ECS709P Introduction to Computer Vision Coursework 1

Aditya Gupta (Student Number: 240754763)

November 2024

1 Transformations

Task b

Figure 1: Original Image

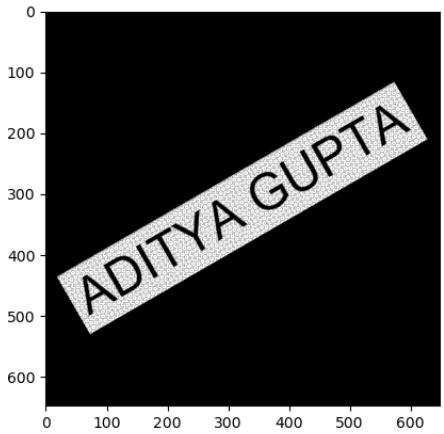


Figure 2: Image rotated by 30 degrees

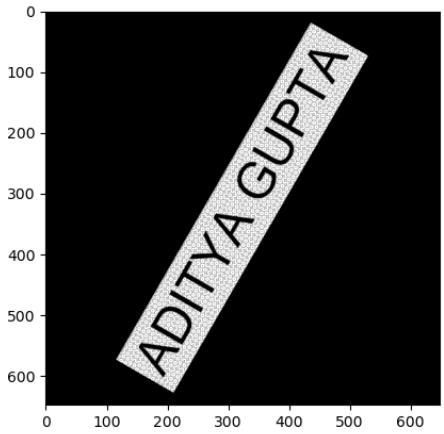


Figure 3: Image rotated by 60 degrees

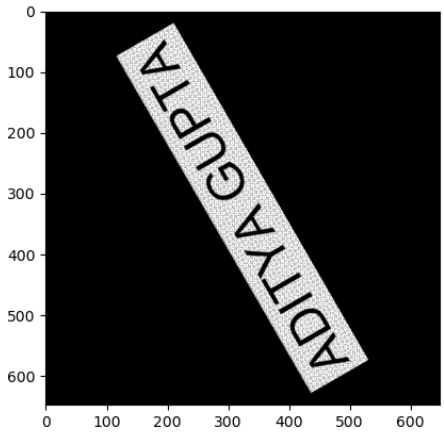


Figure 4: Image rotated by 120 degrees

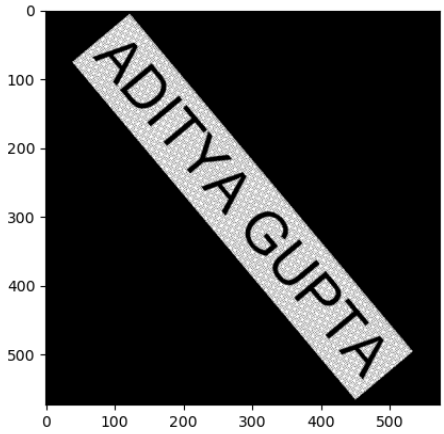


Figure 5: Image rotated by -50 degrees

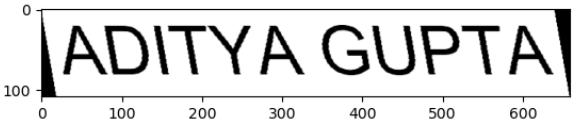


Figure 6: Image skewed by 10 degrees

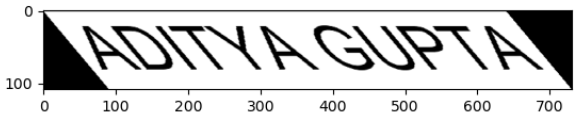


Figure 7: Image skewed by 40 degrees

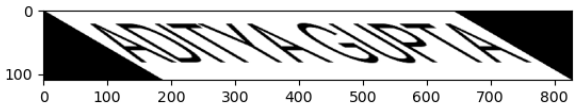


Figure 8: Image skewed by 60 degrees

To analyze the results of applying the rotation and skew transformations to the name image, let's begin by understanding the effect of applying a rotation matrix on the coordinates of the original image.

$$\text{Rotation Matrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

This matrix rotates each point (x, y) based on the angle θ relative to the system's origin. If the value of the angle is positive the image rotates anticlockwise otherwise it rotates clockwise. This transformation rotates the image without changing the size of the image. This is also reflected in Figures 2, 3, 4 and 5 above. The angles 30, 60 and 120 are applied in the first 3 cases and the images rotate by the same angles in the anticlockwise direction. As for the last case, the image rotates by 50 degrees clockwise.

$$\text{Skew Matrix} = \begin{bmatrix} 1 & tan(\theta) \\ 0 & 1 \end{bmatrix}$$

We will now talk about the skew transformations. The skew matrix can distort the image in one direction (x or y). For this task, we have chosen to work with a matrix transformation which performs a horizontal skew. This transformation shifts each point (x, y) horizontally based on their vertical position and angle θ relative to the system's origin. The points in the image will move to the right if the value of the angle theta is positive otherwise they will move to the left if the theta value is negative. We can see this effect in action in Figures 6, 7 and 8. The pixels at the bottom edge of the image start to move towards the right for skew operations of angles 10, 40 and 60, with higher angles making the pixels move further to the right.

For this task, I have chosen to use Nearest Neighbors interpolation for the sake of simplicity. That method just involved casting transformed coordinates to integer values to have valid coordinates. This method has a big disadvantage though; the edges of the images start to become more and more jagged with each transformation which is applied to the image. To fix this issue we can employ other methods of interpolation such as bilinear or bicubic

interpolation to make the edges of the image relatively smooth even when they have been transformed or resized. This will allow us to not lose image quality even after applying hundreds of transformations. Those methods however tend to come with their own set of issues. Employing such methods takes a massive toll on the overall computational efficiency, especially when operating on images with higher resolutions, as such operations are computed for each pixel in the image. In the end, it becomes a tradeoff between computational efficiency and retaining image quality. This is not a big concern in modern systems but certainly something to be aware of.

Task c

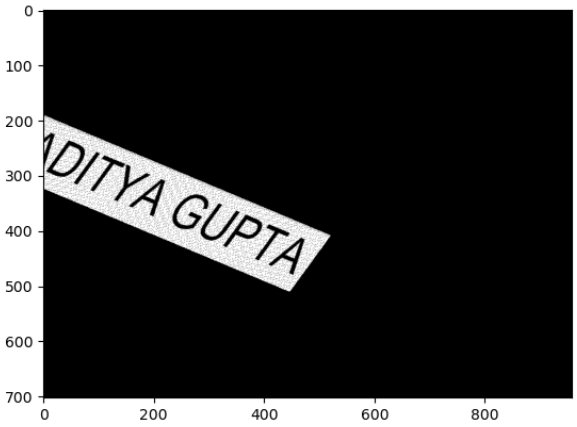


Figure 9: Rotated clockwise by 20 degrees followed by skew of 50 degrees

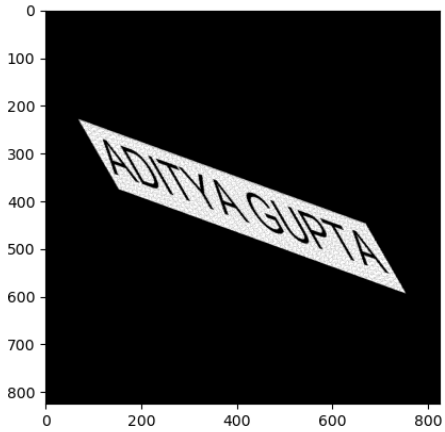


Figure 10: Skewed by 50 degrees followed by a rotation of 20 degrees

Looking at the images it is clear that my method of scaling the images upon transformation seems to move a part of the image outside the frame for case 1. Hence my method of scaling the new image frame seems to fail when multiple different transformations are applied one after the other. This is something which I will investigate further in the future. Aside from that we observe in the images, that the results of applying the transformations differ significantly based on the order of operations. This happens because each transformation involves matrix multiplication between the transformation matrix and the image coordinates matrix. Matrix multiplication is a non-commutative operation, which means that the order of the matrices in the computation can completely change the results. These operations can be summarized using the following mathematical equations (denoted by $\cdot$ for matrix multiplication):

- **CASE 1:**

$$\text{Rotated Image Matrix} = \text{Rotation Matrix}(\theta = 20) \cdot \text{Image Matrix}$$

$$\text{Rotation (20) and Skew (50) Image Matrix} = \text{Skew Matrix}(\theta = 50) \cdot \text{Rotated Image Matrix}$$

$$\text{Overall Transformation} = \begin{bmatrix} 1 & tan(\alpha) \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} = \begin{bmatrix} cos(\theta) * tan(\alpha) + sin(\theta) & tan(\alpha) * cos(\theta) - sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Here θ is -20 degrees and (α) is 50 degrees.

- **CASE 2:**

$$\text{Skewed Image Matrix} = \text{Skew Matrix}(\theta = 50) \cdot \text{Image Matrix}$$

$$\text{Skew (50) and Rotation (20) Image Matrix} = \text{Rotation Matrix}(\theta = 20) \cdot \text{Skewed Image Matrix}$$

$$\text{Overall Transformation} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} 1 & tan(\alpha) \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & tan(\alpha) * cos(\theta) - sin(\theta) \\ sin(\theta) & cos(\theta) + tan(\alpha) * sin(\theta) \end{bmatrix}$$

Here θ is -20 degrees and (α) is 50 degrees.

Since the overall transformation matrices applied to the image differ entirely for each case, the resulting images will also be different.

2 Convolution

Task b

The averaging kernel is a **k**x**k** matrix (where k is the width/height of the kernel) which allows us to calculate the average intensity for each pixel within the affected region. This filter smooths out the image and gives a blurring effect. which smooths out the overall image. It blurs the image by reducing high-frequency details, which can remove noise but it will also severely reduce the sharpness around edges in the image.

(a) Original Image (b) Averaged Image (5x5 kernel)

Figure 11: Comparing Averaged and Original Image

The effect of this filter can be seen in Figure 11 (above). Here I have used **5**x**5** filter to make the effect of the mean filter even more intense to showcase how much it can smooth out the image. I chose the first car image for this task as it had sharp edges and it served as the perfect candidate to display the effects of the the averaging filter. The averaged image makes the edges a lot less prominent and even reduces the effect of strong light in an image.

Task c

1. Kernel A is a 3x3 Gaussian blur filter. This type of filter also causes a smoothing effect when applied to an image. When this filter is applied the pixels in the center are emphasized while the pixels which are closer to the edges of the image

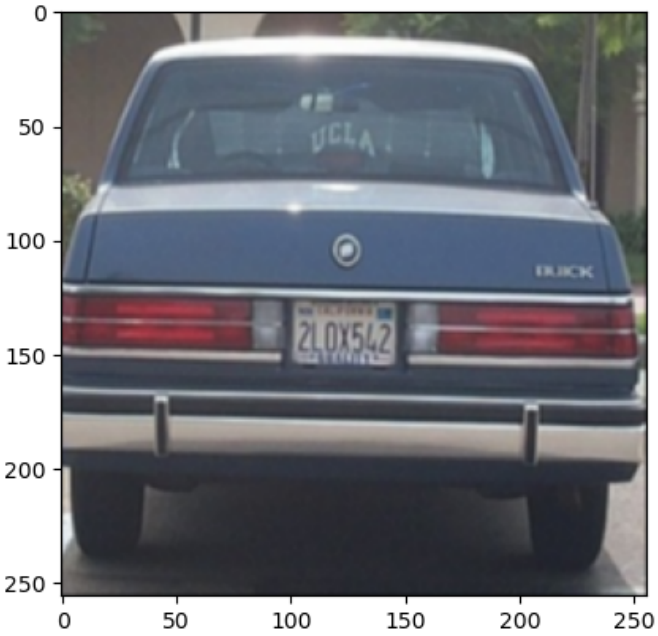


Figure 12: Filter A (Gaussian Blur filter) applied to car-1.jpg

2. Kernel B is the Laplace filter which is used for edge detection. This kernel enhances all edges within the image. The filtered result has emphasized boundaries along the diagonal directions. As it emphasizes all edges and makes the image's attributes sharper, it can make details in the image more prominent but can also make the results more noisy (as can be seen in the image below (Figure 13)). Unlike most other filters which divide by the sum of the kernel, here we cannot do that as the sum of the filter is equal to zero. This can make the pixel values in the filtered image invalid (¿ 255 or ¡ 0) as some values are being multiplied with negative values while some are being scaled by 4 because of the center value. To fix this, I had to apply a thresholding method on the result of the filter to display a valid image. This creates a binary image where all values above the threshold are converted to 255 and all values below it to 0. This is why the image below looks black and white, as the only pixel values after applying the threshold are 0 and 255.

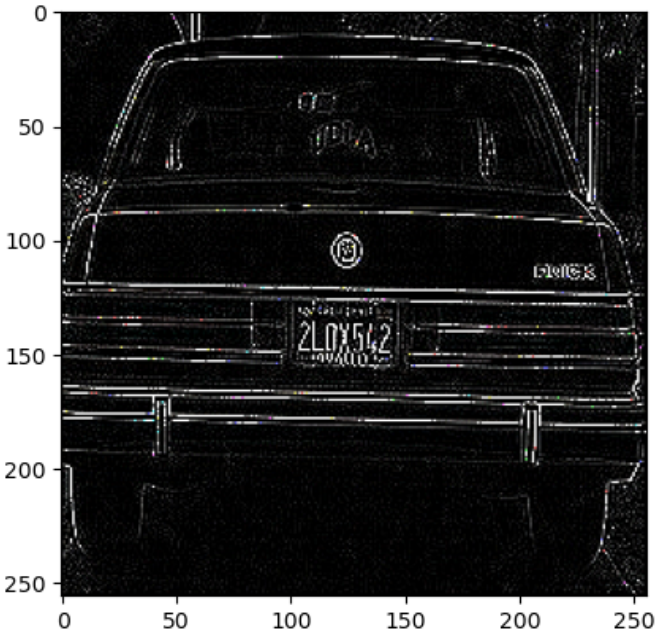


Figure 13: Filter B (Laplace filter) applied to car-1.jpg

Task d

1. Here we filter the image using the Gaussian kernel (Kernel A) twice. Each time the filter is applied the kernel blurs out each edge and makes all sharp details more smooth. This makes the characters on the number plate and model name of the car harder to read. Unlike the mean filter, the Gaussian filter uses a weighted approach

and emphasizes the central pixel values. This means the details haven't completely been obscured even after applying the filter twice.

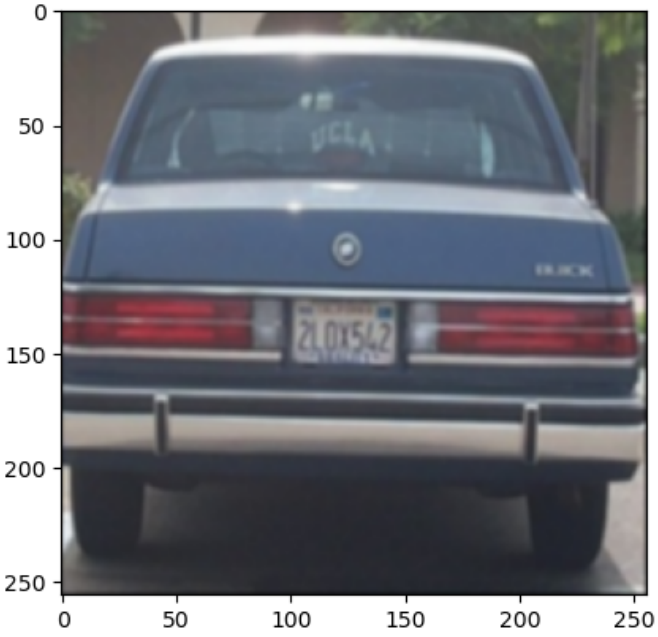


Figure 14: Filter A applied to image twice

2. For this task we apply a 'Gaussian Filter' followed by the 'Laplace edge detection Filter'. We begin by applying Kernel A, which flattens out the existing edges and blurs all of the details. This is followed by applying Kernel B, which tries to enhance the edges in the car image. The resulting image has very faint edges. The effect of the Gaussian blur becomes easy to see in the resulting image as the details closer to the centre of the image are retained while the ones closer to the edges have become quite faint.

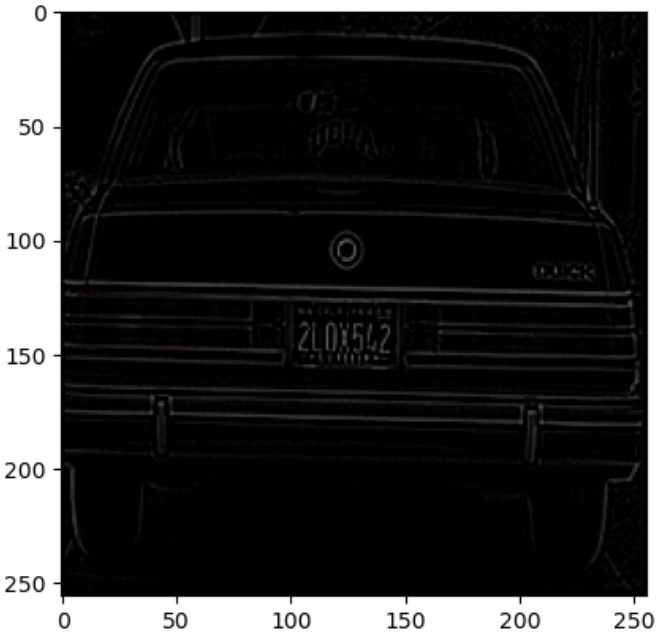


Figure 15: Filter A followed by Filter B is applied to the image

3. Here we apply the kernels in the opposite order to the previous task, which means we begin by enhancing the edges within the image and then blur and smoothen them out. As the edges were already enhanced, the effect of the blur doesn't flatten the edges too much and we end up with an image with lighter edges than the original image, but edges which are still more prominent than in the previous case as can be seen in the image below (Figure 16). Applying the Laplace filter on the image first also increased the noise in the image. There are small white specs close to all of the edges of the cars. The final image doesn't look too noisy as the Gaussian filter de-emphasizes such details and smooths out all sharp points in the image.

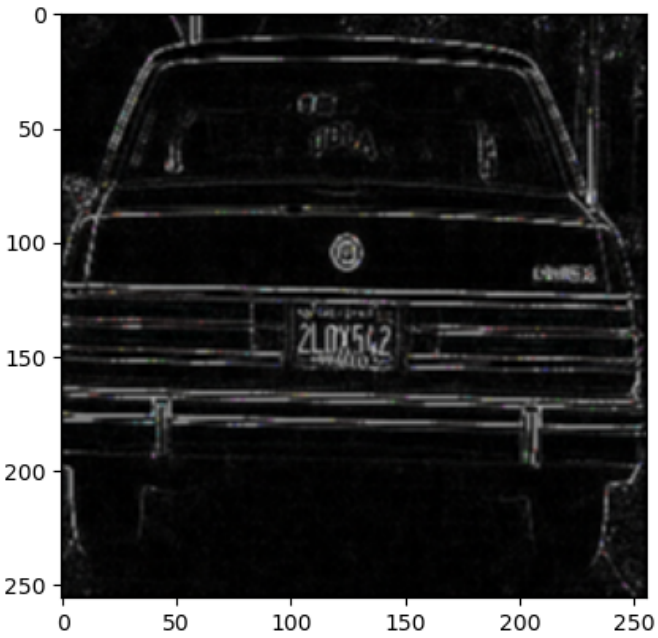
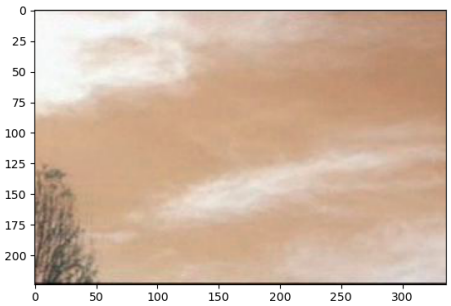


Figure 16: Filter B followed by Filter A is applied to the image

3 Video Segmentation

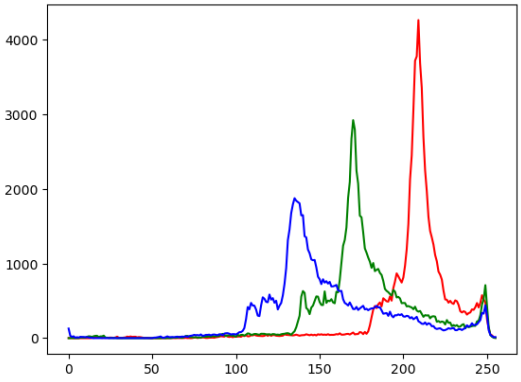
Task a



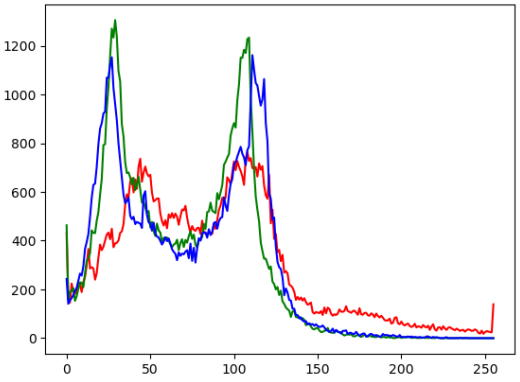
(a) Non Consecutive Frame 1

(b) Non Consecutive Frame 2

Figure 17: Non-Consecutive Frames



(a) Color Histogram Frame 1



(b) Color Histogram Frame 2

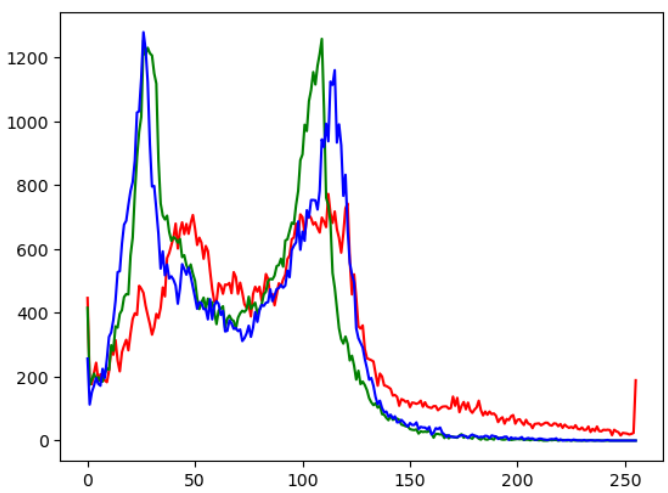
Figure 18: Color Histograms for the Non-Consecutive Frames

Task b

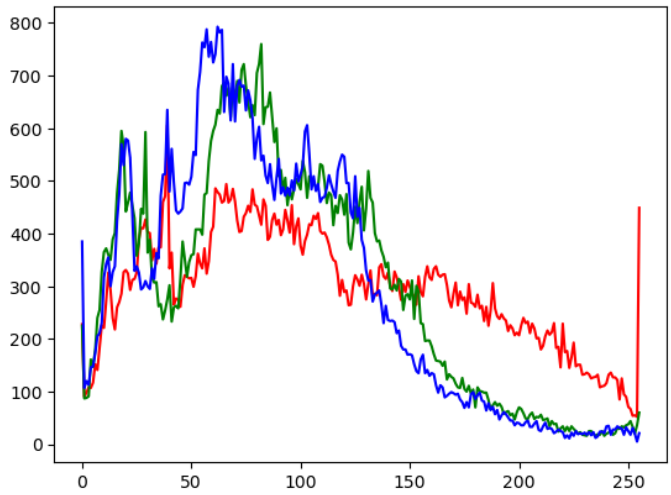
(a) Non Consecutive Frame 1

(b) Non Consecutive Frame 2

Figure 19: Consecutive Frames



(a) Color Histogram Frame 1



(b) Color Histogram Frame 2

Figure 20: Color Histograms of I_t and I_{t+1}

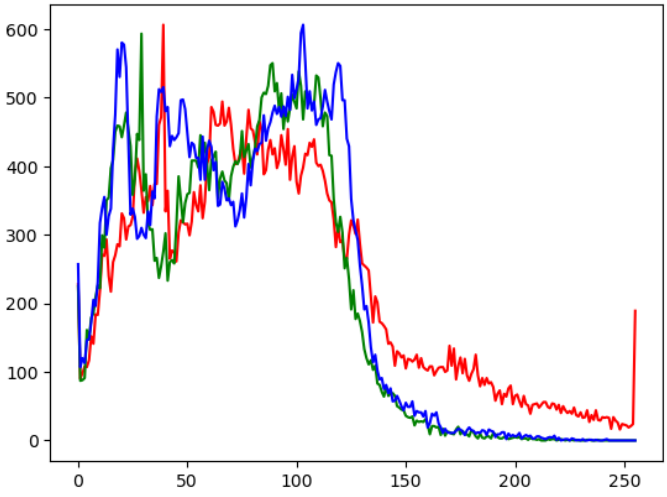
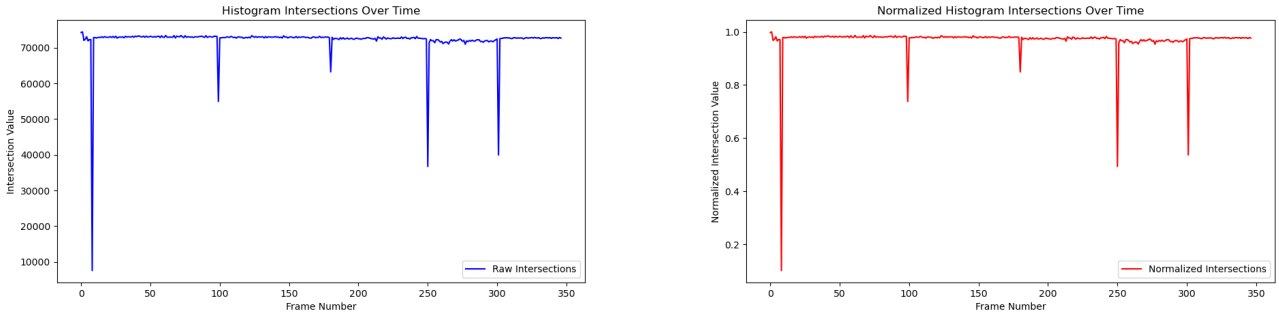


Figure 21: Histogram Intersection of I_t and I_{t+1}



(a) Raw Intersection Results

(b) Normalized Intersection Results

Figure 22: Raw and Normalized Intersection Results for the Video Sequence

Task b

The colour histogram of an image presents a simple yet powerful way to capture the colour distribution of an image. This can be especially useful when comparing frames in a video sequence and can provide insight into the similarity of different images without the use of machine learning techniques. In this task, we find the histogram intersection of all consecutive frames in the video (DatasetB.avi). To compute the intersection we just take the minimum frequency of each pixel value from the two histograms for which we are computing the intersection. The intersection of these histograms serves as a similarity metric. The higher the intersection the more similar are the frames in the video sequence. After finding these intersection values, we normalize them. For normalizing the histograms I used the method specified in the following article: **The simplest Classifier-Histogram Intersection**. *Normalizing just involves dividing the intersection values by the maximum value within the intersection set.* After plotting the values in the raw intersections and the normalized intersections for the video sequence, it is clear that the plots for the raw and normalized versions(Figures 22a and 22b) look nearly identical with the only difference being the scale for the intersection values. For the normalized case it's between 0 and 1 while in the raw intersection it is up to more than 70,000.

Task c

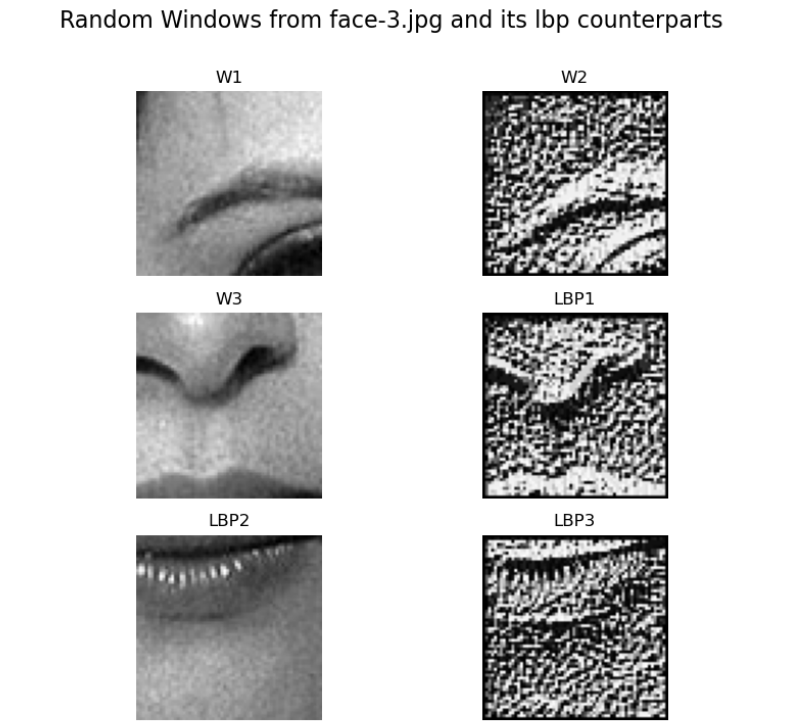
1. *What does the intersection represent for the input video*: The color histogram intersections represent how the overall colour distribution of frames in the video varies over time.

2. *Can a decision about scene changes be made from the histograms*: Whenever there isn't much change in a scene, the overall intersection values tend to stay quite consistent, with only slight variations, as demonstrated in the histogram intersection plots for the video (DatasetB.avi). However, when a scene change occurs, the intersection value drops significantly, indicating a substantial difference between the previous and the current frame. This drop signals that a scene change has taken place.

3. *Robustness of the video histogram to scene changes*: For the video in the provided dataset the colour histogram is quite robust. We know that the colour histogram mainly checks for changes in colour values in each frame (or how the colour distribution changes over time), which means that hypothetically it is possible that if the scene changes to a frame with a very similar colour distribution the histogram intersection might fail to detect it. It could also register false positive scene changes if the colour distribution within the same scene is changing rapidly due to lighting conditions or specific scenarios (for example a video of people in a nightclub).

4 Texture Classification

Task a



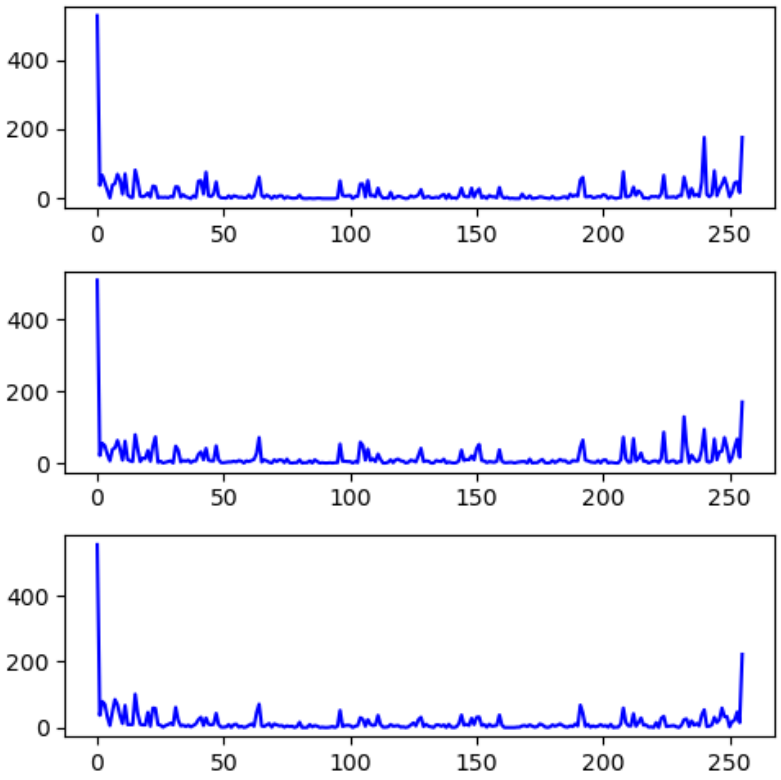


Figure 23: Histograms of LBP Windows

Task b

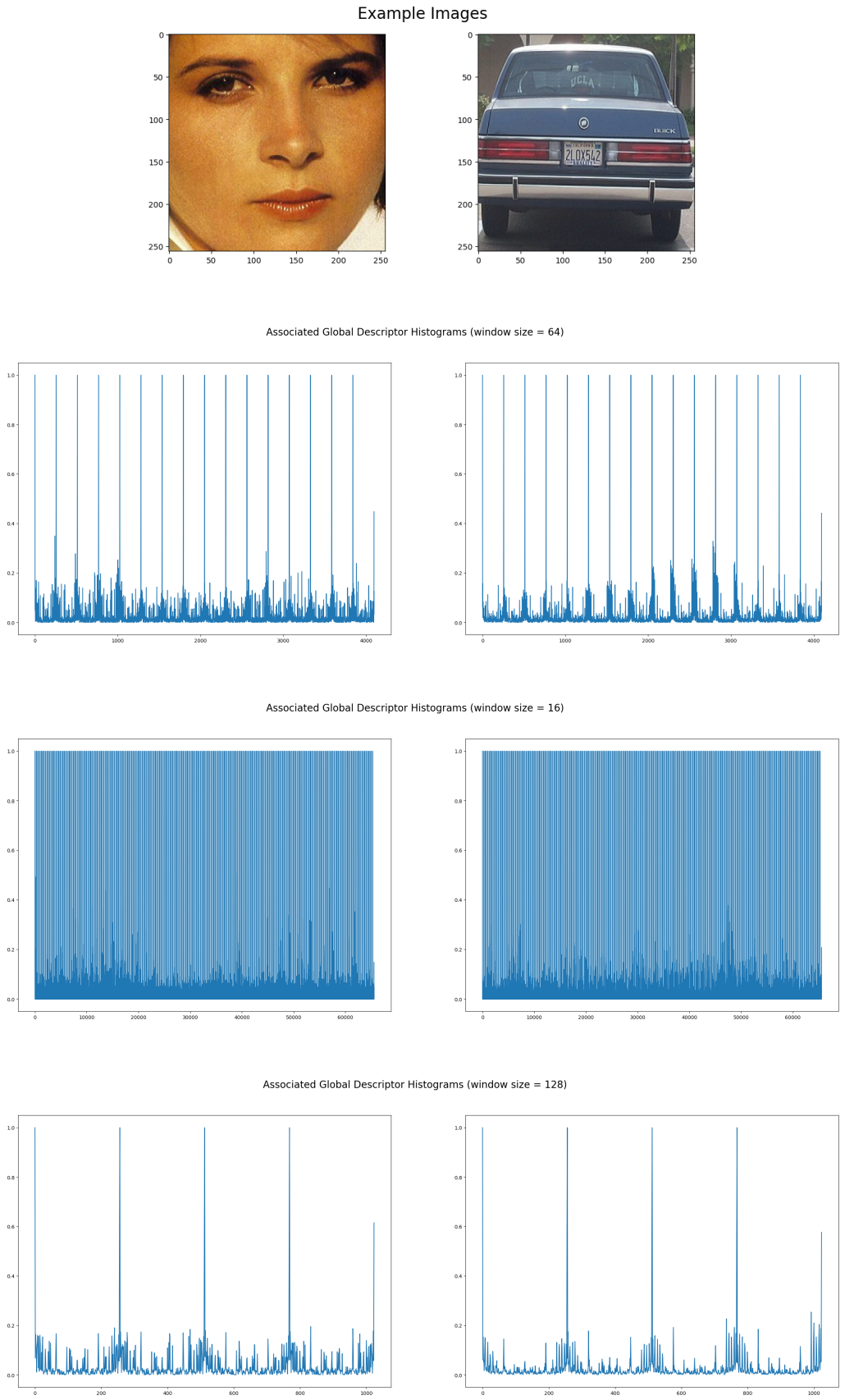


Figure 24: Example Images, and their associated global descriptors for different window sizes

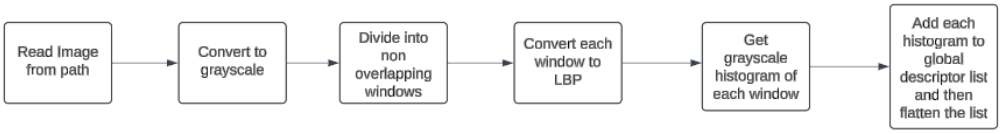


Figure 25: Block diagram explaining how global descriptors for each image are constructed are constructed

For this task, I defined the global descriptor by concatenating all of the LBP histograms of the different windows of the image as described in the question. For classifying different images I used the intersection of the global descriptors as a similarity criteria to define which images are faces or not. The intersection of these descriptors is defined very similarly to the intersection of color histograms with one key difference; As I wanted to get a single similarity score between two images I summed up the intersection values. We know that images "face-i.jpg" (i ϵ $[0, 3]$) represent images with faces while "car-i.jpg" (i ϵ $[0, 3]$) in Dataset A represent images with cars.

I have put the descriptor intersection scores in the heatmap below. I chose a heatmap for this analysis as it allows me to compare all images at once. Here as expected the face images have a high intersection value with other face images. Comparitively the intersection scores for car images among each other are lower. This can be due to the fact that they have more prominent distinguishing features between them (build type, design, colour of the model etc). An interesting observation worth noting here is that image 2 and image 5 have relatively high intersection score (84.89) especially given image 2 is a face and image 5 is a car. The score is comparable to the intersection scores of the car images among each other. This is probably due to the fact that the images have a very similar colour distribution especially when converted to grayscale which might have contributed to the anomalous value. Here I have opted for a window size of 64 pixels.

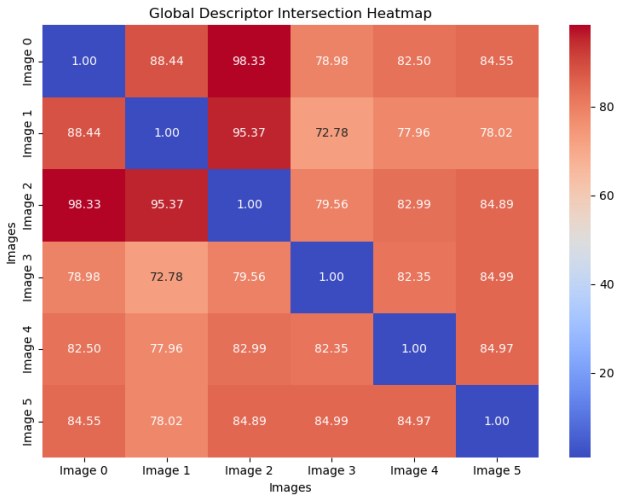


Figure 26: Global Descriptor Intersection heatmap with window size = 64

Task c

For this task, I decrease the window size to 16. As can be seen in the histogram the global descriptors managed to get a lot more information about the images than before, making the classifications more robust. Here we also manage to correctly classify similar-looking cars a lot better. Here the classification criteria is just a high intersection value for things which we know are the same type of image. Car 2 and Car 3 (Images 4 and 5) have a similar shape and are a lot more similar to each other relative to Car 1(Image 3) as they have a more modern build meaning they have smoother edges compared to the harsh edges and blocky design of Car 1. This is also reflected by the intersection score in the heatmap below. Looking at Figure 24, we can see that the histogram of the global descriptor for window size = 16 is quite dense. Based on looking at the descriptor representation and the heatmap values it is clear that a smaller window size manages to capture more subtle details and get more information on local textures in the images.

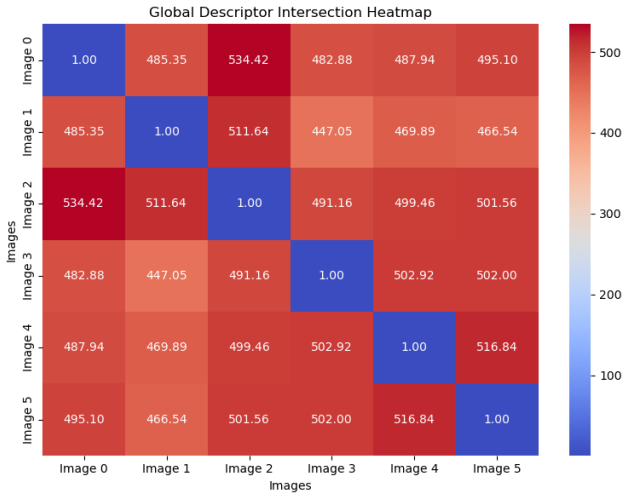


Figure 27: Global Descriptor Intersection heatmap with window size = 16

Task d

Now we move towards using a larger window size of 128 pixels. This provides the worst intersection results of the three cases we have analyzed and it is worse at capturing the similarity between the car images. Based on trying all three window sizes with global descriptor intersection scores as a classification criterion, seems to perform relatively well on face images but struggles when classifying other types of objects. To remedy that it seems that using a smaller window size helps capture subtle changes in the image textures in general and not just for face detection.

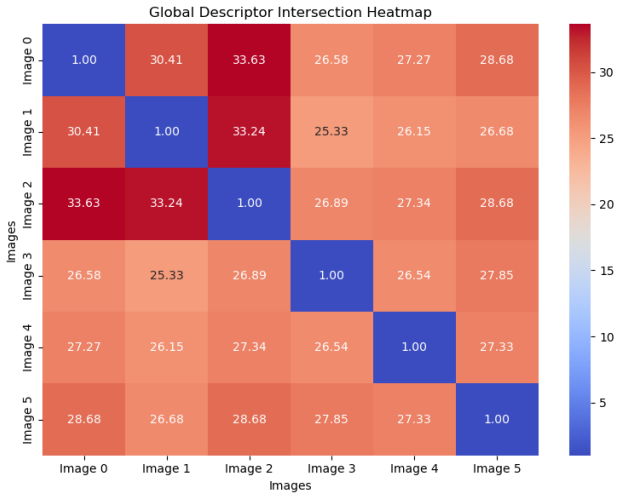


Figure 28: Global Descriptor Intersection heatmap with window size = 128

Task e

One method for extending Local Binary Patterns (LBP) in dynamic texture analysis is Volume Local Binary Patterns (VLBP). VLBP enhances the basic LBP by incorporating spatiotemporal information across multiple frames, meaning it captures information along the X (horizontal), Y (vertical), and T (temporal) axes, with T being represented by the frame count in the video. Unlike traditional LBP, which calculates binary patterns based solely on neighbouring pixels within a single frame, VLBP analyzes binary patterns across a 3D volume of consecutive frames. Specifically, VLBP considers a central pixel alongside its neighbouring pixels in both the spatial (XY plane) and temporal dimensions. Source (Wikipedia)

5 Object Detection

Images for Tasks a and b

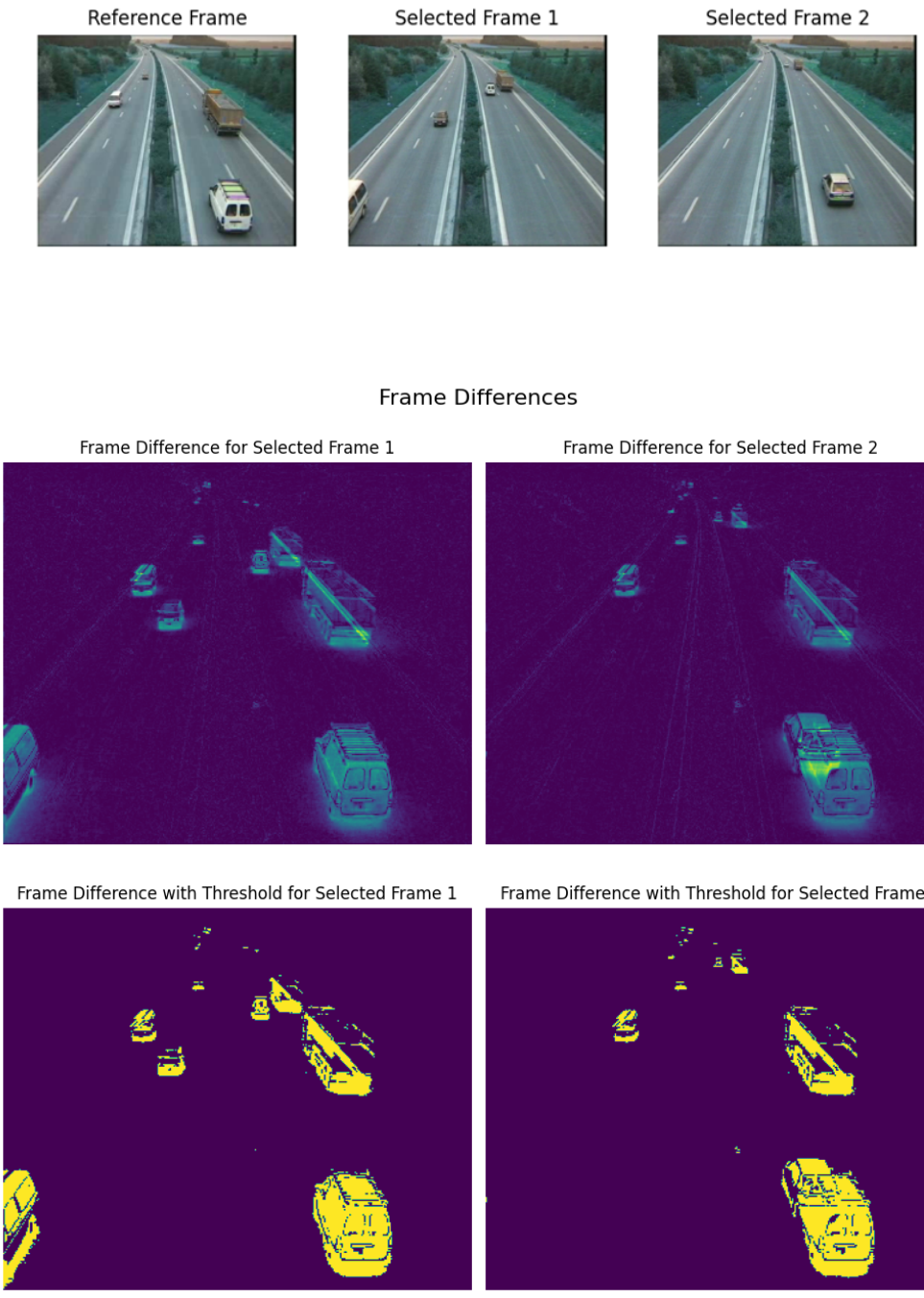


Figure 29: Reference, Selected Frames, and Frame Differences for Task 5a

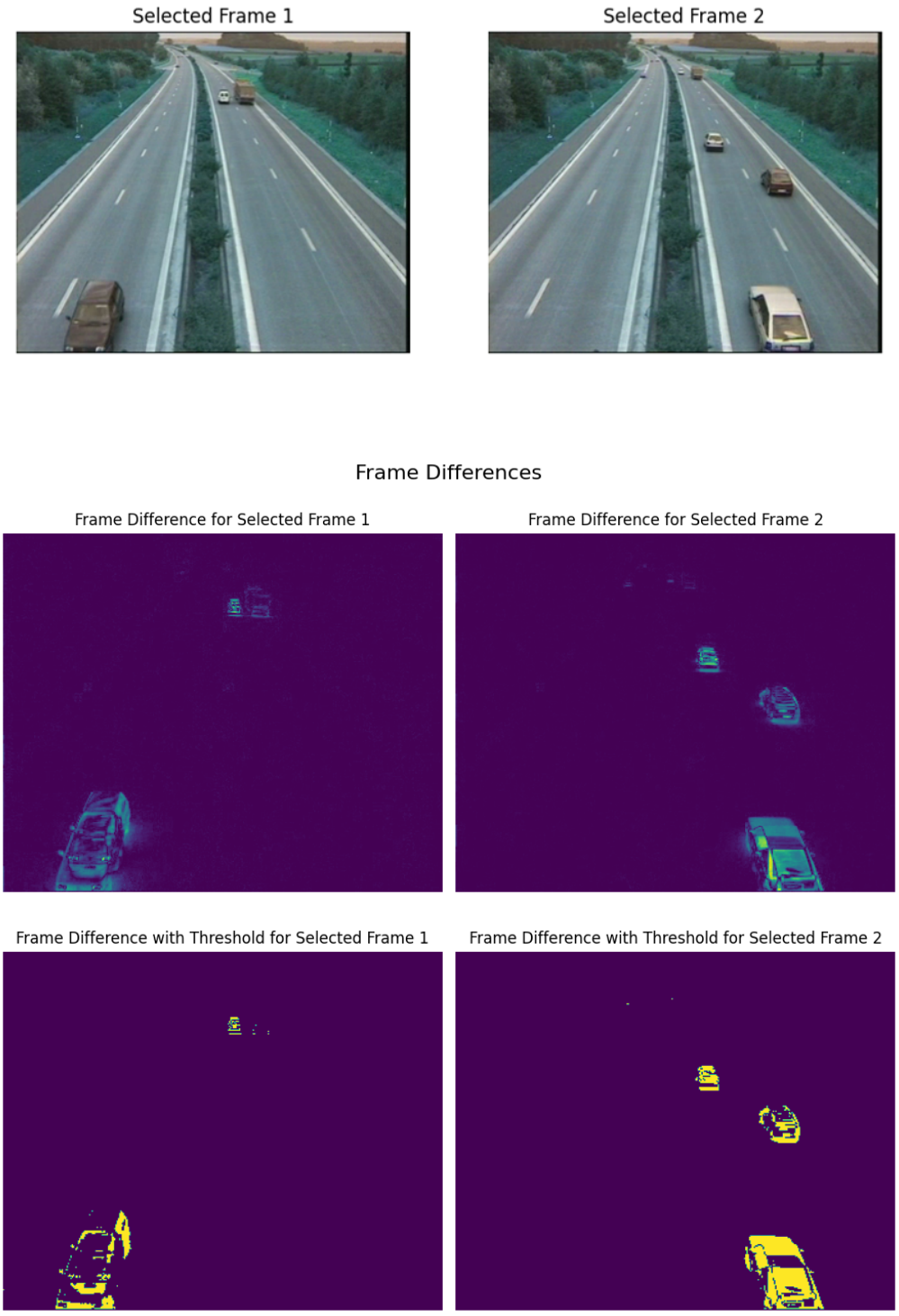


Figure 30: Reference, Selected Frames, and Frame Differences for Task 5b

Comments on Tasks a and b

Picking the classification threshold for this task was quite challenging as its dependent on trial and error, and there doesn't seem to be any other concrete method of determining it. From experimenting with different values for the threshold I came to the conclusion that too low of a threshold value ends up capturing a lot more noise; things which are static can also have fluctuations in their intensity based on how the lighting of the scene changes and usually these changes are subtle, but with a low classification threshold, this noise also gets captured in the frame differences. On the other hand using a higher value for this threshold leads to not enough information within the scene being captured, as sometimes large parts of entire objects would be missing in the frame difference even though they are moving. This made me pick a threshold value with intensity of 40 as that seemed to capture the moving cars, and their shadows and when in later frames they would move further away it's still possible to see the car and it doesn't just turn into noise.

Comparing the results of the frame difference for task (a) which takes the first frame as a reference and the second task which takes the difference between consecutive frames it is clear that the second approach is significantly better at capturing the overall motion of each object. There is also a lot less noise compared to the first case. These improvements can be attributed to the fact that the second case captures change on a smaller time frame as well as the fact that the second case isn't bound by the starting conditions of the scene, which means that any changes in the background due to change in lighting over time or due to some other factors would already be removed by taking the difference for each change. This can be seen in Figures 29 and 30. These figures also reflect the effects of applying the classification threshold and taking the frame difference.

Task c

By using the weighted temporal averaging algorithm, I was able to identify the background reference frame, as shown in the figure below (Figure 32). However, since this is still an averaged image, some artefacts appear in certain areas where objects remain on-screen for extended periods in the video. This creates a sort of after-image effect, resulting in faintly smudged versions of the cars that were on the road. These aren't too visible though, so it still looks like we are just looking at an image of an empty highway road. using this image instead of the first frame would mitigate the issues of using the first frame as this image would be an accurate representation of the starting condition but wouldn't have any objects in the image. This way when a difference is taken the noise from the first frame wouldn't be visible and we will be able to capture the presence of each object properly.

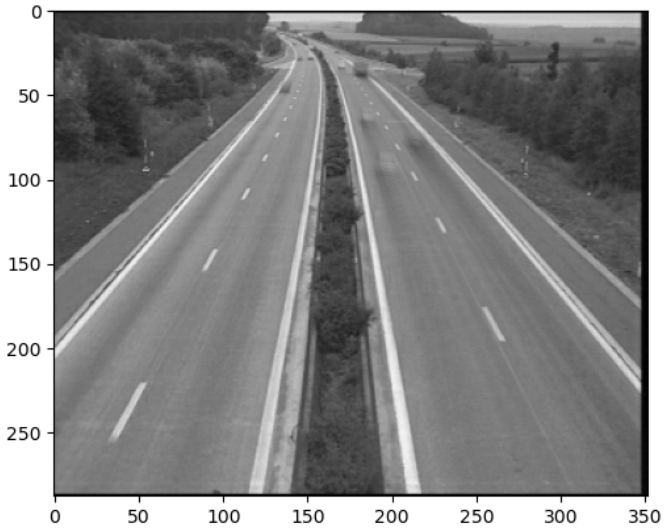
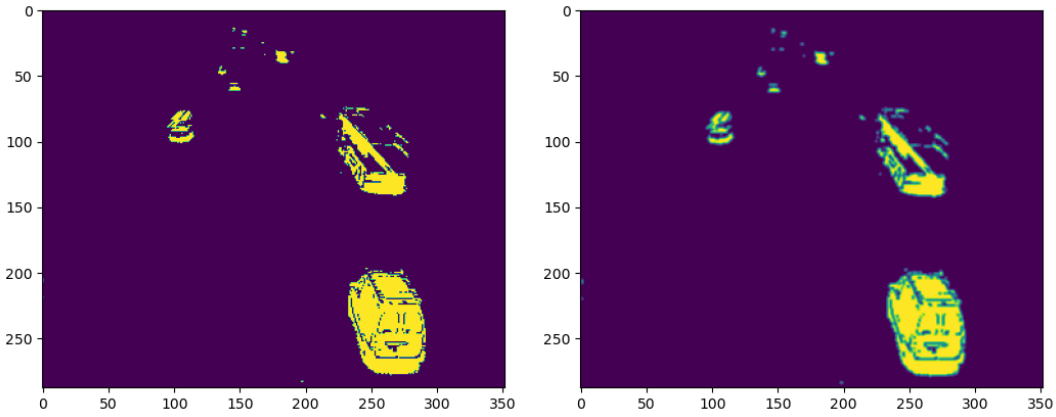


Figure 31: Reference Background Frame created using Weighted Temporal Averaging algorithm

Task d

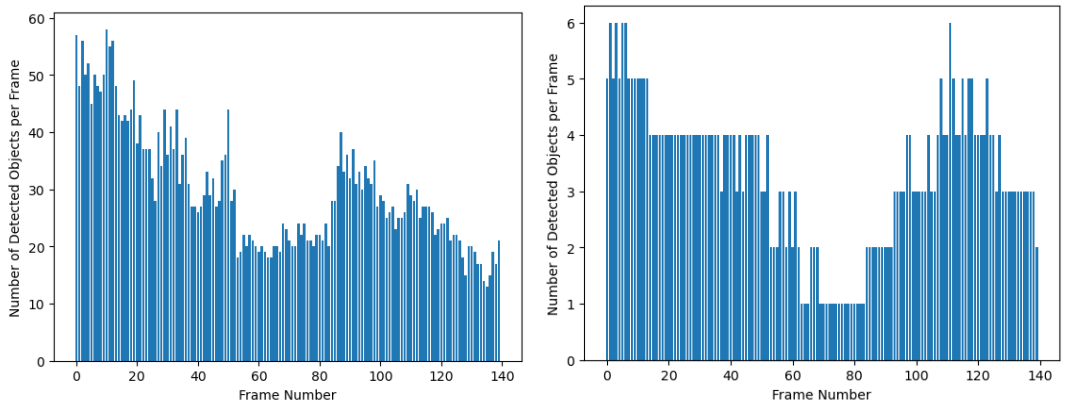
For my solution, I took the difference of each frame in the video with respect to the generated background followed by applying the classification threshold to the difference. To find the number of objects within each scene I made use of a flood fill algorithm which works well with our use case as we have a binary image after applying the classification threshold which is necessary for using this algorithm. This algorithm visits all neighbouring pixels which haven't already been visited and which have a non-zero pixel value. This is a simple way of estimating all onscreen objects, however, this algorithm's results are susceptible to the noise within the image. As the algorithm is only trying to locate non-zero areas in the difference image it can start including noisy spots in the image as moving objects. This can explode the count of moving objects as can be seen in the first bar plot in Figure 33a. In an attempt to fix this, I tried applying different blurring filters in order to denoise each frame. This was quite computationally expensive but gave significantly better results. For this, I applied a convolution kernel for a mean filter and Gaussian blur filter to see how it affects the moving object count. This worked incredibly well and the false positive objects registered by the flood fill algorithm were significantly reduced as can be seen in Figure 33b. I have only included the bar plot for Gaussian blur filter as both the mean filter and gaussian blur achieved the same results.



(a) Frame without Denoising

(b) Frame with Gaussian Blur applied to it

Figure 32: Effect of Denoising Frames in a video sequence



(a) Bar plot without any denoising

(b) Bar plot with frames Denoised using a Gaussian Blur Filter

Figure 33: Motion Detection over Video Sequence