# Project-1 Report
*for*
# Perception for Autonomous Robots (ENPM673)

**Aditi Ramadwar UID: 117093575**

# Problem 1: Detection

## 1a) AR Code detection

Here, I chose the first frame of the test video, Figure 1. To get the edges of the tag, we need to extract the unique features of the image. Hence, by applying a low pass filter on the image, we can get the desired edges. For this application, I used Fast Fourier Transformation to implement a low pass filter and remove the features that are more common in the frame, like the background.

Steps for edge detection:

***Step 1***: I applied Morphological operations on the image using the OpenCV library such as Gaussian Blur, Erosion, Dilation, and Thresholding to get a clean image before converting it in the frequency domain.
***Step 2***: Convert the processed image from spatial domain into the frequency domain using FFT function from the scipy library
***Step 3***: Generate a circular low pass filter mask that will be used for removing the high frequencies and multiply the mask with the derived FFT.

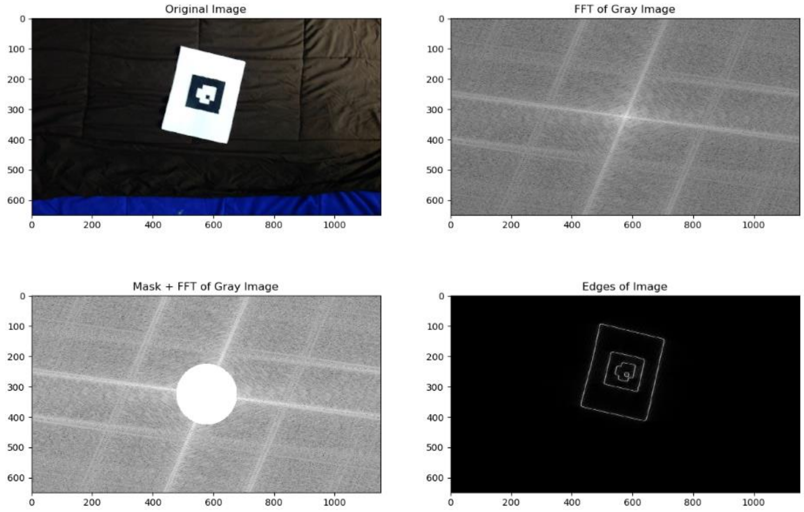


Figure 1

***Step 4***: Convert the modified FFT back into spatial domain using the inverse FFT function from the scipy library to get Figure 2.

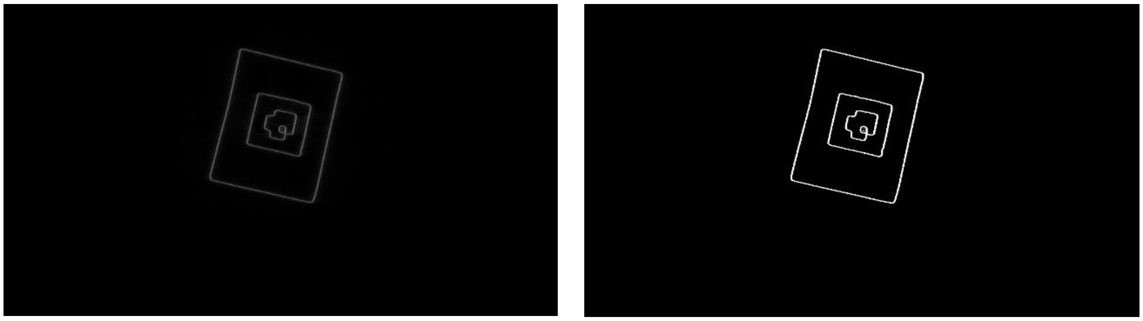


Figure 2

Problems:

The FFT output was not giving out substantially bright edges that can be later used for corner detection. To resolve this, I added a thresholding followed by a blur to make the edges cleaner.

Another problem that I faced was that there is a small part in the video where due to external noise, some prominent unique features were also being captured through the low pass filter which were not a part of the paper or the tag. This noise would give me false detections of paper corners. To tackle this problem, before FFT, I applied the morphological operation of erosion, dilation and blur. This resolved the issue, giving me just the paper edges in every frame.

## 1. b) Decode Custom AR tag

Here I have used to data tags for computing the orientation as well as the tag ID, Figure 3 and Figure 4

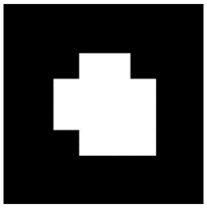


Figure 3

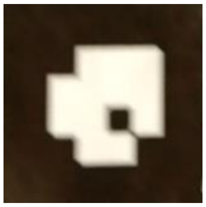


Figure 4

Steps for decoding AR tag:

***Step 1***: Resize the image to 160 x 160

***Step 2***: Apply morphological operations on it such as gray scaling and thresholding

***Step 3***: Divide the image into 8 sections to get it in an 8 x 8 grid of squares

***Step 4***: For each section take the average pixel value to get the true block value of the tag.

***Step 5***: Extract the middle 4x4 pixels to get rid of the padding and to decode the tag for orientation and tag ID

***Step 6***: The last element of the list will represent the orientation of the tag. The value of that element should be 255. If the value is not 255 then rotate your pixels until it is. While doing the rotation, count the number of times the rotations are needed. This can be used later in the future to superimpose any image on it.

***Step 7***: Use the rotated pixel value to decode the tag ID of the marker. Extract the central 2x2 pixels and flatten them out in an array. According to the custom decoding method, which is ordered in the clockwise direction from least significant bit to most significant. So, the top-left square is the least significant bit, and the bottom-left square is the most significant bit. Convert the binary to decimal.

**Result:**

The Tag ID for Figure 3 is 15 where no orientation is needed

The Tag ID for Figure 4 is 7 where a rotation is needed to be done 3 times to obtain the correct orientation

*No specific problems arose as such for this part*

## Tag detection in video frame

a) *Detect Paper in each frame*:

- Run FFT on each frame.
- Pre-process the frame by using morphological operation.
- Detect corners in the frame using the function *cv2.goodFeaturesToTrack*.
- Select two corners, a and b, such that they are closest to the origin, as shown in Figure 5.

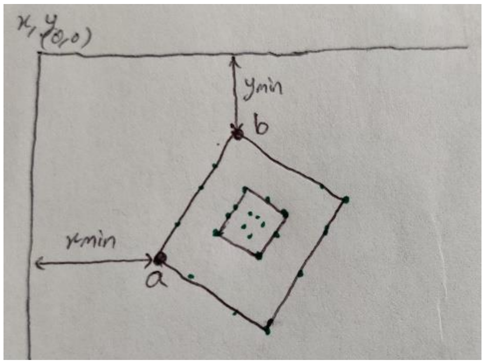


Figure 5

- Using the *np.linalg.norm* function calculate the corresponding farthest points for a and b. The farthest point for a and b would be the diagonally opposite points, c and d respectively. We now have all four corners of the paper.

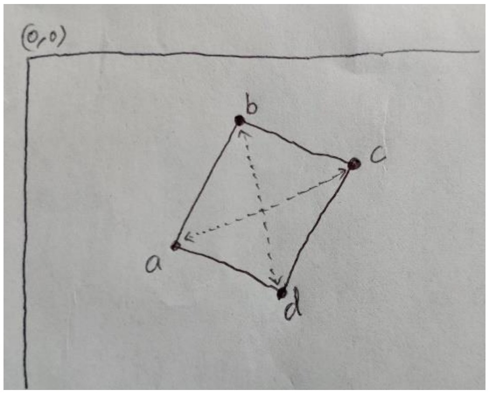


Figure 6

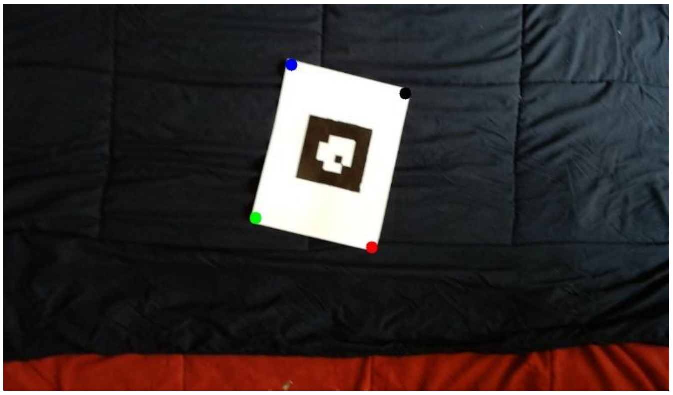


Figure 7

- Now, for consistency, using the two points c and d, I again calculated the farthest points to get new sets of a and b points. For most cases, the points a and b will stay the same as before, but this is done just to avoid a few edge cases where the corners become colinear while the frame is turning.

## b) *Detect tag*

- Now that we have the paper corner points, a, b, c, and d, we can go ahead with detecting the tag corners.
- Using the line equations, we can select all the points that lie inside the paper edges, as shown in Figure 8. This will ensure that the next points that we select are not on the edge of the paper but are part of the tag.

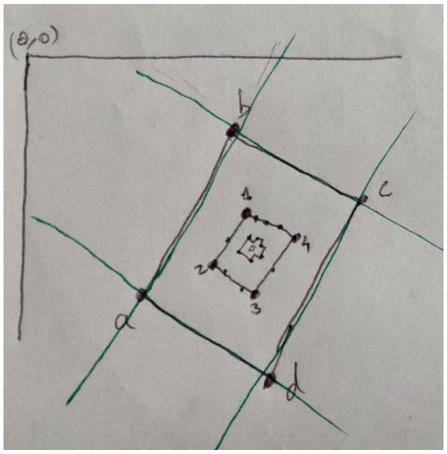


Figure 8

- Now that all the inlier points have been obtained, run the same logic that was used for detecting the paper corners. This will give us the tag corners, 1, 2, 3, and 4, as shown in Figure 9.

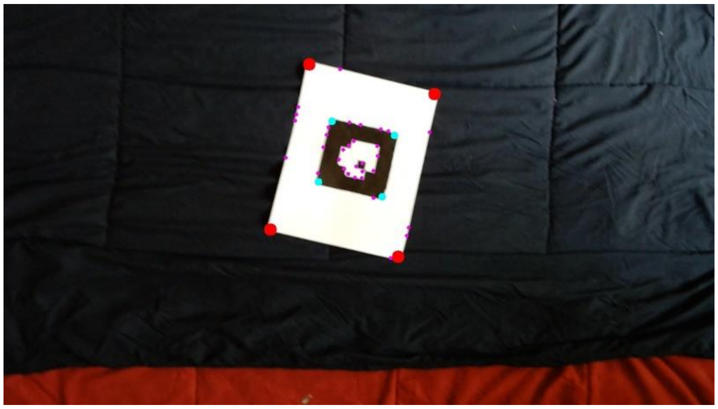


Figure 9

*c) Decode Tag*

- Once tag corner points are obtained, I sorted them in the order as top, left, bottom, right. Using these sorted corner points, I calculated the homography matrix for a 160 x 160 grid. Using this homography matrix, I got the AR tag in a 160x160 grid using inverse warping, such as Figure 10.

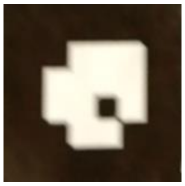


Figure 10

- Now using the logic from tag decoding from Problem 1, get the tag ID and check the number of rotations required to get the tag in the correct orientation.
- I rotated my sorted tag corners by 90 degrees by the number of rotation count.

## Problem 2: Tracking

### Problem 2.a) Superimposing image onto Tag:

Now that I have my correct tag corners, I can superimpose the testudo image onto the AR tag in the frame.
For this, I will derive another homography matrix between the rotated points and the sorted testudo corners.
Using forward warping and the homography matrix, I can superimpose the testudo image back on the original frame

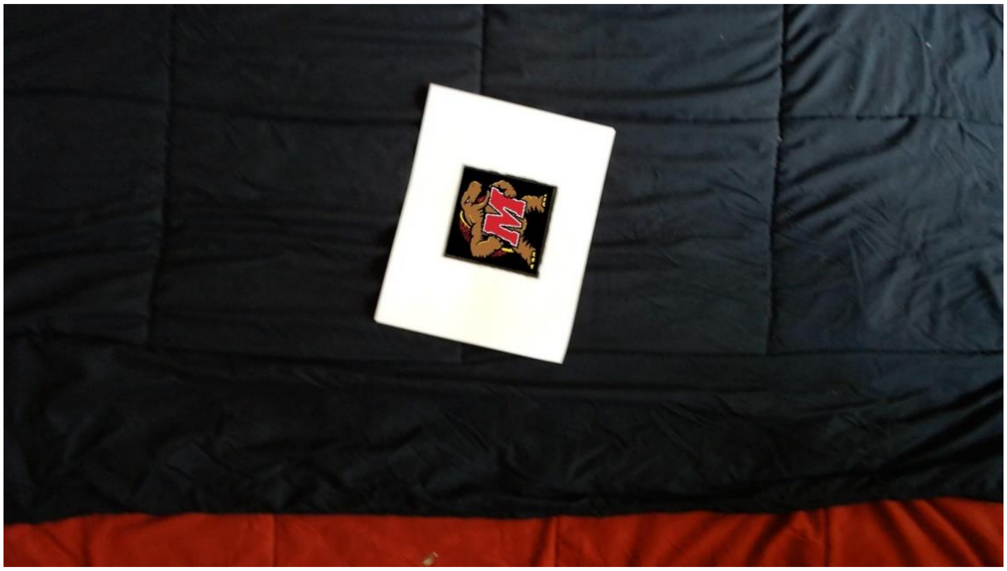


Figure 11

*The video of the Testudo Superimposition can be found here*

Problems:

1) While detecting the corners of the paper and tag, point a and b were shifting in a few frames where the corners were becoming colinear. This resulted in three of the corners getting detected on the same point. This was an undesirable condition since warping would not work at all for those frames. So as a work around I checked the distance between all the four corners with respect to each other and if in any frame the points come too close to each other, I use the corner values of the previous frame to continue warping. This method works for this particular dataset since there aren't a lot of quick changes in the video, so this kind of corner skipping is not too noticeable. Although, a better method should be used to manage the collinearity.

2) Another problem that I was facing is while applying forward warping on the frame to obtain the AR tag, the resulting image was very spotty. This made it difficult to decode the orientation and Tag ID of the AR tag in each frame. As a work around for this, instead of using forwards warping, I used inverse warping which solved this problem.
Although the problem still slightly persists while warping the Testudo image back onto the video frame for a few frames, as shown in Figure 12. This issue can be resolved by changing the shape of the Testudo image and applying bilateral filtering to even out the pixels for a better-quality warping.

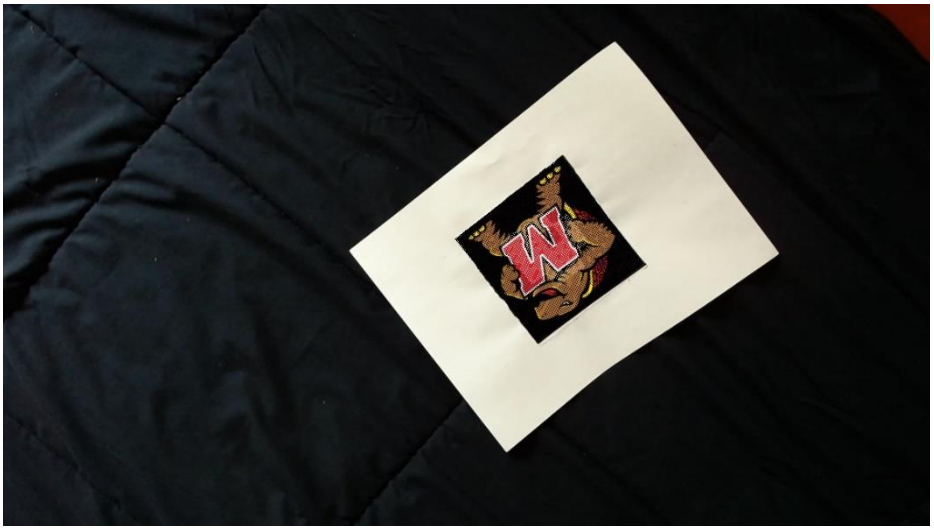


Figure 12

## Problem 2.b) Placing a virtual cube onto Tag

Using the same sorted tag points, we can place a 3D cube on the image.

I derived a 3x4 projection matrix to project the 3D cube points onto the image plane.

The intrinsic matrix is specified as:

$$K = \begin{matrix} 1346.100595 & 0 & 932.1633975 \\ 0 & 1355.933136 & 654.8986796 \\ 0 & 0 & 1 \end{matrix}$$

H is the homography matrix between the rotated and sorted tag corners and the planar 160 x 160 grid

***Step 1***: $B = K^{-1} * H$

***Step 2***: If determinant of B is negative then multiply B with (-1) for further calculations

***Step 3***: We now have b1, b2, b3 which are the respective columns of the B matrix

***Step 4***: We can calculate the rotation and translation of the points using these columns.

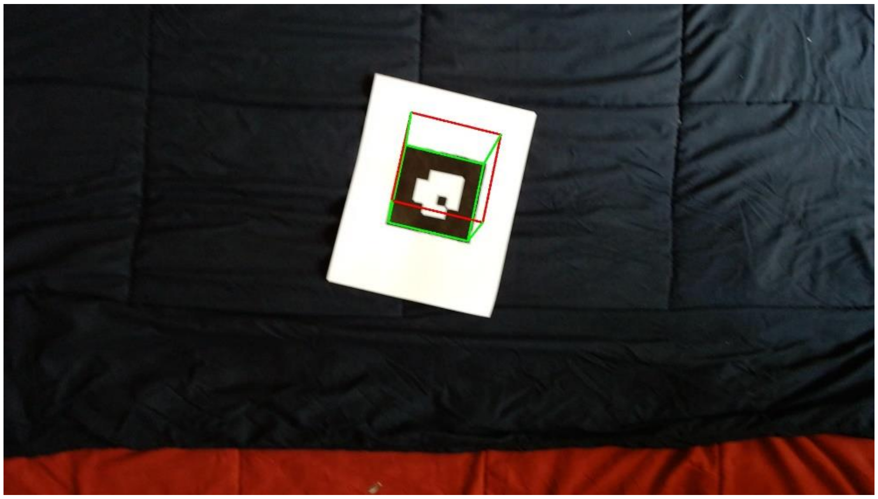
$$[r1, r2, t] = lamda * B \qquad ,\text{where } lamda = \frac{2}{||b1||+||b2||}$$

r3 = r1 x r2

R = [r1, r2, r3]

P = K * [R|t]

***Step 5***: After getting the projection matrix, we can multiply it with the tag corner to get the base of the cube as well as the top face of the cube by giving a height to the cube in the z-axis. I drew the cube edges by using the cv2.drawContours and cv2. Line functions.



*The video of the cube projection can be found here*

*Problem*

At first, I was specifying the height of the cube through z-axis to be +160 but every time I'd try to plot the cube, only the bottom face of the cube would be visible. After following the supplementary document properly, I figured out that due to the OpenCV image coordinates, the origin is on the top left corner, the direction of z-axis would be towards the image rather than away from the image. This issue was solved by making the height negative.

## DexiNed: Dense Extreme Inception Network

It is a robust CNN that has been trained from the scratch. DexiNed has been designed to produce thin edges in order to enhance the visualization of predicted edge maps, for which upsampling is used.

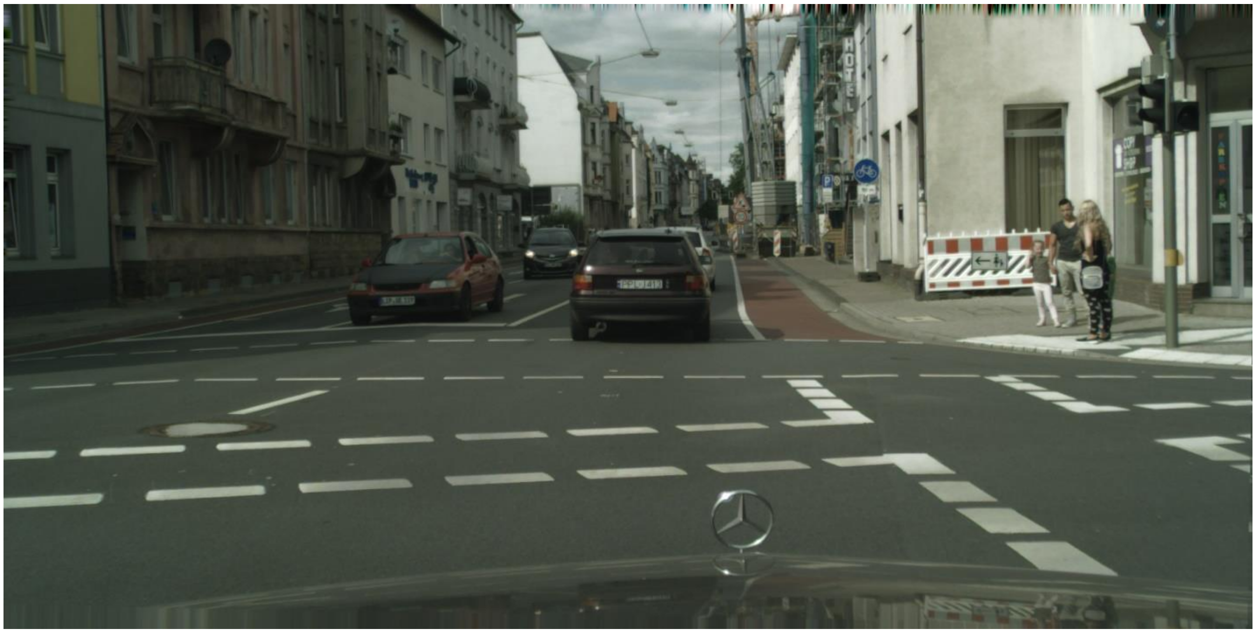
DexiNed consists of an encoder composed of six main blocks. The main blocks are connected between them through 1x1 convolutional blocks.

Each of the main blocks is composed of sub-blocks that are densely interconnected by the output of the previous main block. The output from each of the main blocks is fed to an upsampling block that produces an intermediate edge-map which is used to compose a final fused edge-map.

In each upsampling block, there are two convolutional layers with kernel size 3 × 3, followed by batch normalization and ReLU as the activation function (just the last convolutional layer in the last sub-blocks does not have such activation). The max-pool is set by a 3 × 3 kernel and stride of 2.

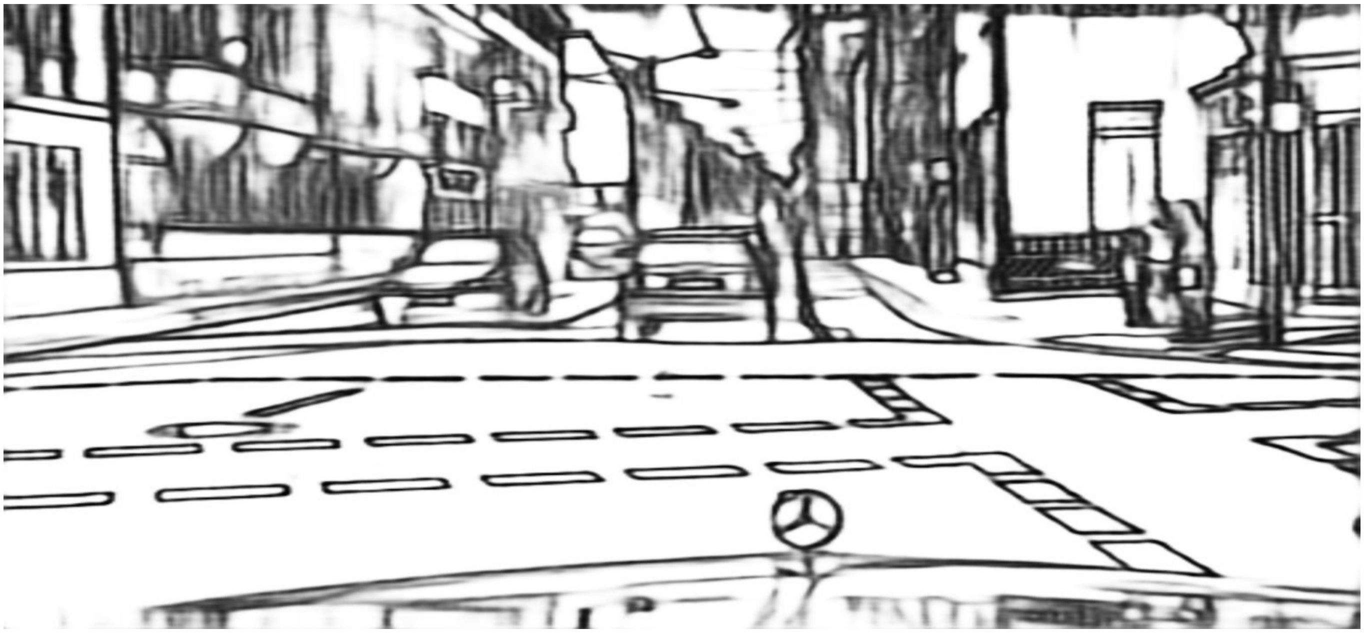
The model is trained on edge detection specific datasets like the Barcelona Images for Perceptual Edge Detection dataset and the multi-cue boundary detection dataset, and non- edge detection specific data sets like Contour Image Database, the Berkeley Segmentation Dataset, NYU Depth Dataset, PASCAL-Context Dataset

The input image used to test the model:

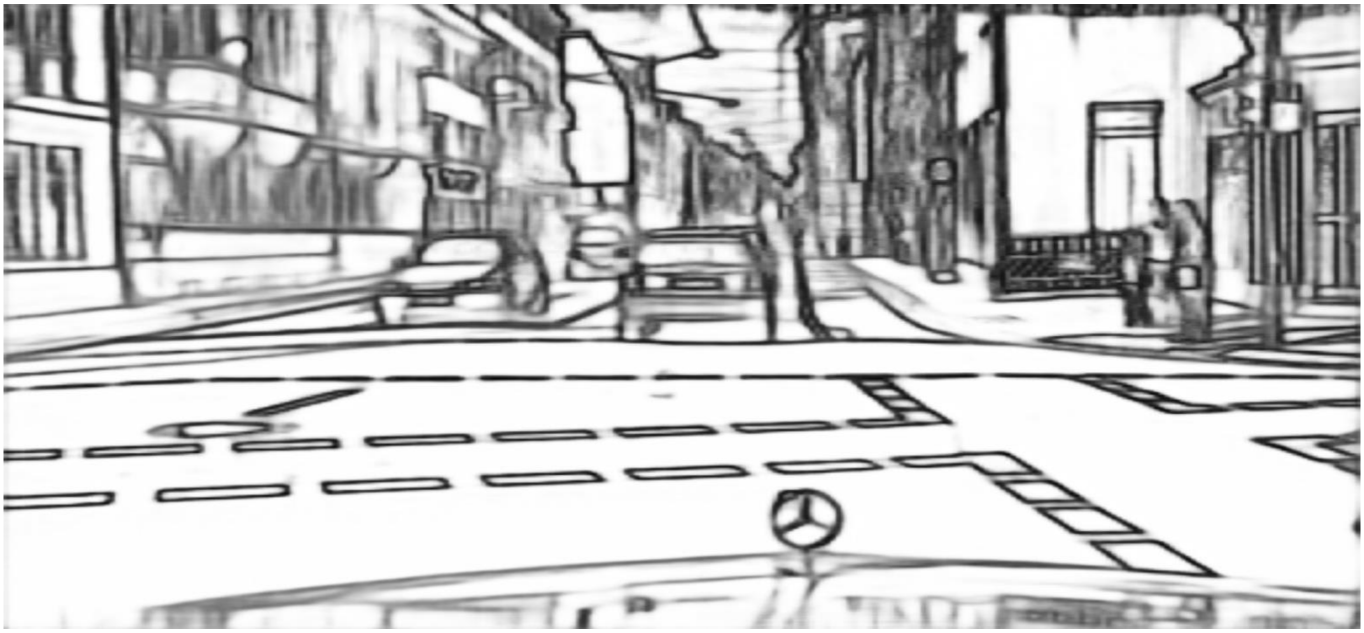


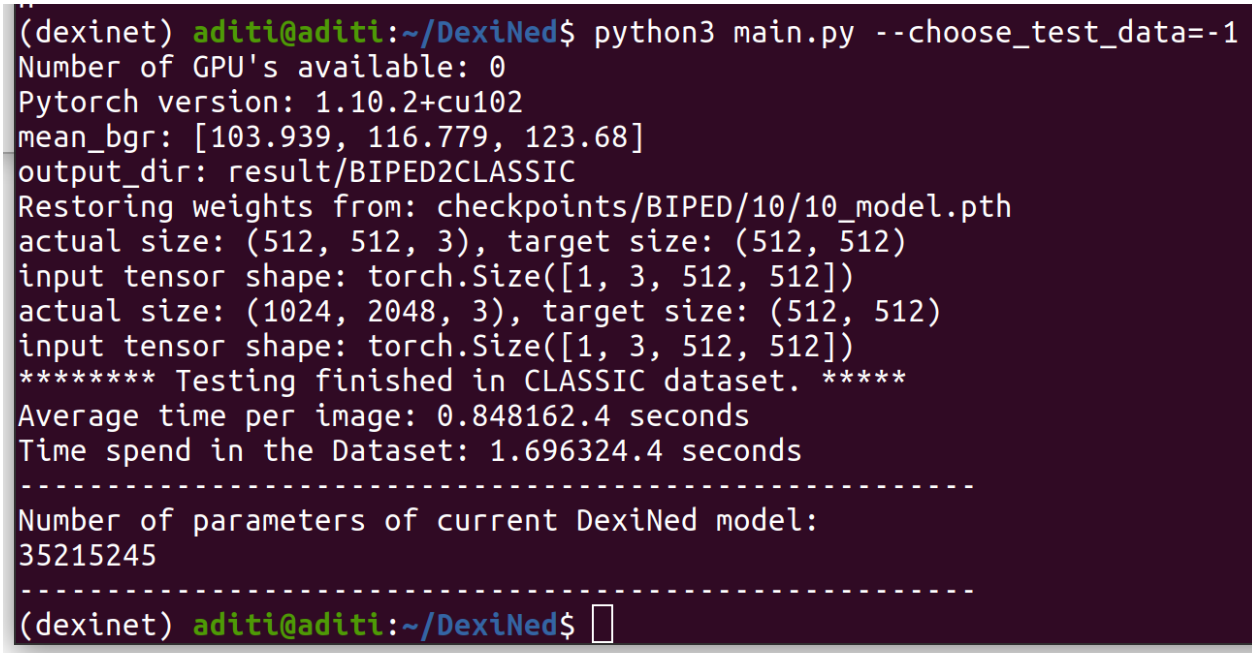
The output produced by the model:

**Fused Output**: The fused output is the concatenation and fusion of the 6 outputs obtained from the upsampling block



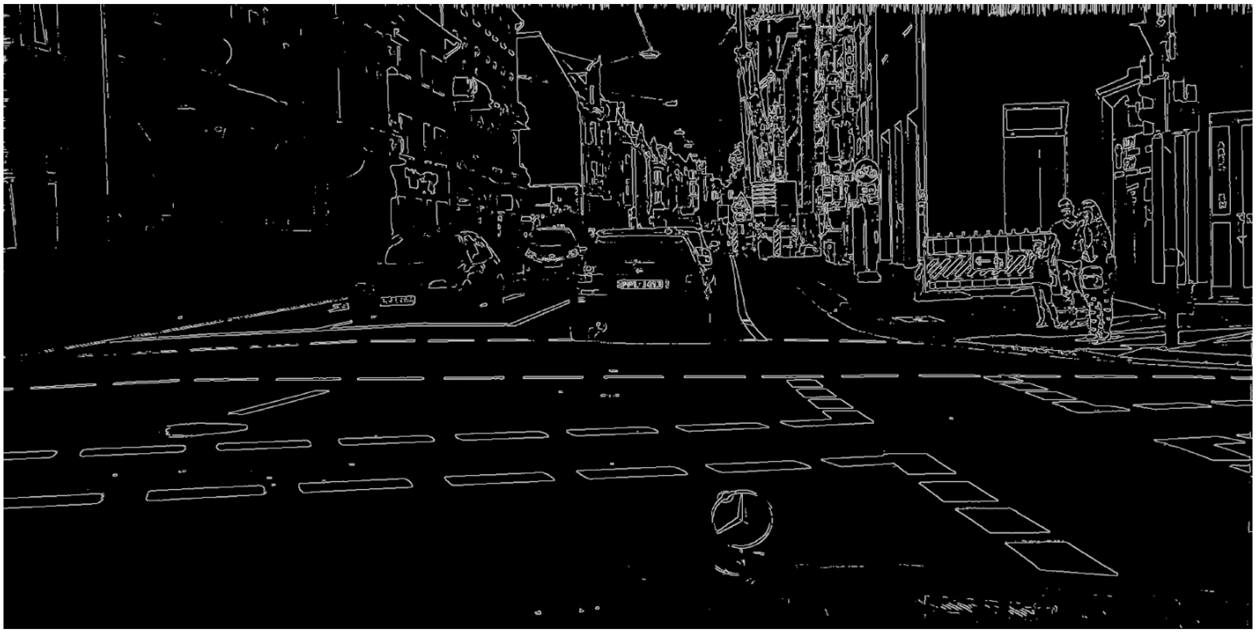
**Averaged Output**: The averaged output is the average of all previous predictions.





```
(dexinet) aditi@aditi:~/DexiNed$ python3 main.py --choose_test_data=-1
Number of GPU's available: 0
Pytorch version: 1.10.2+cu102
mean_bgr: [103.939, 116.779, 123.68]
output_dir: result/BIPED2CLASSIC
Restoring weights from: checkpoints/BIPED/10/10_model.pth
actual size: (512, 512, 3), target size: (512, 512)
input tensor shape: torch.Size([1, 3, 512, 512])
actual size: (1024, 2048, 3), target size: (512, 512)
input tensor shape: torch.Size([1, 3, 512, 512])
******** Testing finished in CLASSIC dataset. *****
Average time per image: 0.848162.4 seconds
Time spend in the Dataset: 1.696324.4 seconds
-------------------------------------------------------
Number of parameters of current DexiNed model:
35215245
-------------------------------------------------------
(dexinet) aditi@aditi:~/DexiNed$ 
```

*Comparison with Canny Edge detection function of OpenCV*



Even though the edges being detected by the canny are more prominent than the result of DexiNed, it can be observed that a lot of information is lost while implementing Canny.