Copenhagen Business School

CDSCO1001U.LA_E21

Foundations of Data Science: Programming and Linear Algebra

Final Report

Creating an IPU (Image Processing Unit) Library in Python

Based on mathematical and theoretical foundations introduced in class

Date of submission: 20 December 2021

Names: Jan Gaydoul, Wiktoria Maria Lazarczyk, Stefano Pontello, Emanuela Zucchetto

Student ID numbers: 149045, 149985, 149883, 149888

Number of pages: 13

Characters: 24,630

Table of Contents

Introduction

Processing images in Python is quite easy, and of course there are already numerous tools and libraries available that make these tasks child's play. The most prominent example is probably the OpenCV library, which offers a powerful tool for all kinds of image processing tasks. With just a few lines of code, images can for example be blurred to any degree. This is made possible by various mathematical operations running in the background, in which many concepts of linear algebra that were introduced in class can be found.

Our goal for this final report is to build such a library for processing images with Python from scratch, applying the underlying mathematical topics discussed in class, such as matrix partitioning or the dot product. Of course, this library cannot be as extensive as OpenCV, but it should allow us to automate some of the processes in image editing. Hereby, our goal is to do so with only a very limited use of existing libraries such as OpenCv or numpy. While we might use parts of these libraries for trivial tasks (e.g. loading the images with OpenCV), we will restrain from using them as much as possible when it comes to the actual image processing, as we aim to apply the mathematical foundations learned in class instead of for example just calling some numpy functions that will apply the underlying mathematical operations automatically.

In the first part of this paper, we will start by introducing the theoretical and mathematical foundations of blurring pictures which we will deploy in the process of building up our library. In the second part, we explain how we put together our library and describe the functionalities it provides.

Theoretical Foundations

RGBA and Grayscale

To get an understanding of how computer programs process and represent pictures, we want to give a quick introduction to the two underlying color concepts: RGBA and the grayscale.

An RGBA value is a group of numbers that specify the amount of red, green, blue and alpha (or transparency) in a color. Each of these component values is an integer from 0 to 255. These RGBA values are assigned to individual pixels; a pixel is the smallest dot of a single color the computer screen can show. Hence, a pixel's RGBA setting tells precisely what shade of color this pixel should display. RGBA values are represented by a tuple of four integer values; for example, red is represented by (255, 0, 0, 255) or white, the combination of all colors, is (255, 255, 255, 255) (Sweigart 2020, pp. 387-388).

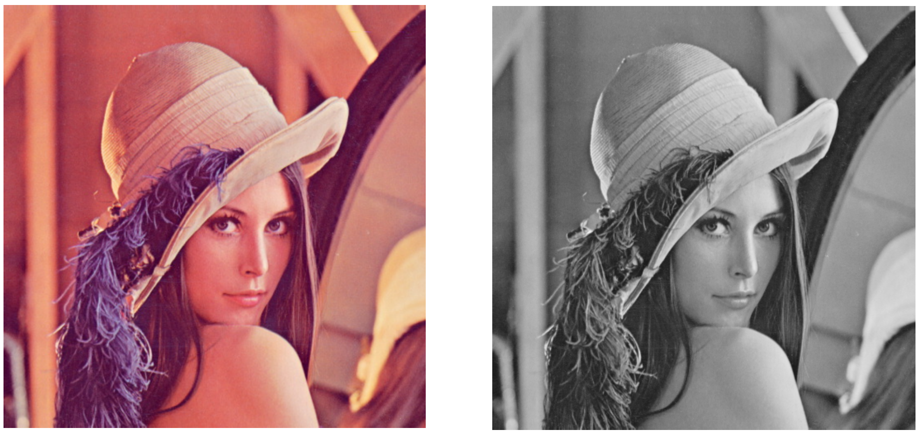


Figure 1: the same image with RGB color scheme (left) and greyscale color scheme (right)

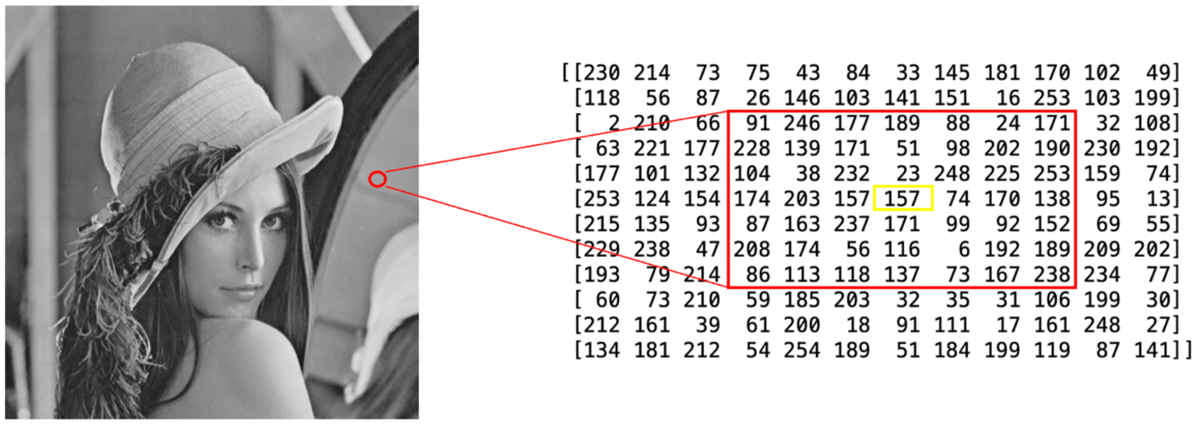
The concept of the grayscale is very similar, but here, the color of a pixel is not determined by four values but only by a single one on scale from 0 (black) to 255 (white), inclusive. The values between 0 and 255 hence represent shades of gray.
Pictures are either in a RGBA or grayscale format. Of course, the RGBA values can represent any shade of gray that the grayscale could represent as well, however, there are different areas of applications that justify using either RGBA or the grayscale specifically. In image processing, it often comes in handy to convert pictures to grayscale before processing them. One example would be the field of Image Recognition, where decreasing the amount of training data, or data attributes (in this case avoiding colors) could make the learning process much faster (Kanan & Cottrell 2012, p. 1). This is why we included a conversion to the grayscale in our library later on as well.

Blurring pictures using Convolution and the Gaussian Blur

Now we know that a picture in a computer program consists of (up to) millions of pixels, all assigned with an RGBA value or a value on the grayscale, depending on the type of picture. This also means that between these pixels (or, since we are talking about millions of them, between groups of pixels), there might be rather sudden changes of colors, which effectively are the color transitions we can see. For example, in the colored picture in Figure 1 above, we could see such a transition between the purple part of the woman's hat and the beige part.
Blurring a picture therefor means nothing else but making the color transition from one (group of) pixel(s) to another one smoother. This happens by averaging out immediate changes in

pixel color intensity; we could hence say that by blurring a picture we remove outlier pixels with a strong intensity (e.g. Partridge and Hussain 1992, p. 115). The next part will introduce the method that is most commonly used in image processing to achieve this: Convolution.

In image processing, convolution basically means changing the value of a pixel according to the values of the pixels surrounding it. This can be well illustrated with an example:



```
[[230 214  73  75  43  84  33 145 181 170 102  49]
 [118  56  87  26 146 103 141 151  16 253 103 199]
 [  2 210  66  91 246 177 189  88  24 171  32 108]
 [ 63 221 177 228 139 171  51  98 202 190 230 192]
 [177 101 132 104  38 232  23 248 225 253 159  74]
 [253 124 154 174 203 157 157  74 170 138  95  13]
 [215 135  93  87 163 237 171  99  92 152  69  55]
 [229 238  47 208 174  56 116   6 192 189 209 202]
 [193  79 214  86 113 118 137  73 167 238 234  77]
 [ 60  73 210  59 185 203  32  35  31 106 199  30]
 [212 161  39  61 200  18  91 111  17 161 248  27]
 [134 181 212  54 254 189  51 184 199 119  87 141]]
```

Figure 2: A 7x7 kernel (red) iterates through a grayscale image and is used to blur the center pixel (yellow) (own illustration)

To blur an image, we take each pixel of the image and look at a rectangular around that pixel. We then create a different matrix of the same size of that rectangular which we will from now on call the “kernel”. In our example we will look at a 7x7 kernel and focus on how to use this kernel to blur an image that uses the grayscale, since our library later on will only blur grayscale pictures as well (RGBA images are converted before the blurring process). This 7x7 kernel consists of 49 pixels with the pixel to be blurred in the middle (the center). Remember: Each of these 49 pixels contains one value, namely the greyscale value on a scale from 0 to 255 of each pixel. Hence, a 7x7 matrix can be constructed that is filled with the respective greyscale values for each single pixel.

The new value of the center of the kernel is now the average of all values of the pixels in the kernel under consideration. We now re-composed the grayscale value for this pixel according to the average of the pixels surrounding it. After completing these calculations, we move on to the next pixel. The 7x7 kernel moves accordingly; for example, if we want to look at the next pixel to the right, the kernel moves one unit to the right as well.

Note: the size of the kernel under consideration (7x7, 15x15, 255x255, ...) ultimately determines to which degree the image is blurred. The larger the kernel is selected, the stronger the blur, since more values are being factored into the average.

The technique we'll use later on to blur images is called the "Gaussian Blur". The difference between the Gaussian blur and the concept introduced above is that in a Gaussian blur, more weight is given to the values of pixels closer to the center than to pixels far away from the center when calculating the new value for the center pixel. This can be achieved by convolving the image with a Gaussian function; Gaussian functions are bell-curved and normally distributed, which facilitates the weighting of the values of the pixels. This concept gets very clear when looking at the three-dimensional representation of a Gaussian function:

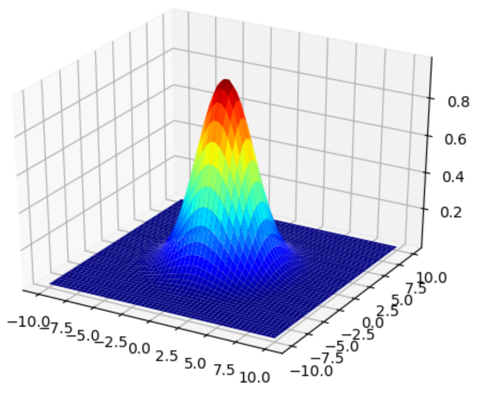


Figure 3: Three-dimensional representation of a Gaussian function. More weight is given to values close to the center

Online image available from: https://stackoverflow.com/questions/52653734/how-to-do-a-3d-plot-of-gaussian-using-numpy, accessed 12/17/2021

After having introduced the crucial theoretical foundations of image processing, we now want to introduce the functionalities of our IPU library.

Library

We included four main functionalities into our library that are building up on each other step by step with the ultimate goal to give the user the possibility to upload an image and then blur either the whole picture or just specific parts of it. The following functionalities cover various steps across this process, starting with retrieving details about the image and the conversion to a greyscale color scheme, to the partitioning into smaller parts and ultimately the blurring.

List of dependencies and their use

As already mentioned, we have largely refrained from using existing libraries. The few exceptions are listed below:

OpenCV:

- **cv2.imread:** function to load the image file and convert it to a numpy array
- **cv2.cvtColor:** function to adjust the channel from BGR to RGB format
- **cv2.imwrite:** used to save the image locally

Numpy:
- **np.reshape** = function used to reshape an array to a desired shape
- **np.sum** = function used to sum the value resulting from the matrix multiplication
- **np.hstack, np.vstack** = used to stack the partitions of the matrix within the blurring process

Matplotlib
- **plt.imshow** = used to convert the Numpy array of pixel values into an image
- **plt.subplots** = function used to display the partitions

List of Functionalities

Our library contains the following functionalities:

Functionality	**Main functions**	**Sub-functions**

Image Information	def __init__
	def getDimension
	def get_img_values_matrix
	def get_width
	def get_height
	def get_rgb
	def get_channels_number
	def showImage
	def isSquare
	def getImageDescription	

Grayscale Conversion	def isGrayscale
	def grayscaleConversionChannel
	def grayscaleConversion	

Image Partitioning and Selection	def img_partitioning
	def show_partitions
	def create_selection
		def add_selection_element

	def clear_selection
	def recompose_img
	def saveImage	

Image Blurring	def blurring
	def blurring_partitions
		def plot_image
		def plot_two_images
		def padding
		def matlab_style_gauss2D
		def convolution2d

Image Information Retrieval

To start it off, we thought it might be useful to implement some functions that give us basic information about the image we want to process. The following functions tell us (1) about size and shape of the image and (2) whether it uses Greyscale or RGBA as its color scheme. The first task can easily be done by checking whether length and width of the image contain the same number of pixels, while the latter can be achieved by counting the color channels of the image; generally, grayscale images only have one channel, while RGB images have three (or four for RGBA pictures) channel as we've mentioned above.

def __init__

Function to initialize an instance of the object Image. It takes an image and a text description as input. First the image is checked as this library only support jpg, png and tiff formats. It then sets several attributes such as the image converted to RGB format, the text description, the width and the height and the number of channels. Moreover, it initializes an empty array that will store the selection for the partitions in the blurring process. Finally, it checks if the image selected contains an alpha channel (only for PNG and TIFF files) and raise an error.

def getDimension

Function to display the height and the width of the image.

def get_img_values_matrix

Function to get image matrix of values

def get_width

Function to access the width attribute of the image

def get_height

Function to access the height attribute of the image

def get_rgb

Function to access the matrix (as np.array) of the image in RGB format

def get_channels_number

Function to access the get the channels number

def showImage

Function to show the image by calling the plt.imshow function

def isSquare

Function to check if the image has a square of rectangular form.

def getImageDescription

Function to access the text description of the image.

Greyscale Conversion

We now know if our image is using greyscale or RGB as its color scheme. If the latter one is the case, we want to convert it to greyscale since this facilitates many further operations.
The following list is the collection of different function to manipulate the image and make it suitable for further processes, such as the blurring.

isGrayscale(self):

This function checks if the image passed as input is grayscale or not. By using a set of conditions, we check if the respective np.array containing the image values contains one channel, if the channels are the same or if the channels are different. In the first two cases the image is grayscale, but since the future blurring operation only works with one-dimensional matrices, the 3-channels grayscale image will need to be transformed to allow only one value per pixel (ranging from 0 to 255). In the case the three channels contain three different values, it means it is RGB and will need to be converted to grayscale for future manipulations. (*ndr. for the purpose of this project we will disregard RGBA images*).

grayscaleConversionChannel(self):

This function is used to convert three-channel grayscale images into one-channel ones. It iterates through the pixel values and takes the first value (since for grayscale images, the 3 channels are the same). It then adds the value to an array which is later reshaped to match the initial image shapes.

grayscaleConversion(self):

This function aims to convert a RGB image into a one-dimensional (one-channel) matrix in order to make it compatible for the blurring process. It first checks if the image is already one-channel grayscale, otherwise it will carry on with the process. It iterates through the image matrix, takes the vector (pixel) containing the three channel values (corresponding to the red, green and blue) and it multiplies them by a corresponding vector (0.299, 0.587, 0.114) (*DynamSoft, 2019*) in order to compute a conversion using the so-called luminosity method, which weights every color based on his corresponding wavelength.

This process can be described as the dot (inner) product between the vector containing the RGB values and a vector containing 3 constants. Using a vector of 3 equal values (average method) is also an option but the quality of the conversion is not the best because the eye react differently to the different colors).

Image partitioning

The goal of this part is to divide the image into a specified number of smaller parts of which the user can then choose the ones he wants to blur. This is based on the idea of matrix partitioning: basically, we can see the image as one big matrix which we then partition into sub-matrices. To ensure the user can then choose the part of the picture he wants to blur, we create variables for each sub-matrix.

Img_partitioning(self)

The function img_partitioning() is used to obtain 16 partitions from the image on which the function is called. The function does not require any parameter besides *self* as it is a function of the class Image.

Firstly, the value of rows and columns is retrieved by using two getter functions. This allows to segment both height and width into 4 different segments, and their starting and ending point are stored in the lists *row_split* and *col_split*. For each dimension, the first three are equal, while the last one could take up to 3 extra elements (the reminder of the division by 4). This decision was made in order to preserve the whole image and not to have any loss of information.

In addition, the grayscale NumPy array version of the image is obtained by calling the function grayscaleConversion() and assigning it to *np_img* and a Python list *parts* to store the partitions is initialized.

The partitions are then obtained by looping *np_*img through two nested for loops, one for rows and one for columns. For this reason, we used the variables ri, rf, ci and cf to set the initial and final index for rows and columns, respectively. By assigning to these variables the values in *row_split* and *col_split* and indexing the array with *np_img[ri:rf,ci:cf]* it is possible to obtain the single partitions. Each partition is then appended to the list *parts.* Once out of the for loops, the list is assigned to the NumPy array np_parts.

Finally, the partitions are shown by calling the function *show_partition().*

Show_partition(self, np_parts)

The partitions are displayed with the use of the function *subplot()* from matplotlib. Through the function *plt.subplot()* we generate a grid of 4 by 4 plots, one for each partition. Then, to perform this operation a nested for loop is used. For each subplot *imshow()* is called so that it would show the given partition and the title is set to be the index corresponding to the given partition. In addition, in all cases axis numeration is set to “off”. Finally through *subplot_adjust()* we set the distance between each plot so that the visualization is pleasant.

Create_selection(self)

This function is created to allow the user to decide on which partitions to perform the blurring. The selected partitions are stored in a list called *selection,* which is an attribute of the Image object.

At first, the user is asked to insert the number of the partition they want to blur. It is important to verify that the input can be converted to an integer and for this purpose we used try and except. In the first case, the string is converted to integer and the if statement is executed, otherwise an error message for the *ValueErro*r is printed.

In the if statement, if the number is not between 0 and 15 (the number corresponding to the partition indexes) the user is asked to enter only valid numbers. if the input is correct the number is appended to the *selection* list, unless already present.

After the execution of the if statement or the except block, the *add_selection_element()* is called to verify if the user wants to add other partitions to the list.

When the process is complete, for each element in the list, the function *blurring_parition()* is called to blur the partition corresponding to that index. Finally, the partitions are recomposed and shown by calling the *recompose_img()* function.

The last step consists in calling the *clear_selection()* to remove all the elements present, so that other blurring activities can be performed to the same image.

Add_selection_element()

This function is used to evaluate whether or not the user wants to blur other partitions. it is positioned within the *create_seleciton()* function so that it is not accessible from the outside. This is verified through an input function and an if statement to match the answer with the following command. To facilitate the process the answer is made lowercase with the function *lower()*.

If the string typed corresponds to “yes”, then the *create_selection()* is invoked again. On the other hand, if the string is neither “yes” or “no” the user is asked to insert a valid answer and the same function is called again.

Clear.selection(self)

This function simply clears the *selection* list by calling *self.selection.clear().*

Recompose_image(self)

In order to recompose the partitions as to show the initial image, this function repetitively uses *hstack()* and *vstack()* functions from NumPy to align together the NumPy arrays representing the partitions in each row and column. Firstly, with the help of *hstack()* the matrices are stacked horizontally to form the 4 rows. Then, these rows are stacked vertically to form the NumPy representation of the image. Finally, through *plt.imshow()* the final array is displayed.

Given that the function is always accessible, we have added an exception handling code in the case of an *AttributeError*. This would rise if the user called this function before *img_partitioning()*, because no np_parts array would be defined. Thus, with the AttributeError a message is printed, stating that first is necessary to partition the image.

Blurring_partitions(self, partition)

Given that python does not support function overloading, *blurring_partitions()* performs the same functionalities as blurring(), but on a NumPy object instead of an Image object.

The call to the function is also in this case done by the Image object we are operating on, but here an additional parameter is required. This parameter corresponds to the index of the partition the user desires to blur.

Given that the function is always accessible, a portion of the code is designed to verify that the inserted parameter corresponds to a partition of the image. Firstly, if the parameter is not of type integer, a *ValueError* rises, specifying that the parameter has to be an integer. Secondly, the value has to be between 0 and 15, which are the indexes associated with the partitions. If this is not the case, an *IndexError* is raised. Finally, it is possible that the function gets called before partitioning the image, so the code tries to assign np_part[partition] to a variable. if this raises an AttributeError, then the *img_partitioning()* function is called and only then the np_part value is assigned.

Normally, this function is called by the *create_selection()* function using each element of the list *selection* as a parameter. However, given that the function is still accessible independently an if statement checks that the value of the parameter is between 0 and 15. Otherwise, an index error is raised.

In addition, at the end of the function the NumPy array corresponding to the blurred partition is re-assigned to the partition at the corresponding index.

Besides these aspects, the function is the same as the one implemented to blur the entire images.

Image Blurring

Now that we have created the sub-matrices we are ready to blur one or several parts of the image.

def blurring_method_selection()

Function to retrieve the user input on the blurring method to be used.

def blur_method_sel()

This function retrieves the user choice on the method of blurring to be applied and calls the respective function which will create the correct kernel, based on the average or the Gaussian method. The predefined kernel size is 49 x 49 pixel, as it was the one that was offering the best results in terms of quality and blur intensity.

def squareKernelAverage()

Function to create a kernel (matrix) containing only number 1s in order to perform the blurring based on the average method. It was done by appending with a for loop a list of number 1s in a list and then reshaped it to match the desired kernel size.

def padding()

Function that adds additional layers of usually 0s so that the size of input image will be preserved when performing the image processing as when performing convolution some pixels are being lost. We could disregard this step and accept the reduced dimensions, however our goal is to perform image processing on parts of the image and then combining it back together meaning the dimension of output image should be equivalent to dimensions of input image. However, our padding layers are constituted of 100s so that the black border (equivalent to 0) is no visible, just the dark grey which is similar to from gray obtained through gray scaling.

def plot_image()

Function to return the image using the plt function figure and imshow

def plot_two_images()

Function to return two images, which will be later used to show the original version of the image and the blurred one.

def matlab_style_gauss2D()

This function returns a kernel of values based on a specified size and standard deviation. We tried to implement this function ourselves but with little success. For the purpose of showing an additional method to the average one, we decided to implement this code sourced by the following post: https://stackoverflow.com/questions/17190649/how-to-obtain-a-gaussian-filter-in-python .

All credits goes to the author: https://stackoverflow.com/users/1461210/ali-m.

This function mirrors the one built-in in Matlab.

def convolution2d()

Convolution function takes two arguments, the image matrix and kernel. Kernel is a matrix acting as filter: depending on its values different image processing operations can be done. Convolution function returns image, which each pixel is determined by multiplying each kernel value by the matching input image pixel values and summing them (Traore et al. 2018).

Conclusion and further improvements

The main issues we faced, which could represent a challenge for further improvements, is the time complexity of some of the functions. As an example, the grayscale Conversion function has an exponential time complexity of $O(n^2)$ due to the two nested for-loops. This could potentially represent a problem when dealing with High Quality images (4K or 8K) as the conversion process slows down significantly.
Moreover, a further improvement of the project could be the ability to perform manipulation on RGB(A) images since the functionalities in our library only blur grayscale images.

We conclude by saying that for us, this final project was a great learning experience which eventually led us to experiment and deal with image data through a mathematical "lens". In our opinion, this small project represents a good foundation for future courses, such as Machine Learning and Natural Language Processing, where algebra operations involving vectors and matrices are common practice.

Finally, we want to mention that this project is also available on GitHub at the following link: https://github.com/stefanopontello/foundations_final_project

References

DynamSoft (May 24, 2019) – “Image Processing 101 Chapter 1.3: Color Space Conversion”, https://www.dynamsoft.com/blog/insights/image-processing/image-processing-101-color-space-conversion/

Kanan, C. & Cottrell, G.W., 2012. Color-to-grayscale: Does the method matter in image recognition? *PLoS ONE*, 7(1), https://doi.org/10.1371/journal.pone.0029740

Partridge, D. & Hussain, K.M., 1992. *Artificial Intelligence and Business Management*, Norwood: Ablex.

Sweigart, A., 2020. *Automate the boring stuff with python: Practical programming for total beginners*, San Francisco, Calif: No Starch Press.

Traore, B.B., Kamsu-Foguem, B. & Tangara, F., 2018. Deep Convolution Neural Network for Image Recognition. *Ecological Informatics*, 48, pp.257–268, https://doi.org/10.1016/j.ecoinf.2018.10.002.