NTNU
Kunnskap for en bedre verden

Department of Computer Science

IT3212 - Data-Driven Software

Assignment 2

Authors:
Birk Strand Bjørnaa
Carl Edward Storlien
Christian Stensøe
Åsmund Løvoll

25.10.2024

Table of Contents

List of Figures

List of Tables

1 Fourier Transform

1.1 Frequency Spectrum

To convert a colored image with RGB values, the mean of the three was measured for each pixel. To examine the frequency spectrum with magnitude of the grayscale image, we utilized `NumPy's fft` module. Since the image is 2-dimensional, the discrete `fft2` transformation was used. The function calculates the Fourier transform of each row first and then calculates the Fourier transform of each column of the result - or vice versa, the exact implementation is not known to us, but the functionality remains the same. This aims to generate the landscape of pixel values in the form of sine and cosine waves with different frequencies and phases as complex values.

We plotted the result as a frequency spectrum showing the magnitudes of the different frequency components. Figure 1 shows the original image converted to grayscale. Its shape is `596*894` pixels. Figure 2 shows the frequency spectrum with magnitudes of the image after Fourier transformation. It is common practice to shift the zero-frequency/DC component from the top-left corner to the center, which is also shown. Frequency in the context of a grayscale image means the variation in pixel intensity.

The shifted frequency spectrum is divided into quadrants. The **bottom-right** quadrant contains positive horizontal and vertical frequencies (row and column), the **top-left** quadrant contains negative horizontal and vertical frequencies, the **bottom-left** quadrant contains negative horizontal and positive vertical frequencies, and the **top-right** quadrant contains positive horizontal and negative vertical frequencies. The low frequencies are surrounding the center of the spectrum, while the high frequencies are close to the edges. The DC component (bright spot in the middle) is the average pixel intensity of the image.

It is normal for natural images to have "decaying" frequencies, meaning more of lower frequencies than higher frequencies[1]. This can be seen in our shifted frequency spectrum as well with brightness around the center. The bright vertical and horizontal line indicates strong vertical and horizontal frequencies in the image. The diagonal lines indicate diagonal patterns or edges in the image. This can be verified in the image as well. We can, for instance, see that her arm is at an angle with sharp transitions between bright and dark.

Figure 1: Original image in grayscale

[1] Wronski 2021.

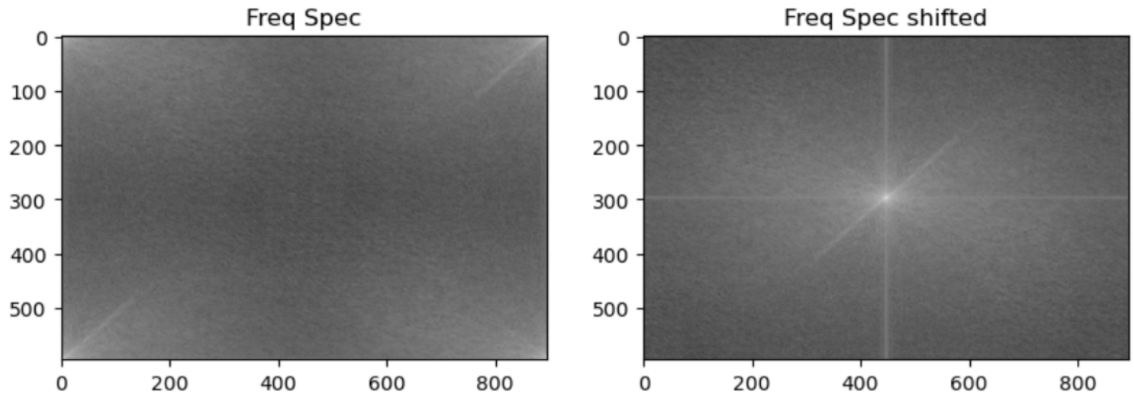
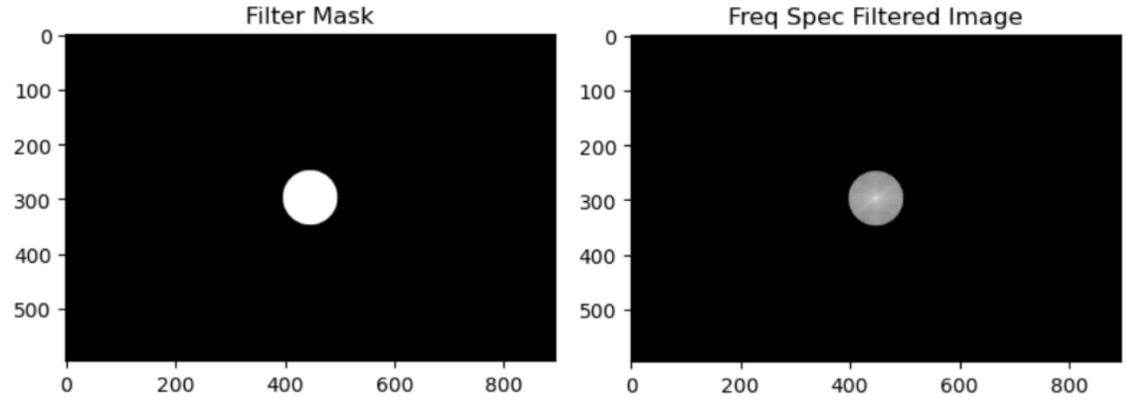


Figure 2: Frequency spectrum with magnitudes

1.2 Low-Pass Filter

After the image is transformed into the frequency domain, it is possible to filter the image based on frequencies. A low-pass filter filters out higher frequencies. To filter the image in the frequency domain, the Fourier coefficients are multiplied by a filter mask. For a low-pass filter, the filter mask has 1s where the desired coefficients are and 0s where the undesired coefficients are. When the Fourier coefficients are rearranged to center the zero component, the filter mask should have 1s near the center of the frequency spectrum and 0s elsewhere. The filter mask was created by setting the elements within a centered circle with `radius = 50` to 1 and 0 elsewhere.

Figure 3a shows the filter mask and figure 3b shows the frequency spectrum after applying the filter.



(a) Low-pass filter mask

(b) Low-pass filtered frequency spectrum

Figure 3: Visualization of the low-pass filter mask and its effect on the frequency spectrum

The Fourier coefficients can be transformed back to an image by applying the inverse transform function `ifft2` which calculates the pixel values. The filtered image is shown in figure 4. Compared to the original image, it can easily be verified that the high frequencies are filtered out by looking at the lack of details in the image. For instance, the belt has no fine-detailed pattern as in the original image. The low frequencies contain a great amount of information, so even though a large portion of the frequency domain has been filtered out, the image is quite similar to the original image. For the purpose of removing high-frequent noise from the image,

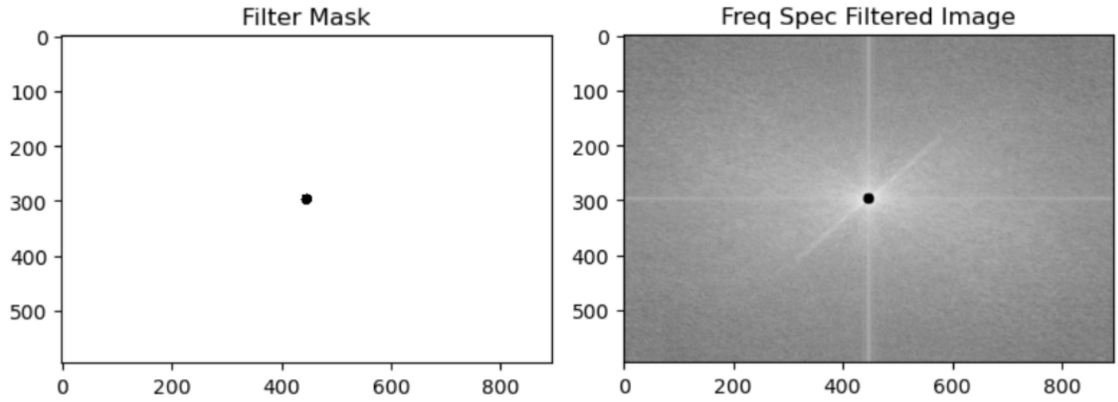
"Ringing" artifacts can also be seen near sharp transitions, which is typical for a low-pass filtered signal[2]. A solution for removing the ringing artifacts is to increase the portion of the frequency range that is kept in the filtering process. Another solution is low-pass filtering without a sharp cutoff in the filter mask. Instead, using a filter with a smooth, gradual cutoff would reduce the likelihood of ringing artifacts[3].

Figure 4: Low-pass filtered image

[2] *Ringing artifacts* 2024.
[3] *Low-pass filter* 2024.

1.3 High-Pass Filter

The opposite filter of a low-pass filter as previously demonstrated is a high-pass filter. Filtering out the lower frequencies can be done in the exact same way with a filter mask by switching the 1s and 0s. Figure 5a shows the filter mask for a high-pass filter and figure 5b shows the frequency spectrum after applying the filter mask. The filter mask in this example filters out only the very lowest frequencies.



(a) High-pass filter mask (b) High-pass filtered frequency spectrum

Figure 5: Visualization of the high-pass filter mask and its effect on the frequency spectrum

The image after inverse transformation from the frequency domain is shown in figure 6. The high-pass filtered image contains sharp transitions such as edges, but the smooth patterns are filtered out. This is ideal for edge detection. Even though the high-pass filtered image contains a significantly greater range of frequencies, it looks less like the original image than the low-pass filtered image. This emphasizes how important the low frequencies are.

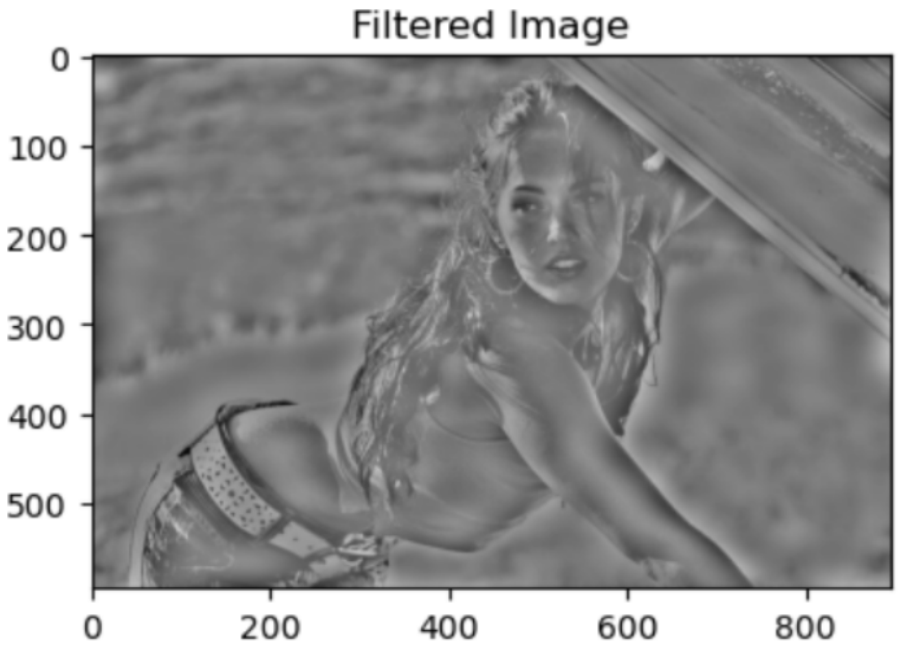


Figure 6: High-pass filtered image

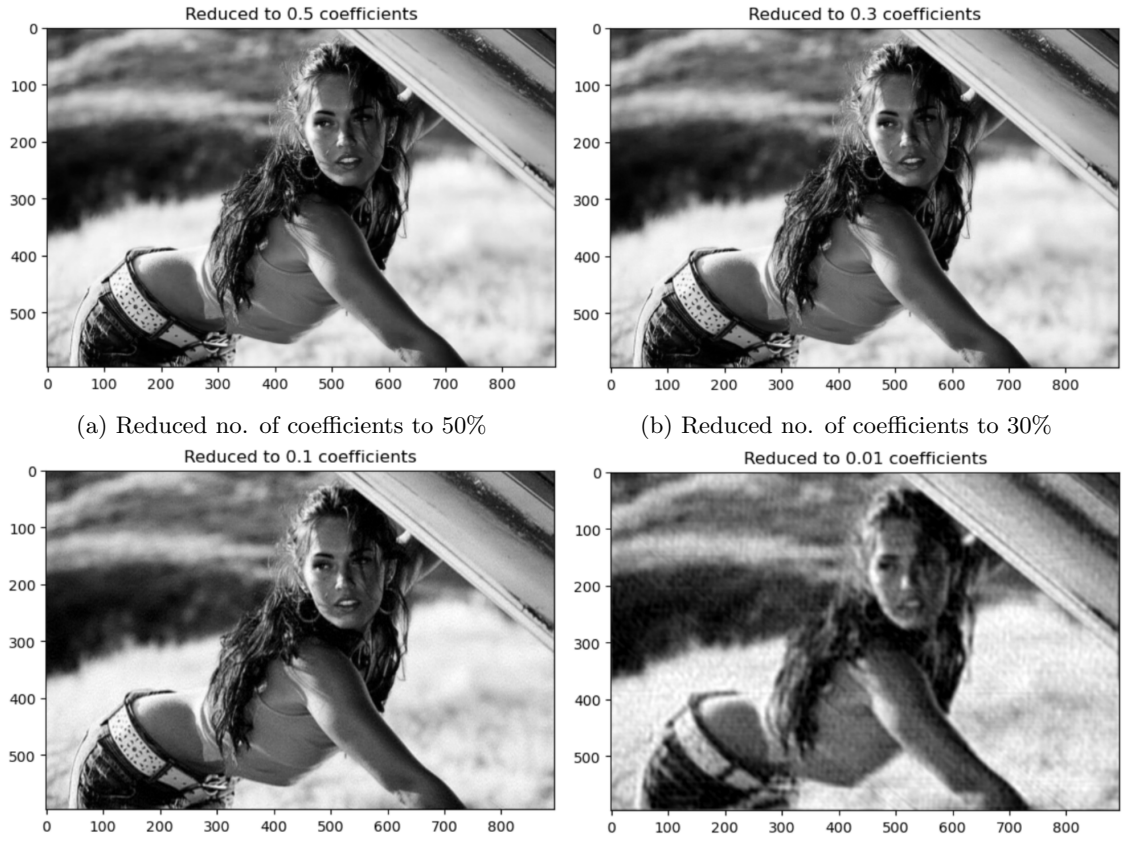
1.4 Image Compression with Fourier Transform

Fourier Transform can be used for image compression by removing the coefficients that contribute less to the representation of the image. The remaining coefficients are stored or transferred. Afterward, the inverse transform of the coefficients brings the image back to the full pixel resolution, but with a lower quality since the compression is lossy.

After transforming the image to Fourier coefficients, the frequencies are sorted based on their magnitude in descending order. Based on the compression ratio, i.e. the number of coefficients to keep, the smaller coefficients are zeroed out. When the image is to be reconstructed, the coefficients are rearranged in a matrix and inverse-transformed back to an image with pixel values.

To accomplish this compression based on percentages, the `argsort()` function of `NumPy` was used with `flatten()` to sort the coefficients. A threshold value was obtained by storing the value of the lowest coefficient to keep. The threshold value was used to create a mask of the coefficient matrix to zero out the values below the threshold.

Figure 7 shows the image after compression with different ratios. The compression ratio is 2, 3 1/3, 10, and 100, respectively. There is no noticeable reduction in the image quality when reducing down to 50% or 30%. A reduction of coefficients down to 10% can still be inverse-transformed back to an image that very closely resembles the original image. When reducing down to 1%, the image becomes grainy. The compression ratio and output quality are not the same for every image. The higher resolution the original image has, the more information can be removed, i.e. coefficients, and still produce a satisfying result because of redundant information[4]. For this exact image with a resolution of `596*894` pixels, a reduction down to 10% is satisfying for the image size in the report.



(a) Reduced no. of coefficients to 50%

(b) Reduced no. of coefficients to 30%

(c) Reduced no. of coefficients to 10%

(d) Reduced no. of coefficients to 1%

Figure 7: Image compression with Fourier Transform at different levels of coefficient reduction

[4]Brunton 2024b.

2 PCA

For this assignment, we selected the *Stranger Things Faces Dataset Grayscale* from Kaggle, featuring 620 grayscale images of 11 characters from the TV series *Stranger Things*[5].

After loading the images into our program (in PIL format), our PCA function took two parameters: `images_array` (grayscale images) and k (number of principal components). First, we converted each image to a NumPy array using `np.array()` and then flattened it into a 1D array using `flatten()`. The flattened images were combined into a 2D array, where each row represented an image and each column represented a pixel, forming a matrix with dimensions 620×7396. To normalize the pixel values, we divided each value by 255, scaling them to a range of $[0, 1]$[6].

Next, we centered the data by computing the mean for each pixel and subtracting it from each image using NumPy's mean subtraction (`images_array_2d` − `mean`), ensuring that PCA focused on variations in pixel intensities rather than their absolute values[7].

To capture relationships between pixels, we computed the covariance matrix using `np.cov()` on the centered data. The covariance matrix had dimensions 7396×7396, representing pixel relationships across the entire dataset.

From this covariance matrix, we calculated the eigenvalues and eigenvectors using `np.linalg.eig()`, which helped us identify the principal components. Sorting the eigenvalues in descending order allowed us to identify the most significant components. We then selected the top k eigenvectors, which represented the directions with the most variance in the dataset.

Finally, we projected the images onto the lower-dimensional subspace formed by these top k eigenvectors using matrix multiplication `np.dot(centered_images, eigenvectors_k)`, reducing each image to k components instead of the original 7396 pixels. The results were stored in the `projected_images` array, representing the compressed versions of the images.

2.1 Reconstruction of Images

When reconstructing the images, we chose to use the top 50 principal components ($k = 50$). This represents a significant reduction from the original 7,396 features. After applying PCA, we clearly observed a loss in detail and sharpness. The images appeared blurry, and it became more difficult to recognize the individuals in them. However, PCA effectively retained the maximum variance in the dataset. While finer details, such as skin texture and wrinkles, were lost, the overall face structure and prominent facial features like the eyes, nose, and mouth were well preserved. In this case, the variance retained was 89.86%.

Figure 8a shows a selection of the original images before PCA was applied, and Figure 8b illustrates the images after dimensionality reduction to $k = 50$, highlighting the loss in finer details while preserving key facial features.

2.2 Experimentation

When experimenting with different values of k, we wanted to see how PCA performed with k values both below and above 50. It is generally recommended to aim to retain enough components to explain around 90-95% of the variance.[8].

From the variance explained plot (see Figure 9) across different values of k, we observe a sharp rise in the variance explained up to about 100 components. Beyond this, the curve begins to flatten, meaning that each additional component contributes less to the overall variance.

[5] Kay 2024.
[6] Harsh 2024.
[7] Dey 2024.
[8] AI 2024.

(a) A selection of the original images. (b) Dimension reducted images, $k = 50$

Figure 8: Comparison of original images and those after dimension reduction

By around 400 components, the variance explained approaches 100%. To balance compression and quality, the optimal number of components is likely to be around 100-150.

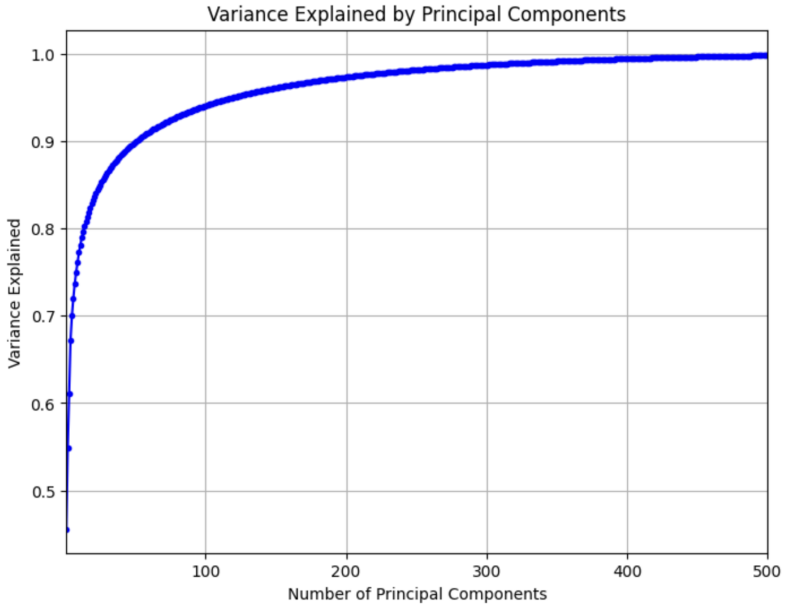


Figure 9: Variance explained by principal components

2.3 Visual Analysis

Knowing this, we looked at images with different numbers of principal components. For images reconstructed with only 10 components, a significant amount of information is lost. The trade-off between compression and quality is too high, resulting in blurred and unrecognizable images.

At 50 components, the quality improves but still lacks clarity. Moving to 100 components, we see a clear improvement in quality and information retained. Distinct facial features such as eyes, nose, and mouth start to become more visible, and the edges become more prominent. Although the images remain slightly blurry, the information loss is moderate, and the individuals become recognizable. At this point, we are explaining approximately 94% of the variance, aligning with our goal of retaining 90-95% variance for an ideal balance between quality and compression.

Increasing to 120 components, the improvement is minimal compared to 100 components, as the explained variance increases by only about 1%. However, at 400 components, the reconstructed images are almost indistinguishable from the originals, with fine details like textures, edges, and contours becoming much clearer.

Figure 10 illustrates the original images alongside the reconstructed images for different values of k, showing the progression in quality as the number of principal components increases. To conclude, aiming for 95% explained variance seems ideal when balancing quality and compression, which corresponds to retaining around 100-150 principal components.

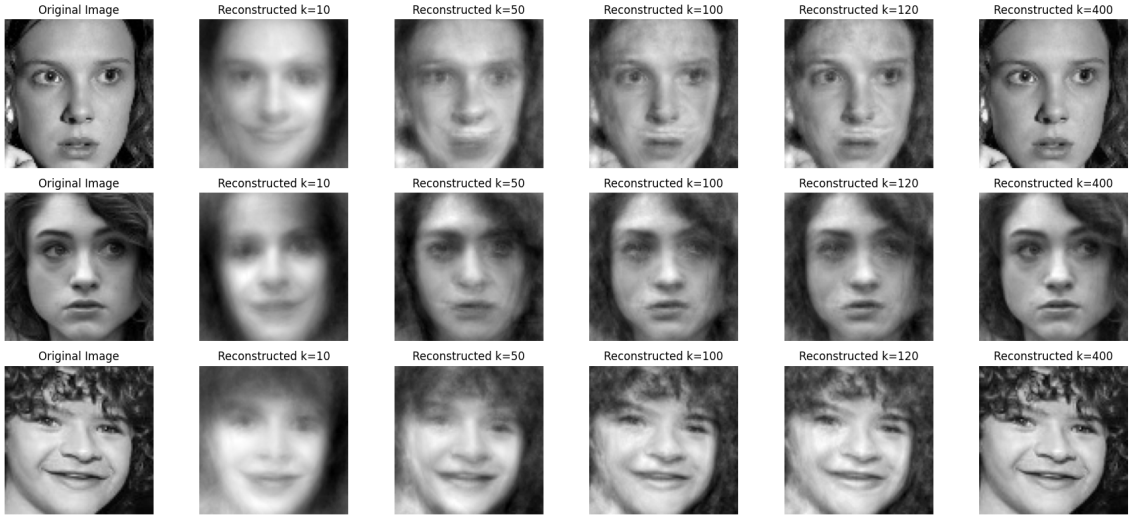


Figure 10: Original images alongside the reconstructed images for different values of k.

2.4 Error Analysis

As we varied the number of principal components, the MSE changed accordingly. Lower values of k (indicating higher compression) resulted in higher MSE, demonstrating greater information loss. On the other hand, higher values of k (indicating less compression) produced lower MSE, suggesting that more information was retained in the reconstructions.

The MSE values we obtained indicate a clear relationship between the number of principal components and the reconstruction error. With only 10 components, the MSE is relatively high (0.00920), indicating significant information loss. As the number of components increased to 100, the MSE dropped to 0.00410, and further reduced to 0.00241 with 150 components. Interestingly, the MSE remained constant at 0.00241 between 150 and 400 components, suggesting that the additional components beyond 150 contribute very little to the reconstruction. This observation strengthens our hypothesis that, beyond a certain point, additional principal components have minimal impact on the visual quality of the image. This implies that around 150 components are sufficient to capture nearly all the significant variance in the images.

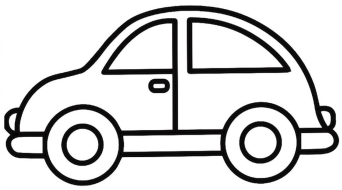
Finally, at 1000 components, the MSE reaches 0.0000, indicating close to perfect reconstruction. The results align with PCA behavior we have seen earlier, where after a certain point, additional components do not significantly enhance reconstruction quality.

3 HOG features

3.1 Figures and results

(a) Ford F150

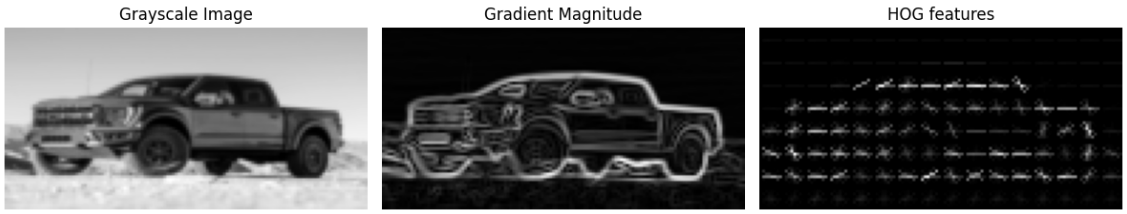
(b) Trees from above



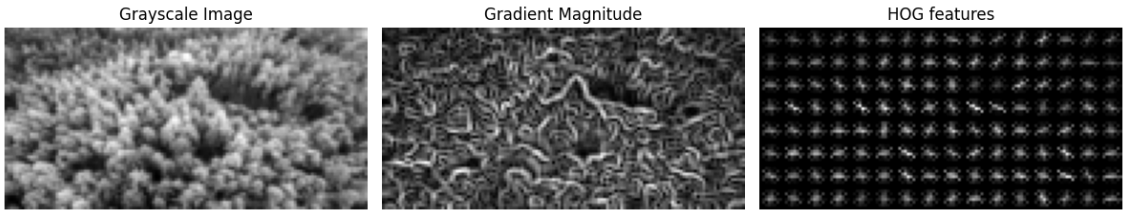
(c) Car drawing

Figure 11: The three images used in this task

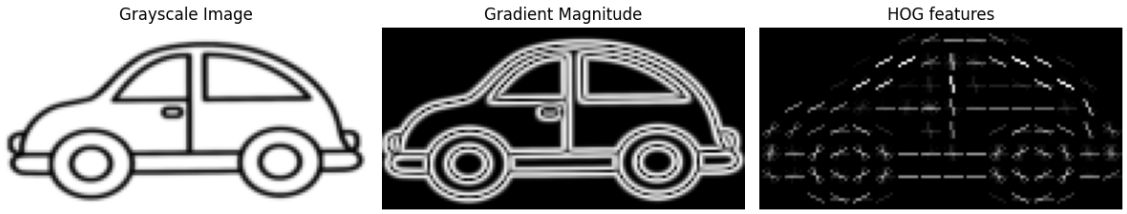
Visualization of HOG image and gradient image



(a) Ford F150



(b) Trees from above



(c) Car drawing

Figure 12: Grayscale image, gradient image and HOG feature image of the three images

All images were reduced to images with (64x128) pixels, then converted to grayscale images. The gradients of the images were produced by applying Sobel filters to the images. The grayscale conversion, Sobel filters and computation of HOG features were all computed with the help of the scikit-image library. The Sobel operator for computing gradients and edges is quite simple and time effective compared to other edge detection operations such as Canny. [9]
As the drawing of a car has more clear edges, and is only black on white, the gradient image is very vel defined. The same goes for the HOG features. On the other hand, the aerial photo of

[9] Haidar 2021

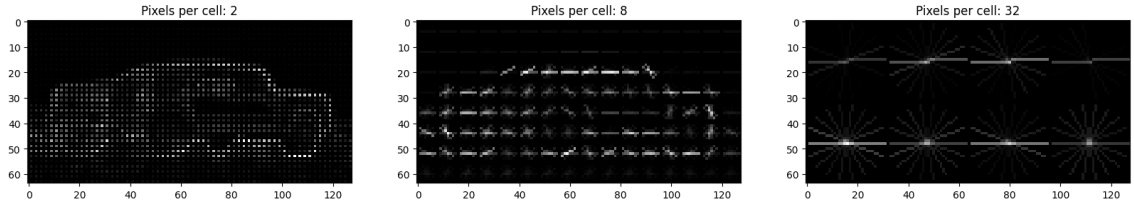


Figure 13: Three visualized HOG features, with different cell size

a forest is highly textured and the HOG captures local gradients but fails to provide a coherent shape representation. This illustrates a weakness of HOG, as it is well suited for object detection when the objective is to identify clear, structured shapes, but it may struggle with scenes where objects are highly textured, densely packed or lack clear boundaries.

3.2 Parameters discussion

Varying the parameters cell size, block size and number of bins yielded very different results. With smaller cells, the resulting HOG was significantly more accurate than with large cells. However, smaller cells leads to higher computational cost and more noise sensitive result, as there are more feature vectors, and slight variations in in pixel intensities are more prominent. Similarly, with smaller block size, the spatial resolution of the image is increased, and so is the computation. With larger block sizes, the resulting HOG features are more noise sensitive.
The amount of bins determines how the gradients within a cell is divided, meaning that fewer bins result in fewer directions within the cells. With 2 bins, edges are only visualized in two directions, and it is very difficult to capture differences in edges. Increasing the number of orientation bins, will lead to edges being detected in more directions.

4 Local Binary Patterns

4.1 Images used and method

(a) Landscape in Italy

(b) Photography of a man

(c) Wool

Figure 14: The three images used in this task

The histogram of the LBP (*Local Binary Patterns*) images is a good representation of the contrasts in the image. When using the basic 8-neighbor method, each pixel in the image is compared to its eight neighbor pixels, and for each pixel with a greater value than the center, one writes a "1", and a "0" otherwise. This is done in the same pattern for all pixels, and the 1's and 0's represent a binary string, with values from 0 to 255. The x-values in the histogram ranges from 0 to 255, and the y-values represent how many occurences there are of each x-value. If there are many values at 255, it can indicate for instance rough surfaces or very similar pixel values for neighboring pixels. As a consequence, with the three images used in the task, the amount of 255-values are far higher for 14a and 14b than for 14c.

4.2 Histograms and discussion

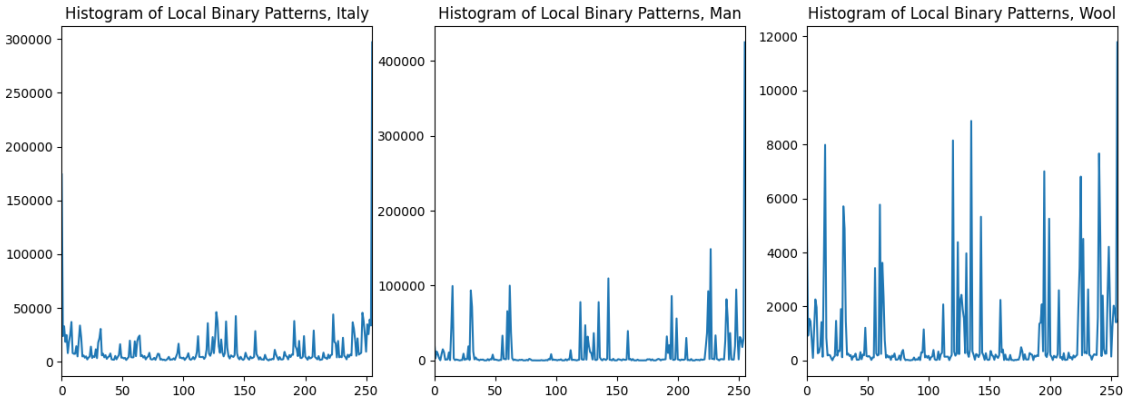


Figure 15: Histogram of Local Binary Patterns for the three images used

From the figure, one can clearly see that the histogram of LBP for the wool image is far more textured than the other two. It is naturally a lot of contrasts in the other two images in the other two images as well, but as they are not as defined, they do not appear using the basic 8-neighbor method.
If another method was used for computing the LBP, the histograms would have looked different. For instance, with the use of Rotation-Invariant LBP or Extended LBP, the histogram for the landscape image would likely show more spread peaks across different intensity values. [10]

[10] Pemalu 2024

5 Blob Detection

The most used parameters for blob detection are defined by **threshold**, **convexity**, **area**, **circularity** and **inertia**. We used the cv2 library to create a `SimpleBlobDetector()`. Through iterating over the pictures in the dataset we detected blobs for each picture.

5.1 Parameters discussion

Threshold operates with a min and max value that the blob-detection algorithm uses to determine which pixels to discard from a blob. A wider range allows more blobs with varying intensity, while a narrow range focuses on a specific intensity. We discovered that for the facial gray-scale images from `strangerthings` dataset the amount of blobs pr. picture increased from mostly 0-2 blobs per picture with min=70, max=350 to 2-4 blobs per picture with min=30-220. We also tried to increase / decrease the range which also resulted in fewer blobs per picture.

Area ensures that only blobs within the specified size range are detected. It helps to exclude noise (too small blobs) or irrelevant large areas (parts of the background or very large objects). For this parameter we chose the range 30-200. We observed that by increasing the lower-bound too much we lost blobs per picture. On the other hand we observed that increasing the upper-bound did not really change blobs per image.This makes sense as we resized the images to a 70px by 70px, and increasing upper-bound does not really change things. By decreasing the lower-bound too much we got way too many blobs and did not represent the picture.

Circularity helps to detect circular objects and discard blobs that are irregularly shaped or elongated. However with the min-ratio set to 0.3 there was minimal difference in number of blobs per picture when we turned the parameter off. On the other hand, when we turned the min-ratio up to 0.4 or more the number of blobs per picture started to decrease a lot, and we lost accuracy. This relates to the fact that the pictures did not consist of a lot of circular shaped objects.

Convexity helps detecting circular or mostly convex objects. Therefore, we ended up with disabling this parameter. When trying out min-ratio 0.2 we already saw a decrease in number of blobs per picture. When increasing the parameter we ended up with zero blobs per picture. As a result, we removed this parameter.

Inertia is a measure of how elongated a blob is. It describes the distribution of the blob's pixels around its axis. This is also a parameter used to detect circular objects. In that case, performance on the amount of blobs detected decreased with turning on the parameter. It also decreased more with a higher min-ratio.[11]

5.2 Conclusion

In conclusion the parameters that stood out as most important in a blob-detection algorithm for these types of face cards were **threshold** and **area**. We saw some improvement by including the **circularity** parameter, but these were minimal. The **convexity** and **inertia** parameteres were discarded as they were not of use in this particular dataset.

Average number of blobs per image	2.02
Average blob size	10.26
Max number of blobs in an image	7
Min number of blobs in an image	0

Table 1: Mapping of average statistics and results after blob detection

[11] *Blob Detection Using OpenCV (Python, C++)* 2024.

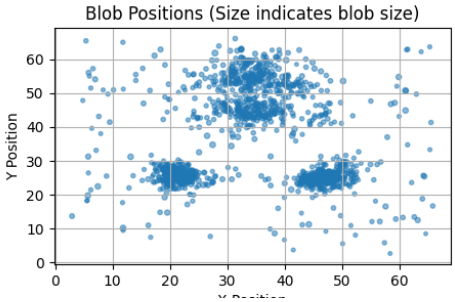


Figure 16: Scatter-plot of blob positions

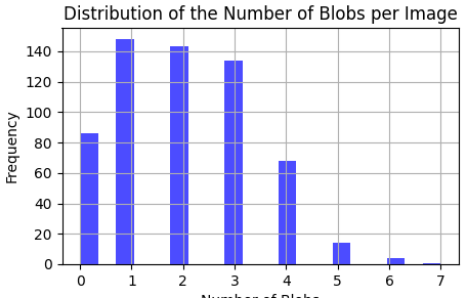


Figure 17: Histogram of blobs per picture

Figure 18: Images after blob detection

6 Contour Detection

Regarding the chosen dataset the clear winner algorithm was contour detection. Which, given the dataset, is not surprising. Images of faces do not often behave as round objects that are easily separated / detected. However all faces have contrasts often to face-features or the background which served as a good dataset for contour detection.

6.1 Advantages and Limitations

There are both advantages and limitations for each algorithm. For blob detection an advantage is that blob detection can detect regions that are not necessarily of a specific shape. It is shape-independent. In addition it works well for spotting round objects. On the other hand, the algorithm is somewhat limited to convex shapes. Concave objects are mush harder for the algorithm to detect. Also, blob detection might not be effective when objects have sharp or irregular edges.

Regarding the contour detection algorithm it is good at finding the precise outlines of objects, making it great for objects with well-defined edges. Contours also give detailed information about object shape, allowing for precise shape analysis, area, and perimeter calculations. On the contrary, contour detection can be very sensitive to noise or small fluctuations in the image. Finally, to get good contour detection, preprocessing steps like thresholding, blurring, or edge detection are often needed, which can complicate the process.

6.2 Parameters discussion

Regarding the threshold values the model tends to overfit the model for values lower than 100. The model tends to underfit for values over 100 and especially over. The threshold turns pixels underneath that colour to 0 and others over to 255[12]. Using the threshold together with $THRESH\_BINARY$ worked by far the best as $THRESH\_TOZERO$ was not able to distinguish contours at all. Neither was $THRESH\_TRUNC$.

For the `cv2.findContours` method we selected parameters $RETR\_EXTERNAL$ as $RETR\_TREE$ and $RETR\_CCOMP$ both overfit the contours.As mentioned above, contour detection often requires some kind of preprocessing. The quality of the contours improved significantly by applying `cv2.GaussianBlur` method with a 5x5-kernel.

6.3 Visual representation and conlusion

Contour detection is advantageous in detecting and segmenting objects with well-defined boundaries, like characters in an image, mechanical parts in an assembly, or shapes in a logo. On the other hand, blob detection works well with detecting round spots as in for example astronomy images (stars, planets) or in biological microscopy images.

Average number of contours per image	3.42
Average contour area	373.77
Average contour perimeter	67.74
Max number of contours in an image	18
Min number of contours in an image	0

Table 2: Mapping of average statistics and results after contour detection

[12] *Image Thresholding in OpenCV* 2024.

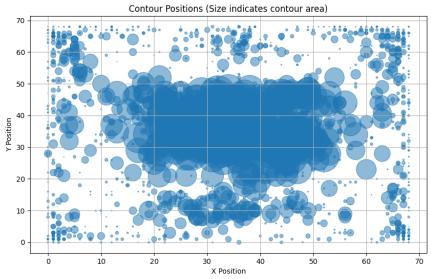


Figure 19: Scatter-plot of contour positions and size

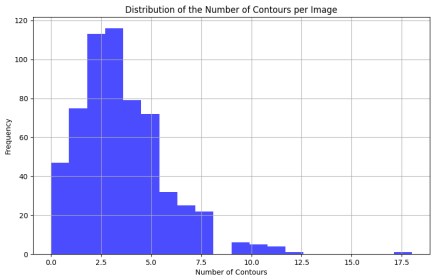


Figure 20: Histogram of contours per picture

Figure 21: Images after contour detection

References

AI, Team Applied (2024). *Principal Component Analysis (PCA) Explained.* URL: https://www.appliedaicourse.com/blog/principal-component-analysis-pca/. (visited on 23rd Oct. 2024).

Blob Detection Using OpenCV (Python, C++) (2024). URL: https://learnopencv.com/blob-detection-using-opencv-python-c/ (visited on 23rd Oct. 2024).

Brunton, Steve (2024a). *Image Compression and the FFT.* URL: https://www.youtube.com/watch?v=gGEBUdM0PVc (visited on 25th Oct. 2024).

— (2024b). *Image Compression and the FFT (Examples in Python).* URL: https://www.youtube.com/watch?v=uB3v6n8t2dQ (visited on 25th Oct. 2024).

Dey, Roshmita (2024). *Understanding Principal Component Analysis (PCA).* URL: https://medium.com/@roshmitadey/understanding-principal-component-analysis-pca-d4bb40e12d33#:~:text=Before%20performing%20PCA%2C%20it%27s%20essential,equal%20importance%20in%20the%20analysis (visited on 23rd Oct. 2024).

Haidar, Lina (2021). *Sobel vs. Canny Edge Detection Techniques: Step by Step Implementation.* URL: https://medium.com/@haidarlina4/sobel-vs-canny-edge-detection-techniques-step-by-step-implementation-11ae6103a56a (visited on 24th Oct. 2024).

Harsh, Patel (2024). *Normalization in Image Preprocessing: Scaling Pixel Values by 1/255.* URL: https://medium.com/@patelharsh7458/normalization-in-image-preprocessing-scaling-pixel-values-by-1-255-111b2fa496d4 (visited on 23rd Oct. 2024).

Image Thresholding in OpenCV (2024). URL: https://learnopencv.com/opencv-threshold-python-cpp/ (visited on 23rd Oct. 2024).

Jaadi, Zakaria (2024). *Principal Component Analysis (PCA): A Step-by-Step Explanation.* URL: https://builtin.com/data-science/step-step-explanation-principal-component-analysis (visited on 23rd Oct. 2024).

Kay, Alireza (2024). *Stranger Things Faces Dataset Grayscale.* URL: https://www.kaggle.com/datasets/alirezakay/stranger-things-faces-dataset-grayscale (visited on 23rd Oct. 2024).

Low-pass filter (2024). URL: https://en.wikipedia.org/wiki/Low-pass_filter (visited on 25th Oct. 2024).

Pemalu, Ikhwan (2024). *Local Binary Patterns (LBP): An In-Depth Exploration of Texture Analysis.* URL: https://medium.com/@ikhwanpemalu22/local-binary-patterns-lbp-an-in-depth-exploration-of-texture-analysis-13aee7fb2c7 (visited on 25th Oct. 2024).

Ringing artifacts (2024). URL: https://en.wikipedia.org/wiki/Ringing_artifacts (visited on 25th Oct. 2024).

Wronski, Bart (2021). *Comparing images in frequency domain. "Spectral loss" – does it make sense?* URL: https://bartwronski.com/2021/07/06/comparing-images-in-frequency-domain-spectral-loss-does-it-make-sense (visited on 25th Oct. 2024).