Heidelberg University

Visual Learning Lab

Exercises for 3D Computer Vision

Titus Leistner
Philip-William Grassal
Raphael Baumgartner
Prof. Dr. Carsten Rother

October 21, 2021

Contents

1 Scientific Python **7**
- 1.1 NumPy . . . 7
- 1.2 Scikit-Image . . . 8
- 1.3 Matplotlib . . . 8
- 1.4 HSV Color Space Conversion . . . 9
  - 1.4.1 Convert RGB to HSV . . . 9
  - 1.4.2 Convert HSV back to RGB . . . 10

2 Image Filtering **13**
- 2.1 Mean Filter . . . 14
- 2.2 Separable Convolutions . . . 15
- 2.3 Maximum Filter . . . 16

3 Feature Matching **17**
- 3.1 Key Point Detection . . . 17
- 3.2 Feature Description . . . 18
- 3.3 Matching . . . 19

4 Fundamental Matrix **21**
- 4.1 Estimation using SVD . . . 22
- 4.2 Algebraic Error vs Geometric Error . . . 22
- 4.3 Robust Estimation using RANSAC . . . 23

5 PyTorch **25**
- 5.1 Tensors . . . 25
- 5.2 Autograd . . . 26
- 5.3 Dataset Class for MNIST . . . 26
- 5.4 Running Pretrained Networks . . . 27

6 Deep Learning Models **29**
- 6.1 Training Loop . . . 29
- 6.2 Fully Connected Network . . . 29

6.3 Convolutional Neural Network 31

7 DSAC **33**
7.1 Line Fitting 34
7.2 Synthetic Image Generation 34
7.3 Circle Loss Function 35
7.4 DSAC Circle Fitting 37
7.4.1 Hypothesis Sampling 37
7.4.2 Soft Inlier Count 38
7.4.3 Refinement with Hard Inliers 38
7.5 Results 39

Bibliography **41**

Introduction

Welcome to the online exercise for Computer Vision: 3D Reconstruction. We prepared seven 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 and corner detection
- Feature matching to find correspondences between two images of the same scene
- Fundamental matrix estimation to find geometric constrains in the scene
- 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, especially for the first exercise. We made the first exercise optional, so feel free to skip it, if you are already familiar with Python and Numpy.

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.

You will receive feedback, but no points for the first optional exercise. All other exercises can be edited in groups. There will be a "Groups" forum to find group partners. We will then switch the Moodle exercises to group modes, hence, you only have to submit one solution per group.

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, Philip-William Grassal, 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:

```
1 # import the numpy module
2 # note that np is the common naming convention
3 import numpy as np
4
5 # create a new 1D array with 4 elements from a Python list
6 a = np.array([0, 0.32, 10, 12], dtype=np.float)
7
8 # perform a basic operation with a floating-point scalar
9 b = a * 42.0
10
11 # use slicing to get the last two elements of this array
12 c = b[-2:]
```

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

1.2 Scikit-Image

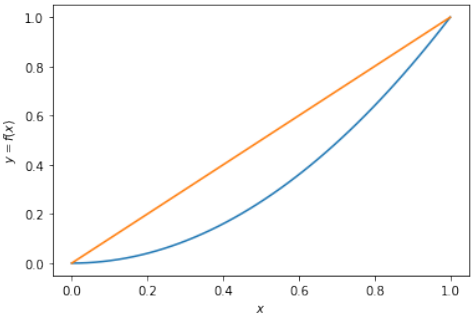
For all exercises we will use scikit-image for basic image loading, storing and manipulation. This module is part of the SciPy ecosystem for scientific computing and rather light-weight and easy to install compared to e.g. OpenCV.

1.3 Matplotlib

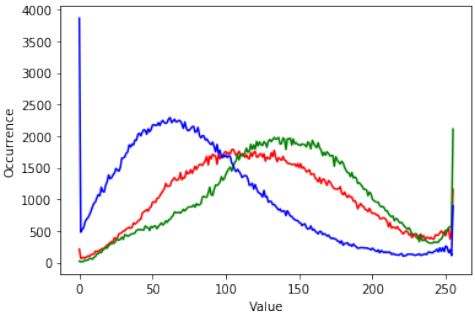
Matplotlib is an extremely powerful library for scientific visualization. In this exercise you will use it to output your processed images as well as some simple function plots and histograms. Let's again look at some minimal example:

```
# 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

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

With the minimal component

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

we can compute the range which is also called chroma

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

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.

0.58	0.03	0.52	0.07	0.46	0.43
0.41	0.19	0.30	0.39	0.25	0.12
0.51	0.19	0.28	0.39	0.16	0.02
0.46	0.17	0.28	0.38	0.77	0.61
0.90	0.55	0.15	0.17	0.61	0.68
0.33	0.12	0.74	0.63	0.33	0.90

(a) f

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

(b) h

0.14	0.35	0.30	0.24	0.06	0.97
0.06	0.54	0.28	0.48	0.10	0.25
0.22	0.77	0.29	0.55	0.15	0.78
0.47	0.40	0.33	0.68	0.49	0.52
0.51	0.22	0.51	0.43	0.90	0.05
0.94	0.18	0.25	0.44	0.05	0.59

(c) g

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

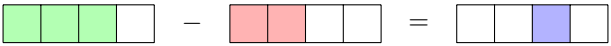


Figure 2.2: Computation of g (right) from precomputed $\tilde{f}$ (left, center)

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:

```
1    for x in range(len(f)):
2        sum = 0
3        for xp in range(x - w, x + w + 1):
4            sum += f[xp]
5        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.

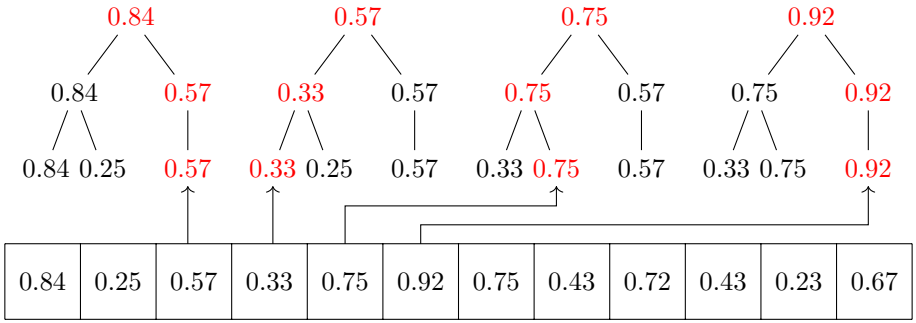


Figure 2.3: Tree based algorithm for fast maximum filter with $w = 1$. To shift the filter window, one leaf node gets replaced by the new value. Only its parent nodes must be reevaluated recursively to compute the new maximum

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.

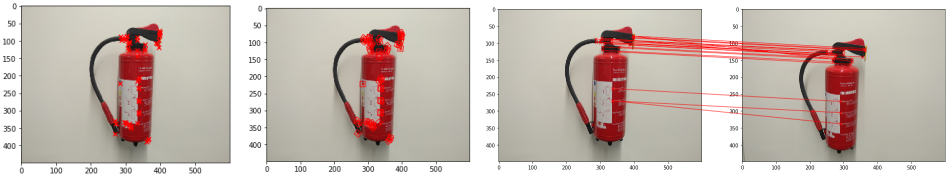
Your task is the implementation of the naïve 1D maximum filter as well as the fast version based on a binary tree. We already implemented the trivial 2D generalization which is used for runtime comparisons on images. The $\mathcal{O}(n \log w)$ implementation should be significantly faster for large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

3. Feature Matching

In this exercise, you will harden your knowledge about key point detection and matching. This is an essential task in computer vision which allows to identify corresponding objects in multiple images of the same scene, e.g. adjacent frames of a video or photos from a famous building found on the internet. To find these correspondences, you will need to go through three steps: key point detection, feature description and matching, see Figure 3.1. All of them will be covered in this exercise. While there are many well-known approaches and implementations for each step, you will implement everything yourself.

3.1 Key Point Detection

Our method should be applicable to all kinds of scenery and thus we can not make any assumption on higher-level contents. As a result, we are looking for lower level image structures rather than complete objects. We require the image structures to be uniquely identifiable across images which is something edges can not provide but corner points can! Hence, those are the key points or interest points we are looking for. In the lecture, you learned about the Harris corner detector which is a tool you



(a) Detection (b) Description (c) Matching

Figure 3.1: a) To find correspondences between views of the same scene, significant spots (key points) are identified in every image. b) Afterwards, a feature descriptor is created by encoding the pixel information of the proximity around a key point. c) Key points of different images are matched based on the distance of their feature descriptors.

are supposed to use in this exercise, as well. It exploits the fact that the image signal must change a lot in all directions around corners. "*Around corners*" is modeled as a proximity window $W(x, y)$ of a pixel (x, y) and the "*change [...] in all directions*" c is computed by moving W in some direction $(\Delta x, \Delta y)$. In summary, we compute c as

$$c(x, y, \Delta x, \Delta y) = \sum_{(u,v)\in W(x,y)} (I(u, v) - I(u + \Delta x, v + \Delta y))^2 \tag{3.1}$$

$$\approx \sum_{(u,v)\in W(x,y)} ([I_x(u, v), I_y(u, v)][\Delta x, \Delta y]^T)^2 \tag{3.2}$$

$$= [\Delta x, \Delta y]Q(x, y)[\Delta x, \Delta y]^T \tag{3.3}$$

with I_x, I_y being the image gradients and $Q(x, y)$ being the structure tensor, a 2×2 square matrix in this case. The first line is essentially the squared distance between W and shifted W. The second step simplifies this equation by using a Taylor approximation of $I(u + \Delta x, v + \Delta y)$ which can be rearranged to obtain Q, encoding all the corner information for every pixel independent of the window shift. The eigenvalues of Q express the change of the image signal around one point. If both eigenvalues are large, we found a corner. Consequently, we compute $Q(x, y)$ for all (x, y) and select the best points via thresholds and non-maximum suppression as our corner key points. You may look at the slides again for more details.

In the exercise, you are asked to implement the Harris detector yourself. You apply it to the images provided by us. Typically, you would want to apply the corner detection algorithm multiple times per image but using different resolutions to discover larger and smaller corners. However, we omit this for simplicity.

3.2 Feature Description

A key point itself does not encompass much information except for its location and color. This makes it difficult to confidently identify the same key point in two images. Hence, we encode information about its neighborhood in a separate feature vector/descriptor which ideally provides us with a unique identifier. The neighborhood is a fixed pixel region around the key point. When detecting corners using multiple resolutions of the image, the pixel region is scaled accordingly. For this exercise, however, this region has the same size for all key points. The region is rotated with respect to the image gradient at the corner point which guarantees that always the same region is chosen in all views of the scene. Finally, the pixel information in each region is aggregated. A very common method is to count the orientation of every pixel's image gradient. Afterwards the counted directions are aggregated as histograms which are then concatenated to represent a feature vector.

In the exercise, you are asked to implement the region extraction, histogram computation and feature vector creation.

3.3 Matching

The final step is about matching key points between two images based on their descriptors. Let $\mathcal{F}_i$ be the set of feature descriptors extracted for key points from image I_i with $i = 1, 2$. If we compute the a chosen distance function $d(\cdot, \cdot)$ between every $f \in \mathcal{F}_1$ and every $f' \in \mathcal{F}_2$, we obtain a distance matrix D with dimensions $|\mathcal{F}_1| \times |\mathcal{F}_2|$. By looking for the smallest distance in every row and column of D, the best match for the respective descriptor (key point) can be found.

In the exercise, you need to compute D with $d(f, f') = \|f - f'\|_2^2$. Afterwards, you can select the best matches.

4. Fundamental Matrix

After having learned how to find point correspondences between pairs of views, you will now discover how these can be utilized to constrain the scene 3D geometry, given the key points were matched correctly. This is important if you want to retrieve camera parameters from the scene and finally reconstruct points in 3D space. Assume that we have a correct match of key points x_1, x_2 from two images I_1, I_2 of the same scene. Then we know that x_1, x_2 represent the 2D image points of the same 3D point X recorded by the respective camera located at C_1 and C_2. If we further assume that both cameras are pinhole-like (perspective projection with no non-linear distortions), the mapping from X to x_1, x_2 is expressed via projections $x_1 = P_1 X$ and $x_2 = P_2 X$. A projection matrix can be decomposed into its intrinsic (projection) and extrinsic (rotation and translation) components $P = K \cdot [R| - C]$. As we only care about the relative motion and rotation between both cameras, we can assume camera C_1 to be at the origin with no rotation and decompose $x_1 = P_1 X = K_1 \cdot [I|0]X$ and $x_2 = P_2 X = K_2 \cdot [R^{-1}| - C_2]$. Note that R^{-1} is also a valid rotation matrix and only chosen for nicer equations in the end.

Figure 4.1 illustrates how we can derive a constraint from this. X is projected onto the image planes of both camera along the viewing ray starting at the respective camera center. Furthermore, the translation vector T from C_1 to C_2 connects both camera centers. We see that T and both rays lie in one plane and hence they are orthogonal to its normal vector. This is the constraint we can construct without

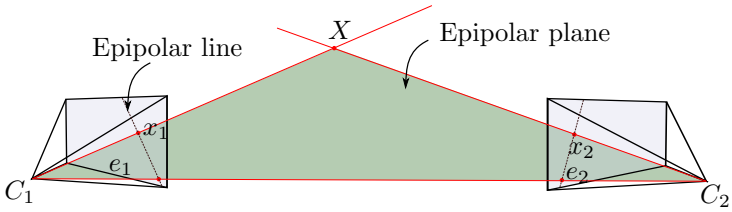


Figure 4.1: Epipolar geometry between two camera views of the same scene. The translation vector between both cameras, the viewing ray through C_1 and X and the viewing ray through C_2 and X lie in the epipolar plane (green).

knowing anything about the cameras or images taken by them. Both viewing rays and T are expressed as vectors in the coordinates of the first camera

$$X = K_1^{-1}x_1 \qquad X - C_2 = RK_2^{-1}x_2 \qquad T = C_2 \tag{4.1}$$

Using the cross product, we find an orthogonal vector which allows us to formulate the epipolar plane constraint

$$X^T(T \times (X - C_2)) = X^T[T]_\times(X - C_2) = 0 \tag{4.2}$$

$$x_1^T K_1^{-T}[T]_\times RK_2^{-1}x_2 = x_1^T F x_2 = 0 \tag{4.3}$$

where $[T]_\times$ is the matrix formulation of the cross product and F is the fundamental matrix. Note because $[T]_\times$ is rank 2, F is rank 2 as well. F encodes the epipolar plane constraint. But what does this get us? First, it will serve as a starting point for recovering the camera intrinsic and extrinsic parameters. In fact, one can already perform a perspective reconstruction using F but it will look very distorted. Second, it relates all points in both views and not only key points. Given some point x_1 in the first image, $x_1^T F$ yields a line (point-line duality of homogeneous coordinates) in the second image where the corresponding point x_2 must lie. This line is called epipolar line, see Figure 4.1. As a result, dense correspondences can be extracted more easily. For more information, look at the slides again.

In this exercise, you are given a set of matched key points (some might be incorrect). Your overall task is to find F.

4.1 Estimation using SVD

First, we assume that all given point matches are correct and precise. We can rearrange Equation (4.3) for each pair of matched key points $(x_1, y_1, 1), (x_2, y_2, 1)$ to

$$(x_1x_2, x_1y_2, x_1, y_1x_2, y_1y_2, y_1, x_2, y_2, 1) \cdot (f_{11}, f_{12}, \ldots, f_{33})^T = 0 \quad . \tag{4.4}$$

Try this yourself on a sheet of paper. If you have eight or more matches, you can combine the resulting equations to one linear system $A \cdot f = 0$ where the right null space of A is $f = (f_{11}, f_{12}, \ldots, f_{33})^T$ which are the vectorized elements of F. We obtain f by solving $\text{argmin}\|Af\|_2$ subject to $\|f\|_2 = 1$ using SVD. This method is often called *direct linear transform* and in our context known as 8-point algorithm.

In the exercise, you will be asked to find F by applying the 8-point algorithm. For a correct and stable result, do not forget to condition the key point coordinates and enforce that F has rank 2 as discussed in the lecture.

4.2 Algebraic Error vs Geometric Error

As even correct matches suffer from a little bit of noise regarding the location of the key points, the 8-point algorithm yields only an approximation of F. By using

SVD, we choose F such it solves the algebraic problem (being in or close to A's right null space) optimally but ignore geometric deviations that result from solving the problem under noise. This becomes visible when predicted epipolar lines do not pass through the key points. Therefore, a geometric error measure must be defined that reports how far epipolar lines miss corresponding key points. You will learn about this error in a later lecture.

We provide functions in the task Jupyter notebook to visualize the epipolar lines and compute the algebraic and geometric errors. Check the quality of F obtained from the 8-point algorithm by using these functions. What do you see?

4.3 Robust Estimation using RANSAC

Keeping the geometric error in mind is one factor to get a reasonable estimate for F. Considering outliers and incorrect matches is another one. The RANSAC algorithm deals with both of them and, therefore, provides a more robust solution to our problem. You should have seen it already in the lecture. While there are slight variations between implementations, the core idea is always the same. We will sketch all steps for our specific problem:

for n iterations **do**
    select at least 8 key point matches
    estimate F using the 8-point algorithm
    compute the geometric error for every match using the estimated F
    if the geometric error of at least m matches is less than threshold t **then**
        we call those matches *inliers*
        if the average error of all *inliers* is less than the smallest error so far **then**
            store F, the *inliers* and the average geometric error
        end if
    end if
end for

In your final task, you will be asked to implement RANSAC following the pseudocode above. Check again the quality of the estimated F. What has changed?

5. PyTorch

This exercise will teach you the basics of PyTorch [5] which are essential for all following deep -learning tasks. It introduces the fundamentals of tensor calculations, autograd and some practical skills like dataset loading and the usage of already implemented and pretrained neural networks.

5.1 Tensors

The tensor class is PyTorchs equivalent to the Numpy ndarray with some additional features. First of all, it tracks gradients which we will see in the next task. A tensor can be stored either on the CPU or GPU. Unfortunately, the tensor syntax is close but not similar to Numpy. Hence, some operations are be called differently. Let's revisit the example from the Numpy exercise:

```
import numpy as np
import torch

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

# create a tensor from this array
t = torch.from_numpy(a)

# move tensor to GPU and back to CPU
t_gpu = t.cuda()
t_cpu = t_gpu.cpu()

# most mathematical operations, indexing and slicing
# work similar to Numpy
s = t * 42.0
r = s[-2:]

# but there are some operations that are named differently
# or may have different parameters
aT = a.transpose()
tT = t.t()
```

Your first task is to get used to the most important tensor operations.

5.2 Autograd

The core feature of tensors is the autograd capability which provides automatic differentiation for any operation that is performed on a tensor. In contrast to other deep learning tools, it is a define-by-run framework which means that the graph of operations is defined at runtime. First, gradients for a tensor must be enabled using `t.requires_grad = True`. Usually, lots of operations are now applied to this tensor, resulting in a scalar loss value. Pytorch stores a reference to the operation that created a tensor in `t.grad_fn`. By calling `loss.backward()`, the gradients for all intermediate results are computed and stored in `t.grad`.

Your task is the to perform the following operations on a random 2×2 tensor x:

$$o = \frac{1}{4} \sum_i (x_i + 2)^2 \tag{5.1}$$

you can check the correctness of the gradients

$$\frac{\partial o}{\partial x_i} = \frac{1}{2}(x_i + 2) \tag{5.2}$$

manually.

5.3 Dataset Class for MNIST

In theory you could just load some image similar to the last exercises, convert it to a tensor and run some neural network on it. However, especially for network training, the `torchvision` package provides some useful helper classes for dataset loading. It is therefore always advisable to implement a dataset class compliant to torchvisions best practice. Please note that although torchvision already includes a dataset class for MNIST, you will implement a simple version of it by yourself to prepare you for the mini projects that might involve your own datasets. A torchvision dataset class must implement the methods `__getitem__(self, i)` which should return the ith data point as a tuple and `__len__(self)` which should return the total number of data points. The helper class `torch.utils.data.DataLoader` can be used for efficient loading of an arbitrary dataset object.

You should also support transforms like `torchvision.transforms.Normalize`. Transforms are callable objects applicable to tensors in the same way as a function. To create your own transform you have to implement the `__call__(self, img)` method. `torchvision.transforms.Compose` takes a list of arbitrary transforms and runs them successively. This is especially useful if you want to perform several data augmentation tasks.

5.4 Running Pretrained Networks

In this task, you will use your dataset class to run a pretrained neural network on the MNIST test dataset. All trained parameters of a network model (which is also a Python class) can be stored and loaded using the state dictionary of a network object.

```
import torch

# instantiate network
model = YourNeuralNetworkClass()

# training code
# ...

# store trained parameters
torch.save(model.state_dict(), 'checkpoint.pt')

# load trained parameters
model.load_state_dict(torch.load('checkpoint.pt'))
```

Your task is the performance validation of the model we provide. Therefore you should compute the accuracy (percentage of correct predictions) and also plot some exemplary MNIST images with their predictions. To analyze problems during training, marking the wrongly predicted data points is also a good idea. Your result should look similar to this image:

6. Deep Learning Models

In this exercise, you will develop several neural network models for image classification. You will gradually improve your architecture by changing architectural details. Furthermore, you will evaluate the performance gain on image data achieved by convolutional neural networks.

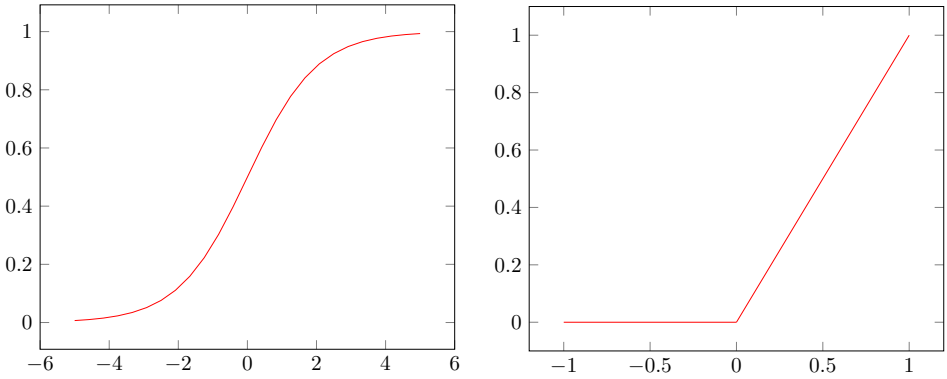
6.1 Training Loop

Before you start to implement the actual network models, you need a training procedure. As shown in the last exercise, training neural networks is achieved with Pytorchs autograd feature. An instance of an optimizer class is used to update the network weights. Neural networks are usually not trained with a single data point, hence a single image, but a batch of images instead. The size of one training batch should be adjusted according to your available memory. If you have access to a GPU, choose the largest batch size that still fits in your GPU's memory. The learning rate determines how much the network parameters are updated within one optimization step. If the batch size changes, the learning rate should be adjusted accordingly. E.g. if you train with twice the batch size you can also double the learning rate.

Your task is to implement the training for one epoch. A training epoch iterates once over the whole dataset. Usually this process is repeated several times until convergence. In Pytorch, the first step is to clear the gradients that are already stored in your optimizer instance. Second, you have to run the network and compute the loss from its output and the ground truth data. Third, the backward pass is performed and last, the network weights are updated by the optimizer. Make sure to also log the current loss after every few iterations to check for convergence while your training is running.

6.2 Fully Connected Network

The first model you will train is a simple fully-connected network. In a fully-connected layer (sometimes also called linear layer), each input dimension is con-



(a) $f_\sigma(x) = \frac{1}{1+\exp(-x)}$ (b) $f_{\text{ReLU}}(x) = \max(0, x)$

Figure 6.1: Comparison of Sigmoid and Rectified Linear Unit (ReLU)

nected to each output dimension.

$$y_j = \sum_i x_i w_{ij} + b_j \tag{6.1}$$

With x and y being the input and output, w and b being the weights and biases. Hence, even for image data, both input and output are represented as a 1D vector without any spatial information. Don't forget to reshape your 2D input to one dimension.

When your initial network is trained, your task is to compare two different activation functions (compare fig. 6.1). The sigmoid function is fully-differentiable and was therefore used in most older image classification networks. However, more recent research showed that the much simpler ReLU (Rectified Linear Unit) works better for most tasks. If your implementation is correct you will see a slight improvement in your results after replacing Sigmoid with ReLU.

Further improvements can be achieved by using batch normalization. Batch normalization subtracts the mean from a layers output and divides it by its standard deviation. This compensates for a high variance between different inputs batches (e.g. mostly thicker numbers vs. mostly thinner numbers). As a result, the network weights only have to be changed slightly which accelerates the training process. In addition, batch normalization also improves generalization of a network and therefore reduces overfitting to the training data. You should therefore observe a smaller difference between validation and training accuracy if batch normalization is applied.

6.3 Convolutional Neural Network

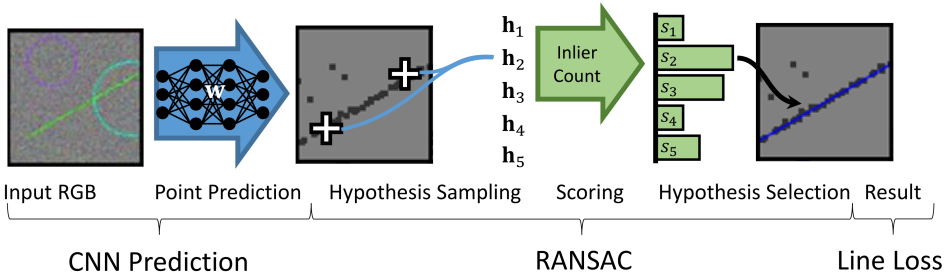
We already discussed one downside of the traditional fully-connected neural network: The input data is essentially a one-dimensional vector, hence it does not utilize spatial information and structures like lines or corners. This problem is addressed by CNNs (Convolutional Neural Networks). Convolutional layers extract spatial information without requiring as many trainable parameters as comparable fully-connected layers. A convolutional layer essentially is a convolutional kernel, covered in a previous exercise, with trained weights.

In order to reduce the dimensions of an image, the convolutional layer can skip a number of pixels, referred to as strides. Strides of e.g. $2, 2$ mean that the convolutional filter is only applied to every second pixel in the image which effectively bisects the outputs width and height. By using this technique the spatial information is condensed by each convolutional layer. For the last part of the network, a small number of fully-connected layers transforms those features to output class predictions. As an alternative to strides, max pooling can be used. Max pooling effectively applies a max filter, as implemented during a previous exercise, to the output of a convolutional layer and therefore also reduces the width and height. This should again improve the network performance slightly.

Please note that, due to the random initialization of your networks trainable parameters, your results may vary. Some changes will lead to only minor improvements that may be below the variance between multiple trainings. To make sure that your performance actually improved, you may have to train the network a few times and compare the results.

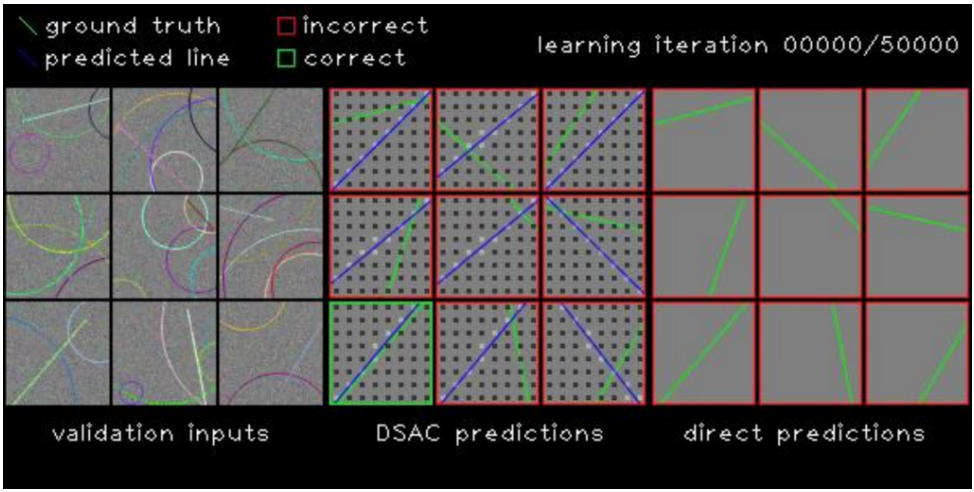
7. DSAC

This last exercise will give you an example of a recent research topic in our group, namely Differentiable Sample Consensus (DSAC) [6]. In the paper, DSAC is applied to the problem of line fitting in noisy image data. Your task is the implementation a similar approach for circle fitting. The whole pipeline is illustrated in the following figure:



As already shown in the lecture, the basic idea of DSAC is a fully-differentiable RANSAC pipeline. Unlike RANSAC, the sample prediction (or in this particular task point prediction) algorithm is replaced by a neural network. To achieve this, all downstream operations must be differentiable. We already prepared the neural network and most of the code skeleton. Your task is the implementation of hypothesis sampling, soft inlier count and loss function. You will also add a refinement step that re-fits the circle to the (soft) inliers.

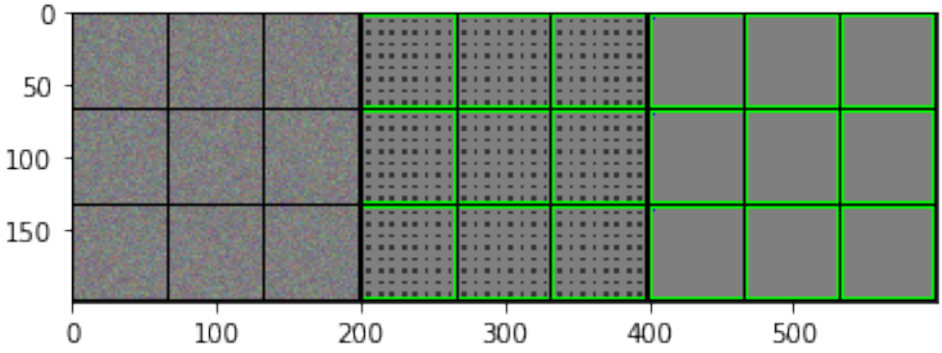
7.1 Line Fitting



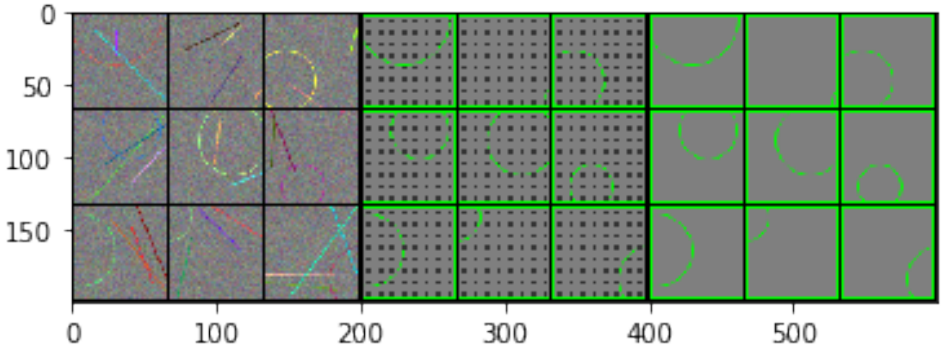
Your first task is to run and inspect the line fitting example which is already implemented. Note that the line fitting code that can be found inside the `vll` module is a really useful reference for all following tasks. The provided training code continuously writes the loss values to a text file. In order to check the training progress, you have to load this file and plot the raw and mean-filtered contents.

7.2 Synthetic Image Generation

Before we can start to implement and train DSAC for circle fitting, we need to generate some data. Many modern Computer Vision methods use synthetic training data, due to cheap generation and free ground truth annotations. We provide code that creates a directory with output images that will first look like this:



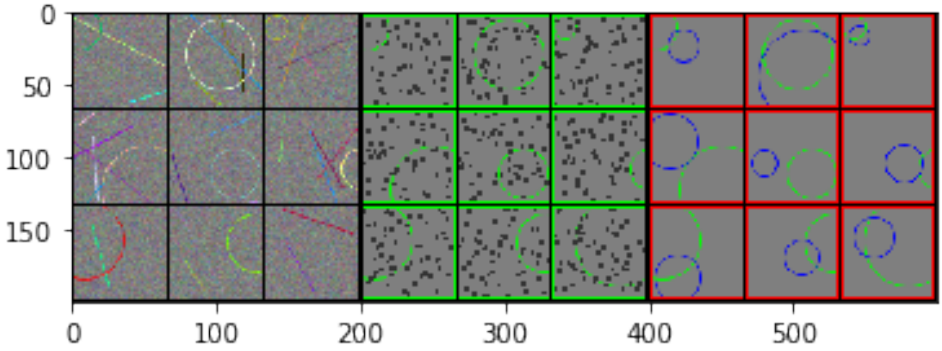
After filling in the blank methods for circle generation, your output should be similar to this:



Note that green circles on the right represent the ground truth data.

7.3 Circle Loss Function

Your next task is the circle loss function. On the right of all plots:



you see the result of a CNN that directly predicts one circle without the DSAC framework and is used as a baseline. Please note, that this method usually converges faster. DSAC might therefore require many more iteration until it outperforms its baseline. If you implemented the loss function correctly, the training of this direct prediction network should work now. The plot above shows the results after 100 iterations. You don't have to run the training for hours, it is sufficient if the loss starts to decrease noticeably. Note that in the beginning, most boxes should be marked red (i.e. incorrect). In case your boxes appear green, you most likely did not scale the loss by the image size.

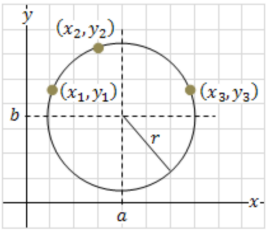
7.4 DSAC Circle Fitting

You now have to implement circle fitting.

7.4.1 Hypothesis Sampling

The next image shows how to fit a circle to three points.

Equation of a circle passing through 3 points (x_1, y_1) (x_2, y_2) and (x_3, y_3).



The equation of the circle is described by the equation:

$$Ax^2 + Ay^2 + Bx + Cy + D = 0$$

After substituting the three given points which lies on the circle we get the set of equations that can be described by the determinant:

$$\begin{vmatrix} x^2+y^2 & x & y & 1 \\ {x_1}^2+{y_1}^2 & x_1 & y_1 & 1 \\ {x_2}^2+{y_2}^2 & x_2 & y_2 & 1 \\ {x_3}^2+{y_3}^2 & x_3 & y_3 & 1 \end{vmatrix} = 0$$

The coefficienta A, B, C and D can be found by solving the following determinants:

$$A = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \quad B = -\begin{vmatrix} {x_1}^2+{y_1}^2 & y_1 & 1 \\ {x_2}^2+{y_2}^2 & y_2 & 1 \\ {x_3}^2+{y_3}^2 & y_3 & 1 \end{vmatrix} \quad C = \begin{vmatrix} {x_1}^2+{y_1}^2 & x_1 & 1 \\ {x_2}^2+{y_2}^2 & x_2 & 1 \\ {x_3}^2+{y_3}^2 & x_3 & 1 \end{vmatrix} \quad D = -\begin{vmatrix} {x_1}^2+{y_1}^2 & x_1 & y_1 \\ {x_2}^2+{y_2}^2 & x_2 & y_2 \\ {x_3}^2+{y_3}^2 & x_3 & y_3 \end{vmatrix}$$

The values of A, B, C and D will be after solving the determinants:

$$A = x_1(y_2 - y_3) - y_1(x_2 - x_3) + x_2y_3 - x_3y_2$$
$$B = ({x_1}^2+{y_1}^2)(y_3 - y_2) + ({x_2}^2+{y_2}^2)(y_1 - y_3) + ({x_3}^2+{y_3}^2)(y_2 - y_1)$$
$$C = ({x_1}^2+{y_1}^2)(x_2 - x_3) + ({x_2}^2+{y_2}^2)(x_3 - x_1) + ({x_3}^2+{y_3}^2)(x_1 - x_2)$$
$$D = ({x_1}^2+{y_1}^2)(x_3y_2 - x_2y_3) + ({x_2}^2+{y_2}^2)(x_1y_3 - x_3y_1) + ({x_3}^2+{y_3}^2)(x_2y_1 - x_1y_2)$$

Center point (x, y) and the radius of a circle passing through 3 points (x_1, y_1) (x_2, y_2) and (x_3, y_3) are:

$$x = \frac{({x_1}^2+{y_1}^2)(y_2 - y_3) + ({x_2}^2+{y_2}^2)(y_3 - y_1) + ({x_3}^2+{y_3}^2)(y_1 - y_2)}{2(x_1(y_2 - y_3) - y_1(x_2 - x_3) + x_2y_3 - x_3y_2)} = -\frac{B}{2A}$$

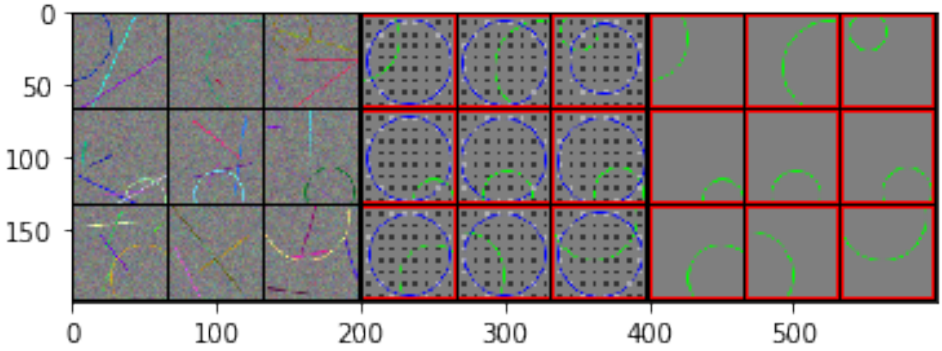
$$y = \frac{({x_1}^2+{y_1}^2)(x_3 - x_2) + ({x_2}^2+{y_2}^2)(x_1 - x_3) + ({x_3}^2+{y_3}^2)(x_2 - x_1)}{2(x_1(y_2 - y_3) - y_1(x_2 - x_3) + x_2y_3 - x_3y_2)} = -\frac{C}{2A}$$

$$r = \sqrt{(x - x_1)^2 + (y - y_1)^2} = \sqrt{\frac{B^2 + C^2 - 4AD}{4A^2}}$$

Please remember that PyTorch offers many handy functions like the computation of matrix determinants. Your circles should interpolate the three points perfectly. In later stages, you should always have at least three inliers.

7.4.2 Soft Inlier Count

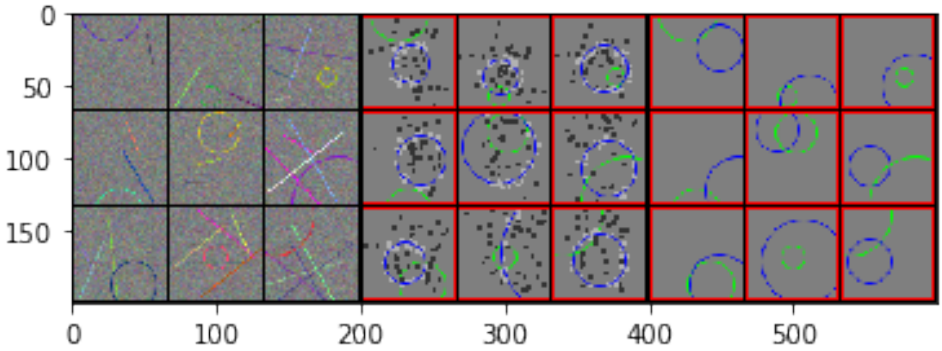
The soft inlier count is a differentiable version of a hard threshold. To calculate it, compute the distance of each predicted point to the circle and apply the sigmoid function. Remember that you can always take a look at the line fitting code for reference. Your code should be running now. The first generated image should look like this:



Note how DSAC chooses circles which almost fill the image (middle), because they have most inliers (bright points).

7.4.3 Refinement with Hard Inliers

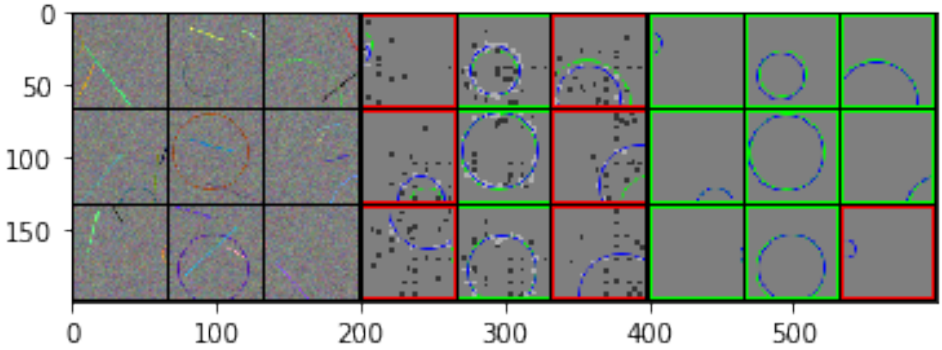
The last additional step is the refinement of your hypothesis by using not only three points, but an arbitrary number of inliers. A description of least-squares circle fit can be found in `circle_fit.pdf` [7]. Fit the circle to all soft inliers above a threshold of e.g. 0.5. The following image shows the refined results:

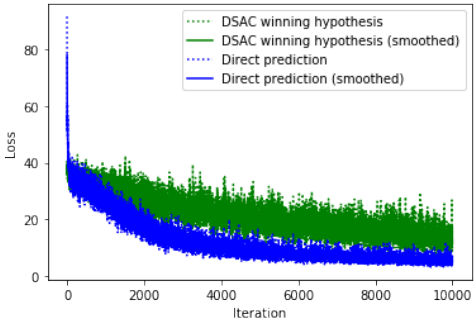


Note how the refined circles go through the white inlier points. As a bonus task, you can adapt the code to fit the circle to all soft inliers instead, using the soft inlier score as weight in the appropriate places.

7.5 Results

You should now train the networks for at least 1000 iterations. In addition, you can play around with parameters and analyze the impact on the results. The fits and losses should look similar to this after 10000 iterations:





Again, if your hardware does not allow that many training iterations, it is sufficient if the loss noticeably decreases.

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] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NEURIPS 32*, 2019.

[6] E. Brachmann, A. Krull, S. Nowozin, J. Shotton, F. Michel, S. Gumhold, and C. Rother, "DSAC - differentiable RANSAC for camera localization," in *CVPR*, 2017.

[7] *Least-Squares Circle Fit - DTC Documentation*. [Online]. Available: https://dtcenter.org/met/users/docs/write_ups/circle_fit.pdf

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