efficient with respect to its overall accuracy, and highlights the problem of template matching as a process, i.e. the need to generate rotated and scaled templates in order to find matches.

Figure 10 shows the accuracy and runtime for each rotation-level permutation on the test data set.

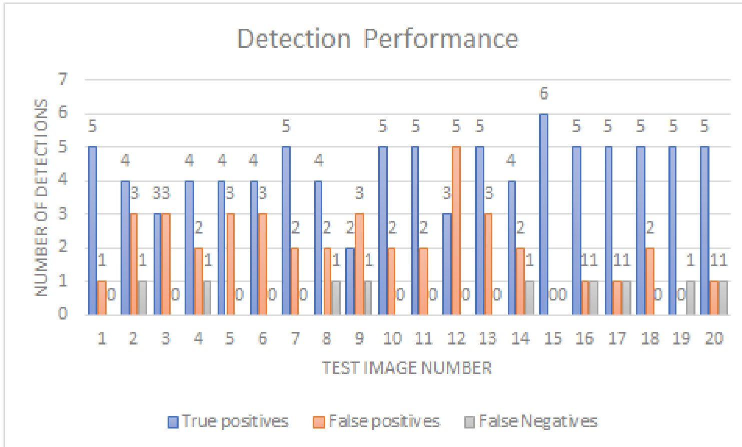

Figure 11: The detection rate of our template matching algorithm with a Gaussian pyramid level of 5, and rotations between 0° - 330° in 30° increments.

The rotation increment of 5 obviously had higher run times than other rotation increments due to the aforementioned reasons, i.e. increased number of total templates. Interestingly, as the rotation increments increased from 20 - 30 for each level, the runtime seemed to plateau. This was likely due to the number of templates not significantly changing, e.g. 15 vs 12 templates per level for increments of 25 and 30 respectively.

Our results clearly show that our chosen rotation-scale had the best overall mean accuracy and runtime, with values of 0.64 and 4086 seconds respectively, with figure 11 showing its detection rate. As can be seen, the algorithm struggled with several images, particularly image numbers 3,9, and 12. Whilst images similar to number 3 (figure 12) had a poor accuracy, their box rotation angles and scales suggest that the algorithm located the correct scale-rotation permutation, but the incorrect template class. This contrasts to image 2, which located false positives of incorrect box permutations completely. Looking at the bounding box size, it is possible that the object size did not fit any of the template scales, i.e. was in-between scale sizes. This is particularly apparent in the case of images 15 and 19, depicted in figure 13, in which the boxes generally fit the objects well, suggesting that they were of the correct scale. This suggests the algorithm has difficulty matching templates to objects which don't share the same scale, again highlighting the issue of template matching as whole, being a scale and rotation-variant process.

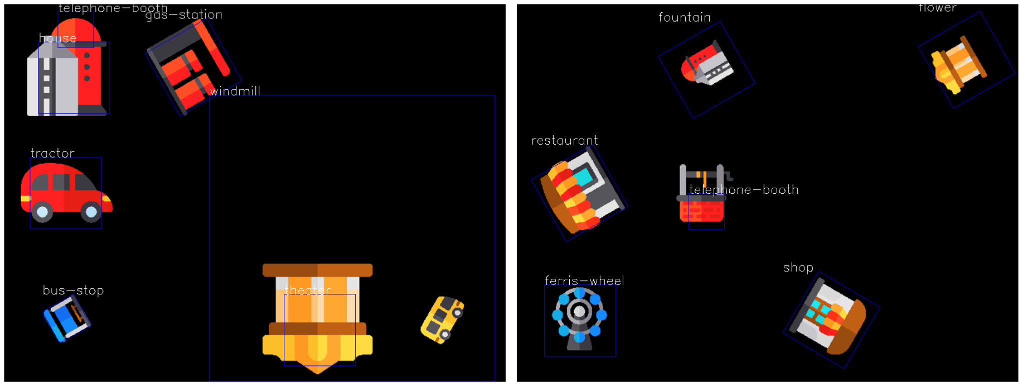

Figure 12: Test images 2 (left) and 3 (right) had poor performances.

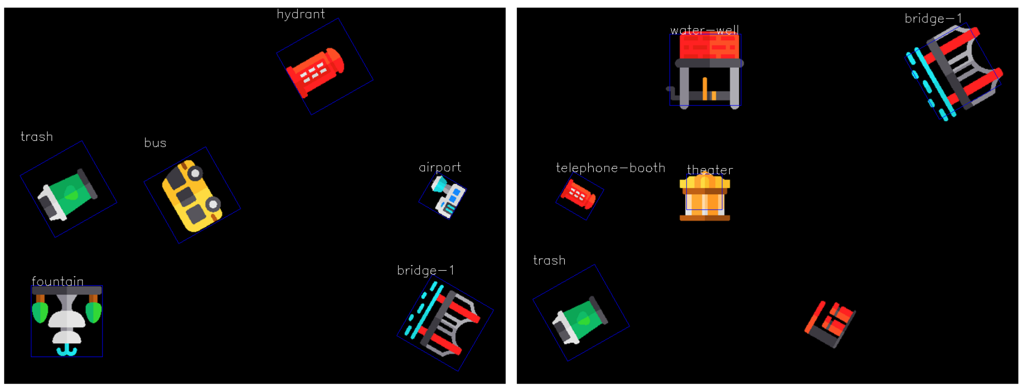

Figure 13: Test images 15 (left) and 19 (right) both had a good true-false positive ratio.

4 Conclusion

In conclusion, we created a python implementation of image convolutions and template matching. Our convolutions matched the results of the built-in function, albeit in a higher runtime. Our template matching had a mean accuracy of 0.64 with a reasonable runtime and space complexity.

Name	**Student ID**	**Contributions**	**Contribution Percentage**
Thomas Vanner	169257026	Template matching code, report writing	50%
Manvir Ubhi	169142191	Convolution code, report writing	50%

5 Appendices

A: The entry point for our program

```
# main.py

import argparse
import config
import sys
import os
import utils
import cv2
import numpy as np
from convolutions import ds_convolution, built_in_convolution
import template_matching as tm
import time

def main():
   ap = argparse.ArgumentParser()

   # Indicates the feature
   ap.add_argument("-f", "--feature", required=True,
             help="The feature to perform: 'convolutions' or 'template_matching'")

   # Indicates the feature - only relevant for template matching
   ap.add_argument("-m", "--mode", required=True,
             help="The mode of operation: 'train' or 'test'")

   # Indicates the image to perform the operation on (convolution)
   ap.add_argument("-i", "--image", required=False,
          help="The image to perform the convolution on")

   args = vars(ap.parse_args())

   feature = args['feature']
   mode = args['mode']

   if feature == config.CONVOLUTIONS_ARG:
      image_dir = args['image']
      if image_dir is None:
         raise Exception("Please enter the path to the image to perform the convolution
             on.")

      if not os.path.exists(image_dir):
         raise Exception("The image at the specified directory was not found.")

      convolute(image_dir)

   if feature == 'optimise':
      optimise_parameters()

   if feature == config.TEMPLATE_MATCHING_ARG:
      # Train template matching
      if mode == config.TRAINING_ARG:
         utils.delete_directory(config.TEMPLATE_OUTPUT_DIR)
         return tm.template_matching(config.TRAINING_DIR, config.TEMPLATE_OUTPUT_DIR)
```

```python
    # Test template matching
    images = tm.test_template_matching(config.TESTING_DIR, config.TEMPLATE_OUTPUT_DIR)

    # Write results to file
    for idx, image in enumerate(images):
        cv2.imwrite(config.RESULTS_DIR+'{}.png'.format(idx+1), image)
```

B: Convolution Function

```python
# convolutions.py

import cv2
import numpy as np
def ds_convolution(image, kernel, boundary_value=0):
    """
    Performs convolutions with the double summation (ds) formula
    Parameters:
    image: A n x m numpy array containing the RGB values for each pixel in the image
    kernel: A n x m numpy array representing the kernel filter
    boundary_value: The value pixels outside of the image boundaries should take; default to
        0.
    Returns:
    An nxm numpy array representing the filtered image
    """

    # Get the image and kernel dimensions
    i_width, i_height = image.shape[0], image.shape[1]
    k_width, k_height = kernel.shape[0], kernel.shape[1]

    # Define the output image
    filtered = np.zeros_like(image)
    kernel_sum = kernel.sum() if kernel.sum() > 0 else 1

    # Iterate each pixel in the image, left to right, top to bottom (x,y)
    for y in range(i_height):
        for x in range(i_width):
            weighted_pixel_sum = 0

            # Iterate over the kernel matrix for each pixel
            for ky in range(-(k_height // 2), (k_height // 2) + 1):
                for kx in range(-(k_width // 2), (k_width // 2) + 1):
                    # Compute image pixel coordinates with respect to the kernel, i.e. the ones
                        under inspection
                    pixel_y = y + ky
                    pixel_x = x + kx

                    # Set default pixel value to passed boundary value
                    pixel = boundary_value
                    # Update pixel value if it lies inside the image boundaries
                    if (pixel_y >= 0) and (pixel_y < i_height) and (pixel_x >= 0) and (pixel_x <
                        i_width):
                        pixel = image[pixel_y, pixel_x]

                    # Transpose local pixel coordinates (-k/2, k/2) back to valid array index
```

```
                (0,k-1)
            weight = kernel[ky + (k_height // 2), kx + (k_width // 2)]

            # Add weighted sum of current pixel to total
            weighted_pixel_sum += pixel * weight

        # Average the collected pixel values
        filtered[y, x] = weighted_pixel_sum

    return filtered

def built_in_convolution(image, kernel):
    """
    Performs convolutions with the built in library function
    Parameters:
    image: A n x m numpy array containing the RGB values for each pixel in the image
    kernel: A n x m numpy array representing the kernel filter
    Returns:
    An nxm numpy array representing the filtered image
    """
    return cv2.filter2D(image, -1, kernel)
```

C: Convolution Test Filter List

```
# main.py
def convolute(image_dir):
    kernels = {
        'identity': np.array([[0,0,0],[0,1,0],[0,0,0]]),
        'v_edge': np.array([[-1,0,1],[-2,0,2],[-1,0,1]]),
        'h_edge': np.array([[-1,-2,-1],[0,0,0],[1,2,1]]),
        'd_edge': np.array([[-1,-1,2],[-1,2,-1],[2,-1,-1]]),
        'gaussian_blur': np.array([[1,2,1],[2,4,2],[1,2,1]]) / 16,
        'gaussian_blur_5x5':
            np.array([[1,4,6,4,1],[4,16,24,16,4],[6,24,36,24,6],[4,16,24,16,4],[1,4,6,4,1]])
            / 256,
        'box_blur': utils.generate_box_blur(3),
        'box_blur_5x5': utils.generate_box_blur(5),
        'sharpen': np.array([[0,-1,0],[-1,5,-1],[0,-1,0]]),
    }

    # Read in image
    image = cv2.imread(image_dir)

    # Guard against invalid image directory
    if image is None:
        raise Exception("The image at the specified directory was not found.")

    # Normalise image RGB values
    image = image.astype(float) / 255.0

    # Define the kernel
    kernel = kernels['gaussian_blur_5x5']

    # Apply convolution to image with written function and built in method
    conv = ds_convolution(image, kernel)
```

```
    built_in_conv = built_in_convolution(image, kernel)

    # Display each convolution side by side for comparison
    horizontal_concat = utils.concatenate_images([image, built_in_conv, conv])
    cv2.imshow('Convolutions', horizontal_concat)

    # Prevent window from destroying immediately
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

D: Pre-processing for Intensity-Based Template Matching

```
# template_matching.py
def pre_process_image(image):
    # Replace transparent background with white
    transparent_mask = image[:,:,3] == 0
    image[transparent_mask] = [255, 255, 255, 255]
    greyscale = cv2.cvtColor(image, cv2.COLOR_BGRA2BGR)

    return replace_pixels(greyscale)

def replace_pixels(image, colour_to_replace=255, with_colour=0):
    """
    Replaces all pixels of specified colour in the image with ones of another colour.
    Parameters:
    image: An n x m x 3 numpy array representing the image
    colour_to_replace: The rgb colour to replace
    with_colour: The rgb colour to fill in
    Returns:
    An n x m x 3 numpy array representing the new image
    """
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    ret, thresh = cv2.threshold(gray, 240, 255, cv2.THRESH_BINARY)

    image[thresh == 255] = 0

    image_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    erosion = cv2.erode(image, image_kernel, iterations = 1)

    return erosion
```

E: Template Matching Gaussian Pyramid Generation

```
# template_matching.py
def create_gaussian_pyramid(image, gaussian, depth=5):
    """
    Creates a gaussian pyramid for a given image
    Parameters:
    image: An n x m x 3 numpy array representing the image
    gaussian: an n x n numpy array representing the Gaussian kernel to apply
    depth: The depth of the pyramid, i.e. how many times to downsample
    Returns:
    An array containing the downsampled images
```

```
    """
    scaled_images = [image]

    for level in range(0, depth):
        sampled_image = subsample_image(scaled_images[level], gaussian)
        scaled_images.append(sampled_image)

    return scaled_images

def subsample_image(image, gaussian, sample_rate=2):
    """
    Subsamples an image
    Parameters:
    image: An n x m x 3 numpy array representing the image
    gaussian: an n x n numpy array representing the Gaussian kernel to apply
    sample_rate: The rate at which to sample the image by
    Returns:
    An array containing the downsampled image
    """
    # Apply low pass filter to image
    image_blur = built_in_convolution(image, gaussian)

    # Get image dimensions
    image_height, image_width = image_blur.shape[0], image_blur.shape[1]

    # Create an array to represent the sub-sampled image, i.e. n/2 x m/2 x 3
    subsampled_image = np.zeros((image_height // 2, image_width // 2, 3), dtype=np.uint8)

    i = 0
    # Sample the original image at the given rate
    for x in range(0, image_height, sample_rate):
        j = 0
        for y in range(0, image_width, sample_rate):
            # Sample blurred image at specified sample rate
            subsampled_image[i, j] = image_blur[x, y]
            j += 1
        i += 1

    return subsampled_image
```

F: Template Matching Rotation

```
# template_matching.py
def rotate_image(image, angle, adjust_boundaries=True):
    """
    Rotates an image
    Parameters:
    image: An n x m x 3 numpy array representing the image
    angle: The angle in degrees by which to rotate the image
    adjust_boundaries: Whether or not to adjust the image boundaries to prevent cut off
    Returns:
    An array representing the rotated image
    """

    # Get image dimensions
```

```
    image_height, image_width = image.shape[0], image.shape[1]
    image_center = (image_width // 2, image_height // 2)

    # Get the rotation matrix for the image
    rotation_matrix = cv2.getRotationMatrix2D(image_center, angle, 1)

    # Compute new dimension boundaries
    abs_cos = abs(rotation_matrix[0, 0])
    abs_sin = abs(rotation_matrix[0, 1])

    # Compute new image boundaries
    horizontal_bound = int((image_height * abs_sin) + (image_width * abs_cos))
    vertical_bound = int((image_height * abs_cos) + (image_width * abs_sin))

    # Realign image centre
    rotation_matrix[0, 2] += horizontal_bound / 2 - image_center[0]
    rotation_matrix[1, 2] += vertical_bound / 2 - image_center[1]

    # Rotate the image with computed matrix
    rotated_image = cv2.warpAffine(image, rotation_matrix, (horizontal_bound,
        vertical_bound))

    return rotated_image
```

G: Template Matching Model Training

```
# template_matching.py
def template_matching(data_dir, template_dir, pyramid_depth=5, rotations=None,
    gaussian=None):
    """
    Train the tempalte matching model by creating templates of specified rotations and depth.
    Parameters:
    data_dir: Testing image directory
    template_dir: The directory to write the templates to
    pyramid_depth: The depth of the Gaussian pyramid
    rotations: The list of rotation angles to apply to the templates
    gaussian: The gaussian kernel to apply to the image before downsampling
    """

    if rotations is None:
        rotations = [x for x in range(0, 360, 30)]

    if gaussian is None:
        gaussian = utils.generate_gaussian(5,5,15)

    image_names = utils.get_files(data_dir, extension='.png')
    for image_name in image_names:
        image = cv2.imread(data_dir + image_name + '.png', cv2.IMREAD_UNCHANGED)
        image_filtered = pre_process_image(image)

        # Create a Gaussian pyramid of given depth for each image
        pyramid = create_gaussian_pyramid(image_filtered, gaussian, pyramid_depth)

        # Create directory for class
        image_class = get_class_name(image_name)
```

```
        class_dir = template_dir + image_class + '/'
        utils.create_directory(class_dir)

        # Rotate each scaled image
        for scale_index, scaled_image in enumerate(pyramid):
            for angle in rotations:
                rotated_image = rotate_image(scaled_image, angle)

                # Save image to png file
                file_name = image_class + "-level" + str(scale_index) + "-rotation" +
                    str(angle) + ".png"
                cv2.imwrite(class_dir + file_name, rotated_image)

    return True
```

H: Template Matching Model Testing

```
# template_matching.py
def test_template_matching(testing_dir, template_dir, threshold=0.5):
    """
    Detects templates in the testing images
    Parameters:
    testing_dir: The directory of the test images
    template_dir: The directory containing the templates
    threshold: The confidence score threshold
    Returns:
    An array of test images with boxes drawn around the detected objects
    """
    testing_images = utils.get_files(testing_dir, remove_extension=False)
    classes = os.listdir(template_dir)
    images = []

    # Iterate over testing images
    for image_num, test_image_name in enumerate(testing_images):
        test_image = cv2.imread(testing_dir + test_image_name)
        boxes = []

        # Iterate over each template class
        for image_class in classes:
            template_class_dir = template_dir + image_class+'/'
            templates = utils.get_files(template_class_dir, remove_extension=False)

            template_pick = {'confidence': 0}

            # Iterate over each template for the given class
            for template_name in templates:
                level, rotation = utils.get_template_config(template_name)
                template = cv2.imread(template_class_dir + template_name)

                width, height = template.shape[:2]
                if width > test_image.shape[0] or height > test_image.shape[1]:
                    continue

                # Perform the template matching, getting the confidence score and top left
                    corner
```

```
            res = cv2.matchTemplate(test_image, template, cv2.TM_CCORR_NORMED)
            min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

            # Bias confidence score for smaller
            max_val = bias_score(level, max_val)

            # Choose best match per template
            if max_val > template_pick['confidence']:
                template_pick = {
                    'image_class': image_class,
                    'permutation': template_name,
                    'res': res,
                    'confidence': max_val,
                    'top_left': max_loc,
                    'width': width,
                    'rotation': rotation
                }

        # Only consider templates which exceed the similarity threshold
        if template_pick['confidence'] > threshold:
            top_left = template_pick['top_left']
            width = template_pick['width']

            # Compute bounding box
            box = [top_left[0], top_left[1], top_left[0] + width, top_left[1] + width]

            # Store match info
            boxes.append({
                'box': box,
                'image_class': template_pick['image_class'],
                'permutation': template_pick['permutation'],
                'rotation': template_pick['rotation'],
                'width': width,
                'conf': template_pick['confidence']
            })

    # Assemble list of detection information, e.g. bounding box, class name, width etc.
    bounding_boxes = [(b['box'], b['permutation'], b['image_class'], b['conf'],
         b['width']) for b in boxes]

    # Non-maxima suppression strategy
    points = non_maxima_suppression(bounding_boxes, 0.2)

    # Draw boxes around the matched templates
    for point_temp in points:
        # Get box level and rotation info
        level, rotation = utils.get_template_config(point_temp[1])
        point = point_temp[0]
        top_left = (point[0], point[1])
        width = point_temp[4]

        p1, p2, p3, p4 = get_rotated_square_coordinates(rotation, width)

        # Draw the box around the object
```

```
            # Write the name of the class above the box
            image_class = point_temp[2]
            test_image = draw_box(test_image,[p1, p2, p3, p4] , top_left)
            test_image = write_text(test_image, image_class, top_left)

        images.append(test_image)

    return images
```

I: Non-Maxima Suppression Strategy

```
# template_matching.py
def non_maxima_suppression(box_configs, threshold):
    """
    Performs a non-maxima suppression strategy on the bounding boxes
    Parameters:
    box_configs: List of box configurations
    threshold: The IOU threshold
    Returns:
    An array containing the selected box configurations
    """

    # Get array of box confidence scores
    scores = np.array([t[3] for t in [b for b in box_configs]])

    # Get array of box coordinates
    boxes = np.array([t[0] for t in [b for b in box_configs]])

    # Every box must have an associated confidence score
    if len(boxes) != len(scores):
        return []

    if len(boxes) == 0 or len(scores) == 0:
        return []

    # Get coordinates of boxes
    boxes = boxes.astype("float")
    x1 = boxes[:,0]
    y1 = boxes[:,1]
    x2 = boxes[:,2]
    y2 = boxes[:,3]

    # Calculate area of each box
    area = (x2 - x1 + 1) * (y2 - y1 + 1)

    # Sort boxes by confidence scores desc
    ranked_boxes = np.argsort(scores)

    # Store list of candidate boxes
    candidate_boxes = []

    # Iterate through each box, removing ones which exceed the IOU of the most confident pick
    while len(ranked_boxes) != 0:
        # Look at each box left in the list
        candidate_index = len(ranked_boxes) - 1
```

```
        candidate_box = ranked_boxes[candidate_index]

        # Add this box to the list of candidate boxes
        candidate_boxes.append(candidate_box)

        # Get a list of all other boxes to compare
        compare_boxes = ranked_boxes[:candidate_index]

        # Get the greatest coordinates for top left of bounding box and smallest for bottom
            right
        x1_max = np.maximum(x1[candidate_box], x1[compare_boxes])
        y1_max = np.maximum(y1[candidate_box], y1[compare_boxes])
        x2_min = np.minimum(x2[candidate_box], x2[compare_boxes])
        y2_min = np.minimum(y2[candidate_box], y2[compare_boxes])

        # Calculate box dimensions >= 0
        width = np.maximum(0, x2_min - x1_max + 1)
        height = np.maximum(0, y2_min - y1_max + 1)

        # Compute the overlap between the candidate box and every other box
        overlap = (width * height) / area[compare_boxes]

        # Remove boxes which have an overlap exceeding the specified threshold
        filtered_boxes = np.where(overlap > threshold)
        eliminate = np.concatenate(([candidate_index], filtered_boxes[0]))
        ranked_boxes = np.delete(ranked_boxes, eliminate)

    return [box_configs[b] for b in candidate_boxes]
```

J: Bounding Box Rotation

```
# template_matching.py
def get_rotated_square_coordinates(angle, side_length):
    """
    Gets the coorindates of the rotated square
    Parameters:
    angle: The angle to rotate the square by in degrees
    side_length: The length of the side of the square
    Returns:
    An array of two value tuples representing the square corners from top left to bottom
        left clockwise
    """
    angle = int(angle)

    # Ensure that the angle of rotation is between 0 and 90 degrees
    relative_angle = math.radians(0)
    if angle < 90:
        relative_angle = math.radians(angle)
    elif 90 <= angle < 180:
        relative_angle = math.radians(angle - 90)
    elif 180 <= angle < 270:
        relative_angle = math.radians(angle - 180)
    elif 270 <= angle < 360:
        relative_angle = math.radians(angle - 270)
```

```python
    # Compute rotation factor
    y = (side_length) / (1 + math.tan(relative_angle))
    x = int(side_length - y)

    # Construct new coordinates
    new_coordinates = [
        (x, side_length),
        (side_length, side_length - x),
        (side_length - x, 0),
        (0, x)
    ]

    return new_coordinates
```

K: Box and Class Name Drawing

```python
# template_matching.py
def draw_box(image, points, offset=(0,0)):
    """
    Draw a box around an arbitrary point in image space
    Parameters:
    image: The image to draw the box on
    points: A four element list containing tuples representing the box corners
    offset: The offset for the corners
    Returns:
    The image with the boxes drawn
    """

    p1, p2, p3, p4 = points
    offset_x, offset_y = offset

    cv2.line(
        img=image,
        pt1=(p1[0] + offset_x, p1[1] + offset_y),
        pt2=(p2[0] + offset_x, p2[1] + offset_y),
        color=255
    )

    cv2.line(
        img=image,
        pt1=(p2[0] + offset_x, p2[1] + offset_y),
        pt2=(p3[0] + offset_x, p3[1] + offset_y),
        color=255
    )

    cv2.line(
        img=image,
        pt1=(p3[0] + offset_x, p3[1] + offset_y),
        pt2=(p4[0] + offset_x, p4[1] + offset_y),
        color=255
    )

    cv2.line(
        img=image,
        pt1=(p4[0] + offset_x, p4[1] + offset_y),
```

```python
        pt2=(p1[0] + offset_x, p1[1] + offset_y),
        color=255
    )

    return image

def write_text(image, text, origin):
    """
    Writes text onto an image
    Parameters:
    image: The image to write the text on to
    text: The text to write
    origin: Where to write the text to (point on the image)
    Returns:
    The image with the text written on
    """

    cv2.putText(
        img=image,
        text=text,
        org=origin,
        fontFace=cv2.FONT_HERSHEY_SIMPLEX,
        fontScale=0.75,
        color=(255, 255, 255)
    )

    return image
```

L: Confidence Score Biasing

```python
# template_matching.py
def bias_score(level, score):
    """
    Applies a bias to the confidence score based on the Gaussian pyramid level
    Parameters:
    level: The level of the template
    score: The confidence score
    Returns:
    The score with bias applied
    """
    if level == 1:
        score *= 0.8
    elif level == 2:
        score *= 0.8
    elif level == 3:
        score *= 0.7
    elif level == 4:
        score *= 0.6
    elif level == 5:
        score *= 0.4
    elif level == 6:
        score *= 0.3

    return score
```

M: Gaussian Kernel Generator

```python
# utils.py
def generate_gaussian(rows, columns, sigma=1):
    """
    Generates a gaussian matrix of arbitrary size.
    Parameters:
    rows: The number of rows for the Gaussian matrix
    columns: The number of columns for the Gaussian matrix
    sigma: The standard deviation
    Returns:
    The Gaussian matrix
    """
    output_matrix = np.zeros((rows, columns))
    output_matrix[rows // 2, columns // 2] = 1.0

    gaussian_matrix = cv2.GaussianBlur(output_matrix , (rows, columns), sigma,
        borderType=cv2.BORDER_ISOLATED)

    return gaussian_matrix
```

N: Miscellaneous Utility Functions

```python
# utils.py
import numpy as np
import cv2
import os
import math
from pathlib import Path
import shutil

def create_directory(name):
    Path(name).mkdir(parents=True, exist_ok=True)

def delete_directory(dir_path):
    shutil.rmtree(dir_path)

def concatenate_images(images):
    """
    Horizontally concatenates a list of n x m images
    Parameters:
    images: A list of n x m images
    Returns:
    An array representing a horizontal concatenation of the input images
    """
    horizontal_concat = np.concatenate(tuple(images), axis=1)

    return horizontal_concat

def generate_box_blur(size):
    return np.ones((size, size), np.float32) / (size * size)

def get_template_config(template_file, class_name=True):
    """
    Gets the level and rotation configuration for a given template file
```

```
    Parameters:
    template: The template file name
    class_name: Whether or not to include the tempalte class name
    Returns:
    A tuple containing its level, rotation, and if selected, class name respectively
    """
    config = template_file.replace('.png', '')
    level_start_index = config.find('level') + len('level')
    level_end_index = config[level_start_index:].find('-') + level_start_index
    level = config[level_start_index:level_end_index]

    rotation = config.find('rotation') + len('rotation')
    rotation_start_index = config.find('rotation') + len('rotation')
    rotation = config[rotation_start_index:]

    return (safe_int_cast(level), safe_int_cast(rotation))

def safe_int_cast(value, default=None):
    try:
        return int(value)
    except(ValueError, TypeError):
        return default

def get_files(directory, extension='.png', remove_extension=True):
    """
    Gets a list of files in a given directory
    Parameters:
    directory: The directory to analyse
    extension: The file extension to filter by
    remove_extension: Whether or not to remove file extension from file name
    Returns:
    The list of files names
    """
    if not os.path.isdir(directory):
        return []

    file_names = [name for name in os.listdir(directory) if name.endswith(extension)]

    if remove_extension:
        file_names = [name.replace(extension, '') for name in file_names]

    return file_names

def generate_gaussian(rows, columns, sigma=1):
    """
    Generates a gaussian matrix of arbitrary size.
    Parameters:
    rows: The number of rows for the Gaussian matrix
    columns: The number of columns for the Gaussian matrix
    sigma: The standard deviation
    Returns:
    The Gaussian matrix
    """
    output_matrix = np.zeros((rows, columns))
    output_matrix[rows // 2, columns // 2] = 1.0
```

```
    gaussian_matrix = cv2.GaussianBlur(output_matrix , (rows, columns), sigma,
        borderType=cv2.BORDER_ISOLATED)

    return gaussian_matrix

def write_to_file(file_dir, text):
    f = open(file_dir, "a")
    f.write(str(text))
    f.close()
```

O: Configuration Variables

```
# config.py
CONVOLUTIONS_ARG = 'convolutions'
TEMPLATE_MATCHING_ARG = 'template_matching'
TRAINING_ARG = 'train'
TESTING_ARG = 'test'

TRAINING_DIR = './input/Training/png/'
TESTING_DIR = './input/Test/'
TEMPLATE_OUTPUT_DIR = './data/templates/'

RESULTS_DIR = './data/results/'
```

P: Optimisation Function

```
# main.py
def optimise_parameters():
    """
    Runs the algorithm on different scale-rotation permutations
    """
    pyramid_levels = [x for x in range(3,6)]
    rotations = [[x for x in range(0, 360, rot)] for rot in range(5,35,5)]
    gaussian_parameters = [[5,5,15]]

    # Create results directory
    utils.create_directory(config.RESULTS_DIR)

    for level in pyramid_levels:
        for rots in rotations:
            for g_n, gaussian in enumerate(gaussian_parameters):
                # Compute/process parameters
                step_size = rots[1] - rots[0]
                row,col,dev = gaussian
                g = utils.generate_gaussian(row, col, dev)
                utils.delete_directory(config.TEMPLATE_OUTPUT_DIR)
                print('training rotation {} level {} gaussian
                    {}-{}-{}'.format(step_size,level,row,col,dev), rots, level)

                # Time how long the algorithm takes
                start = time.time()

                # Train templates on these parameters
```

```
        tm.template_matching(config.TRAINING_DIR, config.TEMPLATE_OUTPUT_DIR, level,
            rots, g)
        new_dir =
            config.RESULTS_DIR+'level{}-rot{}-g-{}-{}-{}/'.format(level,step_size,row,col,dev)
        utils.create_directory(new_dir)

        # Test these templates
        print('testing', rots, level)
        images = tm.test_template_matching(config.TESTING_DIR,
            config.TEMPLATE_OUTPUT_DIR)

        # Write results
        end = time.time()
        time_elapsed = end - start
        utils.write_to_file(new_dir+'time.txt', time_elapsed)

        # Write results to directory
        for idx, im in enumerate(images):
           cv2.imwrite(new_dir+'{}.png'.format(idx + 1), im)

return True
```

References

Rothe, R., Guillaumin, M., and Van Gool, L., 2015. Non-maximum suppression for object detection by passing messages between windows. Vol. 9003. Available from: https://doi.org/10.1007/978-3-319-16865-4_19.

Baskin, C., Liss, N., Mendelson, A., and Zheltonozhskii, E., 2017. Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform.

Shao, Z., Wu, W., Wang, Z., Du, W., and Li, C., 2018. Seaships: a large-scale precisely-annotated dataset for ship detection. *Ieee transactions on multimedia*, 20 (), pp.1–1. Available from: https://doi.org/10.1109/TMM.2018.2865686.

Bing, L. and Babu, J., 2019. "convolution theorem and asymptotic efficiency", a graduate course on statistical inference, pp.295–327.

Wang, D., Li, C., Wen, S., Chang, X., Nepal, S., and Xiang, Y., 2019. Daedalus: breaking non-maximum suppression in object detection via adversarial examples. *Arxiv*, arXiv–1902.

Wang, Y., Li, G., Yan, W., He, G., and Lin, L., 2019. Heterogeneity detection method for transmission multispectral imaging based on contour and spectral features.