[Fall 2024] ROB-GY 6203 Robot Perception Homework 1

Shantanu Ghodgaonkar (sng8399)

Submission Deadline (No late submission): NYC Time 11:00 AM, October 9, 2023
Submission URL (must use your NYU account): https://forms.gle/EPyThuLsYBopQQ3MA

1. Please submit the **.pdf** generated by this LaTex file. This .pdf file will be the main document for us to grade your homework. If you wrote any code, please zip all the **code** together and **submit a single .zip file**. Name the code scripts clearly or/and make explicit reference in your written answers. Do NOT submit very large data files along with your code!

2. You don't have to use AprilTag for this homework. You can use OpenCV's Aruco tag if you are more familiar with them.

3. You don't have to physically print out a tag. Put them on some screen like your phone or iPad would work most of the time. Make sure the background of the tag is white. In my experience a tag on a black background is harder to detect.

4. Please typeset your report in LaTex/Overleaf. Learn how to use LaTex/Overleaf before HW deadline, it is easy because we have created this template for you! **Do NOT submit a hand-written report!** If you do, it will be rejected from grading.

5. Do not forget to update the variables "yourName" and "yourNetID".

Contents

Task 1 Sherlock's Message (2pt)

Detective Sherlock left a message for his assistant Dr. Watson while tracking his arch-enemy Professor Moriarty. Could you help Dr. Watson decode this message? The original image itself can be found in the data folder of the overleaf project (https://www.overleaf.com/read/vqxqpvbftyjf), named `for_watson.png`

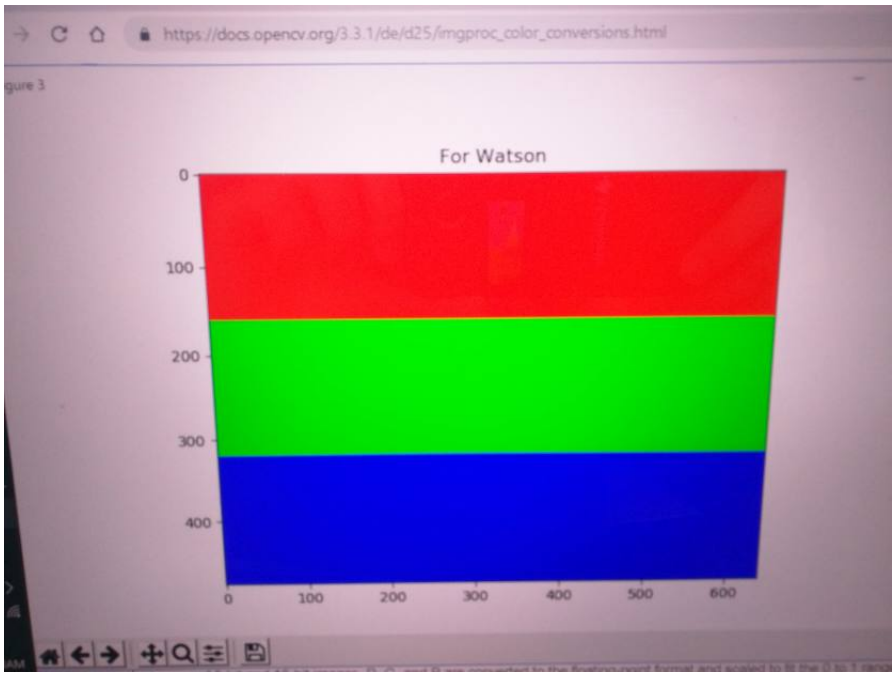

Figure 1: The Secret Message Left by Detective Sherlock

Part A (1pt)

Please submit the image(s) after decoding. The image(s) should have the secret message on it(them). Screenshots or images saved by OpenCV is fine.

Answers:
Here is the decoded image -

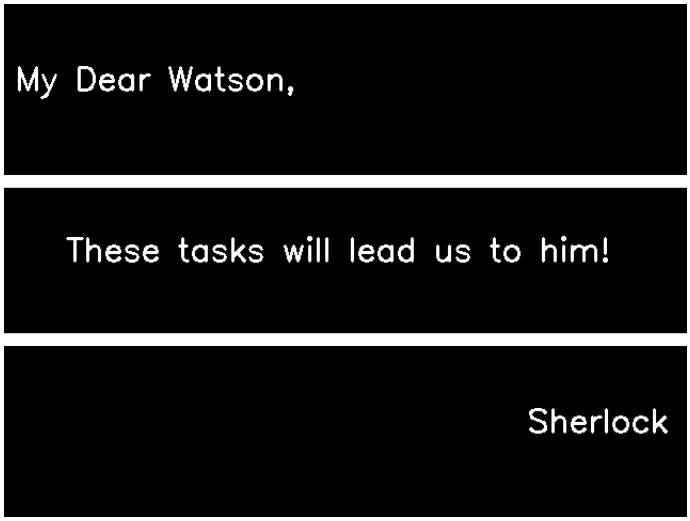

Figure 2: Decoded image shown in Figure 1

Part B (1pt)

Please describe what you did with the image with words, and tell us where to find the code you wrote for this question.

Answers:
For the given image with filename `for_watson.png`, the following code was written -

```python
import cv2 # Import OpenCV for image processing operations
from pathlib import Path # Import Path for file path resolution

if __name__ == '__main__':
# The above condition ensures that the code inside this block runs only when the script is executed directly, not when it is imported as a module in another script.
    # Obtain the absolute path of the image file
    img_path = Path('Q1/for_watson.png').absolute()
    # Read the given image and store it in a variable
    img = cv2.imread(img_path.__str__())
    # Convert image from BGR to Grayscale, which is required for optimising thresholding
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Apply adaptive mean thresholding to the gray image. Here, for each pixel, the
    # algorithm calculates the mean of the pixel values in a 25x25 neighborhood centered around
    # that pixel. It then subtracts the constant C (which is 0 in this case) from this mean value
    # to compute the threshold for that pixel. If the pixel's intensity is greater than this
    # threshold, it's set to 255 (white); otherwise, it's set to 0 (black). The values for the block size and constant C were obtained by brute force trial & error.
    img_decoded = cv2.adaptiveThreshold(
        img_gray, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 25, 0)
    # Display decoded image
    cv2.imshow('Decoded Image', img_decoded)
    # Wait for keyboard interrupt, i.e., keep the image window open until a key is pressed
    cv2.waitKey()
    # close all open windows
    cv2.destroyAllWindows()
    # Store image to a file in the same folder as the original image
    cv2.imwrite((img_path.parent.__str__() +
                 '/for_watson_decoded.png'), img_decoded)
```

The idea for the code was found in the OpenCV documentation at OpenCV: Image Thresholding under the sub-heading of `Adaptive Thresholding`. Here is the github link to the code - q1_ans.py.

Task 2. Deep Learning with Fasion-MNIST (5pt)

Given the Fasion-MNIST dataset, perform the following task:

Part A (2pt)

Train an unsupervised learning neural network that gives you a lower-dimensional representation of the images, after which you could easily use t-SNE from `Scikit-Learn` to bring the dimension down to **Visualize** the results of all 10000 images in one single visualization.

Answers:
The code for this solution can be found here - q2_A_ans.py. Upon execution of the 150 epochs, the below plot shows the t-SNE visualisation of the lower-dimensional space -

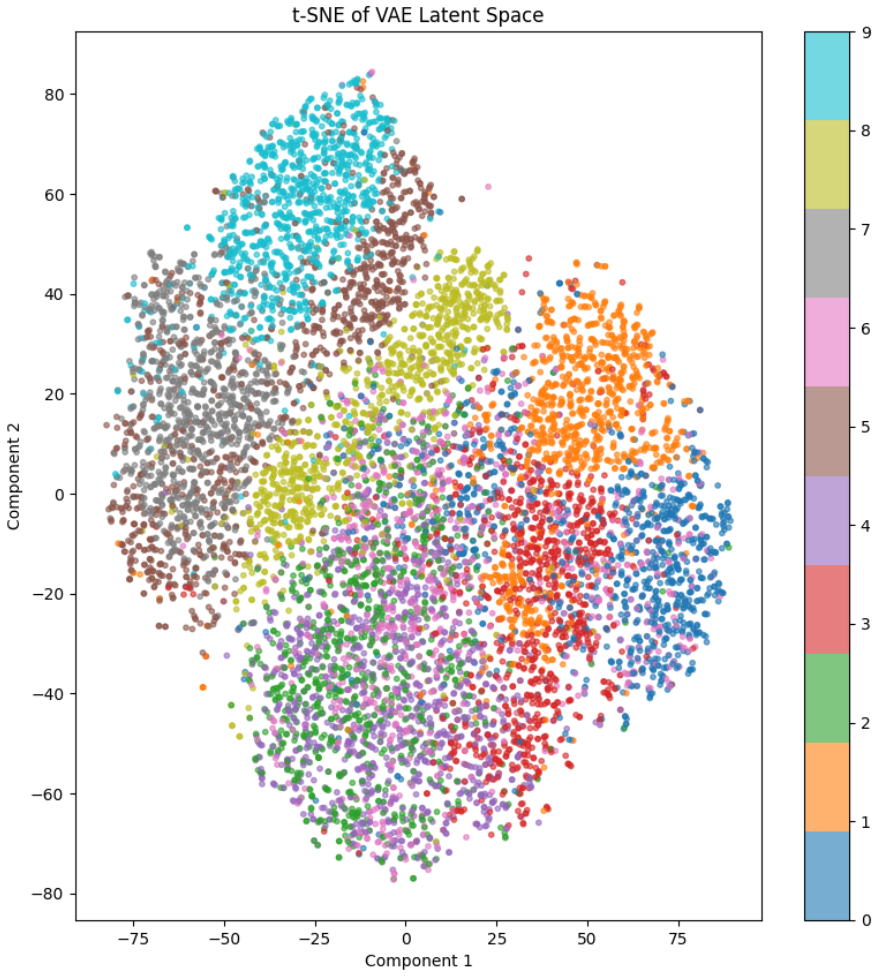

Figure 3: t-SNE visualisation of the lower-dimensional space

The below Figure 4 is a plot of the loss curves for the training and test datasets.

Thus, by observing the loss curves, we can conclude that about 50 epochs are sufficient to train the model. The trained models can be found at the link.

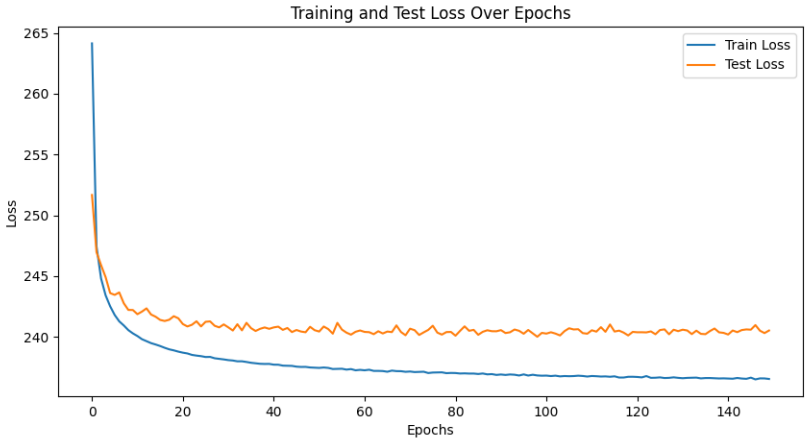

Figure 4: Loss curve visualisation

Part B (3pt)

Take the lower-dimensional latent representation produced in Part A and **train** a supervised classifier using these features. **Visualize** the loss and accuracy curves during the training process for both the training and testing datasets. Discuss your observations on the behavior of both curves. Evaluate the classifier's performance using accuracy or other appropriate metrics on the test set. **Report** your final accuracy, providing examples of correct and incorrect predictions.

Answers:
The code for this solution can be found at q2_B_ans.py and the trained models can be found at the link. The

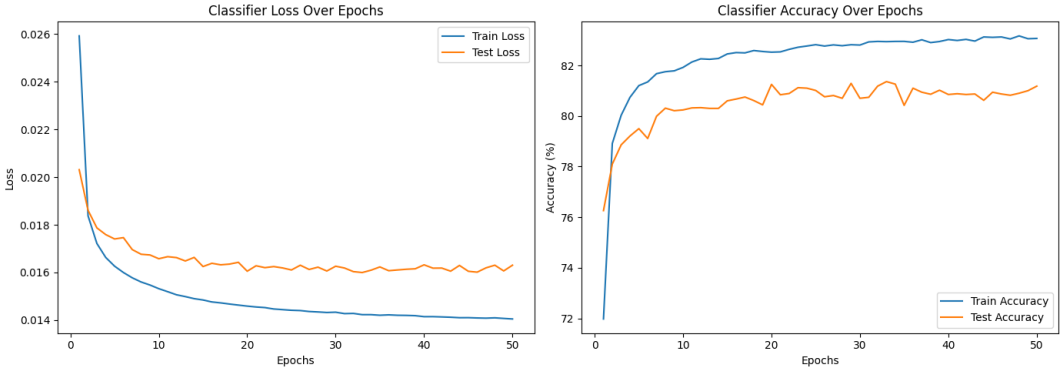

Figure 5: Loss and Accuracy curve visualisation

above figure is a plot of the loss and accuracy curves. As we can see, both the training and testing loss is quite low at 0.014 for the training dataset and 0.0163 for the testing dataset, just after 50 epochs. Similarly, the accuracy for the training dataset was found to be 83.07% for the training dataset and 81.18% for the testing dataset after 50 epochs.

Task 3 Camera Calibration (3pt)

Compare and contrast the intrinsic parameters (K matrix) and distortion coefficients (k1 and k2) obtained from calibrating your camera using two different sets of images. For the first set, take images where the distance between the camera and the calibration rig is **within 1 meter**. For the second set, take images where the distance is **between 2 to 3 meters**. Use the provided pyAprilTag package or other available tools (such as OpenCV's camera calibration toolkit) to perform the calibration and analyze the differences between the two sets. Discuss potential reason(s) for the differences (A good discussion about these reasons could receive 1 bonus point).

Answers:
Here is the link to the code - q3_ans.py.

The rig was essentially a laptop that displayed the image of the 9×7 checkerboard, held in the camera's field-of-view. The camera of choice is the Lenovo Essential FHD Webcam.

For the 1st part, images were taken of the 9×7 checkerboard held within $1m$ of the camera's lens. The code linked above provides a detailed description of the working process for computing the K matrix and distortion coefficients.

A total of 15 images were captured to perform the calibration and the following calibration parameters were found -

$$\text{Total mean re-projection error} = 0.18869793326778977$$

$$\text{Camera Matrix (K)} = \begin{bmatrix} 432.15077202 & 0. & 376.86467699 \\ 0. & 451.18127325 & 258.30382425 \\ 0. & 0. & 1. \end{bmatrix}$$

$$\text{Distortion Coefficients} \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} = \begin{bmatrix} -0.21914991 & 0.19218259 & 0.01419519 & 0.02305682 & -0.07298546 \end{bmatrix}$$

Similarly, for the 2nd part, images were taken of a 9×7 checkerboard held within $1m$ of the camera's lens.

A total of 15 images were captured to perform the calibration and the following calibration parameters were found -

$$\text{Total mean re-projection error} = 0.47347600289462183$$

$$\text{Camera Matrix (K)} = \begin{bmatrix} 7.61457801e+02 & 0.00000000e+00 & 3.08355850e+02 \\ 0.00000000e+00 & 1.01388231e+03 & 2.72467305e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

$$\text{Distortion Coefficients} \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} = \begin{bmatrix} 0.14131 & 15.567 & 0.71407 & 0.0038129 & -40.748 \end{bmatrix}$$

The main reason for the differences between the calibration performed by the two sets is due to the incorrect detection of corners of the cells in the checkerboard. When the checkerboard was placed much closer to the camera, upon plotting the detected corners, it was observed that the algorithm was able to detect the cell corners much more accurately than in the set where the checkerboard was placed much farther away. This inaccurate detection of corners let to a higher re-projection error.

Another reason for the error being higher is the lens distortion. When the checkerboard was placed farther away, it was noticed that the dimensions of the cells of the checkerboard were being distorted and in some cases, warped. This distortion might have also lead to greater error and thus, higher distortion coefficients.

To have a baseline reading, the MATLAB Camera Calibrator tool was implemented on the images of the first set (checkerboard within $1m$ of camera) and the calibration matrix was found to be -

$$\text{Total mean re-projection error} = 0.196925685118350$$

$$\text{Camera Matrix (K)} = \begin{bmatrix} 472.4659 & 0 & 328.5643 \\ 0 & 512.2113 & 261.8880 \\ 0 & 0 & 1] \end{bmatrix}$$

$$\text{Distortion Coefficients} \begin{bmatrix} k_1 & k_2 & p_1 & p_2 \end{bmatrix} = \begin{bmatrix} -0.0509 & 0.0019 & 0 & 0 \end{bmatrix}$$

Task 4 Tag-based Augmented Reality (5pt)

Use the pyAprilTag package to detect an AprilTag in an image (or use OpenCV for an Aruco Tag), for which you should take a photo of a tag. Use the K matrix you obtained above, to draw a 3D cube of the same size of the tag on the image, as if this virtual pyramid really is on top of the tag. **Document** the methods you use, and **show** your AR results from at least two different perspectives.

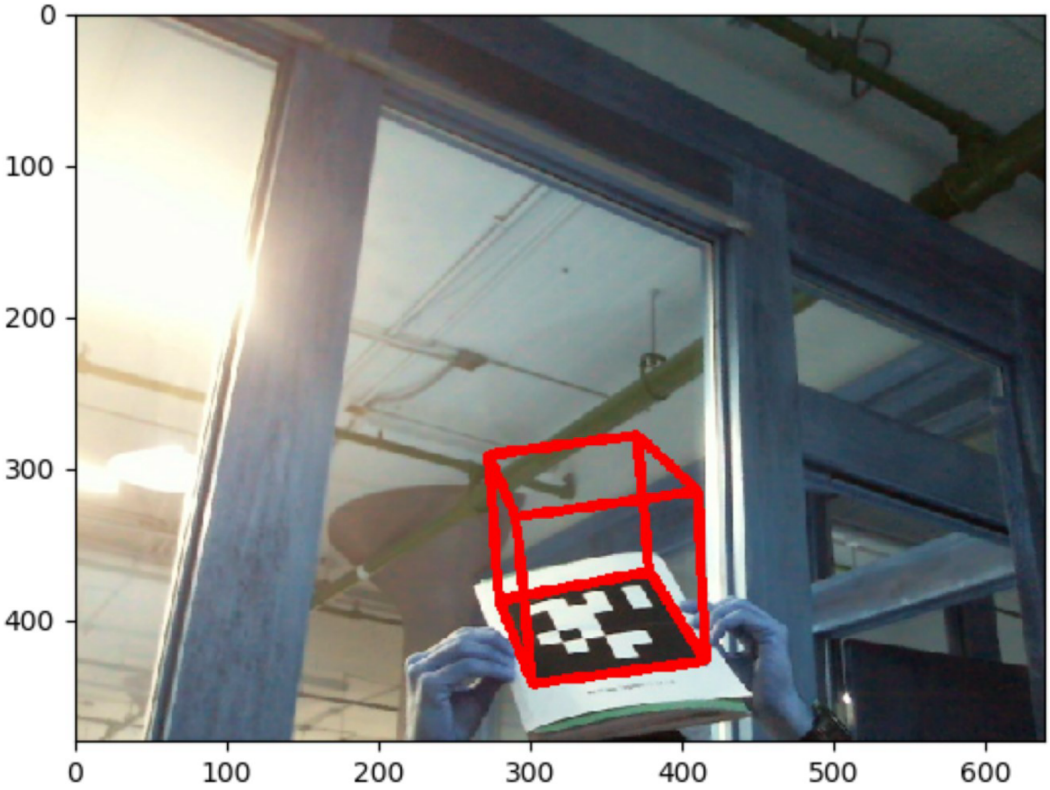

Figure 6: Projected Pyramid on checkerboard

Tips: There are many ways to do this, but you may find OpenCV's `projectPoints`, `drawContours`, `addWeighted` and `line` functions useful. You don't have to use all these functions.

Answers:
Here is the link to the thoroughly commented and documented code - q4_ans.py. Also, here are links to input images and output images. The ArUco Tags were generated using the ArUco markers generator! The chosen ArUco tag was of dictionary 5×5 (100) of size $100mm$ having ID of 2.

The below figures display the output of the code -

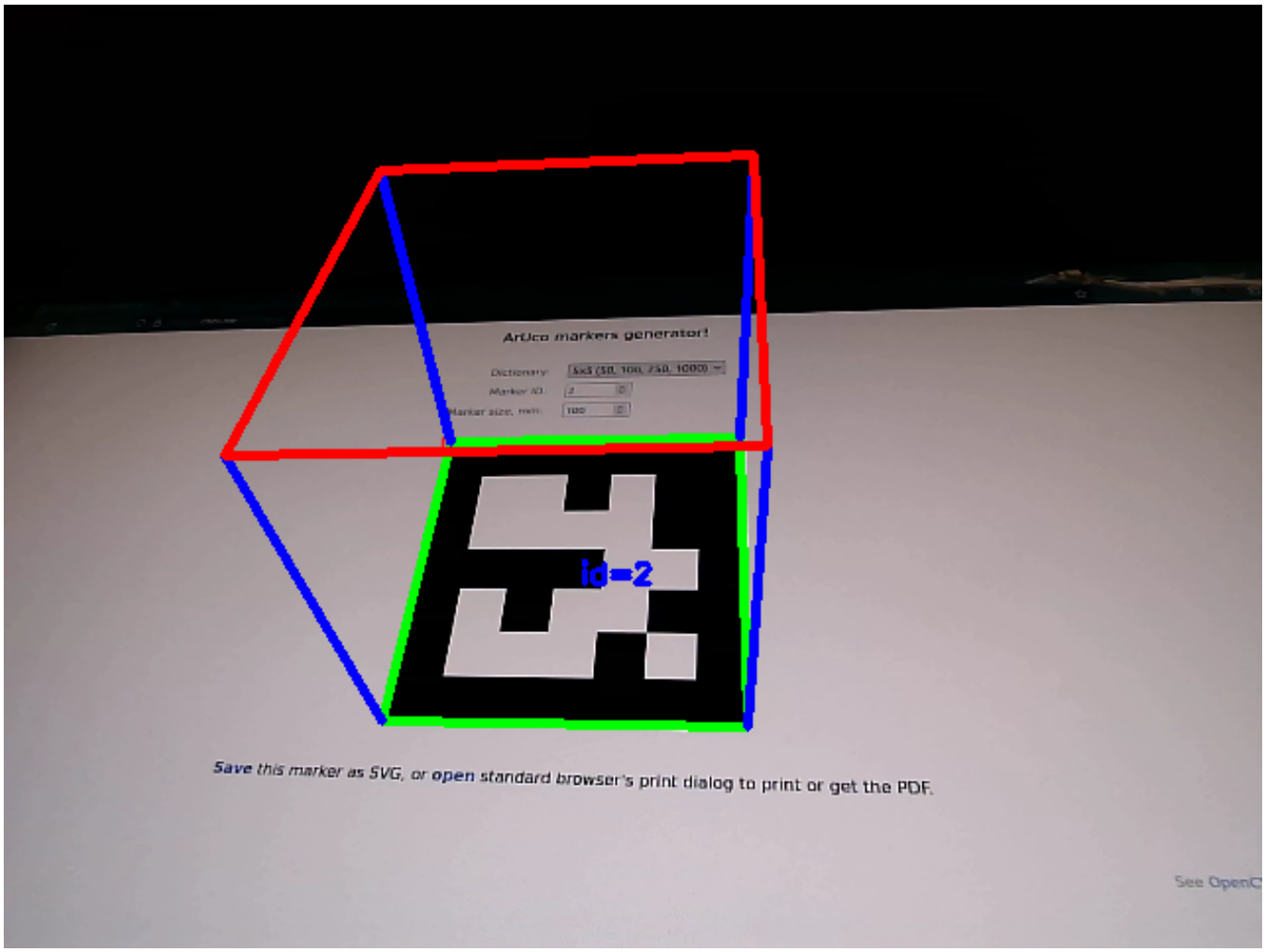

Figure 7: Projected Cube on the first perspective of the ArUco Tag

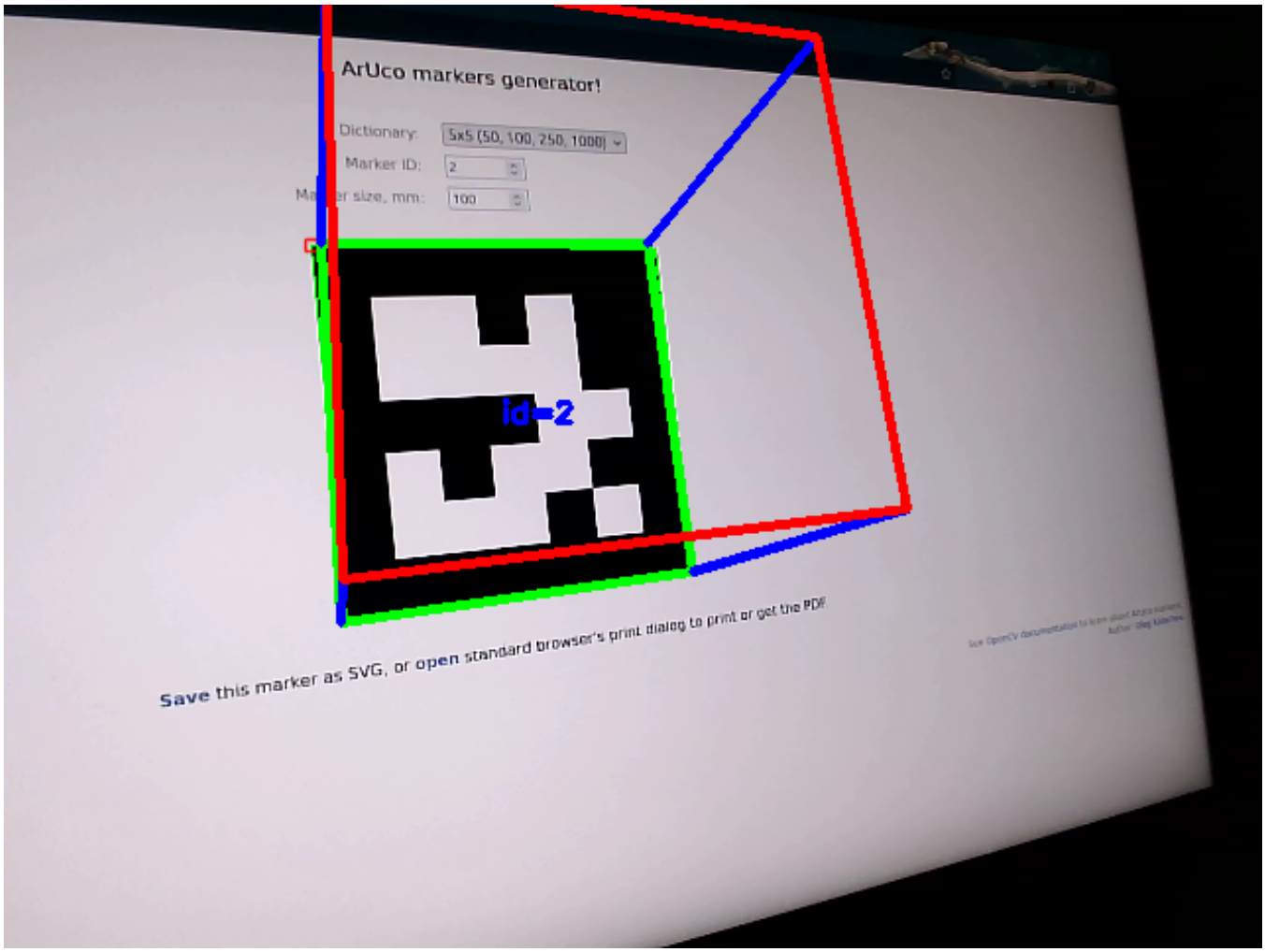

Figure 8: Projected Cube on the second perspective of the ArUco Tag

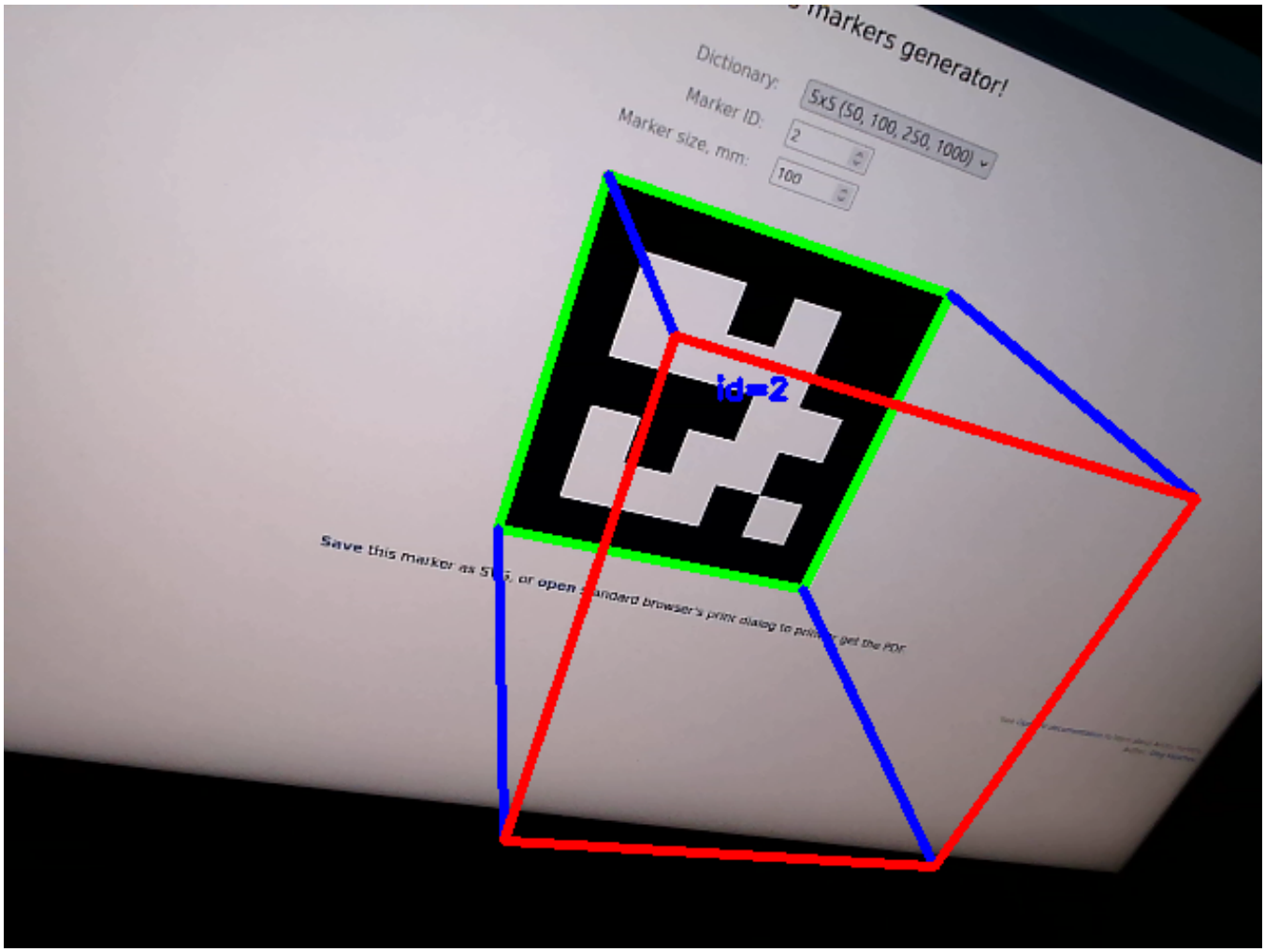

Figure 9: Projected Cube on the third perspective of the ArUco Tag