Computer Vision – Algorithms and Deep Learning

Eyal Gur

June 2, 2024

Part I
Classical Computer Vision

1 Feature Extraction and Matching

We begin by describing two CV tasks, which are *feature extraction* and *matching*, used to extract (describe) some features of interest in an image, and finding the detected features in one image in another image.

Feature extraction (also called *feature description*) and matching are a cornerstone of CV, enabling to interpret and process visual data, with applications in fields such as

- Facial recognition: identifying unique facial features – distances between eyes, nose shape, jawline contours, etc. While traditional methods focus on geometric feature extraction, modern systems predominantly use deep learning (particularly CNNs).
- Object tracking: key features of an object are continuously detected and followed across video frames. Techniques range from basic methods like color tracking to more sophisticated ones like Kalman filtering and CNN-based trackers.
- Anomaly detection: seeks to identify patterns that deviate from the norm. Techniques range from simple statistical methods to complex neural networks, like auto-encoders, to detect outliers.

1.1 Feature Extraction

In feature description we aim at finding features of interest in a given image. The features can be represented using *keypoints* in the image, with a *descriptor* corresponding to each image. In this context, a keypoint is represented by corresponding pixels, and its descriptor is a numeric vector that captures descriptive information regarding the keypoint.

A good descriptor in CV is a set of key points that effectively represent key information about an object in an image. A good descriptor should be invariant to transformation, discriminate between different objects, have manageable size, reproducible across multiple noisy instances, computationally efficient, robust to noises, and more.

1.1.1 The SIFT Algorithm

SIFT. Scale Invariant Feature Transform is a feature extraction(description) method that reduces the image content to a set of keypoints.

The SIFT algorithmic procedure is described as follows:

1. Building the scale-space: first, we ensure that we can identify distinct points in a given image while ignoring noise. To this end, in Figure 1 we first blur each pixel of 300×300 image with a Gaussian noise (notice how the main features are somewhat "blur invariant").

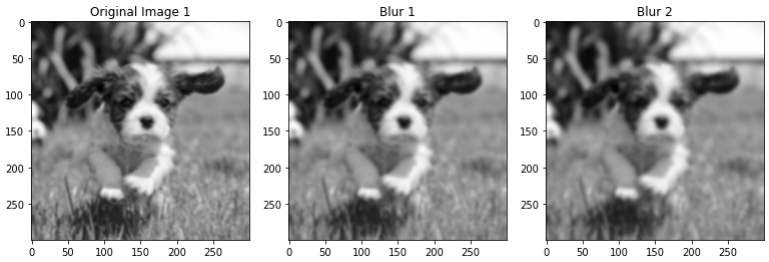

Figure 1: Applying Gaussian noise to each pixel of an image.

   Then, we scale the image to capture the non scale-dependent features (as the main features are expected to be "scale invariant"). We repeat this for different octaves (different scaling dimensions) to get the entire scale space in Figure 2.

2. Difference of Gaussian(DoG): for each octave, DoG subtracts pairs of two consecutive images, which creates another set of images in the same scale (in this example, we will get four images for each octave). The resultant images are used to find key points.

3. Key point localization: in order to find local extremum points over scale and space, one pixel in an image is compared with its eight neighbors as well as the nine pixels in next scale and nine pixels in previous scale. If it is a local extrema, it is a potential keypoint. It basically means that keypoint is best represented in that scale. However, this process results in a lot of key points being generated. So, we drop the key points which do not have enough contrast or are lying along an edge in our test image.

4. Orientation assignment: to make the set of legitimate key points invariant to rotation, we assign each of them a magnitude (intensity of a pixel) and orientation (direction of the pixel) as follows

$$\text{MAG} = \sqrt{G_x^2 + G_y^2} \quad \text{and} \quad \text{ORIEN} = \arctan\left(G_y/G_x\right),$$

   where $G_x, G_y \in \mathbb{R}$ are the gradients (difference in pixels) along the x and y axes.

   Then, we construct a histogram with angles from 0 to 360 degrees along the x-axis with bins of 10 degrees, and the magnitude on the y-axis. Each keypoint pixel is placed

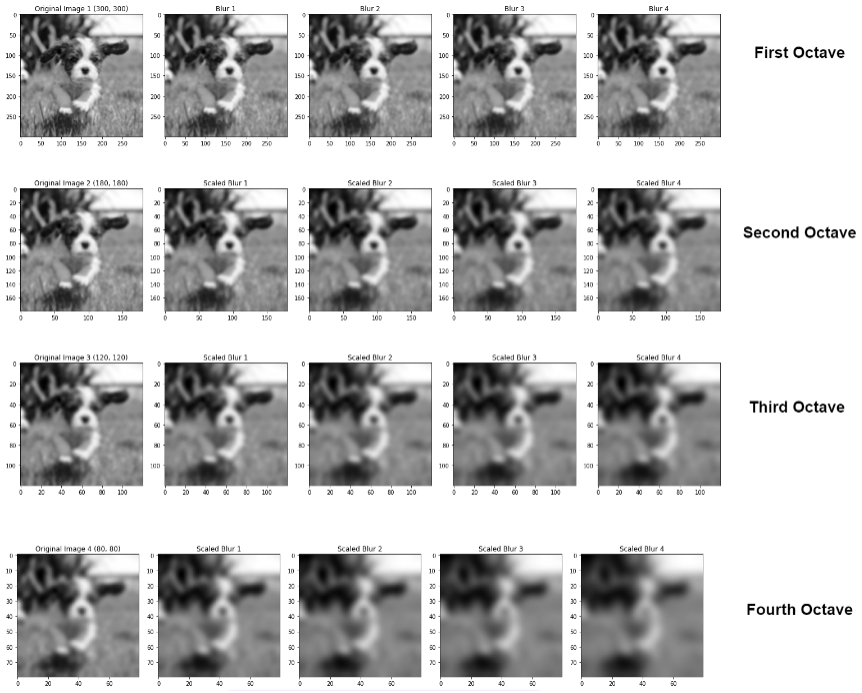

Figure 2: Applying Gaussian noise to different octaves to get the entire scale space of the image.

in its associated angle bin. The highest peak in the histogram is taken, and any peak above 80% of it is also considered to calculate the orientation. It creates keypoints with same location and scale, but different directions. It contributes to the stability of later matching.

5. Keypoint descriptor: the surrounding pixels to the remaining key points are used to make descriptors. Hence, the descriptors are invariant to viewpoint and illumination to a certain extent. A 16×16 grid is formed around the key point, further sliced into 4×4 sub-blocks. For each sub-block, an eight bin orientation histogram is created (similarly to the previous step), so a total of 128 bin values are available. It is represented as a vector to form a key point descriptor. In addition to this, several measures are taken to achieve robustness against illumination changes, rotation etc.

6. Keypoint matching: the keypoints extracted from the previous steps are used for pattern matching in other images. This signifies the importance of SIFT in object detection and image matching.

The following is implementation of SIFT with the `cv2` Python library (after installing the `opencv-python` package). The test image is read and converted to grayscale. The SIFT object is then used to detect and compute the keypoints and descriptors of, and the we highlight the keypoints.

```
path = r'C:\Users\eyal.gur.STAFF\Desktop\dogimage.jpg'  # set correct path
src = cv2.imread(path)
imgGray = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
```

```
sift = cv2.SIFT_create()
keypoints,descriptors = sift.detectAndCompute(src, None)
sift_image = cv2.drawKeypoints(imgGray, keypoints, src)

plt.figure()
plt.imshow(sift_image,cmap='gray')
```

After SIFT is executed, we plot a test image with all obtained keypoints is obtained in Figure 3. Since the descriptor of each keypoint is a numerical vector, they are not plotted.

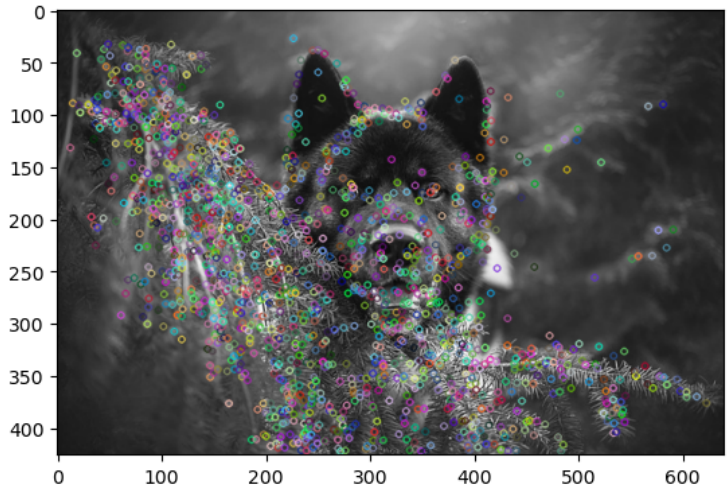

Figure 3: Keypoint detection using SIFT.

1.1.2 The ORB Algorithm

ORB. Oriented FAST and rotated BRIEF algorithm is based on the FAST (Features from Accelerated Segment Test) keypoint detector and a modified version of the visual descriptor BRIEF (Binary Robust Independent Elementary Features). Its aim is to provide a fast and efficient alternative to SIFT.

The following is implementation of ORB with `cv2`.

```
path = r'C:\Users\eyal.gur.STAFF\Desktop\dogimage.jpg'  # set correct path
src = cv2.imread(path)
imgGray = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

orb = cv2.ORB_create()
keypoints,descriptors = orb.detectAndCompute(src, None)
orb_image = cv2.drawKeypoints(imgGray, keypoints, src)

plt.figure()
plt.imshow(orb_image,cmap='gray')
```

Plotting the keypoints, we can see in Figure 4 that ORB detect (faster) a more limited set of keypoints compared to SIFT.

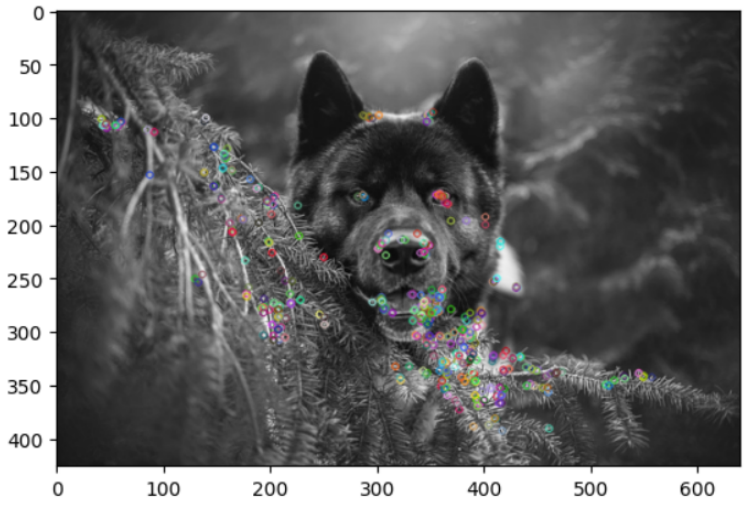

Figure 4: Keypoint detection using ORB.

1.2 Feature Matching

This task involves comparing key attributes in different images to find similarities in a *query image* using a *test image*. Feature matching is useful in many CV applications, including scene understanding, image stitching, object tracking, and pattern recognition. The SIFT and ORB methods, among other methods, can be used for feature extractions (description) for finding keypoints, which can later be used for the matching task.

In *brute force matching*, we compare each keypoint in the *test image* with each one in the query image, and we find the best match using some compatible metric, usually a k-NN algorithm (see below) with some threshold step to choose the best one among the k options. Of course, this method is exhaustive and it might take a lot of time for large images. In the Google Colab notebook we use SIFT and ORB to find keypoints in both test and query image in Figure 5, and to perform brute force matching.

k-NN algorithms. The k-nearest neighbors algorithm is a non-parametric supervised learning method, where the input consists of the k closest training examples in a dataset.

- In *k-NN classification*, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
- In *k-NN regression*, the output is a value for the object. This value is the average of the values of k nearest neighbors. If $k = 1$, then the output is simply assigned to the value of that single nearest neighbor.

Commonly used distance metric for continuous variables is Euclidean distance. For discrete variables, such as for text classification, another metric can be used, such as the *overlap metric*, or *Hamming distance*: the distance between two equal-length strings of symbols is the number of positions at which the corresponding symbols are different.

A drawback of this majority voting classification occurs when the class distribution is skewed (examples of a more frequent class tend to dominate the prediction). A way to overcome this problem is to weight the classification, where the class (or value) of each of the k nearest points is multiplied by a weight proportional to the inverse of the distance from that point to the test point.

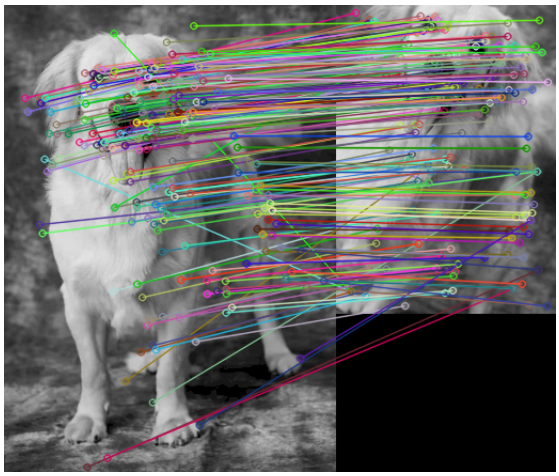

Figure 5: The output of brute force matching using SIFT as a keypoint detector. The brute force matching used the k-NN method with $k = 2$ and a ratio test to threshold the best match. The left figure is the test image, and the right (cropped and rotated) figure is the query image.

Since brute force matching can be slow, another option for matching is *FLANN* (Fast Library for Approximate Nearest Neighbors). This library performs fast approximate nearest neighbor searches in high-dimensional spaces. It contains a collection of algorithms that have been found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. With FLANN, we detect keypoints and descriptors using some algorithms (e.g., SIFT), and then we instantiate a FLANN object using `cv2.FlannBasedMatcher`, instead of the brute force object `cv2.BFMatcher`.

1.2.1 Learning-based Matching Using LoFTR

Both SIFT and ORB are *detector-based* algorithms, meaning that they have a detector phase for detecting keypoints (for example, step 3 of SIFT). *Detector-free* methods remove the feature detector phase and directly produce dense descriptors or dense feature matches. Similar to the detector-based methods, the nearest neighbor search is usually used as a post-processing step to match the dense descriptors.

LoFTR. A detector-free approach to local feature matching, which uses transformers with self and cross-attention layers to transform the dense local features extracted from a CNN backbone. Dense matches are extracted between the two sets of transformed features, and matches with high confidence are selected from these dense matches as output.

LoFTR is invariant to certain transformations, meaning it can handle variations in lighting, angle, or perspective. LoFTR's ability to robustly match features makes it valuable for tasks like *image stitching*, where you combine multiple images seamlessly by identifying and connecting common features.The LoFTR architecture is summarized as follows:

1. Local Feature Extraction: given the image pair I^A and I^B, we use a standard CNN architecture to extract multi-level features from both images. We denote by $\tilde{F}^A$ and $\tilde{F}^B$ the learnable matrices containing the coarse-level feature vectors at $1/8$ of the original image dimension, and by $\hat{F}^A$ and $\hat{F}^B$ the learnable matrices containing the fine-level feature vectors at $1/2$ of the original image dimension.

2. Coarse-Level Local Feature Transform:

   (a) The coarse feature maps are flattened to 1D vectors and added with the positional encoding. This encoding gives each element a unique position information in the sinusoidal format. By adding the position encoding to $\tilde{F}^A$ and $\tilde{F}^B$, the transformed features will become position-dependent, which is crucial to the ability of LoFTR to produce matches in indistinctive regions.

   (b) Local Feature Transformer (LoFTR) Module: extracts position and context dependent local features by interleaving self-attention encoder-only layers with the cross-attention encoder-decoder layers (see LLM document). In the self-attention layers, the input is a single image, while in the cross-attention layers the input are both images. These layers are interleaved N_c times. The output of this module is denoted as $\tilde{F}^A_{tr}$ and $\tilde{F}^B_{tr}$.

   We note that if N is the length of the query and key matrices, then a transformer block has a computation of $\mathcal{O}(N^2)$. Hence, directly applying the standard version of the transformer block in the context of local feature matching is impractical even when the input length is reduced by the local feature CNN. Instead, *Linear Transformers* are used in this context, which reduce the complexity to $\mathcal{O}(N)$ by substituting the exponent in the softmax computation with an approximation

$$\exp\left(\frac{\mathbf{q}^T\mathbf{k}}{\sqrt{d}}\right) \approx \phi(\mathbf{q})^T \phi(\mathbf{k}),$$
$$\phi(\cdot) \equiv \mathrm{ELU}_\alpha(\cdot) + 1,$$
$$\mathrm{ELU}_\alpha(x) \equiv \begin{cases} x, & x \geq 0, \\ \alpha(\exp(x) - 1), & x < 0. \end{cases}$$

3. Matching Module:

   (a) Compute the score matrix $\mathbf{S}$ as $\mathbf{S}_{ij} \equiv \frac{1}{\tau}\tilde{F}^A_{tr}(i)^T \tilde{F}^B_{tr}(j)$, where $\tilde{F}^A_{tr}(i)$ is the i-th feature vector in $\tilde{F}^A_{tr}$, and where τ is some hyper-parameter.

(b) Compute the confidence matrix $\mathbf{P}$ with the the *dual-softmax* operation

$$\mathbf{P}_{ij} \equiv \text{SOFTMAX}\left(\mathbf{S}_{i1}, \ldots, \mathbf{S}_{in}\right)_j \cdot \text{SOFTMAX}\left(\mathbf{S}_{1j}, \ldots, \mathbf{S}_{nj}\right)_i .$$

(c) Based on $\mathbf{P}$, select matches (elements of $\mathbf{P}$) with confidence higher than a threshold of θ. After this filtering, we further enforce the Mutual Nearest Neighbor (MNN) criteria on $\mathbf{P}$ to filter possible outlier coarse matches. By applying MNN on $\mathbf{P}$, for each element of $\mathbf{P}$ we calculate its nearest elements in $\mathbf{P}$ based on some metric. Two elements are considered MNNs if both elements are in the set of MNNs of the other. The resulting coarse matching set of indices is

$$\mathcal{M}_c \equiv \{(i, j) : \mathbf{P}_{ij} \geq \theta \quad \text{and} \quad (i, j) \in \text{MNN}\left(\mathbf{P}\right)\}.$$

4. Coarse-to-Fine Module: after establishing coarse matches, these matches are refined to the original image resolution with the coarse-to-fine module. For every coarse match $\left(\tilde{i}, \tilde{j}\right)$ we first locate its position $\left(\hat{i}, \hat{j}\right)$ at fine-level feature maps $\hat{F}^A$ and $\hat{F}^A$, and then crop two sets of local windows of size $w \times w$. A smaller LoFTR module then transforms the cropped features within each window N_f times, yielding two transformed local feature maps $\hat{F}^A_{tr}\left(\hat{i}\right)$ and $\hat{F}^A_{tr}\left(\hat{j}\right)$ centered at $\hat{i}$ and $\hat{j}$.

   Then, we correlate the center vector of $\hat{F}^A_{tr}\left(\hat{i}\right)$ (the image can be multi-dimensional) with all vectors in $\hat{F}^B_{tr}\left(\hat{j}\right)$ and thus produce a heatmap that represents the matching probability of each pixel in the neighborhood of $\hat{j}$ with $\hat{i}$. By computing expectation over the probability distribution, we get the final position $\hat{j}'$ with sub-pixel accuracy on I^B. Gathering all matches $\left(\hat{i}, \hat{j}'\right)$ produces the final fine-level matches set $\mathcal{M}_f$.

The entire LoFTR architecture is described in Figure 6. After finding the fine-level matches $\mathcal{M}_f$, we can further clean up ouliers in the set by applying the *Random Sample Consensus* (RANSAC) method, which is described in a box below.

RANSAC method. This is a general purpose method for cleaning outliers, and is essentially composed of two steps that are iteratively repeated:

1. A sample subset containing minimal data items is randomly selected from the input dataset. A fitting model (e.g., regression) with model parameters is computed using only the elements of this sample subset. The cardinality of the sample subset must be sufficient to determine the model parameters.

2. The algorithm checks which elements of the entire dataset are consistent with the model instantiated by the estimated model parameters obtained from the first step. A data element will be considered as an outlier if it does not fit the model within some error threshold defining the maximum data deviation of inliers.

The set of inliers obtained for the fitting model is called the *consensus set*. The

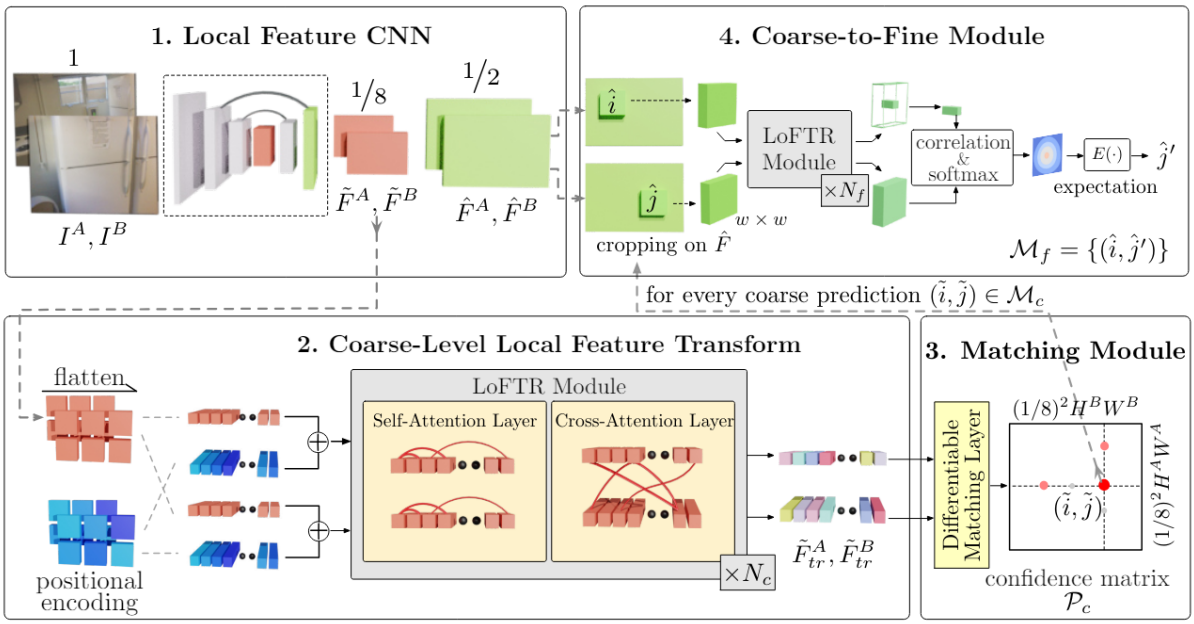

Figure 6: LoFTR architecture.

RANSAC algorithm will iteratively repeat the above two steps until the obtained consensus set in a certain iteration has enough inliers.

In the Google Colab notebook, we implement the LoFTR method (with additional RANSAC) with the `kornia` Python library. This library consists of a set of routines and differentiable modules to solve generic CV problems. At its core, the package uses PyTorch as its main backend both for efficiency and to take advantage of the auto-differentiation to define and compute the gradient of complex functions.

1.3 The Convolution Operation

Convolution is an operation used to extract features from data. For example, consider the vector $\mathbf{v} \equiv (0,0,0,1,1,1,1,0,0,0)$ and the *kernel* (or *filter*) vector $\mathbf{k} \equiv [-1,1]$ with *center* 1 (the center of a kernel is the element of the kernel that has to be multiplied with all elements in the target vector). Assuming a stride of size 1 and a zero padding of size 1 at the left of $\mathbf{v}$ (so all elements of $\mathbf{v}$ are multiplied with the center), the convolution operation then yields

$$\mathbf{v} \star \mathbf{k} = (0,0,0,1,0,0,0,-1,0,0)\,.$$

Notice that the convolution of this filter gives the rate of change in the data (derivatives). The convolved data is called a *feature map*, and is exactly what edge detection filters do.

Another filter of edge detection is the *Prewitt filter*. Its 2D horizontal and vertical variants – $\mathbf{K}_x$ and $\mathbf{K}_y$, respectively – take the 3×3 form

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

This filter calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction, and therefore how likely it is that part of the image represents an edge and how that edge is likely to be oriented.

For a given image $\mathbf{A}$, we calculate its *gradient map* $\mathbf{G}$ using the feature maps

$$\mathbf{G} \equiv \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2} \quad \text{where} \quad \mathbf{G}_x = \mathbf{K}_x \star \mathbf{A}, \quad \mathbf{G}_y = \mathbf{K}_y \star \mathbf{A},$$

where the operations for calculating $\mathbf{G}$ are element-wise.

Another filter used for edge detection is the *Sober filter* defined as

$$\mathbf{K}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}.$$

As the Prewitt filter, this filter is also a discrete differentiation operator and it detects more intricate edges (hence a more cluttered image).

In the Google Colab notebook we implement the two filters for edge detection using `cv2` (though it can be implemented with pure `numpy`). The output is given in Figure 7.

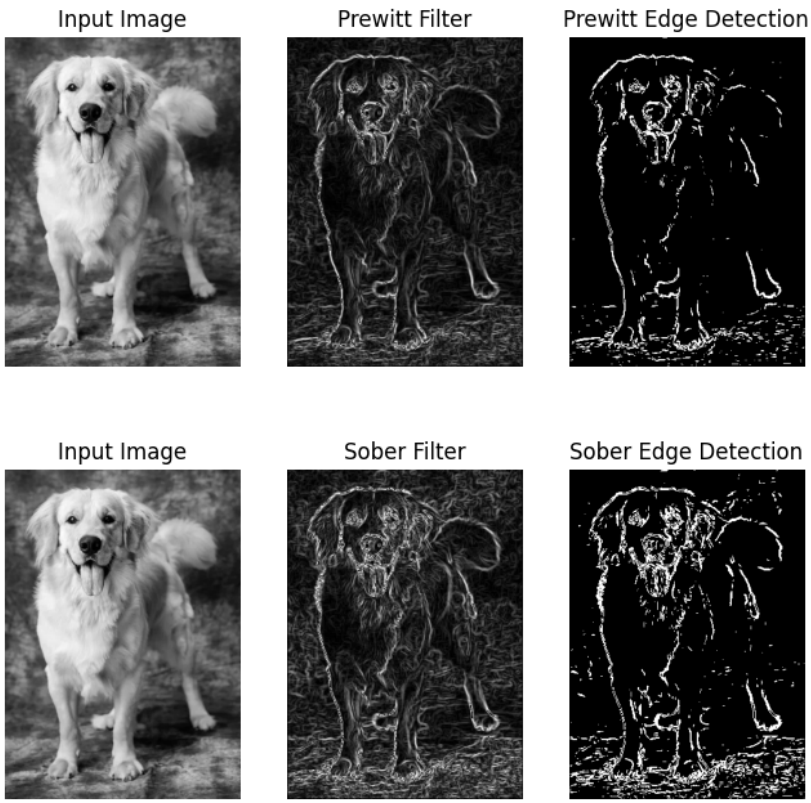

Figure 7: Original image in grayscale, filtered image (gradient map), and the edge detected image (binary map with a threshold of 100 in the spectrum of 0 to 255).

Part II
Computer Vision With Deep Learning

2 Convolutional Neural Networks