Heidelberg University

Visual Learning Lab

Exercises for 3D Computer Vision

Titus Leistner

Raphael Baumgartner

Prof. Dr. Carsten Rother

May 19, 2020

Contents

Introduction

Welcome to the online exercise for 3D Computer Vision. We prepared five tasks to introduce you to the practical aspects of Computer Vision and prepare you for the mini projects at the end of the semester. The exercises will cover the following aspects:

- The Numpy and Matplotlib modules which are essential for any scientific computation project with Python
- Image processing on the example of fast filtering
- PyTorch, tensors and Autograd
- Different neural network architectures for image classification on MNIST
- Differentiable RANSAC as an example for a recent research topic in our group

Please note: We are aware that many of you already have experience with some of those topics. However, we also received many requests from students with very little or no programming experience. Therefore, we made the decision to also include some basic tasks.

Admin

We will publish the exercises on GitHub [1]. Please note, that we might update the repository as well as this document from time to time if we encounter some bugs in the code or mistakes in the tasks. The repository contains installation information and one iPython-notebook (.ipynb-file) per exercise. In the notebooks, you will find some already implemented code and more detailed information about the specific tasks. We assigned a score to each subtask. Please note that you have to score at least 50% of all points for each exercise in order to work on a mini project and get a mark for this lecture.

Each exercise has to be submitted via Moodle before the assigned deadline. Please do not upload more than the .ipynb-file itself. If you have any comments for the corrector regarding your solution, add a new markdown-cell inside the notebook. Please use the forums on Moodle for all your questions. We will create a new forum for each exercise. Questions regarding the installation and setup of the repository can be asked in the forum for the first exercise. You should use the Admin forum for all other admin questions.

Last but not least we want to encourage you to also present unconventional and creative solutions, play around with parameters and do some of the optional tasks.

But most importantly: Have fun, stay healthy and happy despite this difficult time!

Titus Leistner, Raphael Baumgartner and Carsten Rother

1. Scientific Python

This exercise gives you a short introduction into NumPy [2], which is widely used for numerical computation with Python, and Matplotlib [3] for plotting graphs and presenting your results.

Setup

The first task is to setup our repository [1]. You find detailed installation information in the README.md. We tested our setup on Linux (Ubuntu and Arch), MacOS and Windows 10. If you encounter (or solve) any problem regarding the setup, please open a new thread in the forum for the first exercise.

1.1 NumPy

NumPy forms the basis of the Python scientific stack. Its main component is the `numpy.ndarray` class for n-dimensional arrays. Because it is mostly implemented in Fortran, computations using NumPy arrays are way faster than e.g. operations on Python lists. Let's take a look at a minimal example:

```
# import the numpy module
# note that np is the common naming convention
import numpy as np

# create a new 1D array with 4 elements from a Python list
a = np.array([0, 0.32, 10, 12], dtype=np.float)

# perform a basic operation with a floating-point scalar
b = a * 42.0

# use slicing to get the last two elements of this array
c = b[-2:]
```

Your task is to get used to scalar, vector and matrix operations with NumPy.

1.2 Scikit-Image

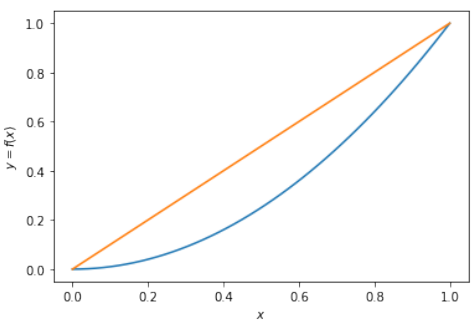
For all exercises we will use scikit-image for basic image loading, storing and manipulation. This module is part of the SciPy ecosystem for scientific computing and rather light-weight and easy to install compared to e.g. OpenCV.

1.3 Matplotlib

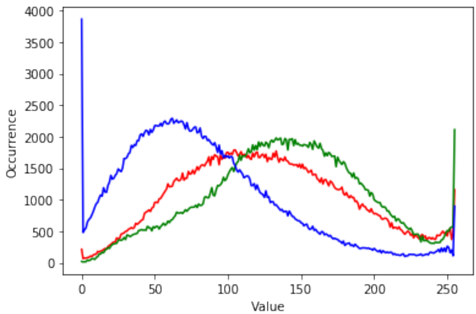
Matplotlib is an extremely powerful library for scientific visualization. In this exercise you will use it to output your processed images as well as some simple function plots and histograms. Let's again look at some minimal example:

```python
# imports
# again, note the naming conventions for np and plt
import numpy as np
import matplotlib.pyplot as plt

# create an array with numbers between 0 and 1
xs = np.linspace(0, 1)

# compute an array for y = f(x) = x^2
ys = xs ** 2.0

# create a new plot for our x^2 function
plt.plot(xs, ys)

# add another plot for f(x) = x
plt.plot(xs, xs)

# define some axis labels
plt.xlabel('$x$')
plt.ylabel('$y = f(x)$')

# present your plot
plt.show()
```

This little code snippet creates the following plot:



Your task is to plot the color histogram of a previously loaded image. To create the histogram data, you are allowed to use an `skimage` helper function. Your result should look similar to this plot:



1.4 HSV Color Space Conversion

The last task of this exercise deals with different color spaces. A color space defines how the color components of an image are stored and processed. Due to its usage in most screens, the RGB color space is the most common and widespread one. However, several other color spaces also play an important role in Computer Vision. E.g. the LAB space is used by many algorithms for image colorization. The HSV space, however, is especially useful for data augmentation. Data augmentation techniques improve the generalization of Machine Learning models without the need for more training data. This is achieved by modification of the existing data. A common data augmentation technique for Computer Vision is the hue rotation of an image. While this is hard to achieve in RGB space, it is trivial in HSV space as the hue is a separate component. Your task is to implement the conversion from RGB to HSV and back, by using Numpy operations only (no scikit-image or other modules allowed).

1.4.1 Convert RGB to HSV

We first compute the value V which is defined as the maximum component in RGB space

$$X_{\text{max}} = \max(R, G, B) = V. \tag{1.1}$$

With the minimal component

$$X_{\text{min}} = \min(R, G, B) \tag{1.2}$$

we can compute the range which is also called chroma

$$C = X_{\text{max}} - X_{\text{min}}, \tag{1.3}$$

as well as the mid range, also called luminance

$$L = \frac{X_{\max} + X_{\min}}{2}. \tag{1.4}$$

The hue component is defined on a full circle. A 360° hue rotation travels through the whole visible color spectrum. We therefore have to find the closest component in RGB space in order to decide for an 120° segment of the circle:

$$H = \begin{cases} 0, & \text{if } C = 0 \\ 60°(0 + \frac{G-B}{C}), & \text{if } V = R \\ 60°(2 + \frac{B-R}{C}), & \text{if } V = G \\ 60°(4 + \frac{R-G}{C}), & \text{if } V = B \end{cases} \tag{1.5}$$

We finally compute the inverse proportion of white, called saturation

$$S = \begin{cases} 0, & \text{if } V = 0 \\ \frac{C}{V}, & \text{otherwise} \end{cases}. \tag{1.6}$$

1.4.2 Convert HSV back to RGB

The opposite direction can be achieved by first computing chroma

$$C = V \times S \tag{1.7}$$

and dividing the 360° spectrum into its components

$$H' = \frac{H}{60°}, \tag{1.8}$$

$$X = C \times (1- \mid H' \mod 2 - 1 \mid). \tag{1.9}$$

We then compute the "pure" RGB components

$$(R_1, G_1, B_1) = \begin{cases} (0,0,0), & \text{if } H \text{ is undefined} \\ (C, X, 0), & \text{if } 0 \leq H' \leq 1 \\ (X, C, 0), & \text{if } 1 < H' \leq 2 \\ (0, C, X), & \text{if } 2 < H' \leq 3 \\ (0, X, C), & \text{if } 3 < H' \leq 4 \\ (X, 0, C), & \text{if } 4 < H' \leq 5 \\ (C, 0, X), & \text{if } 5 < H' \leq 6 \end{cases} \tag{1.10}$$

and finally add the white proportion

$$m = V - C \tag{1.11}$$

to each component

$$(R, G, B) = (R_1 + m, G_1 + m, B_1 + m). \tag{1.12}$$

Your last subtask is to also plot the color histogram for the hue-rotated image. A shift of the color components should be clearly visible.

2. Image Filtering

This exercise teaches you the principles of fast image filter implementations. You will therefore learn how to manipulate pixel data directly and efficiently which is an important skill for many low-level computer vision tasks. Note, that, of course, an implementation in Python will never be "fast" compared to e.g. C or Fortran. However, the algorithms and principles remain the same

Notation

We will refer to an image as a function

$$f(x, y), \quad f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R} \tag{2.1}$$

assigning the amount of received light to each pixel (x, y). We define a colour image as a function

$$f(x, y) = \begin{pmatrix} f_R(x, y) \\ f_G(x, y) \\ f_B(x, y) \end{pmatrix}, \quad f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}^3 \tag{2.2}$$

which maps each pixel coordinate to a 3D vector containing the red, green and blue light components. In the following we will always use the gray scale definition from eq. (2.1) as the generalization to RGB is mostly trivial (just perform the operation for each component independently). A filter, kernel or convolution mask

$$h(x, y), \quad h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R} \tag{2.3}$$

is also an image. Linear filtering, also called convolution, is defined as an operation

$$f * h \to g \tag{2.4}$$

with

$$g(x, y) = \sum_{k,l} f(x + k, y + l)\ h(k, l) \tag{2.5}$$

and g being the filtered output image. Figure 2.1 illustrates the idea.

0.51	0.69	0.26	0.04	0.35	0.06
0.45	0.19	0.30	0.39	0.42	0.31
0.09	0.19	0.28	0.39	0.29	0.02
0.08	0.17	0.28	0.38	0.60	0.33
0.97	0.65	0.80	0.43	0.98	0.93
0.66	0.21	0.55	0.24	0.93	0.65

(a) f

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

(b) h

0.61	0.06	0.19	0.05	0.97	0.14
0.86	0.47	0.05	0.95	0.41	0.29
0.22	0.25	0.29	0.70	0.44	0.55
0.05	0.40	0.34	0.03	0.23	0.47
0.84	0.96	0.33	0.69	0.56	0.54
0.44	0.79	0.15	0.95	0.85	0.54

(c) g

Figure 2.1: An exemplary convolution with a kernel size of 3×3 pixels

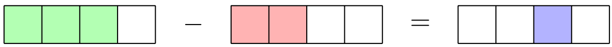


Figure 2.2: Computation of g (right) from precomputed $\tilde{f}$ (left, center)

2.1 Mean Filter

The mean filter computes the mean over a defined window of adjacent pixels. For simplicity, this task only covers one-dimensional data. In deep learning applications this is useful e.g. to improve the interpretability of a fluctuating loss curve. For the one-dimensional case the filtered output is defined as

$$g(x) = \frac{1}{2w+1} \sum_{x'=x-w}^{x+w} f(x') \tag{2.6}$$

using a window size w. According to this formula, a naïve algorithm reads as follows:

```
for x in range(len(f)):
    sum = 0
    for xp in range(x - w, x + w + 1):
        sum += f[xp]
    g[x] = sum / (2 * w + 1)
```

However, this has an extreme time complexity of $\mathcal{O}(nw)$. A much better complexity can be achieved by precomputing "auxiliary" sums $\tilde{f}$:

$$\sum_{x'=x-w}^{x+w} f(x') = \sum_{x'=0}^{x+w} f(x') - \sum_{x'=0}^{x-w-1} f(x') = \tilde{f}(x+w) - \tilde{f}(x-w-1) \tag{2.7}$$

This is also illustrated in fig. 2.2. Your task is, to implement an algorithm with a time complexity of $\mathcal{O}(n)$ by precomputing the values for $\tilde{f}$ and subsequently computing g. Also note that the complexity of this approach does not depend on the window size w at all.

2.2 Separable Convolutions

You already know the principle of convolutions from Convolutional Neural Networks. Convolutions have some useful properties, e.g. commutativity

$$f * h = h * f, \tag{2.8}$$

associativity

$$(g * h^1) * h^2 = f * (h^1 * h^2), \tag{2.9}$$

and distributivity

$$f * (h^1 + h^2) = f * h^1 + f * h^2. \tag{2.10}$$

This task deals with another non-universal property, namely separability. If a convolution is separable, its execution can be accelerated significantly. This is utilized e.g. by the popular MobileNet architecture [4]. An multidimensional n^d separable convolution can be replaced by concatenation of d $1 \times n$ convolutions. For brevity, we will deal with a 2D convolution

$$f * h = f * h^1 * h^2 \tag{2.11}$$

which can be separated into two 1D convolutions h^1 and h^2. Here is a simple example:

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}. \tag{2.12}$$

This reduces the execution complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. In order to separate h we can make use of Singular Vector Decomposition (SVD). SVD decomposes a matrix

$$M = U \Sigma V \tag{2.13}$$

into one diagonal matrix Σ and two orthonormal matrices U and V. Note, that for our separable example

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 0.42 & -0.86 & -0.29 \\ 0.57 & 0 & 0.82 \\ 0.71 & 0.51 & -0.49 \end{bmatrix} \times \begin{bmatrix} 26.46 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0.27 & -0.95 & -0.17 \\ 0.53 & 0 & 0.85 \\ 0.80 & 0.32 & -0.51 \end{bmatrix} \tag{2.14}$$

only the first singular value in the diagonal of Σ is non-zero. Therefore, the first columns of U and V (with one of them multiplied by the singular value) give us our separation into h^1 and h^2. Note that even for non-separable convolutions (multiple non-zero singular values) this method can be used to compute an approximate separation. Your task is to implement a naïve convolution as well as the SVD separation and compare the runtime of both methods.

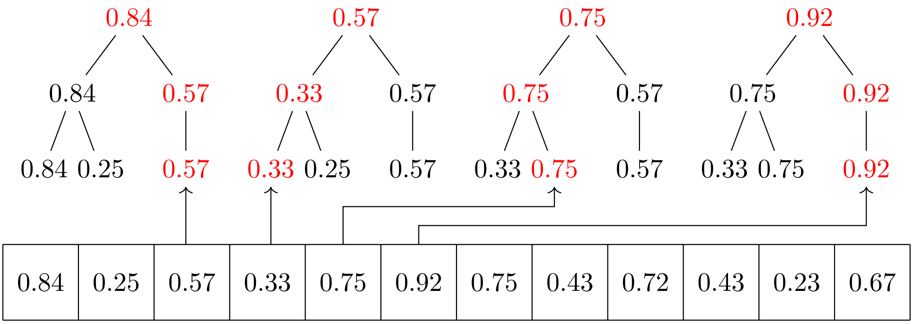


Figure 2.3: Tree based algorithm for fast maximum filter with $w = 1$. To shift the filter window, one leaf node gets replaced by the new value. Only its parent nodes must be reevaluated recursively to compute the new maximum

2.3 Maximum Filter

The maximum filter is a simple example for a morphological filter. We will, again, only cover a one-dimensional example for simplicity. The filtered output is defined as the per-pixel largest value

$$g(x) = \max_{x'=x-w}^{x+w} f(x) \tag{2.15}$$

within a given window $[x - w, x + w]$.

According to eq. (2.15), a naïve algorithm would just enumerate all values within the window for each pixel x. Unfortunately, this simple algorithm has a time complexity of $\mathcal{O}(nw)$. One idea for a faster version is based on a binary tree. As fig. 2.3 illustrates, each parent node gets assigned the maximum value of its children. This has one key advantage over naïve iteration: if the value of one leaf node changes, only its parent nodes need to be updated recursively. As a consequence, we do not need to perform an operation for all $2w + 1$ pixels within our window. Instead, we just update $\lceil \log_2(2w+1) \rceil$ levels of the binary tree. This reduces the time complexity of this approach to $\mathcal{O}(n \log w)$. For each output $g(x)$ the window gets shifted by one pixel. Hence, the leaf node that holds the value of the dropped pixel is overridden by the value of the new pixel. Lastly, all parent nodes need to be updated.

Your task is the implementation of the naïve 1D maximum filter as well as the fast version based on a binary tree. We already implemented the trivial 2D generalization which is used for runtime comparisons on images. The $\mathcal{O}(n \log w)$ implementation should be significantly faster for large window sizes.

Bibliography

[1] T. Leistner, R. Baumgartner, and C. Rother, *Exercises for 3D Computer Vision*. [Online]. Available: https://github.com/titus-leistner/3dcv-students

[2] *NumPy*. [Online]. Available: https://numpy.org

[3] *Matplotlib*. [Online]. Available: https://matplotlib.org

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, 2017.

[5] *Scikit-Image*. [Online]. Available: https://scikit-image.org