Guid to files:

IPCV1_1_FINAL: Main codes to do the first part of the task

IPCV1_2 FINAL: Main codes to do the second part of the task

Others: codes to denoise the images

Part 1:

1.1:

First, we have images of models and scenes as our input images, and we use filters for denoising (such as Gaussian filters). Then, we use the SIFT method to extract key points and descriptors.

Next, we need to match these features to detect them, so we use the predefined FLANN matcher in OpenCV. FLANN (Fast Library for Approximate Nearest Neighbors) is a library that performs fast approximate nearest neighbor searches, making it efficient for large datasets. We use k-NN (k-Nearest Neighbors) search in a more optimal way.

We provide the descriptors of the scene and model images as input to the FLANN matcher to find matches between them. Now, using our function that takes a reference and a model image, we can find the reference in the model using the match function. We then use the findHomography function in OpenCV to find our homography transform matrix. Homography is a transformation matrix that maps points from one plane to another, useful for perspective correction.

There is a mask in homography. If it is None, it cannot find any key points of the reference in the model. For non-zero values, we continue, and using homography, we translate all the pixels of the reference. We can use this as a boundary box, though it is not a very reliable method if it is out of the image or larger than that. Therefore, we need to filter them and choose good boundary boxes.

Next, we have a function that, using for loop, attempts to find an instance of the reference in all scene images, applying the above functions to find detection. The detection indicates the location and a score showing the matches between them. We then use non-maxima suppression (NMS). NMS is a method used to filter out multiple detections by keeping only the best-scoring ones, thus reducing redundancy.

In NMS, we have lots of detections as input, many of which are not useful. This method helps us filter them by sorting them based on their score.

For the image scenes from 1-5, which are for part 1, this method works fine. However, to give us an intuition about part 2 of this project, we also tried this method for scenes 6 to 12 as well, but we could not achieve desired results after fitting the hyperparameters.

1.2:

To extract and identify all instances in images, we need a robust method. Here's an overview of our approaches:

First Solution:

We considered iterating over the process used in part 1: detecting instances, blurring the boxes, and repeating until no instances remain undetected. However, this method was discarded due to the intuition from our observations in part 1 that it would result in rejecting some instances.

Second Solution:

After testing and tuning hyperparameters in part 1, we knew our method could at least detect one instance of each product in the scene. Instead of focusing on SIFT (Scale-Invariant Feature Transform) and descriptor matching, we devised a method to accept all instances, considering image noise and low keypoint detection chances.

We think our model doesn't lose generality but mostly focuses on the problem. This method is completely general for use when we are facing some shelves and products to detect. Because: We want to detect the product on the shelves, and since with the method in part 1 of the project, we know at least one of each instance is found, and also due to the fact that the products next to each other on the shelves are similar to each other and thus they are similar in dimensions, we can use the same grid for the detection of the repeated item in the image since the grid size is the same.

1. Boundary Boxes: Using the method from 1.1, we identify boundary boxes.
2. Brute Force Algorithm: For detected instances, we use a brute force approach. Given that adjacent products are similar, their dimensions and homography matrices (used for projective transformations) are the same. We place a box next to each detected instance, moving it forward to find new detections using a dissimilarity function between instance pixels and box pixels (initially, we used the ZNCC score).
3. ZNCC Score: Zero-Normalized Cross-Correlation (ZNCC) score evaluates the similarity between two image patches.
4. Threshold and Intersection of Union: Boxes with a ZNCC score below the threshold are omitted, and those with a high intersection of union (IoU) with existing detections are also removed. IoU measures the overlap between two boundaries.
5. Grid Adjustment: We refine grid positions within a 10-pixel neighborhood (left/right) and a 5-pixel neighborhood (up/down), selecting the position with the best dissimilarity function.
6. Final Filtering: After calculating the ZNCC score between the grid and the model, we apply a final threshold to remove incorrect grids.

Steps Summary:

- Grid Creation: Generate initial grids.
- Filtering: Apply initial filters to grids.
- Dissimilarity Score Calculation: Compute dissimilarity scores.
- Threshold Application: Apply a threshold to scores.
- Grid Position Adjustment: Adjust grid positions within defined neighborhoods.
- Non-Maximum Suppression (NMS): Select the best score.
- Iteration: Iterate to find the matching product for each grid.

Why ZNCC?

ZNCC provided the best match for our purposes, despite challenges with images containing white milk boxes. We explored alternatives like SSIM (Structural Similarity Index Measure), color histograms, and normalized histograms. Combining color histogram results with SIFT descriptors and calculating the cosine similarity was less effective. Ultimately, we used an RGB version of SIFT, applying it to all three image channels, creating descriptors, and using a FLANN-based matcher (Fast Library for Approximate Nearest Neighbors), which worked better, except for scenes 9 and 12, which required additional denoising. After that, we again used ZNCC to choose or drop the grids since it is faster than the SIFT.

Highlights:

- The first solution might not be effective because it could miss some instances and be time-consuming. In our method, we used many grids, and managing these grids and their filters took a lot of time. We believe there are better algorithms for deriving grids, and changing the order of steps and filters could improve time efficiency, but we couldn't make these changes due to time constraints.
- The advantage of our method in a real-world scenario is its low rejection rate. By employing more hyperparameters, we can accept a greater number of grids and eliminate those with low similarity scores, thereby achieving the project's goal of maintaining a low rejection rate.

Denoising Part:

Choosing an appropriate filter for denoising is challenging, particularly when there is limited information about how the images have been processed. Filters such as blurs can smooth edges or inadvertently add information, complicating the selection process. Different noising methods require related denoising filters and suitable parameters for those filters.

Drawing on our knowledge of autoencoders, which are neural networks used for denoising images in the MNIST dataset, we employed a similar method. Autoencoders are trained by adding noise to input images and then training a deep learning model to reconstruct the original images from the noisy inputs. However, using the same network as MNIST proved ineffective due to the simplicity of the MNIST dataset. Consequently, we explored the DnCNN (Denoiser Convolutional Neural Network) as presented in recent literature. The DnCNN utilizes the

BSDS500 dataset, initially developed for segmentation tasks, for training (referenced: https://github.com/ocimakamboj/DnCNN).

First, we loaded the BSDS500 dataset and applied a Gaussian filter with random parameters to introduce noise. A Gaussian filter smooths an image by averaging pixel values with a Gaussian kernel. We observed that this approach effectively denoised the images.

Addressing Issues in Part 1.2:

To tackle the detection problems in scenes 9 and 12, specifically with white milk boxes featuring text on different colored backgrounds, the RGB SIFT (Scale-Invariant Feature Transform) method was insufficient. Recognizing that the dissimilarity function struggled with this scenario, we considered using Convolutional Neural Networks (CNNs) to train as a classifier for model realization. However, we dropped this idea since it was time-consuming and we didn't have enough time to do so.

Upon further investigation, we identified that the noise in these images resembled salt-and-pepper noise, which is best addressed by a median filter (median blur). The median filter is a nonlinear filter that replaces each pixel value with the median of neighboring pixel values, effectively removing salt-and-pepper noise. Given our existing CNN denoiser, we decided to incorporate salt-and-pepper noise into our inputs sequentially: salt-and-pepper, Gaussian, salt-and-pepper. This approach was unsuccessful, as were attempts using separate networks for each noise type.

We then employed a nested loop to test combinations of parameters for two filters: median blur and bilateral filter. The bilateral filter preserves edges while reducing noise and has four parameters. After testing on 81 images (4 parameters and for each we have 3 different values) from scene 12, we found that a kernel size of 7x7 for the median filter and specific parameter values yielded the best results.

For scene 9, similar testing with a kernel size of 7x7 also led to optimal parameter values.

Conclusion to denoising:

While we successfully identified all instances, we acknowledge that the bilateral filter is not ideal due to its parameter sensitivity across different images. The bilateral filter has two different sigmas which vary in different images, so it could be tricky because of its dependence on a specific sigma for each image.