UNIVERSIDADE FEDERAL
DE SANTA CATARINA

HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

Ultraviolet Fluorescence Image Analysis as Inspection Method for Photovoltaic Cells

Development of an Experimental Setup and an Automatized Image Processing Tool

Bachelor Thesis

Study Program: Energy and Environmental Engineering
at the Hochschule Ruhr West

Timon Benz
10011539,
timon.benz@stud.hs-ruhrwest.de

First examinor: Prof. Dr.-Ing. Marcus Rehm
Second examinor: Prof. Dr. Ricardo Rüther
Supervisor: Ms. Aline Kirsten

Cooperation Partner:
Laboratory of Photovoltaics of the UFSC, Universidade Federal de Santa Catarina

Florianópolis, Brazil, 05-07/2022

Index

List of Figures 4

List of Tables 6

List of Abbreviations 7

1 Background 8

2 Scope and aims of this project 9

3 Central Hypothesis and Questions 10

4 Theory & State of the Art 11

4.1 Formation of UVF by exposure to sunlight 11

4.2 Degradation mechanisms and their interdependence 13

4.3 Present experimental setups for onsite inspection 16

4.4 Basic concepts of Deep Learning for Image Classification 17

4.4.1 Concepts of Neural Networks 18

4.4.2 Convolutional Neural Networks for Image classification 22

4.5 Present Evaluation Tools 24

4.5.1 OpenUVF 24

4.5.2 PV Vision 25

4.6 UVF patterns from earlier studies 26

5 Methodology 27

5.1 Plan of procedures 27

5.2 UVF Image acquisition in this project 28

5.2.1 Fotovoltaica/UFSC in Florianópolis 28

5.2.2 The PV and wind power plant in Tubarão 29

5.2.3 First setup, applied in Tubarão 30

5.2.4 Second setup 31

5.2.5 Third Setup, applied at the Fotovoltaica/UFSC laboratory 32

5.2.6 Comparisons and Conclusions on setups 32

5.3 Processing of UVF images 34

5.4 Testing and evaluation the tool OpenUVF 34

6 Results 35

6.1 Observed UVF patterns 35

6.2 Automatized Image Processing Pipeline 36

6.2.1 General settings, importing libraries and loading the original image 37

6.2.2 Brightness correction 39
6.2.3 Extracting the green colour channel or converting to grey scale 40
6.2.4 Inverting 41
6.2.5 Thresholding 42
6.2.6 Contour Detection and Morphological Operations 44
6.2.7 Filtering contours by criteria 48
6.2.8 Perspective transformation 50
6.3 Cropping cell images from a pre-processed module image 51
6.3.1 Morphological operations to intensify cell borders 51
6.3.2 Filtering cell contours by criteria 52
6.3.3 Cropping each cell by perspective transformation functions 53
6.4 Training a CNN on classification of UVF cell images 54
6.4.1 Creating a data bank of cell images 54
6.4.2 Creating data generators applying data augmentation 56
6.4.3 Load and modify a pretrained neural network 57
6.4.4 Train and save the CNN model 58
6.4.5 Evaluation of the training progress 59
6.4.6 Evaluate the model on test data and do predictions 59
6.5 Results of the Image Processing with the developed pipeline 60

Summary 63

Conclusion 64

References 65

7 Appendix I: How to get started with Deep Learning 67
7.1 How to install Anaconda, keras and Jupyter in Windows 67
7.2 How to install PyCharm with an existing Anaconda environment in Windows 72
7.3 Literature recommendations for Deep Learning 73
7.4 Video Presentation of this work and further links 74

8 Appendix II: Codes of the Image Processing Pipeline 76
8.1 Code: Module Processing Pipeline (Perspective Correction) 76
8.2 Code: Cell Cropping Pipeline 84
8.3 Code: Training a CNN on cell image classification 93

Declaration 98

List of Figures

Figure 1: UVF intensity vs. UV dose for different encapsulants (EVA) and temperatures [2] p.5 11
Figure 2: Current density loss correlated to UV dose and equivalent years of operation in Northern Germany, [2] p.6........ 12
Figure 3: UVF formation and -extinguishing by photobleaching, [1] p.1 14
Figure 4: Hood-structure for UVF Imaging in the field at daylight, (2017) [6] p.2... 16
Figure 5: Mobile UVF system using a monopod, covering about 5 modules at a time (2019) [1], p.2 17
Figure 6: Working scheme of a perceptron - a single neuron [9]........ 18
Figure 7: The prediction of a fully connected layer with 3 input and 3 output values; the drawing based on the illustrations in [7] 19
Figure 8: Illustration of the gradient descent algorithm at the example of one input and one output value; illustration adapted from [7]........ 20
Figure 9: Illustration of the learning workflow: forward propagation, error at the output layer and backpropagation; illustration inspired by the figures in [7] 21
Figure 10: Idealized graphs of performance (error) on training and validation data showing overfitting; on the horizontal axis are the epochs (= nr of iterations), [10]........ 21
Figure 11: The convolutional operation using filtering kernels to create the output feature maps; [11] (left) [12] (right) 22
Figure 12: Feature extraction in a CNN at the example of processing dog images Source: https://www.symmetrymagazine.org/sites/default/files/images/standard/neural_network_visual_final.jpg........ 23
Figure 13: Architecture of a CNN: feature extraction and classifier at the example of the MNIST-dataset, [13]........ 23
Figure 14: Processing steps of the tool OpenUVF including cell and module segmentation, perspective correction and a TensorFlow model applied to detect cracks on each cell, [1] p.4 24
Figure 15: UVF ring pattern (left) and square pattern (right) portrayed by Köntges et al. 2019 [5] p.5........ 26
Figure 16: Plan of procedures as Gantt-chart........ 27
Figure 17: The fotovoltaica/UFSC Solar Energy Research Laboratory, source: Team of the fotovoltaica laboratory........ 28
Figure 18: The ENGIE PV power plant "Cidade Azul" in Tubarão, with multi crystalline (mc-Si), CIGS and amorphous microcrystalline PV power blocks [15] 29
Figure 19: First setup used in this project........ 30
Figure 20: UVF image acquired on 25/04/2022 at Tubarão with the first setup, about 4 cells are well-illuminated........ 31
Figure 21: Second setup composed of three 18 UV-LED flashlights and two 51 UV-LED flashlights Sources: Image bottom left: Amazon, product image of the seller ‘NVTED’, accessed 27/07/2022; Image top left:

Amazon, product image of the seller 'corion', accessed 14/07/2022 31
Figure 22: Whole-module UVF image taken with the third setup at the Fotovoltaica/UFSC PV laboratory, Florianópolis, 20/06/2022 32
Figure 23: UVF patterns observed at the test field and power plant in Tubarão 35
Figure 24: Encapsulant defects observed on the UVF square pattern: (from left to right). Top: busbar corrosion and lateral leakage, cracked front glass, cracks across the cell. Bottom: single circular spots of photobleaching, sometimes merging along a line (crack)............... 36
Figure 25: Overview on the 3 main steps (scripts) of this project: perspective correction on the module image, cropping cell images and classifying cell images with a CNN 37
Figure 26: Brightness correction: original image (top left), gamma=1.5 (top right), gamma=2.5 (bottom left) and both gamma=2.5 and colour histogram equalization (bottom right)........ 39
Figure 27: Blue, green and red colour channel of the brightness-corrected image 40
Figure 28: Inverted image (pre-processed green colour channel)........ 42
Figure 29: Simple thresholding (thresh=105) on the inverted green colour channel 43
Figure 30: Contour detection on the 'raw' thresholded image: 34700 (rounded) contours were detected 44
Figure 31: The influence of the tolerance epsilon at contour approximation, [18].. 45
Figure 32: Loop of morphological operations and contour detection, as implemented in this project 46
Figure 33: The simplified binary image produced by the loop of morphological operations after 5 iterations; 231 external contours are detected (contour limit set to 240 here) 47
Figure 34: When filtering for 4-cornered contour approximations (criterion 7.A), 108 of 231 contour approximations remain........ 48
Figure 35: The module contour, filtered out by applying all the contour criteria..... 49
Figure 36: Original image, module contour and perspective corrected module image 50
Figure 37: Overview on the cell cropping pipeline (the second script) 51
Figure 38: Schematic overview on the folder structure of the cell image data bank 55
Figure 39: Three ways to enlarge image datasets [20] 56
Figure 40: Overview on possible operations for image data augmentation, [20] ... 56
Figure 41: Cell image data base of exemplarily 35 intact cells (on the left) and 35 defect cells (on the right) 61
Figure 42: Graphs of the performance metrics (correct classification rate and loss function value) during the training on cell image classification 62

List of Tables

Table 1: Ambient and Constructive Factors influencing the formation of UVF....... 13
Table 2: Degradation mechanisms and interaction pathways affecting the encapsulant and UVF........ 15
Table 3: Physical/Chemical Parameters and measuring techniques characterizing the encapsulant........ 15
Table 4: Comparison of experimental setups from literature/this project............... 33

List of Abbreviations

PV	Photovoltaic(s), Photovoltaic Energy
IR	Infrared, Infrared Thermography
EL	Electroluminescence
PL	Photoluminescence
UV	Ultraviolet (light)
UVF	Ultraviolet Fluorescence
DIP	Digital Image Processing
OTR	Oxygen Transmission Rate
EVA	Ethylene-Vinyl Acetate, encapsulant material
LED	Light Emitting Diode
CIGS	Copper Indium Gallium Diselenide (type of PV module)
AI	Artificial Intelligence
OpenUVF	“Open Ultraviolet Fluorescence”, open-source inspection tool
API	Application Programming Interface
NN	Neural Network
CNN	Convolutional Neural Network
UFSC	Universidade Federal de Santa Catarina
MNIST	Modified National Institute of Standards and Technology (database)

1 Background

Efficient and reliable onsite inspection methods are gaining importance as the construction of PV power plants is expanding. For large PV installations, time- and cost-efficient failure detection is essential for optimized operation and maintenance. For this purpose, various optical methods as Infrared thermography (IR), Electroluminescence (EL), Photoluminescence (PL) and Ultraviolet Fluorescence (UVF) are employed and under constant development. For each method, the camera, and eventually the light source, can be handheld, or mounted on a drone, also called unmanned aircraft vehicle (UAV), to achieve higher throughputs.

IR is the most widely used optical onsite PV inspection method, as many defects can be detected by the thermal radiation (heating) of the defect component. EL and PL reveal further information on the electrical behaviour of the Si-waver. They are also widely used and take the role of a complement to IR, showing electrically active/inactive areas of the semiconductor. On the other hand, UVF focuses on the degradation of the polymeric encapsulant of the Si-cell, most commonly consisting of EVA (ethylene-vinyl acetate). The degradation of the encapsulant can lead to its discoloration, also called yellowing/browning, which decreases the transmittance of visual light. UVF patterns can show this yellowing as well as humidity and oxygen entrances, which can lead to effects of corrosion. Both mechanisms (discoloration and corrosion) decrease the performance of the PV cell. The discoloration cannot be directly observed on IR or EL images, as the encapsulant is neither a heat source nor electroconductive. Using IR imagery, severe discoloration might be observed indirectly, as the reduced optical transmittance leads to changes in heat transfer mechanisms concerning the cell and the encapsulant.

Similarly, as long as corrosion does not lead to inactive cell areas or heating, it most likely will not be spotted using EL, PL or IR. So, UVF can fill the niche of inspecting the state of the encapsulant and detecting its defects due to climate impacts in early stages.

While a high number of studies on IR, EL, PL and some on UVF were performed in Europe and the USA, there are not yet many studies about the application of these techniques in South America (i.e., in Brazil). UVF mainly depends on climate factors (irradiation, temperature, humidity) and the operation time/"age" of the module. The UVF imagery method has not yet been tested in climate and system conditions of Brazil. Furthermore, systems in Brazil are more recently installed. All this can affect differences in the results of UVF imagery applied in Europe, the USA and Brazil.

The present work focuses on the application of UVF imaging on PV power plants in Brazil, the creation of an experimental setup and the proposal of proceedings for the data analysis of the acquired images. The aim is to propose a method that is suitable for large-scale inspection.

2 Scope and aims of this project

The present project focuses on the application of UVF as an onsite inspection method for PV modules in Brazil.

- **Development of an experimental setup**

Within the course of this project, UVF images are captured using a mobile setup in the field. Efforts to optimize the experimental setup are made whenever possible. Alternatively, options and room for improvement are indicated for future studies.

The experimental setup will be evaluated and, if possible, compared to setups used by other researchers. Adequate criteria for this could be the quality of the images, the costs and the throughput of the method in modules/cells investigated per time unit.

- **Onsite acquisition of UVF images**

UVF images of polycrystalline PV modules in the field are taken, trying to ensure reproducibility and the best conditions that are practically possible. Images of intact as well as defect modules are taken, as well as images from different strings and different PV sites.

- **Evaluation, interpretation and comparison of UVF images**

The acquired UVF images are evaluated regarding the UVF pattern. Eventual pre-processing steps such as image editing using Digital Image Processing, are applied, to test whether they facilitate the evaluation and increase the visibility of UVF patterns and possible defects. If possible, the UVF patterns are matched to UVF patterns known from literature. Common phenomena appearing in several PV cells are named. Any arising assumptions for the causes of the UVF patterns (such as cracks) are noted. A brief comparison between UVF images from experiments and from literature are made to outline which phenomena are case-sensitive and which are not. Arising assumptions about the reasons for this case-sensitivity are noted as well.

- **Development/Application of an appropriate computational analysis**

To foster the automatization and to improve cost-efficiency, the acquired images need to be processed digitally with the aim to facilitate the defect detection. This processing can consist of Digital Image Processing (DIP) using transformations and filters available in image processing software. Available possibilities for further automatized analysis using techniques from the area of Artificial Intelligence (AI) are indicated and given the possibility, applied and tested. The adaption or further development, as far as allowed, of present open-source software such as OpenUVF, is tried as far as the resources of this project (workhours, available skills in Computer Science) allow it.

The in-depth application, testing and optimization of these methods is not the emphasis of the present work, as it lies beyond the possibilities of this project, but, if possible, can be added as a supplement.

3 Central Hypothesis and Questions

The central hypothesis of this work is that UVF is an efficient and cost-effective method for PV module inspection even in different climates. This hypothesis is based on the premiss that the ageing process of the PV cell by UV degradation of the encapsulant is based on the same general mechanisms and therefore comparable between different climates.

To assess this generalization of applicability, the question of case-sensitivity needs to be considered (qualitatively). Due to climate factors influencing the formation of fluorophores, the UVF patterns appearing in Brazil could differ significantly from those of other climates. This raises the question if and how a generalized tool for UVF evaluation can be developed and which main functions it needs to offer for a future entrance into the market of PV assessment tools/services. So far, the main functions are assumed to be detection of PV modules and cells, perspective correction, segmentation of PV cells and the yes/no decision (binary classification) of cracks, compare [1].

Regarding the experimental setup as well as the data evaluation process, a main question is how compromises between complexity/simplicity and generalization/specialization can be achieved.

Finally, the question of usability, applicability, time- and cost-efficiency is important, especially regarding large-scale application and a market entrance in the near future.

4 Theory & State of the Art

Before using UVF as basis for an inspection technique, it is essential to know how it is formed. By presenting the state of the art on how it influences power yield, the importance of UVF as an inspection technique can be justified. Finally, the most common UVF patterns need to be known for the design of a UVF image analysis tool.

4.1 Formation of UVF by exposure to sunlight

The phenomenon of UVF appears in modules that have been exposed to sunlight and the formation of UVF is assumed to be a direct consequence of the UV sunlight irradiation [2]. Incident UV-light degrades the EVA and its additives, the formed degradation products can fluoresce and are therefore visible on UVF images.

Elevated temperatures of the module and therefore of the encapsulant are shown to be a factor which accelerates the formation of UVF [2]. The study by Morlier et al. [2] proposes a saturation-growth modelling of the UVF formation. Within a time interval after the start and before the saturation phase, the UVF-intensity is quadratically correlated to the UV-dose [2], see Figure 1.

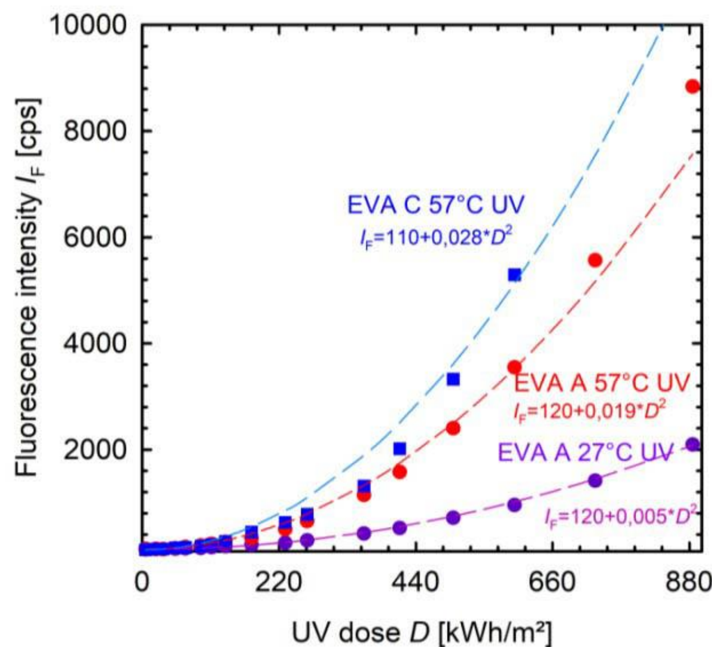


Figure 1: UVF intensity vs. UV dose for different encapsulants (EVA) and temperatures [2] p.5

This degradation proceeds via various degradation mechanisms. Some of these mechanisms and their interaction are not yet well-known and subject to current and future studies.

The creation of UVF often correlates to a ‘yellowing’ or ‘browning’ effect, which is visible for the human eye in later stages [3]. The terms ‘discoloration of the encapsulant’, ‘transparency loss’, ‘transmittance loss’ as well as ‘yellowing’ and ‘browning’ generally refer to

this same phenomenon and are widely used as synonyms. The effect can be quantified as the transmittance loss along the wave spectrum [2], and by the change in yellowing index, a measure for the colour change [3].

The transmittance measurements done by Morlier et al. [2] indicate that the intensity of UV exposure is correlated to the extent of transmittance loss. So, the longer an encapsulant has been exposed to sunlight, the more severe can be the loss its of optical transmission [2].

The study [2] also showed that the current density loss (in percentage) increases linearly with the UVF intensity [2]. And as the UVF intensity is related to the UV dose, a direct correlation between the current density loss and the UV dose can be drawn [2], see Figure 2. These correlations can be used to quantify and measure the ageing and power loss of PV modules by UVF imagery or spectroscopy measurements.

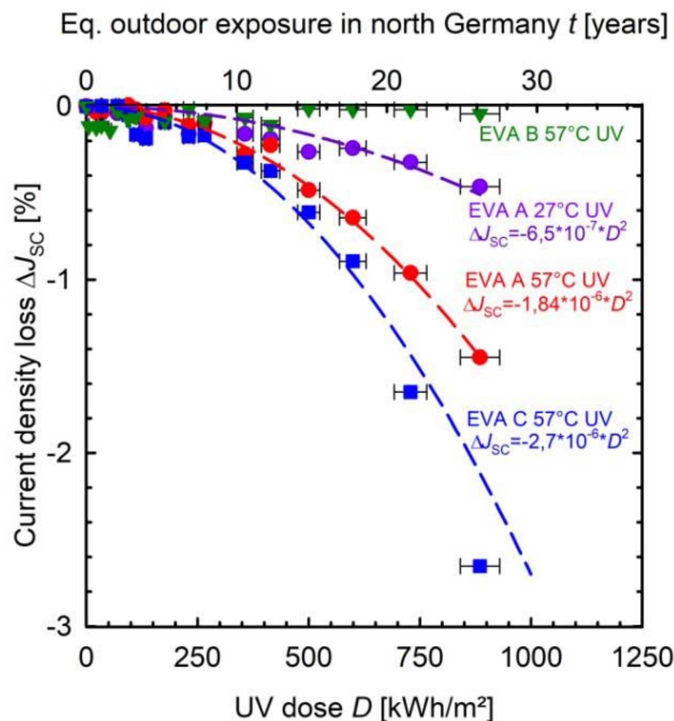


Figure 2: Current density loss correlated to UV dose and equivalent years of operation in Northern Germany, [2] p.6

To obtain a generalized model, the local UV dose of each module/cell can be proposed as an independent variable, and the current density loss as dependent variable. The UV dose is the irradiance integrated over time since the installation and is specific to a module/cell and the climate in which it operates. So, the conversion between the two abscissas in Figure 2 is particular to the climate in which a PV cell operates [2].

Morlier et al. [2] concluded that in Northern Germany, losses due to the discoloration of the encapsulant and ignoring all other degradation effects, will stay below 2% for the first 20 years of operation [2]. However, as the yearly irradiation in of South America is generally notably higher, a higher impact on current and power loss can be expected. It

should also be noted that the power yield on the DC side of a PV power plant is quadratically depending on the current. So, the power loss is proportional to the square of the current loss:

$$P_{DC} = U_{DC} * I_{DC} = R * I_{DC}^2$$

To conclude, further studies of the impact of EVA-degradation on power loss need to be done, especially within different climate zones.

4.2 Degradation mechanisms and their interdependence

Apart from the pathway of transmittance loss, incident UV-light can create various degradation mechanisms, which can interact physically (i.e., by transmission/absorption of UV light and heat sources/sinks) and chemically (by the formation of degradation products, by-products. Alongside the UV irradiation, climate factors and the composition of the module determine when and in which pattern UVF appears, as seen in Table 1.

Table 1: Ambient and Constructive Factors influencing the formation of UVF

Ambient Factors (climate and surrounding)	**Constructive Factors** (concerning the module/cell composition)
• UV-dose (kWh/m²) = UV-irradiation [2] [4] = UV irradiance history integrated over time • UV irradiance-history* (kW/m² for each min) • ambient and cell/module temperature (°C) [2] [4] • air humidity (g water/m³ dry air) [5] [4] • climate (history and combinations in which the above factors occur) [4] • operating time (in years) [2] • albedo-value* • extreme weather events* (e.g. hailstorms, earthquakes) *Factors inferred from the context, not yet proven by experimental studies	• front glass (transparency in UV range, earlier: cerium-containing glasses) [5] • front encapsulation [5] • encapsulant type and materials used between solar cells [5] • UV-pass or UV-cut encapsulants [3] or whether a UV-absorber is used [5] • cell interconnect ribbon [5] • lamination material type and material combinations [5] • combination of applied additives, e.g. oxidation stabilizers, UV absorbers, and the crosslinker [5] • back sheet type / bifacial module [5] • physical impacts (e.g. wrong module handling/montage, mechanical stresses)

Recent studies focused on UVF patterns, fluorescence of different cell parts [5] or focused on the occurrence and interaction of chemical processes related to UVF occurrence [4].

Yellowing/Discoloration probably is the effect which has been investigated most thoroughly and for most years already. Two main reasons for this probably are its visibility to the human eye in later stages and the direct way in which it impacts module power via the transmittance loss [1], [2], [4], [5]. Apart from yellowing, various other climate-induced degradation mechanisms can occur and interact [5]. Each degradation mechanism can affect the formation and pattern of UVF.

Simultaneously to yellowing, another effect called **photobleaching** determines the UVF pattern. Photobleaching designates that oxygen entry causes a local extinction of the UVF light [5]. Likewise at these oxygen entry spots, fluorophores are still present and human-visible yellowing can still occur in later stages, but their UVF light signal isn't visible neither for the camera nor for the human eye. As oxygen enters predominantly from cell borders and through cracks, a characteristic UVF pattern with photobleaching along borders and cracks can be observed [1] [5], see Figure 3.

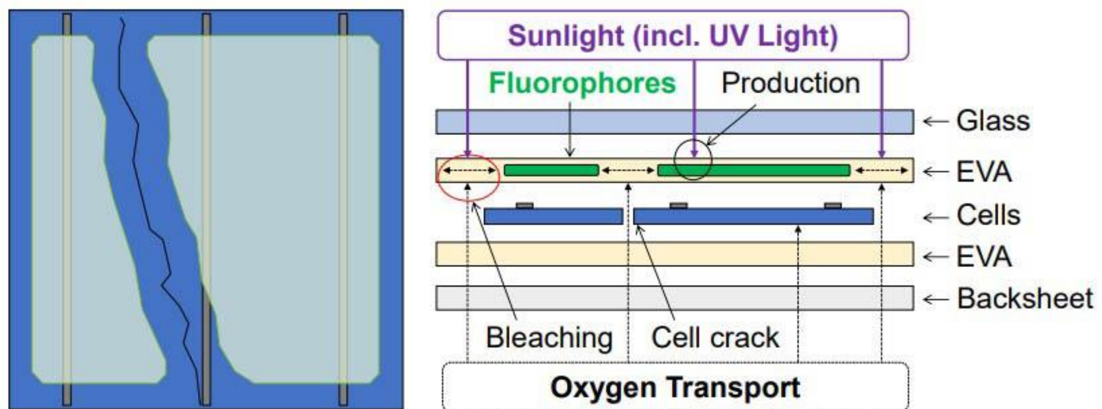


Figure 3: UVF formation and -extinguishing by photobleaching, [1] p.1

Köntges et al. explains in detail under which conditions individual cell parts show UVF patterns and the respective causes [5].

Apart from yellowing/encapsulant discoloration and photobleaching, various other degradation mechanisms can occur and interact, induced by exposure to the climate conditions and detectible or related to patterns in UVF images, see Table 2.

Table 2: Degradation mechanisms and interaction pathways affecting the encapsulant and UVF

Degradation mechanisms	**Interaction pathways**
• Encapsulant discoloration (yellowing, transmittance loss) [2] • Cracks, leakages • Photobleaching [5] • Metallization degradation (corrosion) - of the silver gridline surface by silver oxide formation [4] - by acetic acid formation [3] • Solder bond degradation [4] • Delamination [3]	• transmission/absorption of UV light • heat sources/sinks (i.e. hot spots) • oxygen transmission rate (OTR), local presence/absence of oxygen • formation & diffusion of substances such as UV-absorber, degradation products, acids etc. [5] [4]

For research on these degradation mechanisms and their interaction, usually a combination of inspection methods and/or time series of data are used. Inspection methods measure physical or chemical parameters to observe and quantify the occurrence of UVF and discoloration/transmittance loss. Table 3 shows which parameter is measured by which measuring technique.

Table 3: Physical/Chemical Parameters and measuring techniques characterizing the encapsulant

Physical / Chemical parameter	**Inspection/Measuring technique**
presence, concentration and spatial distribution of fluorophores (degradation products)	UVF imaging
absorption, reflection, transmittance of PV cell components, especially encapsulant and back sheet	UVF spectroscopy
I_SC, U_OC, I-U curve, FF loss	I-U curve measurements, monitoring
colour change (reflected spectrum)	YI (yellowing index measurements) [3]
chemical composition analysis	X-ray photo-electron spectroscopy (XPS) [4] Scanning electron microscopy (SEM) [4]
parameters related to ageing	experiments on field-retrieved modules, accelerated stress testing in UV/heat chambers

EL, PL and IR are not listed in order to focus on inspection techniques that regard the encapsulant properties rather than Si-wafer properties. Naturally, I-V curve measurements do not measure encapsulant properties directly. However, indirect conclusions

from I-V curves on encapsulant degradation can be drawn, assuming unaltered behaviour of the Si-wafer itself. So, Table 3 supplies a list of techniques that can be used for comparison or confirmation of each other.

4.3 Present experimental setups for onsite inspection

To prevent sunlight from superimposing the UVF signal, and to have exclusively UV-excitation (not with the whole sunlight spectrum), UVF measurements are commonly performed at night using UV lights as excitation source. To shoot EL and UVF images at daylight, Morlier et al. developed a hood-structure that can be placed on one module to block the sunlight [6], see Figure 4.

Figure 4: Hood-structure for UVF Imaging in the field at daylight, (2017) [6] p.2

With this method, up to 200 images per hour and one module per image can be taken [6]. When working at night and aiming at images of at least one module at a time, tripods or monopods can be applied to mount the camera and eventual UV lights [1]. Gilleland et. al. developed a setup using a 5m high monopod, see Figure 5.

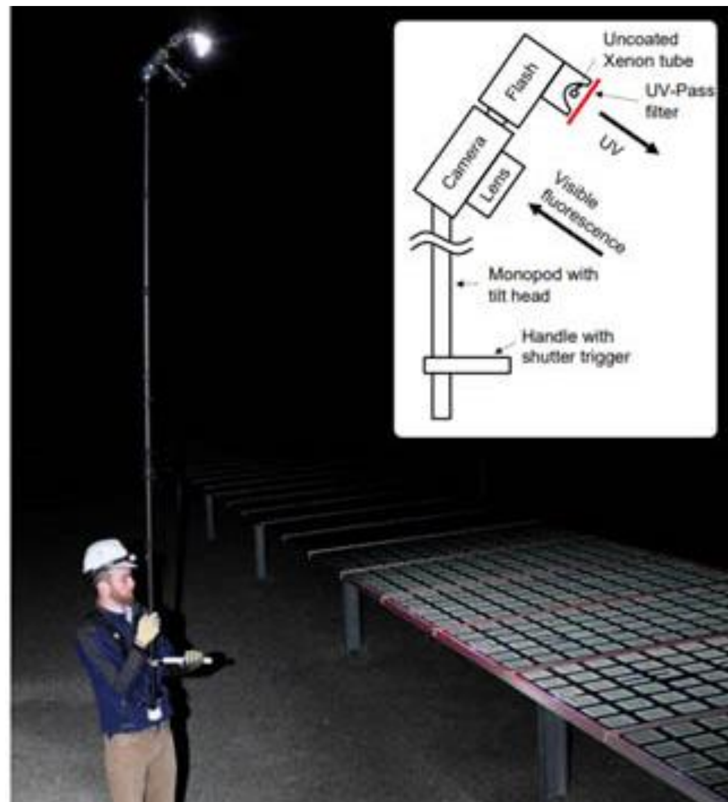


Figure 5: Mobile UVF system using a monopod, covering about 5 modules at a time (2019) [1], p.2

According to the developers, this setup allows throughputs of 1000 modules per hour, or up to 2000 modules/hour in case the modules are mounted in two rows [1].

Similar to IR inspections, a drone-based setup could also be used for UVF imagery. When creating a drone setup for UVF inspection, several criteria/goals need to be fulfilled at the same time, such as a low weight of the drone and its flight stability, enough battery power for UV lights, a low noise camera and for the drone itself, compare [5] p.15. Even when all these criteria are fulfilled, a legal permission for drone flights at night is needed, which can raise problems regarding the local laws or the assurance of the PV plant or the drone. Another compromise is the flight height, either covering a lot of modules at the same time with low resolution, or few modules with high resolution.

After presenting the experimental setups used in this project, a summarizing table comparing all setups is given in 5.2.6 (Table 4).

4.4 Basic concepts of Deep Learning for Image Classification

This section introduces fundamental concepts and principles used in Deep Learning with the aim of image classification. Image classification is often used to tag experimentally acquired images with the main information, here: the presence/absence of defects. Note, that there are other techniques that can be used as alternative approaches, such as

image segmentation or object detection. However, here only image classification is discussed and applied.

4.4.1 Concepts of Neural Networks

Neural Networks can be seen as function which can learn a correlation between the provided input and output data [7]. As the name states, they are inspired by human neurons and consist of several neurons as processing elements connected to each other building a network [7] [8]. The network consists of several layers with different layer types and connection modes for different tasks [8].

The general design of a single neuron is shown in Figure 6 and works as follows [8] [7]: Input signals (numeric values) from several (or all) neurons of the previous layers reach the input node of the neuron. A weighted sum (scalar product) of the input vector and the weights is computed. One weight is assigned to each connection. The resulting value is passed to an activation function, e.g. the Heaviside (unit step) activation function or tanh(). Commonly, a rectified linear unit activation, short 'relu()' is applied. This affects that the weighted sum needs to surmount a certain threshold value to create an output. The activation function also scales the output, so that the values remain in a certain predefined range, e.g. between 0 and 1 [7].

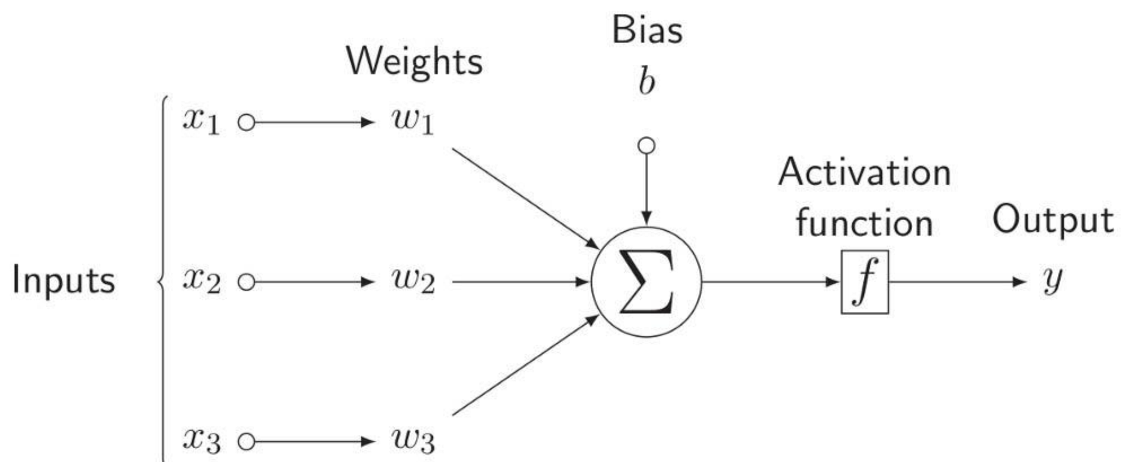


Figure 6: Working scheme of a perceptron - a single neuron [9]

To summarize, a single neuron executes a weighted sum (scalar product) at the input and applies an activation function that includes both thresholding and scaling the output value [7].

Likewise, the creation of a prediction based on input and weight(s) values can be expressed by the following formula [7]:

$$(prediction): output = input * weight \ (+bias)$$

The bias is a different formulation for the thresholding and activation and represents the increase by the activation function when the thresh is surmounted [7].

Within a network of several neurons in each layer, the scalar operations shown above become matrix operations, see Figure 7:

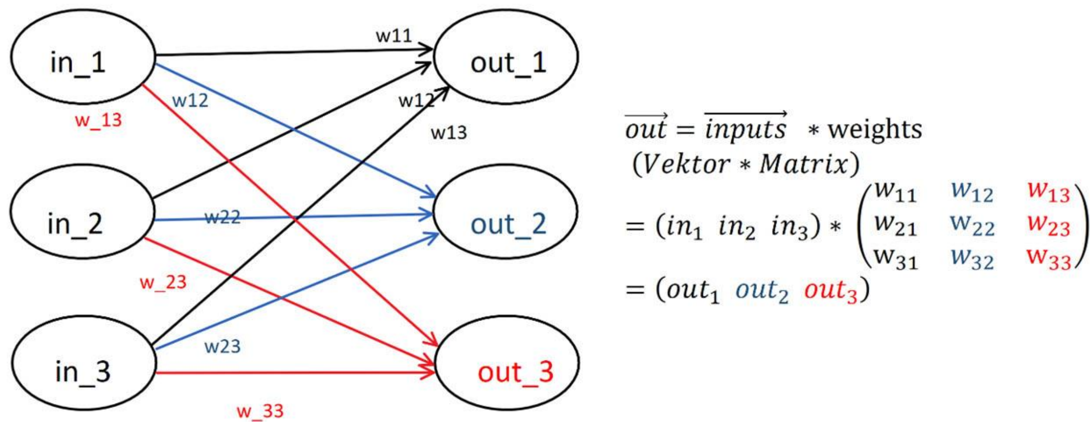


Figure 7: The prediction of a fully connected layer with 3 input and 3 output values; the drawing based on the illustrations in [7]

Layers in which all the neurons are connected to each other (as shown above) are called fully connected layers. For simplicity, the above scheme only portrays the multiplication of the input vector with the weight matrix; the second step of thresholding is not shown.

Given a set of weights, such fully connected layers can create a vector of outputs for a given input vector (make predictions) [7]. In supervised learning, pairs of input and true output (target) data samples are provided to the network and the aim is to learn the right weight (and eventually the right threshold) values [8]. With the information or 'knowledge' stored in the weights, a Neural Network can model the correlation between input and output [8].

To adapt the weights, the gradient descent algorithm can be applied: after the prediction, the obtained output value is compared to the target value, and the absolute difference 'delta' and the error value are computed [7], [8]. There are different error functions, also called loss functions. For simplicity, the squared error is taken here [7]:

$$(absolute\ difference):\quad delta = output - target$$

$$(squared\ error):\quad error = (prediction - target)^2$$

In case of several input and output values for each neuron, these functions have several dimensions. The gradient descent algorithm computes the derivative (1D) or gradient (in several dimensions) of the error function at the current weight position [7], [8]. This gradient is subtracted from the current weight value, which shifts the weight to a position where it yields a lower error [7]. Applying this algorithm several times, the weight is shifted stepwise to better positions, yielding lower error values [7]. Finally, the weight either oscillates around or reaches the optimum position. At the optimum position it would yield the global minimum of the error function, ideally a zero error [7].

The Figure 8 illustrates the gradient descent algorithm using the one-dimensional example of one input and one output value [7]. In the shown case, the current weight is too small and the derivative of the error function (yellow slope) is negative. A value 'change' is computed, based on the derivative, with scaling and applying a learning rate (step size) alpha [7]. All these operations do not alter the sign and only scale the value of the derivative. This value is renamed to 'change' and as the derivative, negative. When subtracting this negative value from the current weight value in step 6., the weight is increased (two minus signs). As desired, this increase shifts the weight closer to the optimal weight value [7].

In the contrary situation that the current weight value is too large, the positive slope is subtracted, so that the weight value is reduced. In both cases, the weight value is shifted, towards the optimal weight position [7].

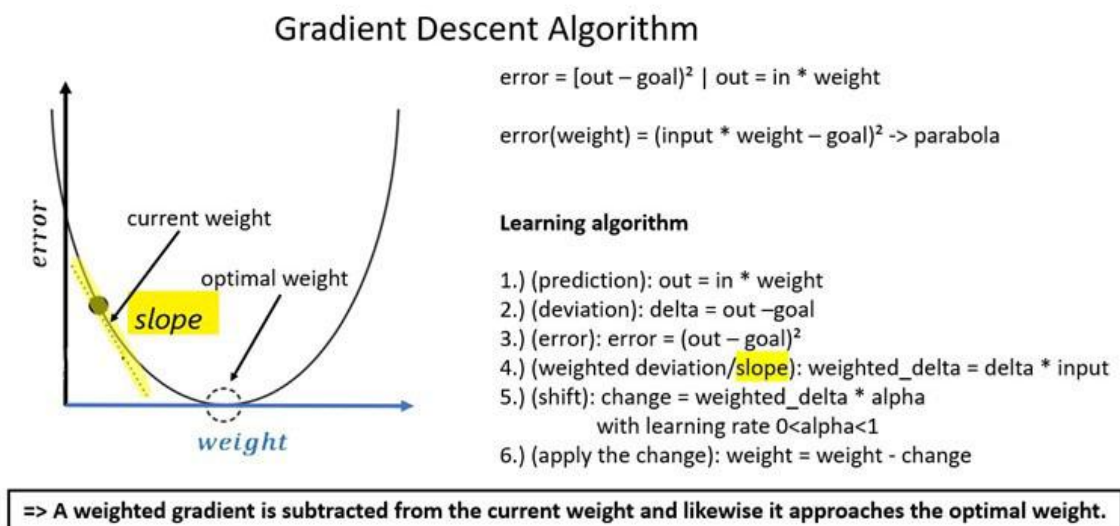


Figure 8: Illustration of the gradient descent algorithm at the example of one input and one output value; illustration adapted from [7]

The gradient descent algorithm corrects one weight value at a time and needs the respective error value and error function (to build the gradient).

There are different variations of the gradient descent algorithm e.g., applying different error functions, inertia when moving the weight or adapting the step size, for more details, consult [7] and [8].

Networks can generally be called 'deep' when they have (far) more than one layer. When training Deep Neural Networks on a databank of input and output data, data samples or batches are processed one after another.

For each sample, the full learning process is executed: first a prediction is made by multiplying the input vector stepwise with the weight matrices of each layer until reaching the output layer. This process of passing the values through the network is called forward propagation, see Figure 9, [7]. At the output layer, the error values are computed, by

handing the output and target (or goal) values to the error function. This yields one error value for each neuron at the output layer. However, one error value for each weight is needed, in the earlier layers as well [7]. For that, the error signal is traced back to previous layers by multiplying it with the transposed weight matrix (weights.T), see Figure 9, [7]. This process is called backpropagation.

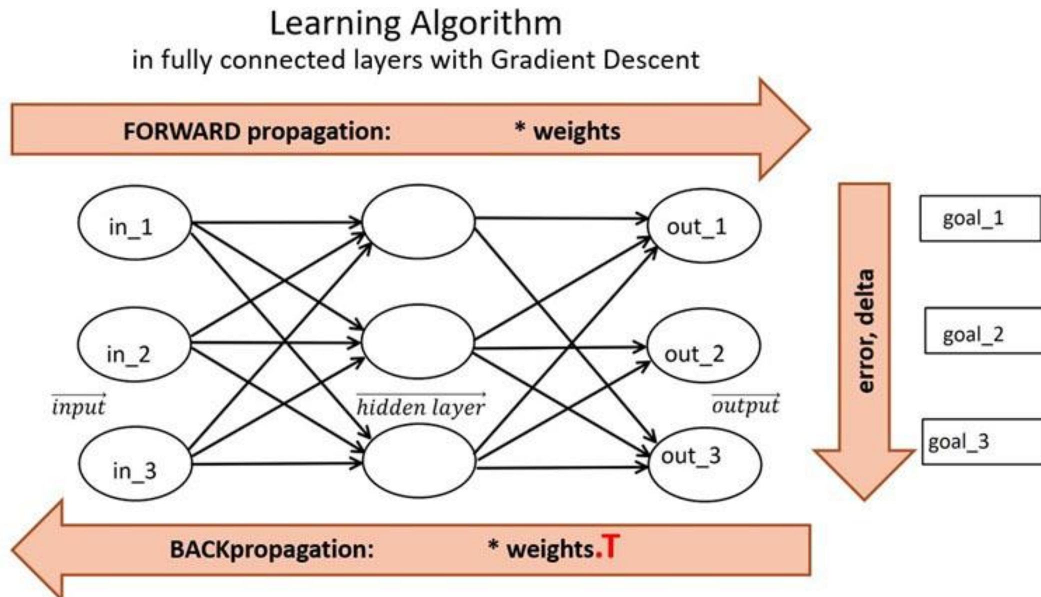


Figure 9: Illustration of the learning workflow: forward propagation, error at the output layer and backpropagation; illustration inspired by the figures in [7]

By executing forward propagation, building the error and backward propagation, one learning step (one correction of each weight) is performed, based on the current batch of data samples [7]. Passing all the data samples of the training dataset in batches to the network is called one training epoch. To verify the performance on unknown data, the network is used to make predictions on the validation data in each epoch. Both the performance metrics on training and validation data are stored in arrays and monitored in the console and via plots. The plots should be similar to the one in Figure 10.

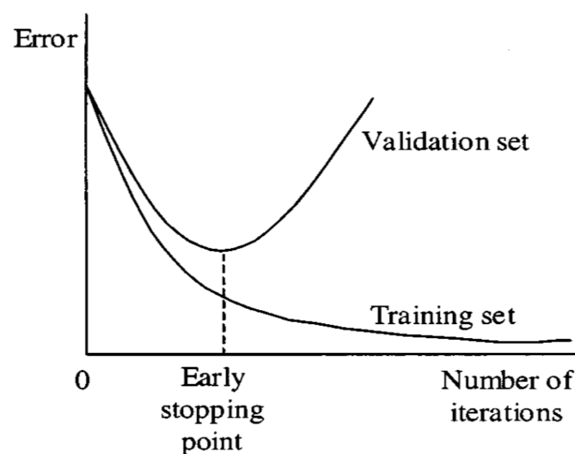


Figure 10: Idealized graphs of performance (error) on training and validation data showing overfitting; on the horizontal axis are the epochs (= nr of iterations), [10]

Overfitting of the Neural Network model occurs, when the performance on validation data stagnates or becomes worse while the performance on training data [7], [8]. From that epoch onwards, the model learns characteristic details of the training dataset, and stops to learn general features [7]. This behaviour can also be considered as ‘learning by heart’ instead of generalizing [7].

Optimizing the learning process has the aims to delay overfitting and to stop the training one epoch before overfitting occurs [7], [8]. Nagidi, [10] and the literature on Deep Learning, e.g. [8] and [7], provide more detailed information on overfitting and methods to delay/prevent it.

4.4.2 Convolutional Neural Networks for Image classification

To process and learn on image data, Neural Networks need special layers and operations: convolutional layers and MaxPooling layers [8]. A convolution is a matrix or tensor operation in which a kernel (a filtering matrix consisting of constants) is passed line by line over the input image [8]. At each position, a matrix-matrix multiplication is performed. The resulting value is stored at the respective position, at the centre of the kernel in a matrix containing the output values called output feature map [8], see Figure 11.

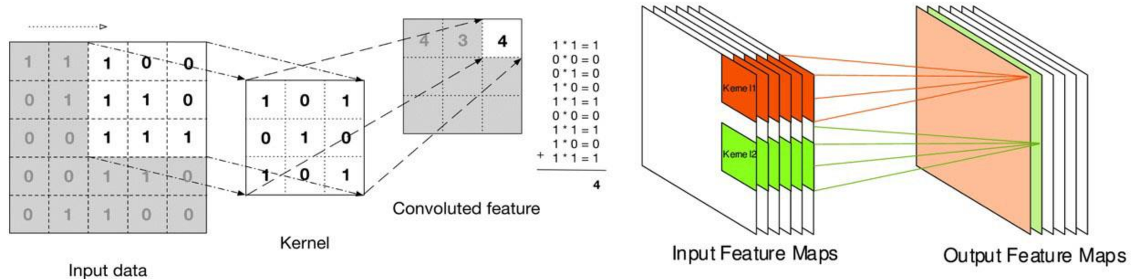


Figure 11: The convolutional operation using filtering kernels to create the output feature maps; [11] (left) [12] (right)

Variations of the convolutional operation can be done by using different kernel sizes, kernel values and shapes and different step widths between positions and lines. Different kernels (image filters) extract different features, for example textures and object borders [8]. Commonly, 32 or 64 filters are applied in one convolutional layer, creating a respective number of output layers [8]. As this generates a great amount of data, the so-called MaxPooling layers are used to filter out maximal values [8].

By stacking convolutional and Max-Pooling layers alternatingly, characteristic image features are extracted. In this feature extraction, the first layers filter small-scale features such as textures while layers deeper within the CNN can extract features that stretch over the whole image, see Figure 12.

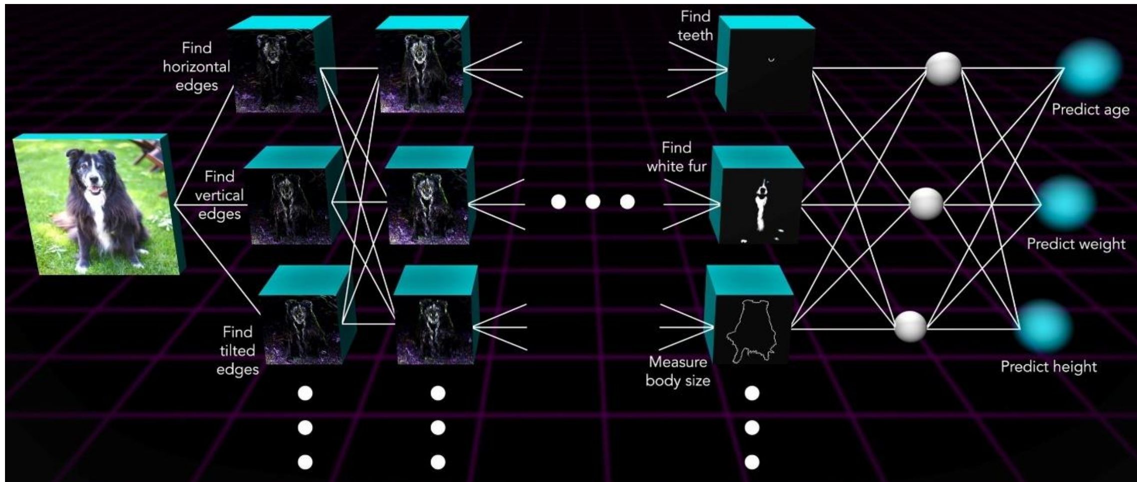


Figure 12: Feature extraction in a CNN at the example of processing dog images,
Source: https://www.symmetrymagazine.org/sites/default/files/images/standard/neural_network_visual_final.jpg

For image classification a block of convolutional and Max-Pooling layers (feature extraction) is used, at the end of which a few (commonly two) fully connected layers are added [8]. The final fully connected layers are called the classifier. The final layer needs to have as many neurons (output values) as there are classes. At the end a softmax-activation function can be used, which scales the output values to the range between 0 and 1, [8]. Each output value of a prediction can then be interpreted as the probability that the input image belongs to the respective class. The target vector is a one-hot coded vector, that means all array entries are equal to zero except for the one at the position of the right class, whose value is one [7], [8].

The Figure 13 shows an exemplary CNN architecture with the example of digit-classification with the MNIST-image dataset.

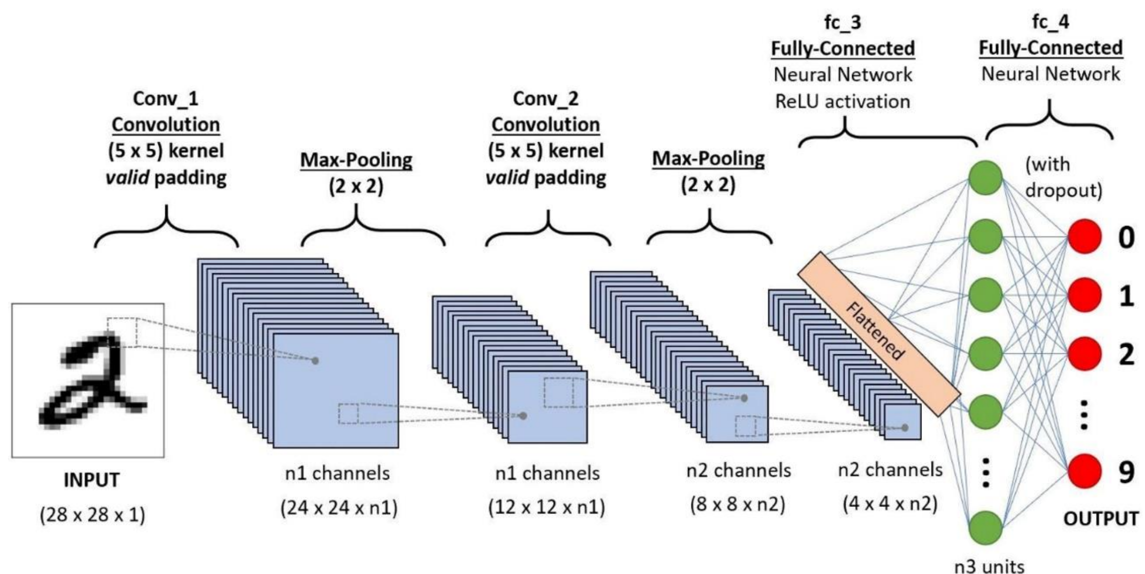


Figure 13: Architecture of a CNN: feature extraction and classifier at the example of the MNIST-dataset, [13]

The short form 'MNIST database' stands for "Modified National Institute of Standards and Technology database", named after the institute which created the dataset. The MNIST-dataset is a set of handwritten digits that is very commonly used to test image processing software.

4.5 Present Evaluation Tools

Single UVF images can be manually edited using common photographic image processing software (such as Photoshop) available as professional versions, free trial versions or open-source software. This method is suited for editing and evaluating small series of images e.g., for the analysis of some cells from laboratory experiments or the evaluation of a few modules. The operating person can then judge on presence/absence and type of encapsulant defects.

But in the case of large-scale PV power plants, a high throughput of modules is needed and the evaluation of their images needs to be automatized, to remain time- and cost-efficient. Some of the available tools that approach this problem are presented in the following subsections.

4.5.1 OpenUVF

Such an automatized (or semi-automatized) image processing and evaluation tool could be the open-source software OpenUVF, which was developed to cover several processing steps of a data evaluation pipeline [1] (see Figure 14).

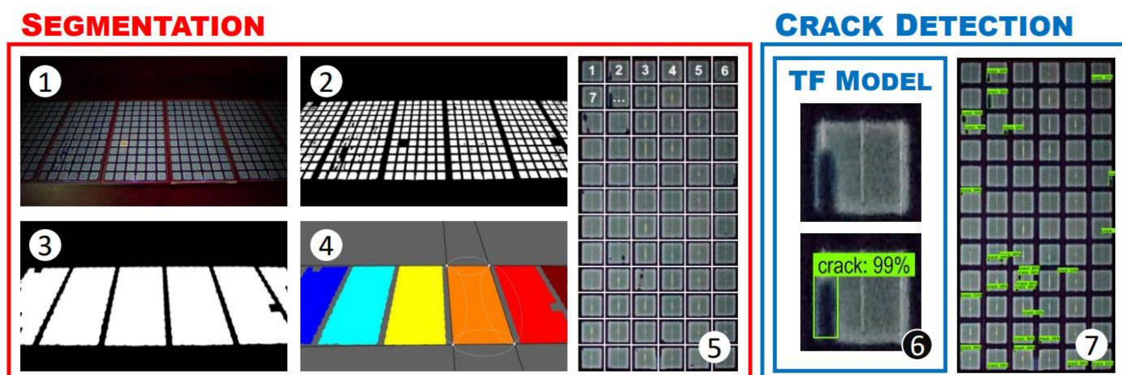


Figure 14: Processing steps of the tool OpenUVF including cell and module segmentation, perspective correction and a TensorFlow model applied to detect cracks on each cell, [1] p.4

OpenUVF uses Python and Matlab scripts, for the application of Deep Learning models and is still being developed [1]. The developers, Gilleland et al., intended to shift all processing steps to Python. The tool can be access on Github[1] and is thought as a base for each user's application [1]. The developers achieved an automatic crack-detection with

[1] https://github.com/southern-company-r-d/OpenUVF

an accuracy of 91.7% [1]. This detection accuracy refers to the binary decision of the Deep-Learning model whether a cell has a crack or not.

Usage in this project: Efforts were made to use and further develop this tool in the present project. However, large parts of the documentation were not finished by the developers, so that the information on how to use the codes needs to be inferred from the code functions and code comments, which resulted to be very time intensive.

The installation was very tricky and took around two days, mainly because the versions and repositories of software packages have changed since the publishment of the tool (2019). For example, the *tensorflow* version got upgraded from 1.x to 2.x, and some central commands like “Sessions” have changed between the versions.

The use of *tensorflow* allows a high potential in terms of functionality, but unfortunately complicates its usage for beginners in the domain of Deep Learning. The use of an API (application programming interface) like *keras* would simplify the coding and allow beginners to understand and develop on the code as well. Naturally, this is a question of the target group and proficiency of the tool’s users.

4.5.2 PV Vision

The tool “PV Vision”[2] was developed for processing of EL images and could potentially be adapted and applied for UVF images, too.

The tool offers scripts for various processing steps, but the installation of PV Vision raised problems, e.g. some docker commands were not working to setup the virtual environment and for unpacking the image. PV Vision uses the coding platform ‘Supervisely’ and problems at accessing and using this platform were encountered, too.

Usage in this project: As both the installation problems and the transfer from EL to UVF images would need a considerable workload, this tool has not been used in the current project. Nevertheless, for users who are familiar with docker and who process EL images, using PV Vision is probably a good approach.

There are more existing tools, and some are being developed at the time of this work. At the present, the aforementioned were thought to be the most relevant for this project.

[2] see project description: pv-vision · PyPI

4.6 UVF patterns from earlier studies

So far, only a few studies have been published on the correlation between climate, module composition or age and the resulting UVF pattern. The existing papers, namely [1], [4] and [5] show a high case-sensitivity.

Two commonly appearing patterns (both in literature and in this project) are the so-called square-pattern and ring-pattern, both showing photobleaching at points of moisture or oxygen entrance, see Figure 15.

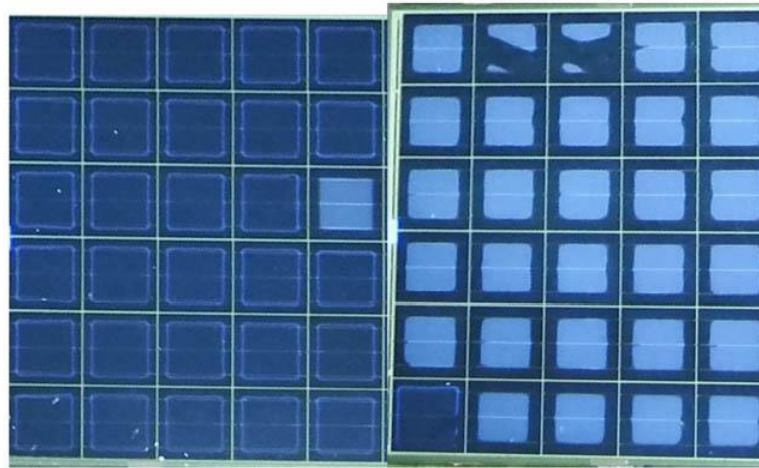


Figure 15: UVF ring pattern (left) and square pattern (right) portrayed by Köntges et al. 2019 [5] p.5

Lateral entrance of oxygen/humidity that diffuses through the slots between cells are assumed to be the reason for the observed square pattern [5]. For the rim (or ring) pattern Köntges et. al. propose the following mechanism: during the production of the module, a UVA absorber was added to the rear EVA to protect the back sheet from degradation, while the front EVA has not got this additive [5]. During the operation, oxygen and the EVA absorber diffuse across the slots between solar cells from the back to the front EVA and above the cell [5]. The superimposition of their effects on the UVF is assumed to cause the rim pattern [5].

Further descriptions of UVF patterns and UVF features as well as more observations on the respective encapsulant defects are presented in [1], [2], [4], [5] and [6].

5 Methodology

The methodology of this work closely follows the activities and goals mentioned in section 2 “Scope and aims of this project” and involves experimental testing as well as testing of software.

5.1 Plan of procedures

The proceeding takes place in two major stages, each consisting of an experimental and a subsequent analytical part. In the first stage, a “prototype”-setup is developed and applied. As disadvantages or room for improvement regarding the prototype are discovered in practice, these findings are considered for the development of the second experimental setup. The feasibility of this second experimental setup stands under the condition of low-costs and short-delivery times.

After the following image acquisition phase with the second setup, the image processing is developed, finalized and the thesis report is being written. All the phases can be seen in the Gantt-chart in Figure 16.

Figure 16: Plan of procedures as Gantt-chart

The cooperation partner, the *Laboratory of Photovoltaics of the Universidade Federal de Santa Catarina*, supports this project by giving access to laboratory equipment and providing the necessary devices (camera, tripod, UV lights and complementary equipment for experiments). The laboratory grants access to their own PV power plant test site as well as sites of chosen cooperation partners. In addition to this, services such as transportation to and from the power plants are provided by the laboratory.

The author wants to thank the UFSC laboratory of Photovoltaics for their generous support for this project. This research project is based on a recently signed (April-May/2022) cooperation agreement between the *UFSC - Universidade Federal de Santa Catarina* and the *HRW – Hochschule Ruhr West University of Applied Sciences*, the current university of the author.

Special thanks are to be directed to Prof. Ricardo Rüther and Aline Kirsten of the UFSC Photovoltaics Laboratory as well as to Prof. Marcus Rehm of the HRW Hochschule Ruhr West University of Applied Sciences for their contributing work on creating this cooperation and likewise enabling this project.

Thanks to all the supporters of this project for their generous help.

5.2 UVF Image acquisition in this project

This section first presents the sites where UVF images were taken. Then, the developed experimental setups are presented, discussed and compared.

5.2.1 Fotovoltaica/UFSC in Florianópolis

The Fotovoltaica/UFSC is the Solar Energy Research Laboratory of the Federal University of Santa Catarina (UFSC - Universidade Federal de Santa Catarina). The Fotovoltaica/UFSC laboratory performs research and development as well as demonstration projects in the sector of solar energy. A self-presentation and further information on projects are presented on their website[3] and YouTube channel[4].

Figure 17: The fotovoltaica/UFSC Solar Energy Research Laboratory, source: Team of the fotovoltaica/UFSC laboratory

The laboratory is located in Canasvieiras in the North of the island of Florianópolis (27°S, 48°W) [14], the capital city of the federal state Santa Catarina. Various testing setups are operated at the laboratory grounds. The test setups include polycrystalline, monocrystalline, thin-film and CIGS PV modules, the use of fixed structures and trackers. The solar monitoring station disposes of various pyranometers and experimental measuring techniques under development.

[3] https://fotovoltaica.ufsc.br/sistemas/fotov/en/

[4] https://www.youtube.com/channel/UCG7j_EffB_2teLxAomPA3fA

For this project, whole-module UVF images of 9 polycrystalline PV modules operating at the laboratory grounds were taken. The modules have been operating for about 2,5 years and are a part of a comparative study, see [14].

5.2.2 The PV and wind power plant in Tubarão

The first test site is the 3 MW PV power plant called "Usina Solar Cidade Azul", operated by ENGIE[5]. The PV power plant is located at the municipality Tubarão, in Santa Catarina, Southern Brazil (28° South, 49° West) [15]. The plant was built for the purpose of research and development in 2014 [15].

The plant counts/disposes of 20 thousand modules in total [15], installed on ground-mounted racks, directed towards the North [15].

The power plant consists of a wind turbine and three PV blocks of multi-crystalline (mc-Si), thin-film (CIGS) and thin-film amorphous-microcrystalline silicon (a-Si) modules, respectively [15], see Figure 18.

Figure 18: The ENGIE PV power plant "Cidade Azul" in Tubarão, with multi crystalline (mc-Si), CIGS and amorphous microcrystalline PV power blocks [15]

In this project, UVF images were acquired only on some of the 4,199 Yingli Solar 245Wp polycrystalline modules [15]. The other PV module types show no UV Fluorescence, because of their different composition/encapsulant.

[5] https://www.engie.com.br/institucional/sobre-a-engie/

5.2.3 First setup, applied in Tubarão

All UVF images in this project were taken at night, so that no cover against the light was needed. The experimental setups consist of an arrangement of UV-LED flashlights mounted on a tripod (eventually with a metal structure on top) and the handheld camera. Handholding the camera allowed quick acquisition of images from different angles and distances from the module. A right angle between the camera direction and the module surface would be ideal but could not be achieved due to the inclination of the modules.

The experimental setups of this project are presented and afterwards compared to experimental setups described in literature. The setups used here mainly vary in the number and arrangement of UV flashlights.

The first experimental setup consisted of a camera tripod with two UV-LED flashlights of 51 LEDs each and powered by 3 AA batteries.

Figure 19: First setup used in this project

Different digital cameras were tried, and in the end, a smartphone camera with HDR resolution was used, as it showed the best performance regarding the automatic focus and brightness adjustment, while being easy to handle in the darkness.

A UV filter, common in professional photography, was tested, but didn't prove any visible improvement so that it wasn't applied for image acquisition.

Evaluation

The first setup worked well for acquiring images of one to four cells at a time, mainly due to the small radius of the two flashlights' light beam. The illumination is very inhomogeneous, so that only these few cells in the centre of the image show UV Fluorescence. This caused problems at the image processing when trying to detect the cell contours.

This first setup is very cheap, only 10-20$ for the flashlights are needed in case a smartphone and a tripod are already present in the laboratory inventory.

Application

This first setup was developed and tested at the Photovoltaic laboratory of the UFSC and applied for UVF image acquisition at PV power plants in Tubarão, Santa Catarina, Brazil. 158 images of cells from 67 modules of the Tubarão power plant were taken at the Tubarão power plant. 177 images of cells from 38 modules were taken on another PV test field in Tubarão, which is part of a project involving 8 test fields in different climates in Brazil. Several images were needed for one module, because the small light beam of the flashlights did not allow to illuminate a whole module at a time.

Figure 20: UVF image acquired on 25/04/2022 at Tubarão with the first setup, about 4 cells are well-illuminated

5.2.4 Second setup

As the main problem of the first setup was the inhomogeneous illumination, more UVF flashlights were acquired. By the time of this work, the supply of UV-lights was a challenge. Especially large-area or high-powered UV-light sources are not or very rarely produced and sold in Southern America. To avoid long delivery times and custom issues, more UV flashlights were bought and used throughout this project.

The second setup consisted of 5 flashlights, with 156 UV LEDs in total, see Figure 21:

Figure 21: Second setup composed of three 18 UV-LED flashlights and two 51 UV-LED flashlights

Sources: Image bottom left: Amazon, product image of the seller 'NVTED', accessed 27/07/2022; Image top left: Amazon, product image of the seller 'corion', accessed 14/07/2022

The second setup was used to take some whole-module images at the PV laboratory of the UFSC in Florianópolis. But as exactly these images were taken again with the third setup with better illumination, the images of the third setup were finally used and the ones from the second setup were discarded.

5.2.5 Third Setup, applied at the Fotovoltaica/UFSC laboratory

The third setup consisted of 7 flashlights with 258 LEDs in total. The flashlights were distributed over the same metal tripod structure as before, and a professional camera was used, setting a longer aperture time. Some of the photos were taken by two persons, one lifting the UV lights' structure, the other person taking the photo.

As mentioned before, this setup allowed to acquire better whole-module images at Fotovoltaica/UFSC in Florianópolis (see Figure 22).

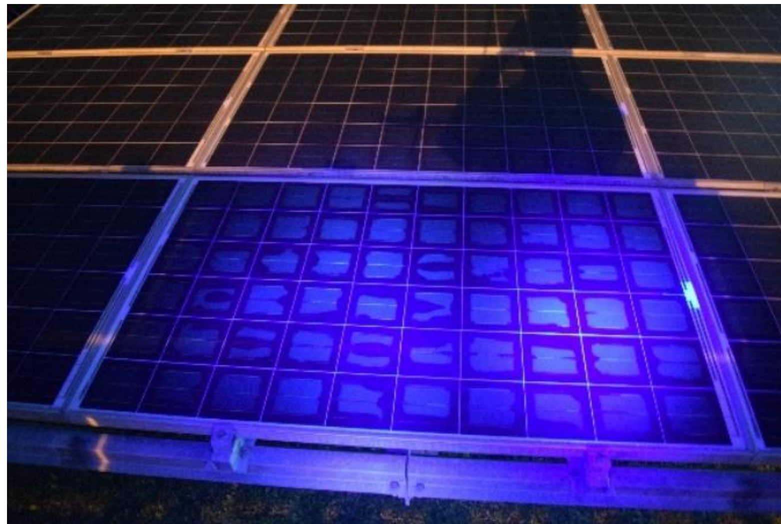


Figure 22: Whole-module UVF image taken with the third setup at the Fotovoltaica/UFSC PV laboratory, Florianópolis, 20/06/2022

5.2.6 Comparisons and Conclusions on setups

Homogeneity and power of the UV illumination turned out to be the most important factors when taking UVF images. These two factors generally allow a a better image acquisition with increasing size and price of the UV lights used. Table 4 compares the setups from the benchmark and the setups from this project, ranked by their scale.

Table 4: Comparison of experimental setups from literature/this project

Light source	**Price** (lights + structure)	**Illuminated area**	**Throughput** [modules/hour]	**Project**
2 flashlights, (=102 LEDs)	20-30 $	4-16 cells	about 45*	First setup (this project)
5 flashlights, (= 156 LEDs)	50-80 $	about 20 cells	about 45*	Second setup (this project)
7 flashlights, (=258 LEDs)	70-90 $	about one module, 30 - 45 cells well illuminated	about 45*	Third setup (this project)
hood structure + 2 LED arrays	50-250 $ (estimated)	1 module (60 cells)	up to 200	Setup of Morlier et al. for daylight inspection, [6], Northern Germany, 2017
monopod with photography flash	<3.500 $	up to 5 modules	1000 (one row) 2000 (two rows)	Setup of Gilleland et al., USA, 2019 [1], p.5
drone with UV-LEDs	3.000-6.000 $ (estimated)	12 – 16 modules, depending on flight height	Up to 720 (assuming 6 battery changes, flight time 8-10 min)	Drone setup of Köntges, Morlier et al. [5] p.15

* The throughput of 45 modules/hour with the setups of this project were calculated based on experimental trials trying to optimize image quality, not aiming at a high throughput, so they are not representative, as image acquisition could be done more rapidly.

Note: the price variances of 20 $ or more are due to price differences of different sellers and depend on which quality the tripod shall have.

In this comparison, the camera price is purposely excluded, because in the current project, smartphone cameras performed nearly as well as professional cameras. With the third setup, a professional camera was used because the aperture time could be set easier and the lens itself is larger, so that more light can be caught.

It is assumed that a smartphone or professional camera is already present in most of the PV laboratories or PV inspection companies (e.g., from PL or EL image acquisition). Furthermore, it can be assumed that (within a certain range) a high camera price difference does not directly provide an equivalent quality improvement, as the UVF patterns observed in this project were clearly visible by the human eye and generally not as detailed as EL images.

5.3 Processing of UVF images

At first, it was tried to adapt available image processing tools from previous studies. Especially *OpenUVF* and *PV Vision*, mentioned in Section 4.5.1 and 4.5.2 . Various issues were faced at installing and adapting these tools to the UVF images of this project. Due to installation problems, these tools could not be used here.

Based these issues and the project goals of this work, goals/criteria were set for the design of a new tool:

Criteria for Image Processing Tools:

- **Automatization**: high throughputs possible, few manual operations;
- **Efficiency**: performance at the given task, computational effort, processing time, accuracy of defect detection, performance compared to a human evaluating the images;
- **Generalization**: adaptability/compatibility with UVF images from other setups;
- **Usability:** user-friendliness (e.g. graphical surface), simplicity of the programming surface (e.g. using a high-end API like *keras* instead of *tensorflow*) or conciseness of scripts and codes, quality of the documentation.

5.4 Testing and evaluation the tool OpenUVF

The tool *OpenUVF*, by Gilleland et al. [1] was well-automatized – two to three scripts needed to be executed, some in Python, some in Matlab. *OpenUVF* was efficient on the images of the developers (91,7 % accuracy [1] p.9). However, it failed to detect the module contours in images of this project. So, probably the morphological operations used need to be adapted and supposedly the classification network retrained. So, it is not yet generalized and in most of the cases it probably cannot be applied to images from another setup without adaptations[6]. It should be noted that generalization is often very hard to achieve in Computer Vision and Deep Learning tasks and probably the most difficult goal in the list above.

In the present form, it is hard for a beginner user to understand and apply *OpenUVF*, because of its short documentation the usage of *tensorflow* without API. During this project, contact was established to the developers, and their help via instructions by e-mail helped to get started with the tool. The developers pointed out that during their project, the time-resources were insufficient for a more complete documentation.

These obstacles led to a change in the strategy of this project: instead of adapting a present tool, a new one shall be developed, with special attention to the set criteria.

[6] It should be noted that this is hard to achieve, and generally can only be achieved by training on a large-scale image databank of UVF images from different climates and countries.

6 Results

The observed UVF patterns including visible encapsulant defects and the developed image processing workflow are presented and discussed in this section.

6.1 Observed UVF patterns

In this work, the square pattern was present in the great majority of the cases (70-90%), some showing a transient state between square and ring pattern, a minority (10-20%) showing only the ring pattern, and the rest showing no UV Fluorescence. A few modules arbitrarily showed some cells with an intense square pattern and other cells without any fluorescence at all, see Figure 23.

Figure 23 shows the main UVF patterns observed at the test field and power plant in Tubarão (from left to right): the square pattern, the square pattern with defects (here broken front glass), the rim (or ring) pattern and some cells arbitrarily showing no UVF or UVF in the square pattern within one module.

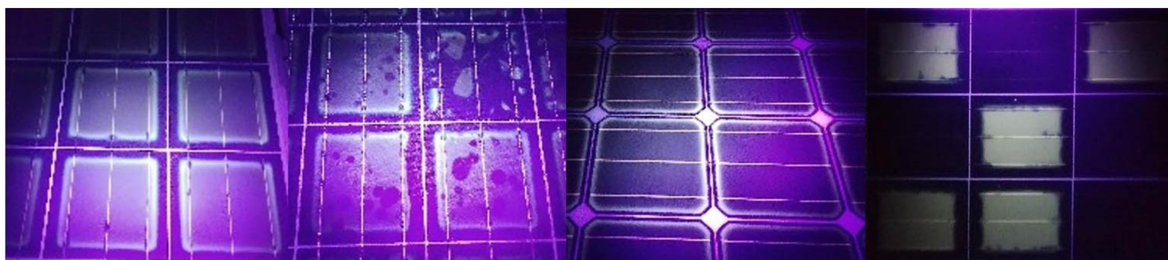


Figure 23: UVF patterns observed at the test field and power plant in Tubarão

Exact percentages for each UVF patterns' share are not named here, because only a small share of the modules was inspected and consequently, they are not representative for the whole power plant. In addition to this, transient patterns were observed, showing UVF in a square pattern with a brighter rim, visible in Figure 24, at the top left. It is assumed that the square pattern slowly turns into the rim pattern, due superimposing degradation effects[7]. Likewise, these transient images could not be clearly classified into one of the two patterns.

Consequently, it was decided to focus the current research on the defect detection on the square pattern. On the bright square area, encapsulant defects are easily visible due to the photobleaching they cause.

[7] The exact reasons for this remain unclear, maybe the UV absorber or oxygen diffusion ways play a role here.

The following (assumed) defects could be observed on the square pattern:

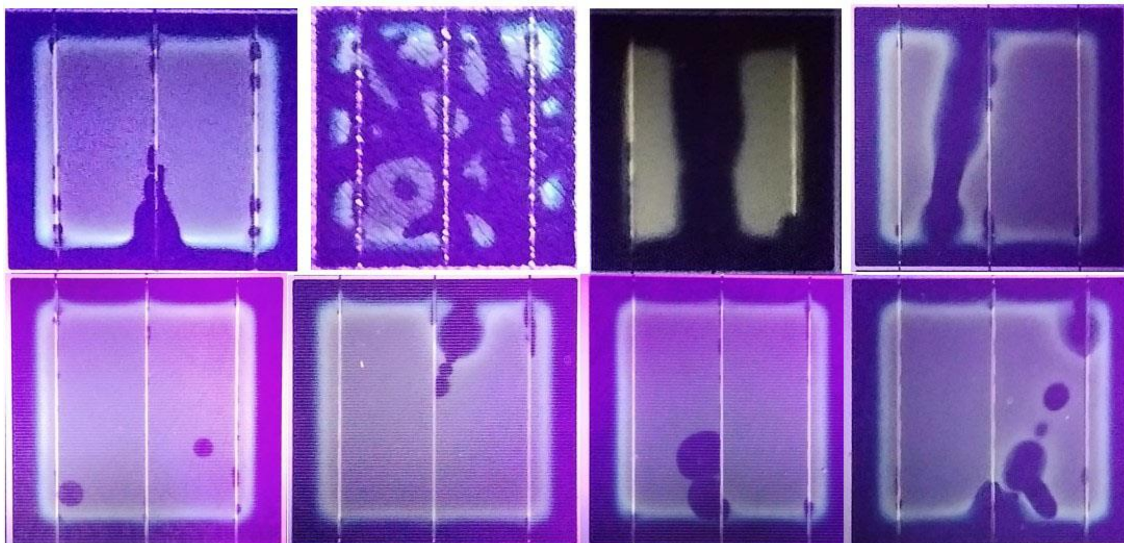


Figure 24: Encapsulant defects observed on the UVF square pattern: (from left to right).
Top: busbar corrosion and lateral leakage, cracked front glass, cracks across the cell.
Bottom: single circular spots of photobleaching, sometimes merging along a line (crack).

The formation of cracks and busbar corrosion causing photobleaching along lines across the square region has been described by [1] and single black spots were observed by [5], p.3. Köntges et al. [5] showed exemplarily that a black spot on the UVF image lies on top of a 1-2 mm cell crack seen on the EL image [5], p.3. So, it can be assumed that at the dark spots, the encapsulant is damaged, e.g. by a crack or a puncture. Humidity and/or oxygen enter and possibly the cell has a cell crack already or is developing secondary defects such as corrosion due to the leakage of humidity/oxygen.

Due to the lack of a large-scale image data acquisition, a detailed analysis of the case-sensitivity by comparing UVF patterns with those from literature could not be performed in this project. The development of an image processing tool shall enable this in the future.

6.2 Automatized Image Processing Pipeline

For the development of a UVF-processing tool, *OpenCV* was chosen as image processing library and Python as programming language. This is a common choice for beginners and students, ensuring a good readability and usability of the developed codes.

In order to allow a concise overview on the program workflow, the program should only consist of a few scripts, in this case three (see Figure 25). The first script has the aim to get the module contour and to do perspective correction on the module. This is an important pre-processing step ensuring that the module and the cells aren't distorted, but clean-shaped geometric as if the camera was directed in a right angle (90°) on the module plane. The second script has the aim to get cell contours and to extract images of

single cells (cropping), both on the original image, as well as on a thresholded binary (black and white) image. The third script creates, trains and evaluates a Convolutional Neural Network (CNN) performing binary classification stating whether a cell is defect or not, see Figure 25.

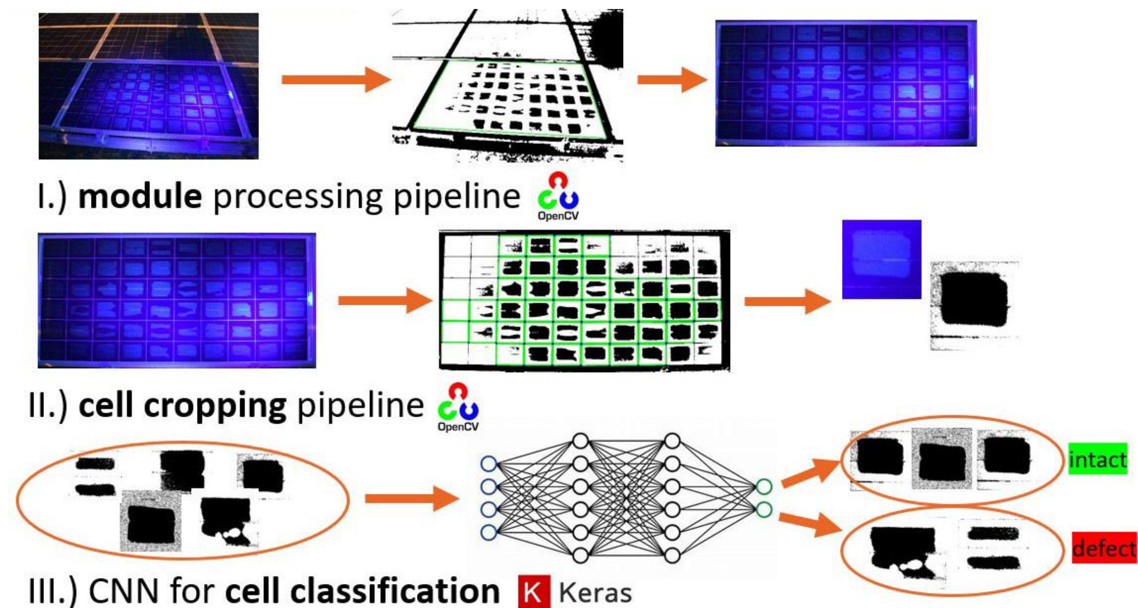


Figure 25: Overview on the 3 main steps (scripts) of this project: perspective correction on the module image, cropping cell images and classifying cell images with a CNN

These three scripts are explained stepwise in the following sections, starting with the script for perspective correction in this section (6.2). The second script is explained in 6.3 and the third in section 6.4. The first two scripts use the library *OpenCV*, the third uses *keras*. The appendix sections 8.1 to 8.3 contain the full code of the respective scripts.

At this point, it should be noted that generally, other processing pathways are possible as well, for example object detection of cells could be performed directly on the original image. Because of the complexity and high computational effort, this option was not implemented in this project.

6.2.1 General settings, importing libraries and loading the original image

First, the libraries ‘cv2’ (*OpenCV2*), ‘numpy’ (numerical python) and ‘pyplot’ (python plotting) are imported. Installing these libraries can be done by “pip install [package name]” commands, or “conda install […]” in case you are using an Anaconda virtual environment. The latter is recommended and was done in this project, so that the libraries for different projects do not interfere with each other.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

A customized function for displaying images was developed, which (by default) portrays images in 10 % of their original size. This small percentage was chosen because images of high resolution (HDR, 4000x3000 pixels) were used. The final command “waitKey(0)” affects that the image is shown until any keyboard button is pressed.

```
def custom_imshow(title, img, perc=0.1):
    w = img.shape[1]   # original width
    h = img.shape[0]   # original height
    # downscaling of the window size & showing the original image
    WW = int(w * perc)
    HH = int(h * perc)
    cv2.namedWindow(title, cv2.WINDOW_NORMAL)
    cv2.resizeWindow(title, WW, HH)
    cv2.imshow(title, img)
    cv2.waitKey(0)
```

At the top of the script a section with general settings was included, to change the most important processing parameters without scrolling down to the respective section each time. The parameter ‘SHOW’ determines whether created images in intermediate steps shall be shown on the screen or not. Similarly, the parameter ‘SAVE’ determines whether these intermediate images shall be saved to the workspace directory.

The other parameters are duly explained in the respective sections.

```
### GENERAL - SETTINGS
SHOW = True
SAVE = True
take_green_channel = True
simple_thresh = 150
G_thresh = 105
contour_limit = 240
EPSILON = 0.05
```

To start the pre-processing, an image is saved in the current working directory (the folder in which the Python script is saved, too). The original image is now accessible as a variable and shown on the screen.

```
### 1.) LOAD ORIGINAL image
img_name = 'DSC_0650'
img_original = cv2.imread(img_name + '.JPG')
custom_imshow(img_name + " original", img_original)
```

The example image is shown in Figure 22 and was taken with the third setup at the fotovoltaica/UFSC Laboratory in Florianópolis.

6.2.2 Brightness correction

In this project, most of the images have an inhomogeneous illumination, due to the problems at buying a large-area UV-light mentioned in section 0. To compensate this inhomogeneous illumination, brightness correction techniques are applied on the original image.

As all images of this project were taken at night or in a dark room, and as the UVF signal is not very bright, the original image was made brighter using the so-called gamma method. The gamma method is an exponential scaling using (1/gamma) as exponent [16]:

$$output = (\frac{input}{255})^{\frac{1}{gamma}} \cdot 255$$

In the formula above, 'input' is the pixel value in the original image and the output is stored as pixel value on the output image [16]. In this project, gamma values of 1.5, 2.5 and 3.5 were used. These values were found and adapted manually by trial-and-error and visual evaluation of the obtained image quality. The code implementing this in a function 'gammaCorrection()' was adapted from a tutorial presented in [16].

Furthermore, the colour channels were equalized in a step called "colour histogram equalization". This affects that the pixel intensities of the R, G and B channel are normalized. To implement this, the code of the function "equalize_hist_color()" was taken from the respective book section in "Mastering OpenCV", see [17], p.199ff. The code for the brightness correction can be found within the full script in the appendix, section 8.1.

The images with brightness correction using different values of gamma are shown in Figure 26:

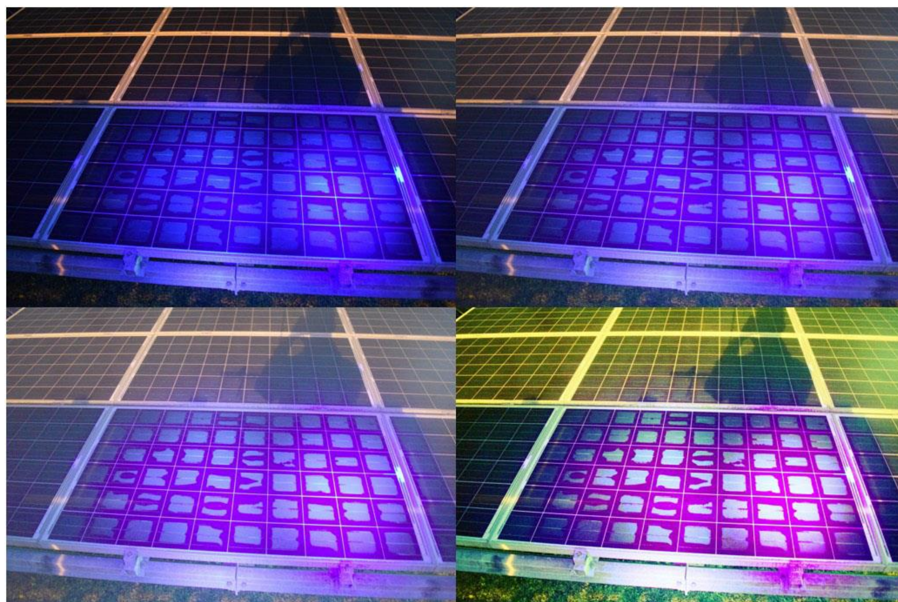


Figure 26: Brightness correction: original image (top left), gamma=1.5 (top right), gamma=2.5 (bottom left) and both gamma=2.5 and colour histogram equalization (bottom right)

For the image processing in this project, gamma=2.5 was chosen and colour histogram equalization applied afterwards, as shown in Figure 26 on the bottom right.

6.2.3 Extracting the green colour channel or converting to grey scale

To reduce the amount of information and to perform thresholding and contour detection later, transforming the image to grey scale is needed. For this, the function 'cvtColor()' from cv2 can be used to convert the colour image from RGB-format to grayscale.

For a more detailed analysis in this project, the brightness-corrected and colour-histogram-equalized images were split into colour channels. In other words, the layers of red pixels, green and blue pixels were each taken separately and interpreted and shown as a greyscale image. The implementation using array-slicing is inspired by code sections in [17].

```
### 3.A) Extracting the green colour channel
B = img[:, :, 0]# splitting into colour channels
G = img[:, :, 1]
R = img[:, :, 2]
# img_without_red = img[:, :, 0]

if take_green_channel:
    custom_imshow('GREEN channel', G)
    cv2.imwrite(img_name+'_GREEN channel.jpg', G)
    #custom_imshow('RED channel', R)
    if SAVE: cv2.imwrite(img_name+'_RED channel.jpg', R)
    #custom_imshow('BLUE channel', B)
    if SAVE: cv2.imwrite(img_name+'_BLUE channel.jpg', B)
    img_name = img_name + "_G"
    simple_thresh=G_thresh
    img_gray = G

### 3.B) GRASCALE
if take_green_channel==False:
    img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
else:
    img_gray = G
```

As can be seen in this code section, extracting the green colour channel was treated as an alternative to grayscaling, and controlled via the logical parameter 'take_green_channel', set in the general settings at the beginning of the script.

This colour-channel-splitting allowed interesting observations on the acquired colour channel images, see Figure 27.

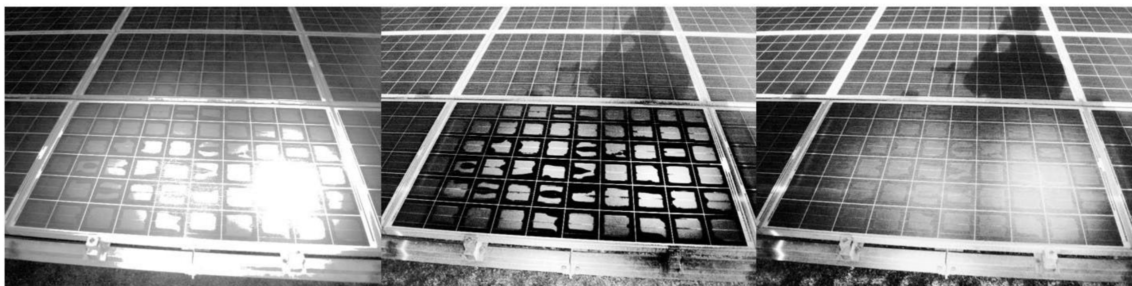


Figure 27: Blue, green and red colour channel of the brightness-corrected image

The **blue** channel seems to show light which has been reflected on the module glass surface, nearly without alteration, especially without significant energy loss, so that the wavelength is not increased, and the colour remains blue.

On the **green** colour channel, the UVF-pattern appears very intensely and with a high contrast. This observation suits to the theory of light-absorption and re-emission during the effect of fluorescence: the blue/UV excitation light is absorbed in the outer atomic shells of the substances (degradation products) in the EVA, and afterwards re-emitted with a longer wavelength and less energy. This wavelength increase affects a colour change from the UV/blue excitation light to the green UVF signal.

The **red** channel shows scattered light and, if present, ambient orange/red light sources. When this image was taken, there was an orange security light in the back of the photographer. This light can be seen clearly on the red colour channel, especially by the shadow of the photographer on the module in the background (see Figure 27). The UVF light did not create this shadow, because it was in front of the photographer. Apart from disturbing light, scattered light is shown, diffuse light that has been reflected several times and therefore has an increased wavelength and red colour.

It should be noted that the red colour channel can be eliminated (set to zero), as in the commented line creating 'img_without_red'. However, in this project, mostly the green colour channel was used.

6.2.4 Inverting

As the pre-processing prepares the contour detection, a definition for foreground and background is needed, with the contour as the separating line (border) of an object in the foreground. The contour can be understood as the outline or silhouette, distinguishing a shape from the background. In *OpenCV* by default white objects are considered the foreground and black areas the background.

On the greyscale image, the module frame appears bright and the background dark. So, at the first glance this is a favourable condition for contour detection according to the above definition. But as one module frame is close to the others, there is no clear edge between the module frame(s) and the background. On the other hand, the inner edge between the frame and the lateral PV cells has a well-pronounced contrast and sharpness. So, the module contour shall be drawn along the inner border of the module frame. To have the PV cells as bright foreground and the frame as background, the grayscale image is inverted. The inversion can be done by calling the built-in function "bitwise_not" from *OpenCV* [17]:

In step 3, variables are named/overwritten respectively when taking the green channel:

```
if take_green_channel:
   […]
   img_gray = G
   simple_thresh=G_thresh
```

```
    […]
### 4.) INVERTING
img = cv2.bitwise_not(img_gray)
title = img_name + " inverted grayscale after mB and dilate"
if SHOW: custom_imshow(title, img)
if SAVE: cv2.imwrite(title+'.jpg', img)
```

Though “bitwise_not” can be the logical inversion (1 to 0 and vice versa), it performs an inversion of the greyscale image here.

The resulting image, either the inverted grayscale image or the inverted green colour channel, shows a high contrast at module borders and cell borders. Figure 28 shows the inverted green colour channel. The high contrast and sharp edges are favourable conditions for the contour detection later.

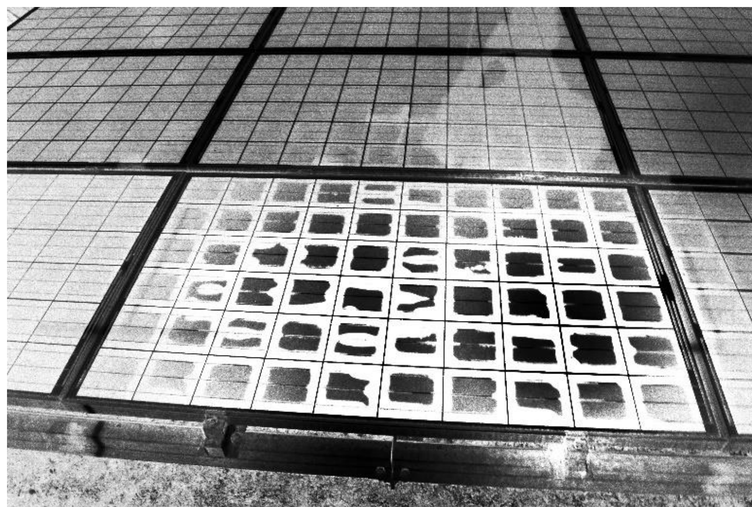


Figure 28: Inverted image (pre-processed green colour channel)

6.2.5 Thresholding

Although contour detection is technically possible at this stage, it is not recommended, because it creates rough, unsteady contours. Commonly the last step before contour detection is thresholding. Thresholding generally denominates any method that converts a greyscale image (with pixel values 0…255) to a black and white image (with pixel values either 0 or 255).

Simple thresholding sets a limit for pixel values, for example 110, and any pixel with a value above is set to 255 (white) and any pixel value underneath is set to 0 (black). The advantage of this method is its simplicity, both for understanding and implementation. The disadvantage is that it cannot cope with inhomogeneous images, e.g. being one region brighter than others. So, simple thresholding only preserves well-pronounced edges.

Figure 29 shows the simple thresholding on the green colour channel. As desired, module (and cell) borders are well visible.

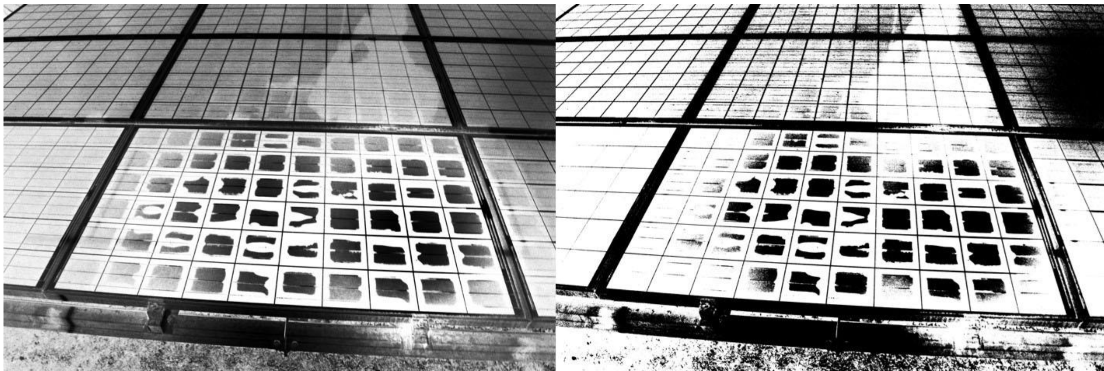


Figure 29: Simple thresholding (thresh=105) on the inverted green colour channel

There are several more elaborated and complex thresholding techniques, such as **adaptive thresholding, Otsu's method** and **canny edge detection**. As they are more sophisticated, they allow to preserve minor edges that are more difficult to perceive. Although they form the state-of-the-art thresholding techniques and are highly powerful, they were not used in this project: after a series of trials, it was noted that the resulting black-white images contain too much information, and too many contours were detected. So, they could be estimated as 'too elaborated' for the simple detection of the module contour in this step. But naturally, these methods offer potential for more generalization, when developing a tool for diverse UVF images, taken with different setups at different sites.

In this project, simple thresholding was applied following the instructions in [17] and adapting the thresh value manually by trial-and-error and visual evaluation of the black-white image quality. For the implementation, the function cv2.threshold() can be used, specifying the thresh value:

```
### 5.) SIMPLE THRESHOLDING
th, img_thresh = cv2.threshold(img, simple_thresh, 255, cv2.THRESH_BI-
NARY)
title = img_name + " simple thresh "+str(simple_thresh)
custom_imshow(title, img_thresh)
if SHOW: custom_imshow(title, img_thresh)
if SAVE: cv2.imwrite(title + ".jpg", img_thresh)
```

In the script, the "simple_thresh" variable with the value 150 is applied after grayscaling, and "G_thresh" with the value 105 if the green colour channel was taken, because only one colour channel yields lower pixel intensities than all three. These thresh-variables are defined at the top of the script in the section for general settings.

6.2.6 Contour Detection and Morphological Operations

As this step is more complex, the code explanations are divided into three sections:

A.) Contour Detection and Approximation
B.) Morphological Operations
C.) Loop of Morphological Operations and Contour Detection

These (development) steps and concepts are explained in the following sections.

A.) Contour Detection and Approximation

OpenCV offers the function '**cv2.findContours()**' to perform contour detection on grey-scale or black-and-white images. Explanation on parameters and details are available in the *OpenCV* documentation and in manuals such as [17].

To customize and apply 'cv2.findContours()', a function '**detect_contours()**' was written, enclosing contour detection, approximation and plotting the results.

First, the function differentiates between detection of all or only external contours. In our case, the module contour is an external contour (later cell contours are external, too). The variable 'contours' saves the returned list of [x, y] coordinates of each point of each contour. The length of the contour is the number of its points. If, as a very simplistic example, there are 3 contours of length 4, 3 and 7, the variable contour will look like this:

```
contours = [ [ [x1, y1], [x2, y2], [x3, y3], [x4, y4] ],
             [ [x1, y1], [x2, y2], [x3, y3] ],
              [ [x1, y1], [x2, y2], [x3, y3], [x4, y4], [x5, y5], [x6, y6], [x7, y7] ] ]
```

The contours that 'cv2.findContours()' hands back for the binarized UVF images are far more and far longer. In this project, there were generally 1000 to 100000 contours of lengths between 3 and a few hundreds or thousands of points. Each point is one pixel of the contour line. These are rough estimations of the dimensions; the number and length of contours highly depend on the input image in the concrete case. So, due to noise, far too many contours were detected and the number and length of the contours need to be reduced. On the example image, about 34700 contours were detected, see Figure 30.

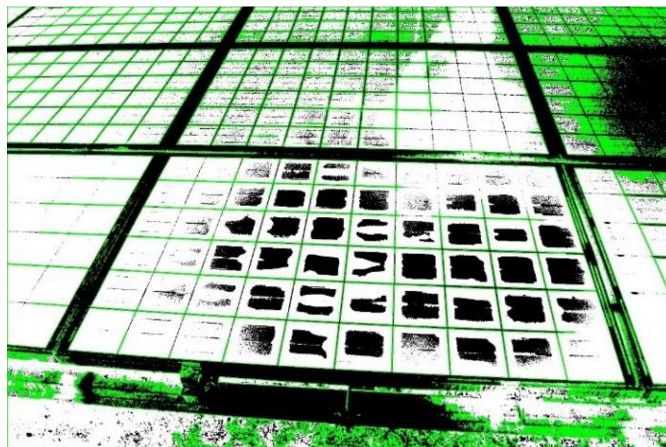


Figure 30: Contour detection on the 'raw' thresholded image: 34700 (rounded) contours were detected

To reduce the length of contours (the number of points per contour) the contours can be approximated (represented by fewer points). The function 'cv2.findContours()' offers such an approximation in form of the keyword-argument 'CHAIN_APPROX_SIMPLE', which helps as a first step to reduce the amount of data.

In case of the module contour, which is geometric and four-cornered, only the four corner points should be needed to describe its whole shape. For a more drastic simplification, a polynomial approximation 'cv2.approxPolyDP()' is used. As first argument it takes the contour itself, and secondly a tolerance called epsilon:

```
cv2.approxPolyDP(cnt, epsilon, True)
```

The lower epsilon is, the closer the approximation remains to the original contour and vice-versa [18], see Figure 31.

Figure 31: The influence of the tolerance epsilon at contour approximation, [18]

So, high values for epsilon allow the approximation to deviate more from the original contour and the approximated contour becomes more geometrical [18]. In this project, 5-15 % of the circumference (arclength) of the contour are taken as values for epsilon, by default 5%. The parameter epsilon is also included in the list of settings at the beginning of the script.

The reason for the high number of detected contours is the noise in the image. Small white and black noise dots, that were not eliminated by thresholding are detected as contours. To eliminate the image noise, morphological operations are used to simplify the black-white image further. In this script, simplification also aims at erasing cell contours to have a clear module contour.

B.) Morphological Operations

Morphological operations (also: morphological transformations) are kernels that are passed over binary (black-and-white) images altering the shapes or stressing structures such as object borders in a specific way [17].

Two basic morphological operations are dilation and erosion. Dilation expands white areas (foreground objects) while erosion expands black areas (the background regions) [17]. The combination of first eroding and then dilating is called opening, the other way around (first dilating and then eroding) is called closing [17]. The opening and closing

operations use the same kernel for both steps [17]. There are more morphological operations such as gradient operations, and every morphological operation can be customized by its varying the kernel size and kernel structure. More details are available in the respective literature, e.g. [17] as an introduction.

In the present case, the aim is to eliminate the black cell borders and salt and pepper noise. Therefore, a combination of the dilation operation and the so-called medianBlur filter were applied. Likewise, white regions are expanded and noise which is smaller than the kernel size of the medianBlur-filter is eliminated.

C.) Loop of Morphological Operations and Contour Detection

In the presented script, dilation, filtering and contour detection are performed in a loop, to stepwise simplify the image and monitor the obtained contours. The number of detected contours is monitored and considered as a measure for the simplicity of the image. In this loop, the following compromise is needed: the image needs to be simplified enough, to facilitate the contour detection but not too much in order to not alter the image in a way that the obtained contour becomes unprecise, meaning that it does not lay exactly over the module contour in the original image. In addition to this, a too long series of morphological operations can 'destroy' the module contour i.e. by merging its area with the area of a neighbouring module. With respect to these aims and limitations, two abortion criteria were applied: the loop is stopped if either the contour limit or the iteration limit is reached, see Figure 32.

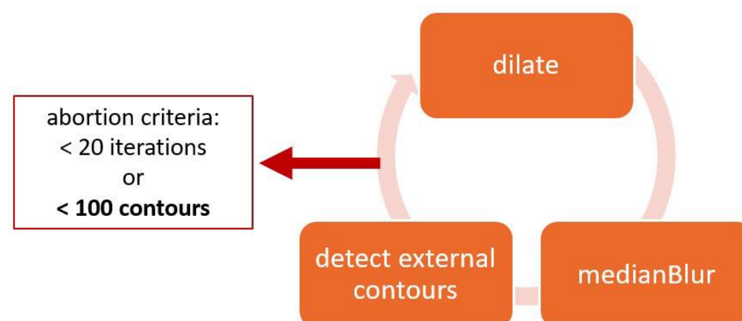


Figure 32: Loop of morphological operations and contour detection, as implemented in this project

The contour limit and the iterations limit can be varied, based on the amount of noise and consequently the extend of simplification needed. As one main parameter, the "contour_limit" also appears in the general settings section at the top of the script. The iterations limit was kept to 20 and normally is not reached. The following code implements this loop with the two abortion conditions:

```
### 6.)  MORPHOLOGICAL OPERATIONS LOOP ----------------
max_iter = 20
# create the kernel for smoothing images
kernel_averaging_10_10 = np.ones((10, 10), np.float32) / 100
kernel_averaging_5_5 = np.ones((5, 5), np.float32) / 25
```

```
kernel_size_3_3 = (3, 3)
mB_kernel_size = 9

img = img_thresh # choose input image
for i in range(max_iter):
    img = dilate(img, cv2.MORPH_CROSS, kernel_size_3_3)
    img = cv2.medianBlur(img, mB_kernel_size)
    total_nr_contours, contours = detect_contours(img,
                                                    which='external',
                                                    eps=EPSILON,
                                                    iter=i + 1,
                                                    draw=True,
                                                    save=True,
                                                    plots=False)
    # ABORTION criterion: contour limit
    if total_nr_contours <= contour_limit:
        print("Less than " + str(contour_limit) + " contours reached,
quitting morphology loop at iteration " + str(
            i + 1))
        stop_iter = i + 1
        break
```

The number of iterations is limited within the head of the loop, while the contour limit is implemented via an if-statement with a break command.

After the loop execution, the simplified image has less contours than the specified limit. Depending on the image (i.e. the illumination and thickness of the module frame), the module contour can be broken during this simplification. In that case, the contour limit needs to be adjusted manually. For the current image, a contour limit of 240 contours was used, because after further morphological operations the contour opened at the lower right anchoring clip. The other settings in this loop (which morph. operations, kernel sizes etc.) need to be adjusted whenever images from a different setup or module type are processed. In the present case, the loop stopped after 5 iterations, reaching less than 240 contours (231), see Figure 33.

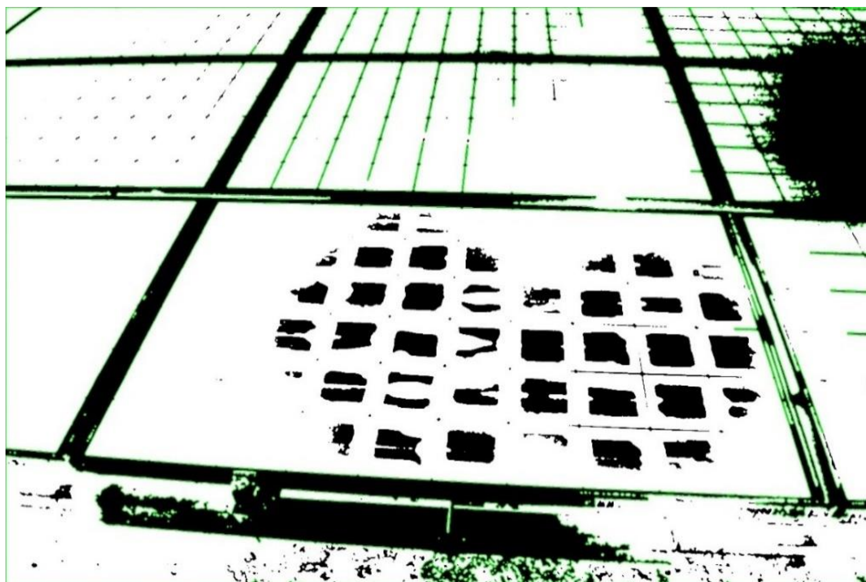


Figure 33: The simplified binary image produced by the loop of morphological operations after 5 iterations; 231 external contours are detected (contour limit set to 240 here)

The contour limit is a highly relevant parameter here: if it is too high, the image is not simplified enough, and the module contour is not yet clearly visible (e.g., still attached to some cells). If the contour limit is too low, the module contour is probably open at some point and then cannot be detected. If the image quality is bad, this range of possible contour limit values might be very narrow or disappear. In the latter case, a detection of the module contour with this script is not possible and the pre-processing needs to be altered. For this reason, homogeneous illumination and reasonable image quality are important.

6.2.7 Filtering contours by criteria

The desired module contour is now 'filtered out' from the remaining contours. This can be done by accessing the properties of the obtained contours and checking specific conditions for the module contour. The more specifications are done for the module contour, the better will work the contour filtering.

The following criteria were used to filter out the module contour:

7.A) The module contour has 4 corners.

7.B) The module contour area is within 20 to 95 % of the total image area.

7.C) Opposite sides of the module contour do not deviate more than 60° and 40° in their directions. (parallelism)

7.D) The module contour is the largest contour of those fulfilling the conditions A to C.

In the example image, the criterion 7.A already reduces the number of contours from 231 to 108, see Figure 34.

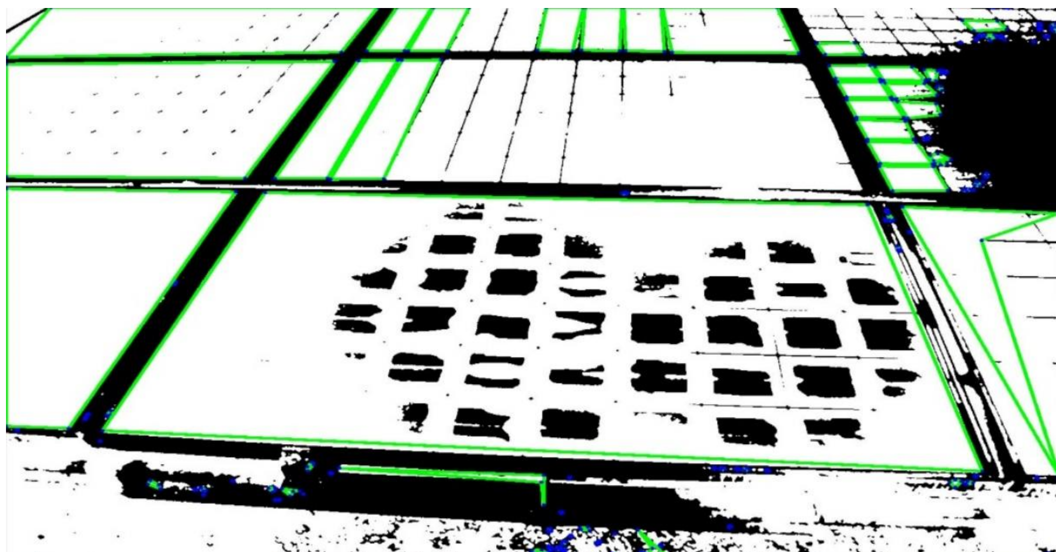


Figure 34: When filtering for 4-cornered contour approximations (criterion 7.A), 108 of 231 contour approximations remain

The criteria are formulated as logical statements and implemented as such, so that each criterion is either true or false. As the formulation of 7.D implicates, a logical AND-combination of all the criteria is used to decide which contour is the module contour.

While the criteria 7.A, 7.B and 7.D could be easily implemented by accessing the contour properties via *OpenCV*, the implementation of **7.C (parallelism)** was done manually by defining and calling functions. The aim of 7.C is to check whether opposite sides are relatively parallel, to exclude very distorted contours and to allow nearly trapezoidal contours. Nearly trapezoidal contours occur frequently due to the camera perspective and the inclination of the module.

As a first step, the four corner points are identified, by assigning them to variables named after the corners, e.g. “lower_left” or “upper_right”. Then, each side is computed as a vector in polar coordinates, with its magnitude and its angle towards the positive x-direction. The difference between these angles for left/right and upper/lower side are calculated respectively. These two angular differences specify the difference of the direction of opposite sides and are measures for the parallelism. In this project, module contours typically were nearly trapezoidal, with left and right-side directions deviating more than upper and lower side directions, which are nearly horizontal. So, for the larger angular difference a limit of 60° is set, for the lower angular difference 40° respectively. These limits were found experimentally and sometimes elevated to 60° and 80°, relying more on the other contour criteria.

When applying all the contour criteria to the approximated contours, the module contour is extracted, see Figure 35.

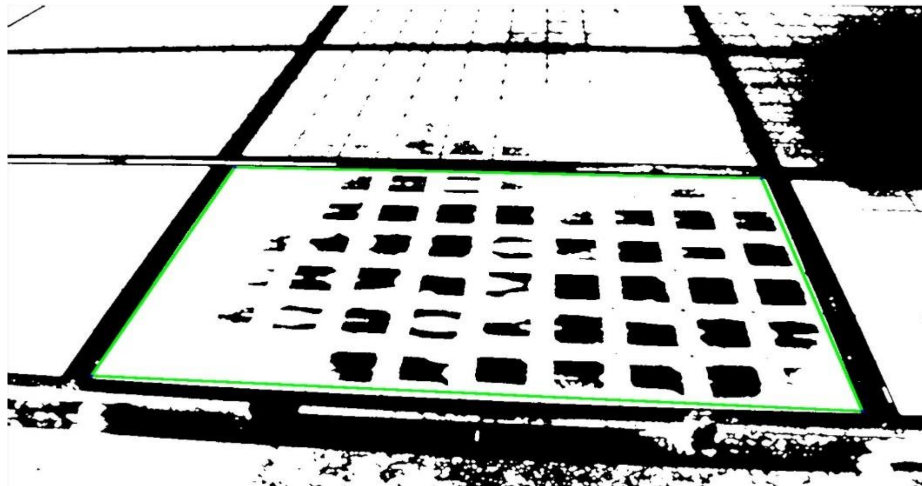


Figure 35: The module contour, filtered out by applying all the contour criteria

On the majority of the whole-module images of this project, the contour filtering could be applied successfully. Whenever the module contour was not found, it was more likely due to bad illumination or pre-processing (thresholding and morphological operations) that the module contour was not clearly visible. It turned out to be a good strategy to use several criteria for the contour, each one having a certain tolerance and to then build the

logical AND-connection of all of them. Likewise, all the real-world knowledge on the sought contour is applied in the code.

The code for contour filtering is added in the appendix, in section 8.1 within the full script.

6.2.8 Perspective transformation

As the last step of the module-processing script, a perspective transformation is applied, cropping out the module and correcting the camera perspective.

Perspective correction means that the image is transformed in a way that the apparent angle between camera direction and module plane is corrected to 90°, orthogonally directed at the module plane.

The perspective transformation is performed using built-in functions of *OpenCV* and orienteering at tutorials on the implementation, especially [19].

To prepare the perspective transformation, the four corner points are identified, side lengths calculated as well as average width (dx) and average height (dy). The coordinates of the 4 identified points forming the module contour are the input points; the target points are synthesized from average width and height:

```
target_points = np.float32([[0, 0], [dx, 0], [0, dy], [dx, dy]])
```

A transformation matrix is computed applying the function “**cv2.getPerspectiveTransform()**” on the input and output points.

```
matrix = cv2.getPerspectiveTransform(input_points_rim, target_points)
```

The function “**cv2.warpPerspective()**” is applied on the input image, using the obtained transformation matrix and specifying the target image size.

```
img_color_out = cv2.warpPerspective(img_color_out, matrix, size)
```

The obtained perspective-corrected image is shown in Figure 36 on the right.

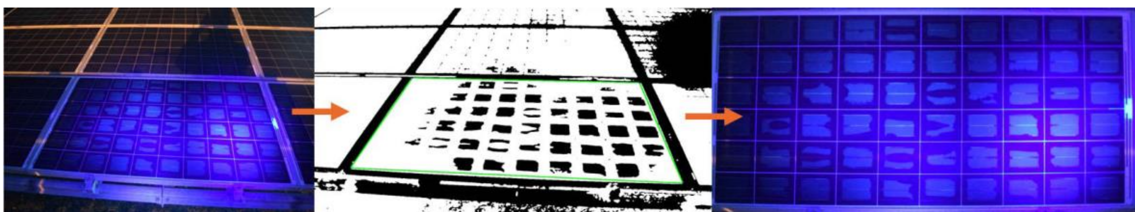


Figure 36: Original image, module contour and perspective corrected module image

The complete code for the perspective correction step is included in the appendix, section 8.3.

6.3 Cropping cell images from a pre-processed module image

The second pipeline starts with the perspective-corrected module image and has the aim to cut out (crop) images of single cells. For that, the detection of the cell contours is necessary. The structure of this cell-cropping pipeline is very similar to the one of the module-processing one. The full script is included in the appendix, section 8.2. The main steps are portrayed schematically in Figure 37:

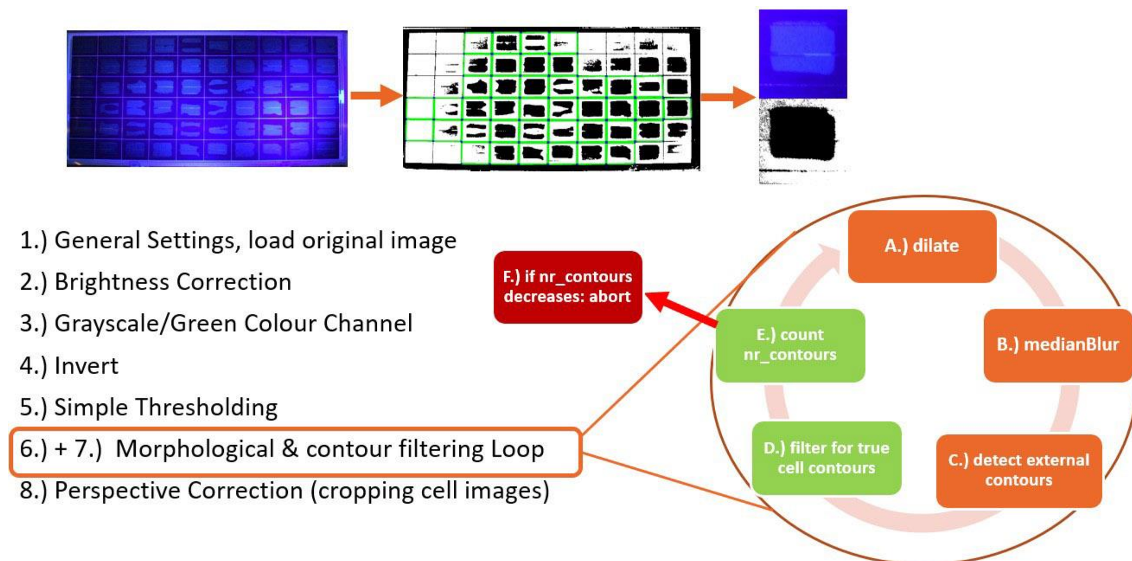


Figure 37: Overview on the cell cropping pipeline (the second script)

In fact, this script was developed based on the module-processing script and altered to get cell contours and crop images of single cells. The pre-processing steps 1 to 5 are identical to the module-processing script. For later use on large-scale datasets, a pre-processed and perspective-corrected image can be handed over. This saves the computational effort for executing steps 1 to 5 again. However, in this experimental stage, this approach was chosen to have more flexibility and to occasionally be able to modify the pre-processing of each script.

6.3.1 Morphological operations to intensify cell borders

In processing step “6.) + 7.) Morphological and contour filtering Loop”, the aim is now to reinforce cell borders, not the module border. Therefore, the morphological operations need to be adapted. Previously, they erased cell borders to get the module borders, now the aim is to strengthen cell borders.

Generally, the dilation operation and the medianBlur-filter can also be used here, but when applying them for noise reduction, the following problem occurs: some of the salt

and pepper noise dots are larger than the distance between two neighbouring cells (the border). So, if a smaller filter kernel is applied, not all the noise is filtered out. Contrarily, if a larger filter kernel is used, it erases the borders between two neighbouring cells and their contours melt together.

The simplification has these limitations, and it is hard to generally estimate which extent of simplification is sufficient and which extent is too high, judging by only the total number of detected contours as was done before. In other words, it is hard to determine automatically when to stop the morphological loop, having simplified the image enough but not too much.

6.3.2 Filtering cell contours by criteria

To solve this problem of when to stop the loop, the contour filtering step was included into the loop of contour detection and morphological operations. In form of pseudocode, the loop of steps 6 and 7 executes the following steps repeatedly, see steps A-F in Figure 37:

A dilate

B apply the medianBlur filter

C detect all contours and approximate them

D filter for true cell contours using cell contour criteria

E count the true cell contours and save the number in an array as a history

F if the current n° cell contours is equal or higher than the previous one, continue with step 1, else quit the loop execution and continue with the program

The steps A to C are generally performed in the same way as before, with the exception that eventually the kernel sizes (of dilation and the medianBlur filter) were increased stepwise or first increased and later decreased, to enhance and speed up the process of reinforcing cell borders. These options were not included in the final codes, because they also bring the danger of rapidly destroying cell contours.

The last step (F) relies on the fact that at some point, the morphological operations will destroy a cell contour by merging two neighbouring cells together. If that happens, this last step (F) re-establishes the previous list of all cell contours and quits the loop, so that the maximum number of detected cell contours is used.

Furthermore, the contour criteria in step 4 were adapted to be cell contour criteria, using the following 3 criteria:

- A cell contour has **4 corners.**
- In a cell contour, **opposite side lengths are equal** (rhombus or square), allowing a certain tolerance.

- A cell contour has about the area of 1/60 of the module area.

The last criterion uses the fact that each of the inspected modules had 60 cells. The number should of course be adapted in the code if different modules are inspected. In the great majority of the cases, these criteria were sufficient for filtering out true cell contours in the current project. Single erroneous cell contours creating distorted images need to be deleted manually (by a human).

For observations during the development of the criteria and for the validation of their efficiency, parameters supplying information on the contours were saved as histories (in arrays, one entry for each loop execution) and can be plotted. The most important history parameter is the number of true cell contours, as this parameter is used in the abortion criterion. Examples for other parameters are the total number of contours, the (average) number of points per contour or the distribution of the contour areas.

6.3.3 Cropping each cell by perspective transformation functions

The previous loop delivers a list of true cell contours, having each four points indicating the four corners of the cell, respectively. In this last step, each cell shall be cropped out and the image of one single cell shall be saved.

All the steps for that are analogous to the perspective correction of the module. The steps are scaled down and applied one-by-one to each detected cell. The cell images are saved in a folder named after the module image and with the cell index, to create individual filenames.

This procedure uses the fact that the perspective transformation function “cv2.warpTransform()” does not only correct the perspective, but also crop the image. Additionally, in case the perspective correction on the module image did not work perfectly, a slight correction is automatically applied to each cell, assuring a good image quality of each cell image.

Note: It is also possible to crop images of cells directly from the original image, without correcting the perspective based on the module contour. At first sight, this seems more straight-forward and advantageously: this option would need less computational effort and does the tasks of the two presented scripts in only one. However, this way it is more difficult to distinguish true cell contours from contours around noise, especially if the module appears very inclined or is not homogeneously illuminated. Likewise, several sources of errors appear, at least when working on the images acquired in this project. For images acquired with more elaborated setups and better illumination, cropping cell images directly could be a good idea to save coding steps and computational effort.

The cell-cropping (by perspective correction functions) is both applied on the perspective-corrected module image and on the binarized (thresholded) perspective corrected

module image. The obtained coloured and black-white cell images are stored in respective folders. By applying this script to several perspective-corrected module images, a databank of cell images can be created.

6.4 Training a CNN on classification of UVF cell images

With the previous scripts, cell images could be created in an automatized or semi-automatized way. The automatization aims at enabling the creation of a large databank of cell images. The image databank can then be used to train a Neural Network model with the aim of automatized defect detection, as shown in Figure 25.

In this project, a demonstration (or prototype) script is developed that makes the decision whether a cell has a defect or not. This task definition can be denominated as binary image classification. To classify the cell images a Convolutional Neural Network (CNN) is trained to assign one or several tags (class names) to an image. The class names in this project are “defect” and “intact”. By developing the code further, a multiple classification could be trained, to specify which defects are present on a given cell image.

For the implementation the Deep Learning libraries *keras* (front-end) and *tensorflow* (backend) were chosen, to create an efficient CNN while keeping the code readable. An instruction on how to install *keras* and how to set up a deep learning coding environment is provided in the appendix, sections 7.1 and 7.2.

The third and last script performs initialization, training, validation and testing of a CNN for cell image classification and is presented in this section step by step. This script is also written in python, so that all three scripts could be called from a main program or console later. The full code is added in the appendix, section 8.3.

Literature recommendations for learning to know the fundamental structures and concepts of Neural Networks and Convolutional Neural Networks are provided in section 7.3 of the appendix.

The fundamental principles of Neural Networks and convolutional neural Networks (CNN) were summarized in the sections 4.4.1 and 4.4.2. The code of the third and last script, which is based on these concepts, is presented step by step in the following sections: 6.4.1 to 6.4.6.

The implementation and code structures of this script are based on the code examples in the book “Deep Learning with Keras and Python”, by François Chollet [8].

6.4.1 Creating a data bank of cell images

From this section onwards, the steps of the third script are explained step by step:

1 **Annotate** the images (here: by saving them in respective subfolders), (section 6.4.1)

2 **Loading** labelled images from a databank (section 6.4.1)

3 **Splitting** the databank into **training, validation and test dataset** (section 6.4.1)

4 Creating **data generators** applying **data augmentation** (section 6.4.2)

5 Load and modify a **pretrained** neural network (section 6.4.3)

6 **Train** and save the CNN model (section 6.4.4)

7 Evaluation of the training progress (section 6.4.5)

8 Evaluate the model on test data and do predictions (section 6.4.6)

After these steps, the Neural Network model can be loaded and used to classify images.

For the first step, annotation, the cell images that were created by the previous scripts can be manually split into the respective classes. To take a simple case, just binary classification between “intact” and “defect” cells is treated here. So, within the file manager, all intact cell images are copied to a folder called “intact”, all defect ones into the folder called “defect” (annotation). This is done manually by the operating person and a difficulty appears when very small defects are present, or when there is uncertainty about whether the UVF pattern shows a defect or not. For simplicity and for demonstration purposes, these cases are left aside in this project.

By developing this program further, respecting these cases and classification into several classes (one for each defect type) should be possible.

A folder called “original” with all defect and intact images in respective subfolders needs to be created manually. Given the file path to this folder and a list of class names, the script splits the data bank into training, validation and test data sets, while respecting the classes as subfolders, see Figure 38.

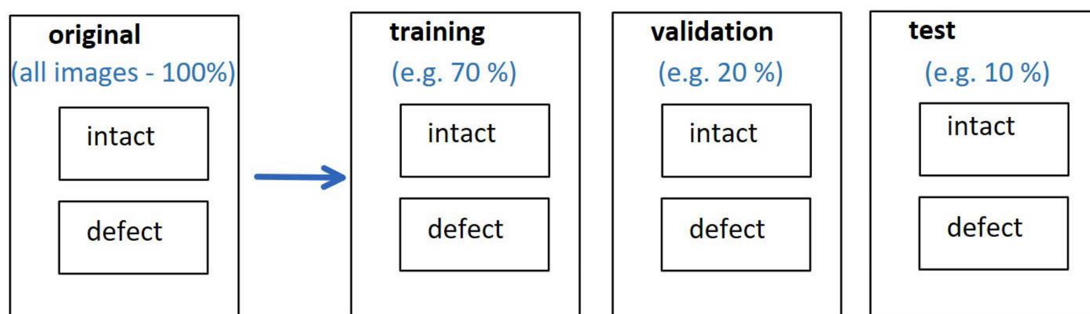


Figure 38: Schematic overview on the folder structure of the cell image data bank

For example, 70% of the images of each class are copied to the respective subfolders in the training folder, 20% to the validation folder and 10% to the test folder. The three new folders and their subfolders are created by the script, not manually. All the copying commands are executed by the script as well, to save time. Only the first assignment of cells to classes by saving them in a respective subfolder within “original”, needs to be done manually by a person. The full code for this first step is in appendix, section 8.3.

6.4.2 Creating data generators applying data augmentation

To create an access to train, validation and test images, image data generators are created for each data subset. Data generators are iterable objects similar to functions that provide one data sample each time they are called [8]. The generators used here provide a batch of pre-processed images with their class name inferred from the subfolder name.

The more image data is available, the potentially better can the CNN learn generalized features. So, a way of enhancing training is to enlarge the image dataset. There are three general approaches for this: to collect more, to synthesize or to augment data [20], see Figure 39.

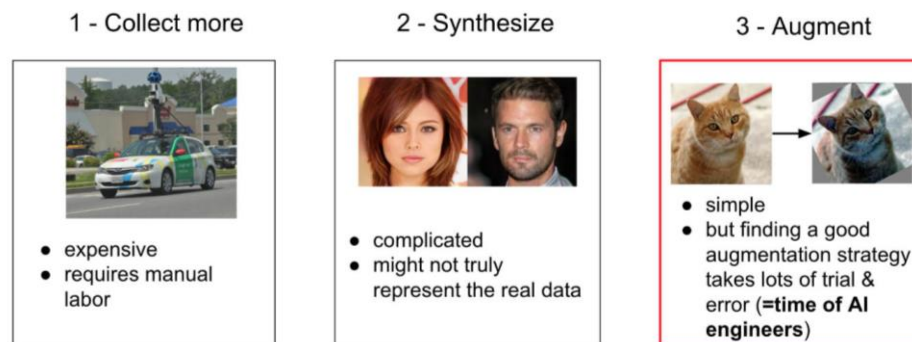


Figure 39: Three ways to enlarge image datasets [20]

As the advantages and disadvantages in Figure 39 state, for a given image dataset, data augmentation probably is the most appropriate solution. Naturally, options 1 and 3, both collecting more real data and augmenting it, would yield the quantitatively and qualitatively the best dataset. In this project, only data augmentation is used. Data augmentation means to create more sample images by applying random operations such as rotations, horizontal/vertical shifts or zooming into the image [8]. An overview of possible operations is provided by Figure 40.

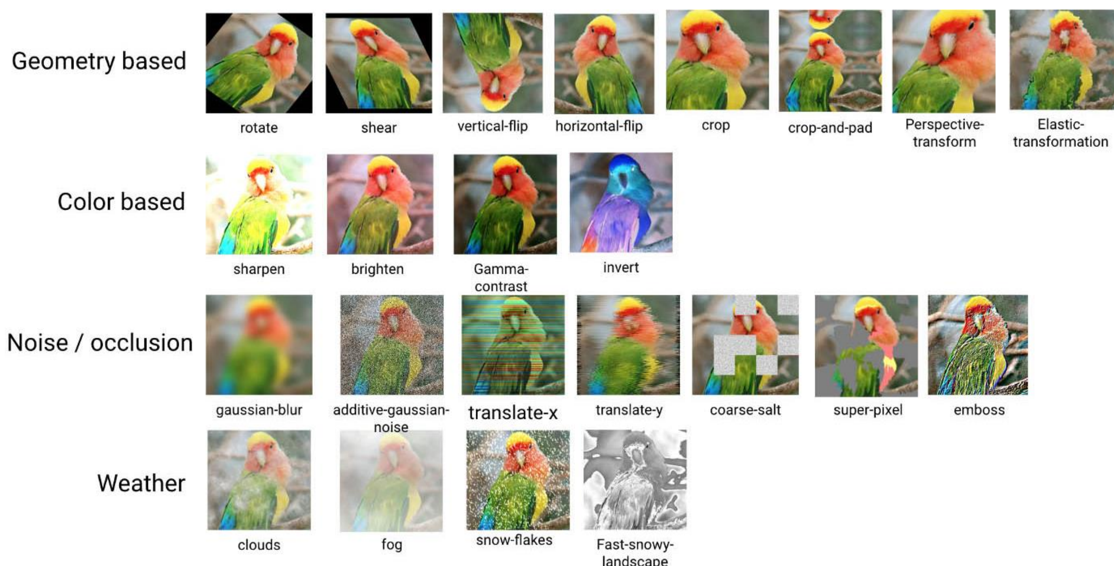


Figure 40: Overview on possible operations for image data augmentation, [20]

In this project, only geometry-based operations are used. They are implemented using *keras*' built in options for data generators: when creating an image data generator using the function "ImageDataGenerator()" the operations for data augmentation can be enabled and specified as key-word arguments [8]. Then, using the "flow_from_directory" argument, the respective data path is provided. Likewise, data generators for training, validation and testing are created, [8].

```
### STEP 2 ### create_DataGenerators
# image data generators with data augmentation
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1. / 255,
                                   rotation_range=40,  # data generator with
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,  # data augmentation
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=False,
                                   )
train_generator = train_datagen.flow_from_directory(train_dir,# configuring train gener-
ator
                                               target_size=(150, 150),
                                               batch_size=images_per_batch,
                                               class_mode='categorical')
validation_datagen = ImageDataGenerator(rescale=1. / 255)  # simple data generator
validation_generator = validation_datagen.flow_from_directory(validation_dir,
                                               target_size=(150, 150),
                                               batch_size=images_per_batch,
                                               class_mode='categorical')
test_datagen = ImageDataGenerator(rescale=1. / 255,
                                  dtype="float32")  # simple data generator
test_generator = test_datagen.flow_from_directory(test_dir,  # configuring
                                              target_size=(150, 150),
                                              batch_size=images_per_batch,
                                              class_mode='categorical'
                                              )
print("Data generators successfully created.")
```

6.4.3 Load and modify a pretrained neural network

An efficient method for applications of CNN is to re-train a network model that has been trained before, instead of training a model from scratch [8]. This technique is called transfer-learning. It accelerates the training progress and allows to proceed rapidly to fine-tuning. The pre-trained network is already able to extract features such as textures and shapes [8].

The library *keras* offers to load pretrained models, in the present script the so-called VGG16 model is loaded. To not modify the feature extraction, the layers are frozen, meaning that their weights cannot be modified [8]. To allow the model to learn the UVF patterns, two fully connected layers are added at the end of the network, each consisting of 256 neurons.

```
### STEP 3 ### load & modify a pretrained neural network
from keras.applications import VGG16
from keras import optimizers
from keras import models
from keras import layers
import os

conv_base = VGG16(weights='imagenet',  # initializing weights
```

```
                include_top=False,   # whether the classifier is included
                input_shape=(150, 150, 3))   # shape of the input image tensor

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(len(classes_list), activation='softmax'))
# freeze the convolutional base
print("Number of weights before freezing weights: ", len(model.traina-
ble_weights))
conv_base.trainable = False
print("Number of trainable weights after freezing weights: ", len(model.train-
able_weights))
```

6.4.4 Train and save the CNN model

Before training, the CNN model needs to be compiled stating the error function (also called loss function), the optimizer and the error metrics. In the present case, common choices for image classification tasks are done (see parameters of the compile() function in the code).

After compiling, the CNN model is ready to learn on the training data. The training is performed by calling *keras*' function 'fit_generator()' with the respective arguments. To specify how many times a generator needs to be called, the 'steps' variables are specified. The following conditions should be fulfilled:

$$number\ of\ available\ images = steps_per_epoch * batch_size$$

The "number of available images" means the number of images in the respective subset (training, validation and testing). So, for each subset an individual batch size can be set and different numbers of steps will be needed.

Note: The variable names can deviate from the above designations: For the training data, the variable "batch_size" is called "images_per_batch".

The aim is that after (steps) batches of (batch_size) samples, all the available samples of the data subset have been handed over.

```
### 4.) TRAIN and SAVE the CNN model
model.compile(loss='categorical_crossentropy',
            optimizer=optimizers.RMSprop(lr=2e-5),
            metrics=['acc'])
history = model.fit_generator(train_generator,
                            steps_per_epoch=images_per_batch,
                            epochs=10,
                            validation_data=validation_generator,
                            validation_steps=36)
```

Finally, the model is saved and can be loaded afterwards and from any other script for later use.

6.4.5 Evaluation of the training progress

The history variable that is handed back from the model.fit() command contains the accuracy metrics obtained at each epoch, both on training and validation data. In this case, it contains the correct classification rate the model achieves on the training and validation image after each epoch. These metrics are used to observe and validate the training progress and to verify that the model performance increases.

```
### STEP 5 ### evaluate the training progress
# use the pyplot library to draw diagrams of the training progress and
test results
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_loss']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='training')
plt.plot(epochs, val_acc, 'r', label='validation')
plt.title('Correct classification rate training/validation')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='loss Training')
plt.plot(epochs, val_loss, 'r', label='loss Validierung')
plt.title('Loss function value training/validierung')
plt.legend()
plt.show()
```

The above code section uses the pyplot library to plot the metrics, correct classification rate and the loss function value as graphs. Due to the random initialization of the weights, the diagrams can be different from execution to execution.

6.4.6 Evaluate the model on test data and do predictions

The final step of the CNN training script is to evaluate the CNN model on the test data. The test data are the images that have been spared so far and are still unknown to the CNN. This is important for judging the ability of the CNN to generalize and classify new, unknown cell images. Within *keras*, the command model.evaluate() is used, together with the test image data generator. This command hands back one set of performance metrics. Here the value of the loss function and the accuracy (correct classification rate) are handed back.

```
### STEP 6 ### Evaluate the model on test data
test_datagen = ImageDataGenerator(rescale=1. / 255)
test_generator = test_datagen.flow_from_directory(test_dir,
                                                  target_size=(150, 150),
                                                  batch_size=10,
                                                  class_mode='categorical')
test_loss, test_acc = model.evaluate_generator(test_generator,
```

```
steps=11)
print('Correct classification rate on test data:', test_acc)
```

The obtained model can be applied using, the model.predict() command of *keras* [21]. For a given image, the command hands back probabilities for each class [21].

By extracting the maximum prediction, the predicted class can be obtained with the respective name 'defect'/'intact', consult the tutorial [22]. More details on the command for predictions are available on the webpage of the *keras* documentation, see [21].

6.5 Results of the Image Processing with the developed pipeline

None of the **225 images of about 4 cells** that were acquired with the **first setup** at Tubarão could be processed automatically, because the illumination is too inhomogeneous. This causes inhomogeneous pixel intensities, which is an obstacle for automatized image processing, even if brightness correction is applied. The pre-processing, especially the simple thresholding applied in this project cannot cope with these inhomogeneities, as the operations assume that common features (e.g., cell borders etc.) have similar pixel intensities across the image area. The cell borders result interrupted and cannot be extracted.

In future versions, the mentioned advanced thresholding techniques could possibly solve this problem. This would highly increase the generalization and flexibility, allowing UVF images acquired with just one or a few flashlights to be evaluated automatically.

The current version of the tool was tested on **51 whole-module images** acquired at the laboratory Fotovoltaica/UFSC with the **third setup**. Of these, 35 whole-module images of the 7 modules from the ground row were selected manually for their good image quality. The presented processing pipeline was executed on these 32 images. In 20 cases, the images could be processed well, especially the module contour was precisely detected and cell images could be extracted. On 9 images, the module contour was detected imprecisely, so that some parts of the module were missing/cut on the perspective-corrected image. On the remaining 3 images and on samples of the initially discarded images, the module contour could not be detected. The main reason was always inhomogeneous illumination causing interruptions in the module contour; either holes at the well/bad illuminated parts, at the anchoring clip or due to connection to cell borders.

The second script for cropping cell images was executed on 10 of the obtained perspective-corrected module images. For each image, between 3 and 39 true cell contours were detected and the respective (binary and colour) cell images cropped. On average, 24 of the total 60 cells were detected and cropped on each module. Among these, no cell contour was detected erroneously, so that 240 high-quality cell images from 7 different modules were obtained, saved in a file of the respective image name. To avoid redundant

data, the folder with most cell images was chosen for each module, yielding 126 cell images, of which redundance can be excluded.

On 22 of these **126 cell images**, only the busbars were visible, no UVF. As these 22 images do not add any information for the training of the CNN, they were discarded. **35 clearly defect** and **35 clearly intact cell images** were selected, see Figure 41.

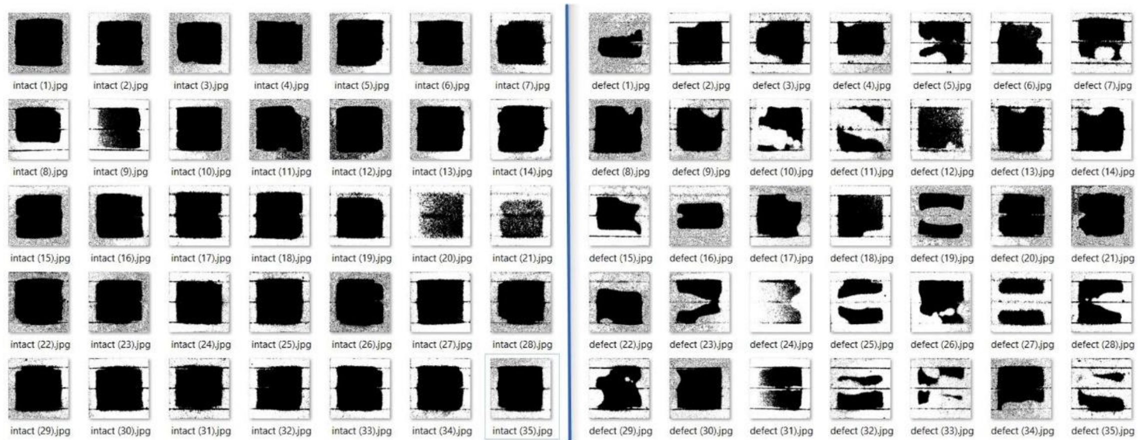


Figure 41: Cell image data base of exemplarily 35 intact cells (on the left) and 35 defect cells (on the right)

The remaining 34 images show slight or small defects or features where it is not clear whether they appear due to a defect or due to bad illumination.

Here, the number of cell images (70) is not high enough to be considered as an appropriate image data bank for Deep Learning. When collecting more data in the future, there will probably be more intact cells than defect ones. Such a data base is called unbalanced[8] data. Methods for training CNN on unbalanced data can be found in the respective literature on Deep Learning, e.g. [8].

For simplicity and for demonstration purposes, in this project a CNN model was trained only on the 35 images of each class. The training script splits the images into 22 (65%) for training, 8 for validation (20%) and 5 (15%) for testing in each class.

[8] Balanced training data means that the same number of samples is available for each class.

The training yielded the following metrics[9], see Figure 42.

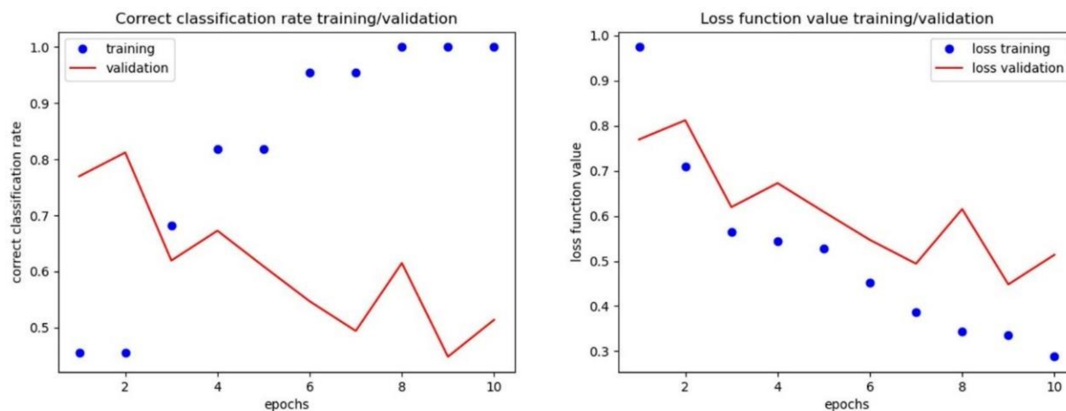


Figure 42: Graphs of the performance metrics (correct classification rate and loss function value) during the training on cell image classification

In the diagram, the epoch at which overfitting occurs, can be detected: it is the first epoch at which the performance on training data is better than on validation data. In the present case, overfitting occurs from the third epoch onwards, see Figure 42. From the third epoch, the network classifies training images better than validation images (see correct classification rates), and the loss function value on the validation data is higher than on training data. So, here the training is executed once more, only for two or three epochs[10].

The loss value represents the error signal that is used to correct each weight (backpropagation, see section 4.4.1). So, the loss value can be considered as an estimate on how wrong the current prediction is or how much the network can still learn. The decreasing loss values on the training data show that the model cannot learn more on the given images. To learn more, it would need more image data, not more epochs. At this point, the mentioned strategies for data augmentation and methods against overfitting can be applied to further optimize the training.

Due to the very low number of cell images, the obtained CNN performance is not representative for the potential of the technique. However, the **correct classification rate** obtained values, that are on average[11] higher than 50% (random assignment): **between 45 and 70 %.** The values change from execution to execution due to the random initialization of the two fully connected layers that were added as a classifier.

[9] Note that the history of training and validation metrics vary at each execution due to the random initialization.

[10] This should be done while preserving the same initialization, because the optimal number of epochs depends on the initialized weight values. This is not implemented in the script presented here. Information and tutorials for implementing this can be found in [8].

[11] based on 5 code executions

Summary

In the course of this project, an experimental setup as well as an image processing pipeline were built to acquire and process UVF images.

First, a literature review was done to explain how UVF forms and to portray the state of the art in PV module inspection via UV Fluorescence. Based on this benchmark, experimental setups were build using UV flashlights mounted on a metal structure on top of a tripod. UVF images of polycrystalline PV-modules were shot at Tubarão and Florianópolis in Santa Catarina in the South of Brazil. On these UVF-images, the so-called square-pattern and ring pattern as well as various other patterns were observed. On the square pattern, defects in the encapsulant were well-visible on the UVF image due to the photobleaching they cause (extinction of the UVF light). The most common encapsulant defects were: punctual encapsulant leakages, cracks, corroded busbars and modules showing some cells with/completely without any UVF. Some patterns were hard to identify and classify because they might belong to a transient state between the square and the ring pattern.

For the image processing and analysis with the aim of defect detection, a search for available tools on the internet was done, and there are good approaches for processing pipelines, but no ready-to-use tool for UVF image analysis could be found. OpenUVF is one of these approaches, which could be developed further, given the required expertise in *tensorflow* and Deep Learning. For (undergraduate) engineering students, coding in *tensorflow* probably means a too high need of expertise, which restricts the target group of the tool. As an alternative suggestion, the development of a tool using *OpenCV* and *keras* was started in this project.

The developed tool consists of three main steps, each done in one respective script: correcting the perspective of the module image, cropping individual cell images and training and evaluating a CNN on the classification of the cell images.

The first two scripts use *OpenCV*, which can be easily installed using “pip install cv2”, or “conda install cv2” or similar commands. The third script uses *keras* and instructions on how to install *keras* and its dependencies are given in the appendix of this work. This work aims at allowing everyone to install *OpenCV* and *keras* and execute the three scripts one after another. This shall serve as a prototype of processing tool for UVF images and as a basis for further development.

In the future and given the appropriate image processing tools, comparative studies on UVF images from different countries and climates could be done, in order to establish UVF as a state-of-the-art inspection method for PV cells.

Conclusion

In the field of optical inspection methods, UVF currently has a niche-position next to the well-established Thermography (IR) and Electro- and Photoluminescence (EL and PL). Research is being done on the application (setups, image processing) and the provided information (cell defect visibility), but the knowledge about the UVF pattern interpretation is not yet sufficient to understand all the available information: there are still unexplained UVF patterns and the correlations between the causes (climate, module composition, age) and the UVF pattern are not yet well-investigated.

On the long term, comparative studies on UVF patterns regarding different climate zones, module compositions and time series of data would be needed to fully understand the formation and causes of the UVF patterns.

This project can be seen as an important contribution to this research, showing UVF patterns appearing in Santa Catarina, Brazil. Also, this work focused on developing a suitable image processing software, as presented in the summary above. Facing issues with available software, a set of four general goals and criteria for (UVF) image processing software was set: automatization, efficiency, generalization and usability. Following these goals, the libraries were chosen which have a reputation to be beginner-friendly and which are taught at universities: *OpenCV* for image processing and *keras* for Deep Learning techniques.

The proposed tool can process UVF images once the general parameters of thresholding and morphological operations have been adjusted. With these adjustments, a high number of UVF images can be processed automatically. So, the tool is semi-automatized and semi-generalized. Together with this work and the presentation on YouTube[12], the tool shall be as user-friendly and applicable as possible.

Due to the development and modification of experimental setups and logistical reasons, not enough module images were acquired to mount a large cell image data bank. Due to this, the performance metrics of the CNN trained in this project are not representative for the potential of the technology. The potential can be estimated by regarding at the 91,6% correct classification rate which Gilleland et. al. achieved with OpenUVF [1]. Furthermore, the application of Deep Learning methods on Thermography is already state of the technology and widely applied for PV inspection. The current project only demonstrated binary classification, but the program capacity can be extended to classification into multiple classes, one for each of the most common encapsulant defects.

To summarize, this work is a contribution to the development of experimental setups and processing tools for UVF images. With further research, development and data collection UVF can become a useful complement to Thermography and Electroluminescence as optical PV inspection methods.

[12] https://www.youtube.com/watch?v=KVd6Grq8Pdg&t=96s

References

[1] W. B. H. J. B. R. Braden Gilleland, „High Throughput Detection of Cracks and Other Faults in Solar PV Modules Using a High-Power Ultraviolet Fluorescence Imaging System,“ in *PVSC 2019*, 2019.

[2] A. Morlier, M. Siebert, I. Kunze, S. Blankemeyer und M. Köntges, „Ultraviolet fluorescence of ethylene-vinyl acetate in photovoltaic modules as estimation tool for yellowing and power loss,“ researchgate, Emmerthal, 2018.

[3] A. S. S. T. J. O. G. T. Kshitiz Dolia, „Early Detection of Encapsulant Discoloration by UV Fluorescence Imaging and Yellowness Index Measurements,“ researchgate, 2018.

[4] F. Li, V. S. Pavan Buddha, E. J. Schneller, N. Iqbal, D. J. Colvin, K. O. Davis und G. TamizhMani, „Correlation of UV Fluorescence Images With Performance Loss of Field-Retrieved Photovoltaic Modules,“ *IEEE Journal of Photovoltaics,* 2021.

[5] A. M. ,. G. E. E. F. B. K. a. J. L. Marc Köntges, „Review: Ultraviolet Fluorescence as Assessment Tool for Photovoltaic Modules,“ *IEEE JOURNAL OF PHOTOVOLTAICS,* 2019.

[6] M. S. I. K. G. M. a. M. K. Arnaud Morlier, „Detecting Photovoltaic Module Failures in the Field During Daytime With Ultraviolet Fluorescence Module Inspection,“ *IEEE Journal of Photovoltaics,* 2017.

[7] A. Trask, grokking Deep Learning, Manning Publications, SMTEBOOKS, 2019.

[8] F. Chollet, Deep Learning with Python and Keras, german title: Deep Learning mit Python und Keras, translation of Knut Lorenzen, mitp, 2018.

[9] D. T. Kim, „TEKnology blog post: [Perceptron] Single and Multi-Layer Perceptron,“ 02 08 2019. [Online]. Available: https://kimdanny.github.io/deep-learning/single-multi-perceptron/#4-3-outputting-final-outcome-eg-softmax. [Zugriff am 28 07 2022].

[10] J. Nagidi, „Dataaspirant: HOW TO HANDLE OVERFITTING IN DEEP LEARNING MODELS,“ 24 08 2020. [Online]. Available: https://dataaspirant.com/handle-overfitting-deep-learning-models/. [Zugriff am 28 07 2022].

[11] O'REILLY, „O'REILLY: Chapter 4. Major Architectures of Deep Networks,“ [Online]. Available: https://www.oreilly.com/library/view/deep-learning/9781491924570/ch04.html. [Zugriff am 02 08 2022].

[12] M. P. Véstias, „A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing,“ 07 2019. [Online]. Available: https://www.researchgate.net/figure/Input-and-output-feature-maps-of-a-convolutional-layer_fig1_334819564. [Zugriff am 02 08 2022].

[13] Atharva, „Classifying breast cancer tumour type using Convolutional Neural Network (CNN — Deep Learning),“ 04 10 2019. [Online]. Available: https://towardsdatascience.com/how-to-fight-breast-cancer-with-deep-learning-ab28e42e4250. [Zugriff am 02 08 2022].

[14] M. B. R. R. Anelise Medeiros Pires, „Performance assessment of bare and anti-reflective coated CdTe photovoltaic systems in comparison to multicrystalline Si in Brazil,“ *Progress in Photovoltaics Research and Applications,* 07 2021.

[15] U. E. M. M. A. L. R. d. N. R. R. Aline Kirsten Vidal de Oliveira, „Aerial Infrared Thermography of a Utility-Scale PV Plant After a Meteorological Tsunami in Brazil,“ 2018.

[16] „lindevs,“ lindevs, 24 07 2021. [Online]. Available: https://lindevs.com/apply-gamma-correction-to-an-image-using-opencv/. [Zugriff am 22 07 2022].

[17] A. F. Villán, Mastering OpenCV 4 with Python, Birmingham - Mumbai: Packt, 2019.

[18] k. &. atul, „TheAILearner - Mastering Artificial Intelligence,“ 22 11 2019. [Online]. Available: https://theailearner.com/2019/11/22/simple-shape-detection-using-contour-approximation/. [Zugriff am 27 07 2022].

[19] G. Code, „Warp Perspective with OpenCV | Document Scanner,“ 04 02 2022. [Online]. Available: https://www.youtube.com/watch?v=SQ3D1tlCtNg. [Zugriff am 24 07 2022].

[20] B. Özmen, „InsightDataScience,“ 27 03 2019. [Online]. Available: https://blog.insightdatascience.com/automl-for-data-augmentation-e87cf692c366. [Zugriff am 27 07 2022].

[21] keras documentation, „keras documentation: Model training APIs,“ [Online]. Available: https://keras.io/api/models/model_training_apis/. [Zugriff am 02 08 2022].

[22] „tutorialspoint: Keras - Model Evaluation and Model Prediction,“ [Online]. Available: https://www.tutorialspoint.com/keras/keras_model_evaluation_and_prediction.htm#:~:text=Model%20Prediction.%20Prediction%20is%20the%20final%20step%20and,None%2C%20max_queue_size%20%3D%2010%2C%20workers%20%3D%201%2C%20. [Zugriff am 02 08 2022].

7 Appendix I: How to get started with Deep Learning

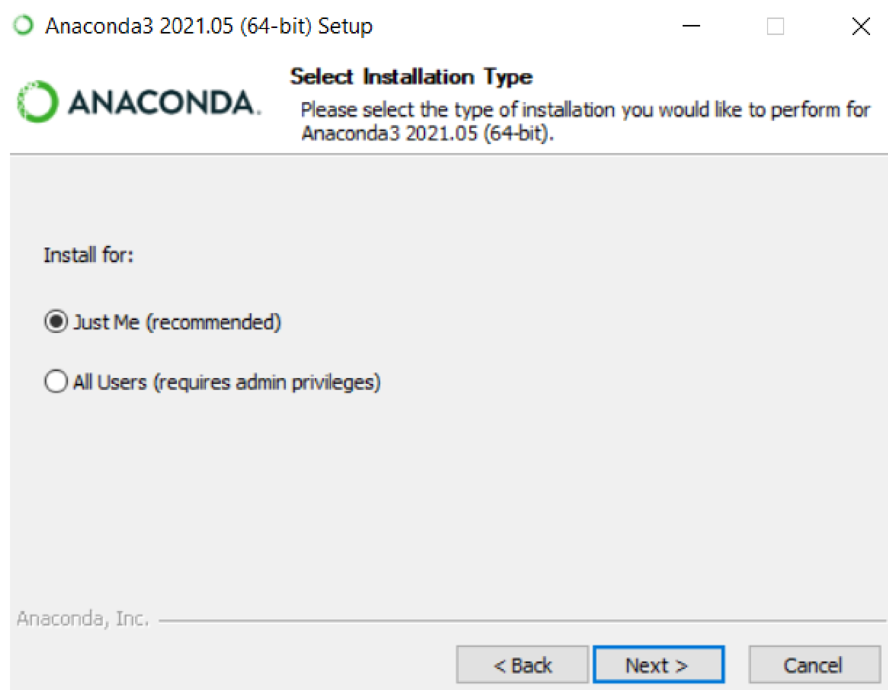
7.1 How to install Anaconda, keras and Jupyter in Windows

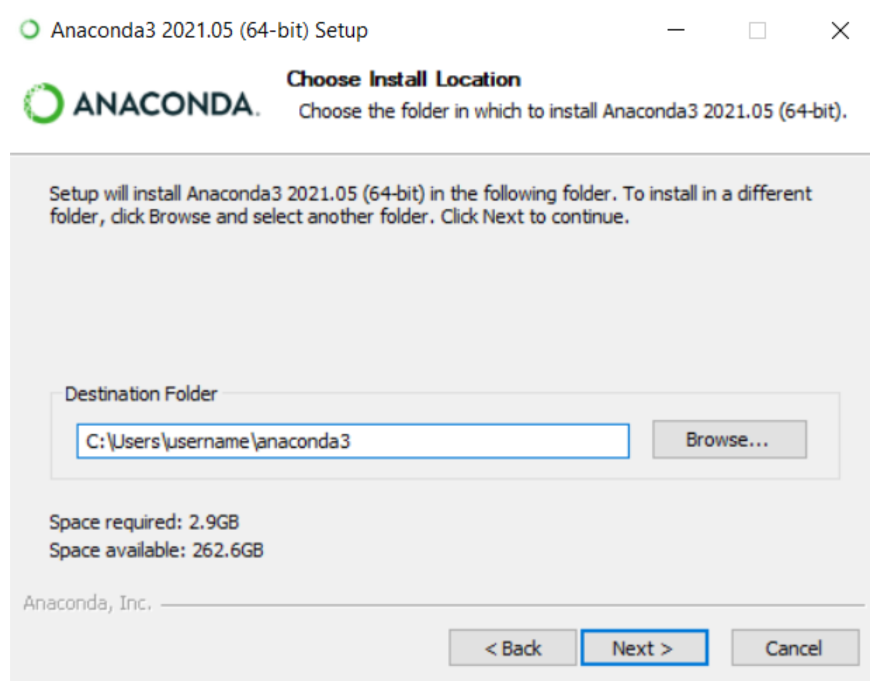
Timon Benz, written 06.10.2021, checked 06/2022

Note: These instructions are an update (adaption) of the article "How to install keras with a *tensorflow* backend for deep learning", written by Red Huq, published on the blog inmachineswetrust on 2017-07-28: How to install Keras with a TensorFlow backend for deep learning | In Machines We Trust

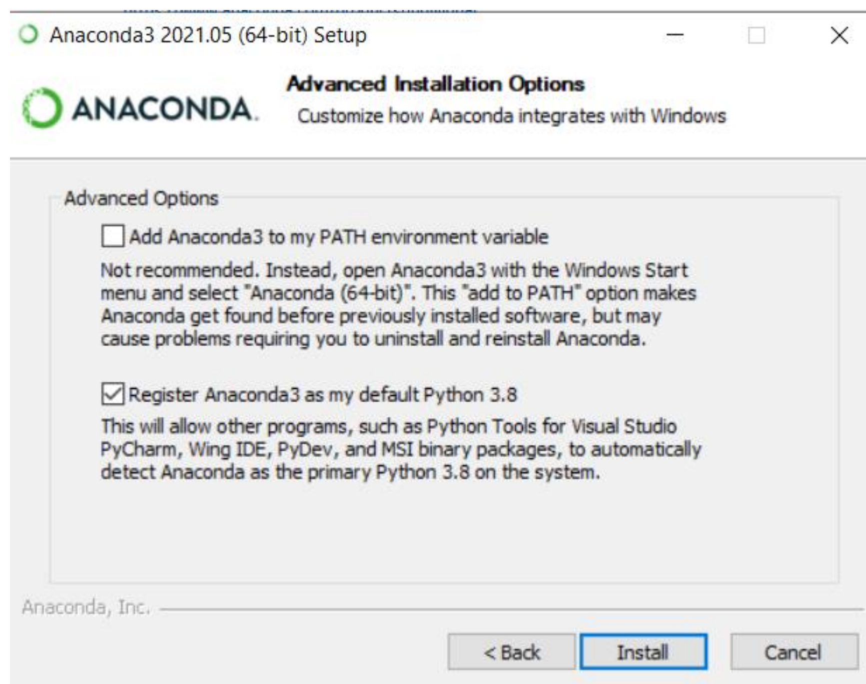
https://inmachineswetrust.com/posts/deep-learning-setup/ .

1.) **Download and install Anaconda**
https://www.anaconda.com/products/individual
download the .exe-file and execute it, you'll be asked for administrator rights





- set an easy file-path and note it down, we will need it in the next step



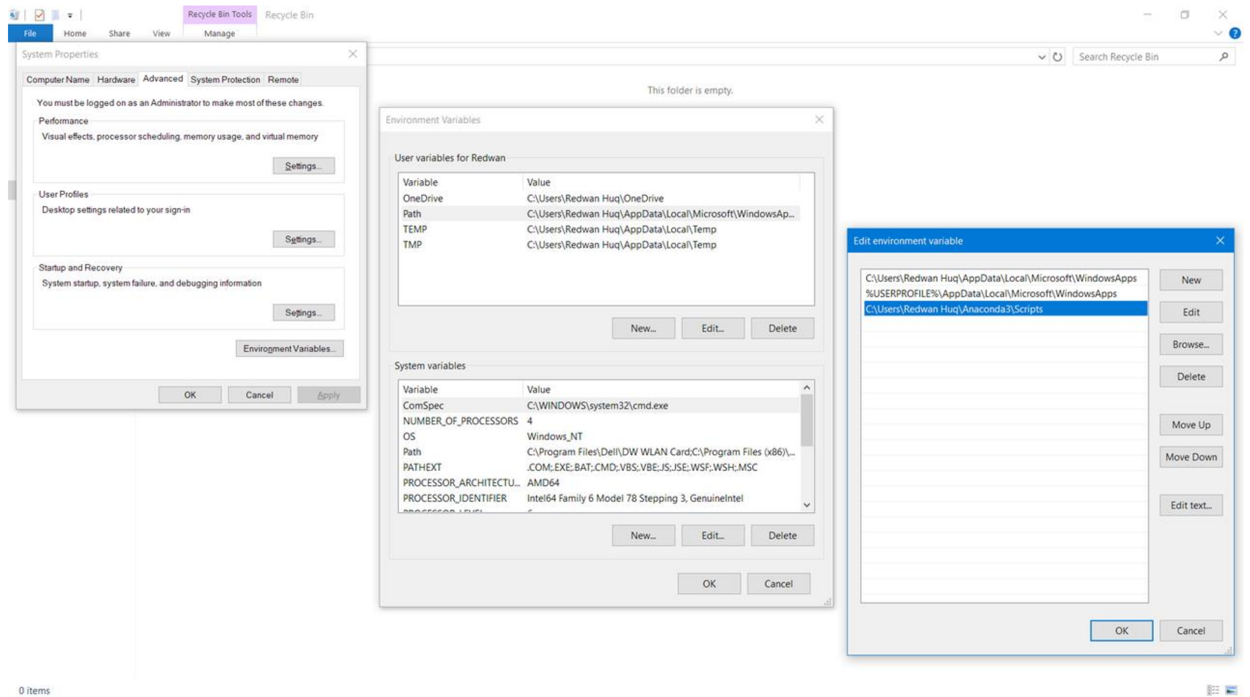
- I personally choose only the second option and the path as PATH environment variable in Windows in the next step

2.) Add Anaconda to the Windows PATH environment variables

Quote from the original blog post: “

1. Open the Start menu, start typing "environment" and select the option called *Edit the system environment variables*
2. Select the *Environment Variables* button near the bottom

3. In the top section containing user variables, select the one called *Path* and choose to edit it
4. Create a new variable whose name refers to the location of the "Scripts" folder, which is inside whichever folder you chose to install Anaconda

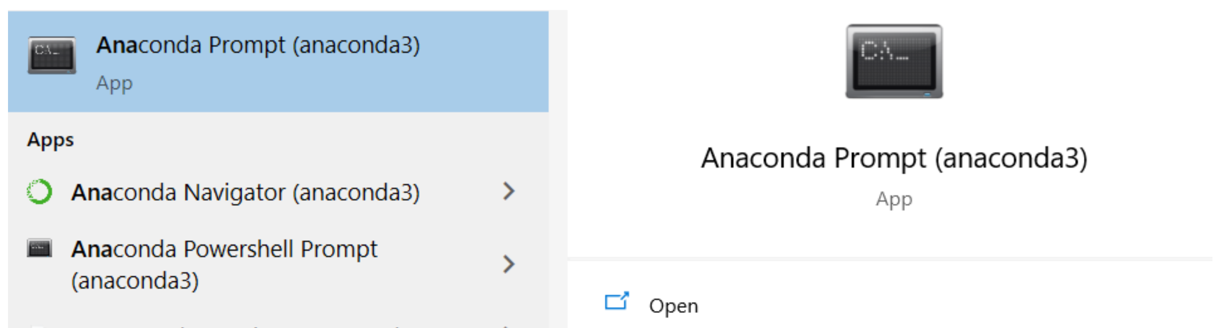


"

source: https://inmachineswetrust.com/posts/deep-learning-setup/

3.) Set up a virtual environment, install jupyter and keras (and eventually tensorflow)

Click on the Windows start menu, start typing “Anaconda” and click on “Anaconda Prompt (anaconda3)” to open the Anaconda shell.



The Anaconda Navigator can also be used for installing packages (e.g. libraries) within virtual environments. Here we are using commands for conda (a package manager) for the settings and installations.

Run the command:

conda create --name deeplearning python

In the shell, it looks like this:

```
(base) C:\Users\timon>conda create --name deeplearning python
```

After installation commands like this one conda prompt will list available packages and ask for confirmation of their installation. Type “y” and press “Enter” to confirm.

Activate the deep learning environment:

activate deeplearning

```
(base) C:\Users\timon>conda activate deeplearning

(deeplearning) C:\Users\timon>
```

The “(deeplearning)” indicates you are inside the virtual environment.

Install jupyter, an IDE (intergrated developing environment) that uses iPython notebooks – a special coding format allowing formatting and step-by-step execution of your code.

conda install jupyter

The above steps of this main step 3.) were in accordance with the instructions of Red Huq in his blog post. From here on, I proceed differently because as the packages “keras”, “tensorflow”, “jupyter” and also the “python” (language-)package are frequently updated, we need to find a combination of compatible versions. This is the “critical point” of the installation, if neither my instructions here nor the ones of Red Huq, please search for the compatibility of the packages on the internet, for example using the search terms: “keras tensorflow python compatibility”. The compatibility between these 3 packages is probably most important, and jupyter as IDE can probably substituted by other Python-IDEs.

In the present (10/2021 and 06/2022) compatibility can be achieved by downgrading Python to Python 3.6 and using the version Keras 2.3.1 . New Keras versions already contain tensorflow, so tensorflow doesn’t need to be installed separately.

conda install python=3.6

conda install keras=2.3.1

(Sidemark: Note: Red Huq`s article says to install “pandas” and “scikit-learn” using “conda” and then install keras and tensorflow using the package manager “pip”:

Original commands – due to compatibility and version changes: DON`T use these:

```
conda install pandas

conda install scikit-learn

pip install tensorflow
```

```
pip install keras
```

Because this didn't work for me, I recommend to always use conda and not to install tensorflow explicitly – it is included in new keras versions.)

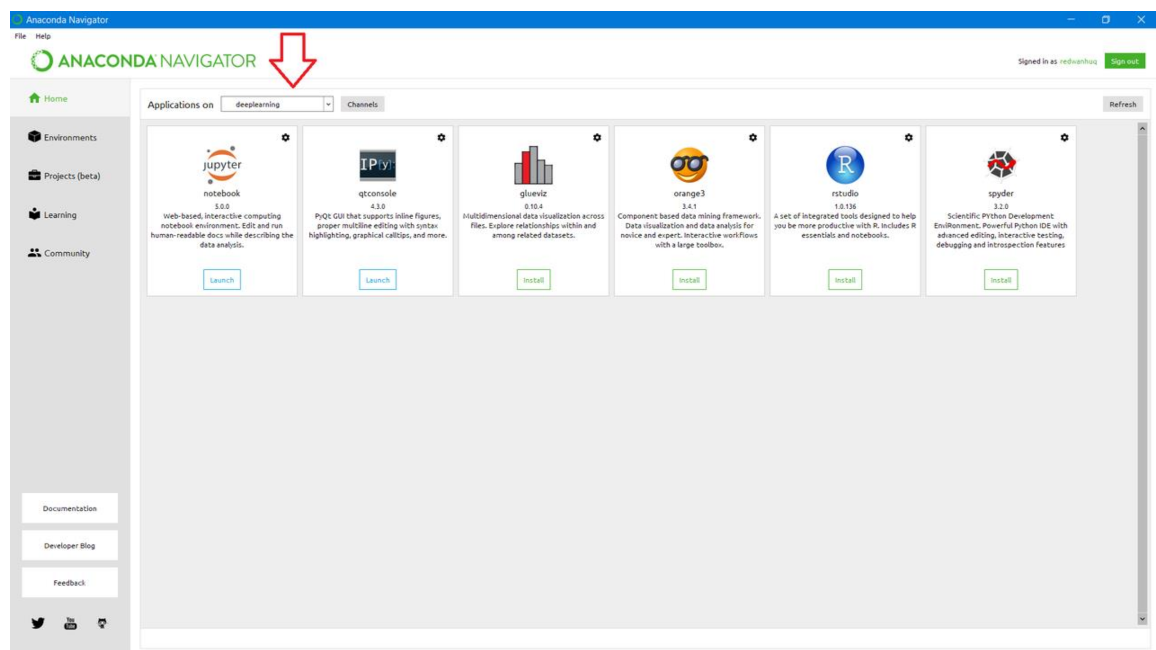
The shell listed that now (10/2021) tensorflow 2.1.0 is included within keras 2.3.1.

The last step – about getting started and opening your first Jupyter notebook from which you can use keras – is still the same as in Red Huq`s article:

4. Verifying the installation

“A quick way to check if the installation succeeded is to try to import Keras and TensorFlow in a Jupyter notebook. Here are two ways to access Jupyter:

1. Open Command prompt, activate your deep learning environment, and enter `jupyter notebook` in the prompt
2. Open Anaconda Navigator (use the Start menu shortcut), switch to your deep learning environment in the *Applications on* drop-down menu, and then choose to open Jupyter



The first option is a lot faster. If you missed a step or made a mistake, you can always remove the conda environment and start over.

```
conda remove --name deeplearning --all
```

Otherwise, you should have TensorFlow and Keras ready to go. Go forth and start and building! As always, don't hesitate to leave your comments below.“

Source: https://inmachineswetrust.com/posts/deep-learning-setup/

I personally use the first method, explicitly:

1.) Open “Anaconda Prompt (anaconda3)” via the Windows start menu.
2.) **activate deeplearning**
3.) **jupyter notebook**, this opens jupyter in your browser. Select a folder you want to save your notebook in, and press “New” – “Python 3”.

Good luck with your deep learning projects!

7.2 How to install PyCharm with an existing Anaconda environment in Windows

As the Anaconda environment was created to develop Deep Learning Codes using *keras*, a user will want to code in his/her preferred IDE (PyCharm, Spyder, …). PyCharm is a common choice and recommended, so I personally use PyCharm and describe how to code in PyCharm within the existing Anaconda environment. The steps for other IDEs might be similar.

Timon Benz, written 15.10.2021, checked 06/2022

Note: These instructions are based on the approach of the article/blog post “How to setup PyCharm with an anaconda virtual environment already created | by Aseem Bansal | towards-infinity | Medium”, written by Aseem Bansal, published on the blog medium on 2018-03-28.

1.) **Download and install PyCharm Community Edition**
Download PyCharm: Python IDE for Professional Developers by JetBrains
https://www.jetbrains.com/pycharm/download/#section=windows
download the .exe-file and execute it, you’ll be asked for administrator rights

Install PyCharm but when you are asked to connect an interpreter, don`t choose to download a Python interpreter, but choose

Existing interpreter/environment and then the folder path to the **python.exe** file inside the “deeplearning” folder created when setting up the deeplearning virtual environment using Anaconda. This is important because Python, *keras*, Tensorflow etc. must all use compatible versions, so also one single Python version compatible to all the libraries and packages!
You might be asked twice: once for a python interpreter – then choose the file path to the python.exe file inside the folder of the virtual environment. The second time, you will be asked for a virtual environment, there choose **existing environment / existing Anaconda environment** and give the file path to the anaconda environment, in our case called “deeplearning”, or to the .conda folder. In my case this field was filled automatically, I didn`t change it and it worked.

Next time I install this, I will add screenshots.

2.) Create a new project, a new python file and test whether keras and tensorflow work

Create a new project or easier: probably there will be an automatically generated project with a “main.py”. That file is a test whether the Python Interpreter works (e.g. is well connected via file path) so, please execute it. It shouldn`t give any errors, it should print the message “Hi PyCharm”.

Please create a new python file, by **right clicking on the project folder -> new -> python file** and type the following (or similar) commands to see whether *keras* and tensorflow work, too:

```
print("Testing Deeplearning libraries")

from keras import layers
```

The execution of this file should give an output equal/similar to this:

Testing Deeplearning libraries
Using TensorFlow backend.
Process finished with exit code 0

In that case, the set-up was successful! Congratulations and good luck with your Deep Learning projects!

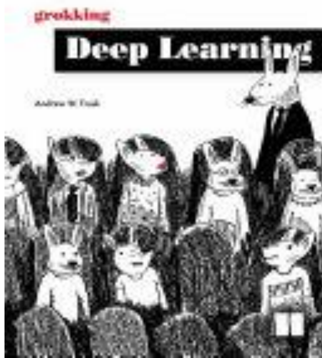
Timon Benz

7.3 Literature recommendations for Deep Learning

In order to successfully apply Deep Learning, it is essential to know at least its theoretical basics. Therefore, reading at least the first chapters of the following works is highly recommended. It is recommended to start with the following book, as it provides a stepwise and easy-to-read introduction to Neural Networks:

Grokking Deep Learning

Source of literature information



Author : Andrew Trask

Publisher : Manning Publications

Total Pages : 336

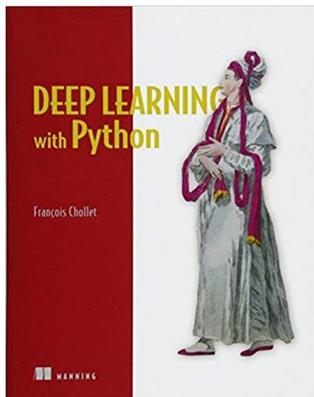
Release : 2019-01-25

ISBN 10 : 9781617293702

ISBN 13 : 1617293709

Language : EN, FR, DE, ES & NL

Having understood the basic structures and the learning algorithm, consult this book for explanations of advanced learning techniques:



François Chollet - Personal Page (fchollet.com)

access, e.g. via this link: Downloading Deep Learning with Python 1st Edition - libribook

Deep Learning with Python, written by François Chollet, 2017

François Chollet developed the *keras* library for Deep Learning and works for Google, his book is very popular in the domain of Deep Learning, many tutorials and Codes that you can find online are based on his explanations and Code examples.

Note: Personally, I use the german version “Deep Learning mit Python und keras”, published 2018, mitp. So, in case page numbers, formulations etc. differ, it is due to that.

There is also a new edition, published in 2021, which is probably even more up to date.

7.4 Video Presentation of this work and further links

A presentation of this work is available on the YouTube channel of the fotovoltaica/UFSC Solar Energy Research Laboratory, see the respective links:

Presentation Video of this thesis project on YouTube:

https://www.youtube.com/watch?v=KVd6Grq8Pdg, Please note that at that state the final results of the third script (training the CNN on cell classification) were not yet obtained. However, a general explanation of the third script is included.

YouTube Channel of the fotovoltaica/UFSC laboratory:

https://www.youtube.com/channel/UCG7j_EffB_2teLxAomPA3fA

Homepage of of the fotovoltaica/UFSC laboratory:

https://fotovoltaica.ufsc.br/sistemas/fotov/en/

8 Appendix II: Codes of the Image Processing Pipeline

8.1 Code: Module Processing Pipeline (Perspective Correction)

```
# MODULE PIPELINE

import cv2
import numpy as np
import matplotlib.pyplot as plt

def custom_imshow(title, img, perc=0.1):
   w = img.shape[1]   # original width
   h = img.shape[0]   # original height
   # downscaling of the window size & showing the original image
   WW = int(w * perc)
   HH = int(h * perc)
   cv2.namedWindow(title, cv2.WINDOW_NORMAL)
   cv2.resizeWindow(title, WW, HH)
   cv2.imshow(title, img)
   cv2.waitKey(0)

def gammaCorrection(src, gamma):
   invGamma = 1 / gamma
   table = [((i / 255) ** invGamma) * 255 for i in range(256)]
   table = np.array(table, np.uint8)
   return cv2.LUT(src, table)


def equalize_hist_color(img):
   """Equalize the image splitting the image applying cv2.equalizeHist()
    to each channel and merging the results, source:'Mastering OpenCV p.199'
"""
   channels = cv2.split(img)
   eq_channels = []
   for ch in channels:
      eq_channels.append(cv2.equalizeHist(ch))
      eq_image = cv2.merge(eq_channels)
   return eq_image


def build_kernel(kernel_type, kernel_size):
   """Creates the specific kernel: MORPH_ELLIPSE, MORPH_CROSS or MORPH_RECT"""
   if kernel_type == cv2.MORPH_ELLIPSE:
      # We build a elliptical kernel
      return cv2.getStructuringElement(cv2.MORPH_ELLIPSE, kernel_size)
   elif kernel_type == cv2.MORPH_CROSS:
      # We build a cross-shape kernel
      return cv2.getStructuringElement(cv2.MORPH_CROSS, kernel_size)
   else:   # cv2.MORPH_RECT
      # We build a rectangular kernel:
      return cv2.getStructuringElement(cv2.MORPH_RECT, kernel_size)


def dilate(image, kernel_type, kernel_size):
   """Dilates the image with the specified kernel type and size"""

   kernel = build_kernel(kernel_type, kernel_size)
   dilation = cv2.dilate(image, kernel, iterations=1)
   return dilation


def draw_contour_points(img, cnts, color, do_squeeze=True):
   """Draw all points from a list of contours"""
```

```
    for cnt in cnts:
        if do_squeeze == True:
            cnt = np.squeeze(cnt)

        for i, p in enumerate(cnt):
            p = p.reshape(1, -1)[0]  # converts array to tuple
            if len(p) == 2:
                cv2.circle(img, p, 12, color, -1)
    return img


def draw_contour_outline(img, cnts, color, thickness=1):
    """Draws contours outlines of each contour"""

    for cnt in cnts:
        cv2.drawContours(img, [cnt], 0, color, thickness)


def get_corner_points(contour):
    """This function identifies the relative positions of four corner points of
a contour,
     it takes tour points and assigns Top_left, top_right, bottom_left, bot-
tom_right to them
     using the coordinate sum (x+y) as criterion
    (x+y) is maximal at the bottom right, minimal at the top left"""
    ### ----------------- get CORNER POINT COORDINATES ---------------------###
    contour = np.squeeze(contour)
    # print("squeezed contour = ", contour)
    pointsums = []  # np.zeros((4,))
    for i in range(4):
        pointsums.append(contour[i][0] + contour[i][1])

    # extract lower right corner point coordinates
    lower_right = np.array((2,))
    max_ps = max(pointsums)
    index_lower_right = pointsums.index(max_ps)
    lower_right = contour[index_lower_right]

    # upper left corner
    upper_left = np.array((2,))
    min_ps = min(pointsums)
    index_upper_left = pointsums.index(min_ps)
    upper_left = contour[index_upper_left]

    all_indices = set([0, 1, 2, 3])
    identified = set([index_upper_left, index_lower_right])

    # difference -> remaining points
    remaining = all_indices.difference(identified)  # set of indices
    rem = np.array(list(remaining))  # list of indices

    # check location of remaining points
    point_1 = contour[rem[0]]
    point_2 = contour[rem[1]]

    if point_1[0] > point_2[0]:  # if x_coordinate of point 1 is greater than x
of point 2
        upper_right = point_1  # the first point is more to the right (top
right)
        lower_left = point_2
    else:
        upper_right = point_2
        lower_left = point_1

    # (human) CHECK whether corner coordinates are correctly assigned
    # only for debugging
```

```
    # check_img = color_image.copy()
    # check_img = cv2.putText(check_img, 'top left', upper_left, cv2.FONT_HER-
SHEY_SIMPLEX, 7, (0, 0, 255), thickness=10,
    #                         lineType=8, bottomLeftOrigin=False)
    # check_img = cv2.putText(check_img, 'top right', upper_right,
cv2.FONT_HERSHEY_SIMPLEX, 7, (0, 0, 255), thickness=10,
    #                         lineType=8, bottomLeftOrigin=False)
    # check_img = cv2.putText(check_img, 'lower left', lower_left,
cv2.FONT_HERSHEY_SIMPLEX, 7, (0, 0, 255), thickness=10,
    #                         lineType=8, bottomLeftOrigin=False)
    # check_img = cv2.putText(check_img, 'lower right', lower_right,
cv2.FONT_HERSHEY_SIMPLEX, 7, (0, 0, 255),
    #                         thickness=10,
    #                         lineType=8, bottomLeftOrigin=False)
    # check_img = cv2.drawContours(check_img, [contour], -1, (0, 255, 0), 10)
    # custom_imshow("check", check_img)
    # cv2.imwrite("coordinates check img.jpg", check_img)
    return upper_left, upper_right, lower_right, lower_left


def distance(pt1, pt2):
    import math
    dx = pt1[0] - pt2[0]
    dy = pt1[1] - pt2[1]
    t = dx * dx + dy * dy
    return math.sqrt(t)


def get_rectangle_sidelengths(upper_left, upper_right, lower_right,
lower_left):
    # get distances
    dist_x_top = distance(upper_left, upper_right)
    dist_x_bottom = distance(lower_left, lower_right)
    dx = (dist_x_bottom + dist_x_top) / 2  # average WIDTH

    dist_y_left = distance(lower_left, upper_left)
    dist_y_right = distance(lower_right, upper_right)
    dy = (dist_y_left + dist_y_right) / 2  # average HEIGHT
    return int(dx), int(dy)



def detect_contours(img, which='all', eps=0.05, iter=1, draw=False,
save=False, plots=False):
    if which == 'all':
        contours, hierarchy = cv2.findContours(img, cv2.RETR_LIST,
                                               cv2.CHAIN_APPROX_SIMPLE)  #### retrieve
ALL METHOD !
    elif which == 'external':
        contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL,
                                               cv2.CHAIN_APPROX_SIMPLE)  #### retrieve
EXTERNAL METHOD !
    nr_contours = len(contours)
    print("detected contours: '{}' ".format(nr_contours))
    color_image = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
    if draw == True:
        # Draw the outline of all detected contours:
        image_contours = color_image.copy()  # copy color image to show the re-
sults (be able to draw colored lines)
        draw_contour_outline(image_contours, contours, (0, 255, 0), 5)
        title = img_name + " - " + str(nr_contours) + " contours found in itera-
tion " + str(iter)
        custom_imshow(title, image_contours)
    if save == True:
        cv2.imwrite(title + '.jpg', image_contours)
```

```
    # CONTOUR APPROXIMATIONS - source: opencv.org tutorial py contour features
    img_contour_approximations = color_image.copy()
    cont_approx_lengths = np.zeros(len(contours) + 1)
    approx = []
    for i, cnt in enumerate(contours, 1):
        epsilon = eps * cv2.arcLength(cnt, True)
        current_approx = cv2.approxPolyDP(cnt, epsilon, True)
        if draw == True:
            cv2.drawContours(img_contour_approximations, [current_approx], 0, (0,
255, 0), 10)
            draw_contour_points(img_contour_approximations, current_approx, (255,
0, 0), do_squeeze=False)
        cont_approx_lengths[i] = len(current_approx)
        approx.append(current_approx)
    if draw == True:
        title = img_name +" "+str(len(approx)) + " contour approximations epsi-
lon " + str(eps) + " iteration " + str(iter)
        custom_imshow(title, img_contour_approximations)
        cv2.imwrite(title + '.jpg', img_contour_approximations)

    # compute average values to be returned as metrics
    total_nr_contours = nr_contours

    return total_nr_contours, approx


######################################
### OPERATION MODE - SETTINGS
EPSILON = 0.05
simple_thresh = 150
contour_limit = 240
take_green_channel = True
G_thresh = 105
SAVE = True
SHOW = False
######################################


### 1.) LOAD ORIGINAL image
img_name = 'DSC_0650'
img_original = cv2.imread(img_name + '.JPG')
custom_imshow(img_name + " original", img_original)


### 2.) BRIGHTNESS CORRECTION
gamma = 2.5   # change the value here to get different result
img_gamma = gammaCorrection(img_original, gamma=gamma)
title = img_name + "corrected with gamma = " + str(gamma)
if SHOW: custom_imshow(title, img_gamma)
if SAVE: cv2.imwrite(title+'.jpg', img_gamma)

img_gamma_equalized = equalize_hist_color(img_gamma)
title = img_name + ' gamma correction '+str(gamma)+' and color histogram
equalization'
if SHOW: custom_imshow(title, img_gamma_equalized)
if SAVE: cv2.imwrite(title+'.jpg', img_gamma_equalized)

img = img_gamma_equalized # choose with which image the processing shall con-
tinue


### 3.A) Extracting the green colour channel
B = img[:, :, 0]# splitting into colour channels
G = img[:, :, 1]
R = img[:, :, 2]
# img_without_red = img[:, :, 0]
```

```
if take_green_channel:
   custom_imshow('GREEN channel', G)
   cv2.imwrite(img_name+'_GREEN channel.jpg', G)
   #custom_imshow('RED channel', R)
   if SAVE: cv2.imwrite(img_name+'_RED channel.jpg', R)
   #custom_imshow('BLUE channel', B)
   if SAVE: cv2.imwrite(img_name+'_BLUE channel.jpg', B)
   img_name = img_name + "_G"
   simple_thresh=G_thresh
   img_gray = G


### 3.B) GRASCALE
if take_green_channel==False:
   img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
else:
   img_gray = G


### 4.) INVERTING
img = cv2.bitwise_not(img_gray)
title = img_name + " inverted grayscale after mB and dilate"
if SHOW: custom_imshow(title, img)
if SAVE: cv2.imwrite(title+'.jpg', img)

### 5.) SIMPLE THRESHOLDING
th, img_thresh = cv2.threshold(img, simple_thresh, 255, cv2.THRESH_BINARY)
title = img_name + " simple thresh "+str(simple_thresh)
custom_imshow(title, img_thresh)
if SHOW: custom_imshow(title, img_thresh)
if SAVE: cv2.imwrite(title + ".jpg", img_thresh)


### 6.)  MORPHOLOGICAL OPERATIONS LOOP
max_iter = 20
# create the kernel for smoothing images
kernel_averaging_10_10 = np.ones((10, 10), np.float32) / 100
kernel_averaging_5_5 = np.ones((5, 5), np.float32) / 25
kernel_size_3_3 = (3, 3)
mB_kernel_size = 9

img = img_thresh # choose input image
for i in range(max_iter):
   img = dilate(img, cv2.MORPH_CROSS, kernel_size_3_3)
   img = cv2.medianBlur(img, mB_kernel_size)

   total_nr_contours, contours = detect_contours(img,
                                                 which='external',
                                                 eps=EPSILON,
                                                 iter=i + 1,
                                                 draw=True,
                                                 save=True,
                                                 plots=False)
   # second ABORTION criterion
   if total_nr_contours <= contour_limit:
      print("Less than " + str(contour_limit) + " contours reached, quitting
morphology loop at iteration " + str(
          i + 1))
      stop_iter = i + 1
      break


### 7.) FILTER CONTOURS BY CRITERIA (to get the module contour)

## 7.A.) ------ 4 CORNER criterion
```

```
color_image = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
color_image_copy = color_image.copy()
cont_4_corners = []
for i, cont in enumerate(contours, 1):
   nr_corners = len(cont)   ### 4 CORNER criterion
   if nr_corners == 4:
      cv2.drawContours(color_image_copy, [cont], -1, (0, 255, 0), 20)
      draw_contour_points(color_image_copy, [cont], (255, 0, 0))
      cont_4_corners.append(cont)
nr = len(cont_4_corners)
title = str(nr) + ' contours with 4 corners after ' + str(stop_iter) + ' iter-
ations'
custom_imshow(title, color_image_copy)
cv2.imwrite(title + ".jpg", color_image_copy)
print("number of contours with 4 corners: ", nr)

## 7.B.) ------ ABSOLUTE AREA criterion: 20 to 95 % of image area
conts_possible_size = []
shape = img_original.shape  # get image_size
image_size = shape[0] * shape[1]
for cont in cont_4_corners:
   contour_area = cv2.contourArea(cont)  # contour size (area)
   if contour_area > image_size * 0.2 and contour_area < image_size * 0.95:
      conts_possible_size.append(cont)
print("number of contours between 20 and 95 % of the image area: ",
len(conts_possible_size))

## 7.C.) ------ ANGLES criterion -> parallelograms, trapezoids, rhombuses ...
from math import atan
import math

def get_angle(p1, p2):
   dx = p1[0] - p2[0]
   dy = p1[1] - p2[1]
   if dx != 0:
      alpha = atan(dy / dx)
   else:
      if dy > 0:
         alpha = math.pi / 2
      elif dy < 0:
         alpha = -math.pi / 2
      else:
         print("Error: dx = 0 and dy=0 in arctan(dy/dx) one line has length
0")
   return alpha


# set angular TOLERANCES
higher_tolerance_degrees = 80
t_h = higher_tolerance_degrees * math.pi / 180  # tolerance for higher angular
difference in rad -> between and pi (polar vectors)
lower_tolerance_degrees = 60
t_l = lower_tolerance_degrees * math.pi / 180

conts_accepted_angles = []
for cont in conts_possible_size:
   if len(cont) == 4:
      upper_left, upper_right, lower_right, lower_left = get_cor-
ner_points(cont)
      # get the 4 angles of each side with respect to positive x direction
      alpha_lower = get_angle(lower_right, lower_left)
      alpha_upper = get_angle(upper_right, upper_left)
      beta_left = get_angle(lower_right, upper_left)
      beta_right = get_angle(lower_right, upper_right)

      # get similarity measures for the angles
      beta_difference = abs((beta_left - beta_right))  # in absolute radians
```

```
      alpha_difference = abs((alpha_lower - alpha_upper))

      # rank the measures of similarities - one will (most probably) be larger
than the other
      angle_diffs = [alpha_difference, beta_difference]
      higher_difference = max(angle_diffs)
      lower_difference = min(angle_diffs)

      # FIRST pair of similar angles
      if higher_difference > -t_h and higher_difference < t_h:
         if lower_difference > -t_l and lower_difference < t_l:
            conts_accepted_angles.append(cont)

print("number of contours with acceptable angles: ", len(conts_accepted_an-
gles))

parallelograms = conts_accepted_angles

## 7.D.) -------- RELATIVE AREA CRITERION
sorted_approx = sorted(parallelograms, key=cv2.contourArea, reverse=True)  # -
SORT contours by area
if len(sorted_approx) > 1:
   module_contour = sorted_approx[0]  # extract the LARGEST CONTOUR (by area)
elif len(sorted_approx) == 1:
   module_contour = sorted_approx
else:
   print("No module contour could be found, try to lower the angular re-
striction, change thresholdig or epsilon.")


color_image_copy2 = color_image.copy()
cv2.drawContours(color_image_copy2, [module_contour[0]], -1, (0, 255, 0), 20)
draw_contour_points(color_image_copy2, [module_contour[0]], (255, 0, 0))
custom_imshow("largest contour with 4 corners", color_image_copy2)
cv2.imwrite("largest contour with 4 corners.jpg", color_image_copy2)

# SAVE the module contour (!)
np.save(img_name + "_module_contour.npy", np.asarray(sorted_approx[0]))


### 8.) ------ PERSPECTIVE CORRECTION of the MODULE image
def get_corner_points(contour):
   ### ----------------- get CORNER POINT COORDINATES ---------------------###
   contour = np.squeeze(contour)
   # print("squeezed contour = ", contour)
   pointsums = []  # np.zeros((4,))
   for i in range(4):
      pointsums.append(contour[i][0] + contour[i][1])

   # extract lower right corner point coordinates
   lower_right = np.array((2,))
   max_ps = max(pointsums)
   index_lower_right = pointsums.index(max_ps)
   lower_right = contour[index_lower_right]

   # upper left corner
   upper_left = np.array((2,))
   min_ps = min(pointsums)
   index_upper_left = pointsums.index(min_ps)
   upper_left = contour[index_upper_left]

   all_indices = set([0, 1, 2, 3])
   identified = set([index_upper_left, index_lower_right])

   # difference -> remaining points
   remaining = all_indices.difference(identified)  # set of indices
   rem = np.array(list(remaining))  # list of indices
```

```
    # check location of remaining points
    point_1 = contour[rem[0]]
    point_2 = contour[rem[1]]

    if point_1[0] > point_2[0]:  # if x_coordinate of point 1 is greater than x
of point 2
        upper_right = point_1  # the first point is more to the right (top
right)
        lower_left = point_2
    else:
        upper_right = point_2
        lower_left = point_1
    return upper_left, upper_right, lower_right, lower_left


def distance(pt1, pt2):
    import math
    dx = pt1[0] - pt2[0]
    dy = pt1[1] - pt2[1]
    t = dx * dx + dy * dy
    return math.sqrt(t)


def get_square_sidelength(upper_left, upper_right, lower_right, lower_left):
    # get distances
    dist_x_top = distance(upper_left, upper_right)
    dist_x_bottom = distance(lower_left, lower_right)
    dx = (dist_x_bottom + dist_x_top) / 2  # average WIDTH

    dist_y_left = distance(lower_left, upper_left)
    dist_y_right = distance(lower_right, upper_right)
    dy = (dist_y_left + dist_y_right) / 2  # average HEIGHT

    dxy = int((dx + dy) / 2)  # average SIDELENGTH (cells are square-shaped)
    return dxy


def get_rectangle_sidelengths(upper_left, upper_right, lower_right,
lower_left):
    # get distances
    dist_x_top = distance(upper_left, upper_right)
    dist_x_bottom = distance(lower_left, lower_right)
    dx = (dist_x_bottom + dist_x_top) / 2  # average WIDTH

    dist_y_left = distance(lower_left, upper_left)
    dist_y_right = distance(lower_right, upper_right)
    dy = (dist_y_left + dist_y_right) / 2  # average HEIGHT
    return int(dx), int(dy)


## 8.A) ------ get CORNER POINTS
contour = module_contour  # choose contour
upper_left, upper_right, lower_right, lower_left = get_corner_points(contour)

## 8.B) ----- get DISTANCES
dx, dy = get_rectangle_sidelengths(upper_left, upper_right, lower_right,
lower_left)

## 8.C) ------ PERSPECTIVE TRANSFORM
input_points = np.float32([[upper_left], [upper_right], [lower_left],
[lower_right]])
target_points = np.float32([[0, 0], [dx, 0], [0, dy], [dx, dy]])

## 8.D) ------ compute TRANSFORMATION MATRIX, input & target POINTS
r = 50  # rim width - added to ensure that the whole module, all border cells
are entirely shown
```

```
input_points_rim = np.float32([[upper_left[0] - r, upper_left[1] - r], [up-
per_right[0] + r, upper_right[1] - r],
                        [lower_left[0] - r, lower_left[1] + r],
[lower_right[0] + r, lower_right[1] + r]])
input_points = np.float32([[upper_left], [upper_right], [lower_left],
[lower_right]])
# upper left corner is taken as (0,0) in the new image's coordinate system
target_points = np.float32([[0, 0], [dx, 0], [0, dy], [dx, dy]])

## 8.E) ------ PERSPECTIVE TRANSFORM on COLOR image (original)
img_color_out = img_original.copy()
size = (dx, dy)
matrix = cv2.getPerspectiveTransform(input_points_rim, target_points)
img_color_out = cv2.warpPerspective(img_color_out, matrix, size)
custom_imshow("original_image", img_original)
custom_imshow("img_color_out", img_color_out)
cv2.imwrite(img_name + "_pers_corrected.jpg", img_color_out)

# perspective transform on COLOR image WITH RIM
img_rim_out = img_original.copy()
matrix = cv2.getPerspectiveTransform(input_points_rim, target_points)
size = (dx + 2 * r, dy + 2 * r)
img_rim_out = cv2.warpPerspective(img_rim_out, matrix, size)
custom_imshow("img_rim_out", img_rim_out)
cv2.imwrite(img_name + "_pers_corrected_with_rim.jpg", img_rim_out)
```

8.2 Code: Cell Cropping Pipeline

```
# CELL PIPELINE

### 0.) IMPORTING LIBRARIES and defining functions

import cv2
import numpy as np
import matplotlib.pyplot as plt

def custom_imshow(title, img, perc=0.1):
   w = img.shape[1]   # original width
   h = img.shape[0]   # original height
   # downscaling of the window size & showing the original image
   WW = int(w * perc)
   HH = int(h * perc)
   cv2.namedWindow(title, cv2.WINDOW_NORMAL)
   cv2.resizeWindow(title, WW, HH)
   cv2.imshow(title, img)
   cv2.waitKey(0)


def gammaCorrection(src, gamma):
   invGamma = 1 / gamma
   table = [((i / 255) ** invGamma) * 255 for i in range(256)]
   table = np.array(table, np.uint8)
   return cv2.LUT(src, table)


def equalize_hist_color(img):
   """Equalize the image splitting the image applying cv2.equalizeHist()
    to each channel and merging the results, source:'Master OpenCV p.199' """
   channels = cv2.split(img)
   eq_channels = []
   for ch in channels:
      eq_channels.append(cv2.equalizeHist(ch))
      eq_image = cv2.merge(eq_channels)
   return eq_image
```

```
def get_rectangle_sidelengths(upper_left, upper_right, lower_right,
lower_left):
   # get distances
   dist_x_top = distance(upper_left, upper_right)
   dist_x_bottom = distance(lower_left, lower_right)
   dx = (dist_x_bottom + dist_x_top) / 2  # average WIDTH

   dist_y_left = distance(lower_left, upper_left)
   dist_y_right = distance(lower_right, upper_right)
   dy = (dist_y_left + dist_y_right) / 2  # average HEIGHT
   return int(dx), int(dy)


def get_corner_points(contour):
   ### --- get CORNER POINT COORDINATES
   contour = np.squeeze(contour)
   # print("squeezed contour = ", contour)
   pointsums = []  # np.zeros((4,))
   for i in range(4):
      pointsums.append(contour[i][0] + contour[i][1])

   # extract lower right corner point coordinates
   lower_right = np.array((2,))
   max_ps = max(pointsums)
   index_lower_right = pointsums.index(max_ps)
   lower_right = contour[index_lower_right]

   # upper left corner
   upper_left = np.array((2,))
   min_ps = min(pointsums)
   index_upper_left = pointsums.index(min_ps)
   upper_left = contour[index_upper_left]

   all_indices = set([0, 1, 2, 3])
   identified = set([index_upper_left, index_lower_right])

   # difference -> remaining points
   remaining = all_indices.difference(identified)  # set of indices
   rem = np.array(list(remaining))  # list of indices

   # check location of remaining points
   point_1 = contour[rem[0]]
   point_2 = contour[rem[1]]

   if point_1[0] > point_2[0]:  # if x_coordinate of point 1 is greater than x
of point 2
      upper_right = point_1  # the first point is more to the right (top
right)
      lower_left = point_2
   else:
      upper_right = point_2
      lower_left = point_1

   # (human) CHECK whether corner coordinates are correctly assigned
   # only for debugging
   # check_img = color_image.copy()
   # check_img = cv2.putText(check_img, 'top left', upper_left, cv2.FONT_HER-
SHEY_SIMPLEX, 7, (0, 0, 255), thickness=10,
   #                        lineType=8, bottomLeftOrigin=False)
   # check_img = cv2.putText(check_img, 'top right', upper_right,
cv2.FONT_HERSHEY_SIMPLEX, 7, (0, 0, 255), thickness=10,
   #                        lineType=8, bottomLeftOrigin=False)
   # check_img = cv2.putText(check_img, 'lower left', lower_left,
cv2.FONT_HERSHEY_SIMPLEX, 7, (0, 0, 255), thickness=10,
   #                        lineType=8, bottomLeftOrigin=False)
```

```
    # check_img = cv2.putText(check_img, 'lower right', lower_right,
cv2.FONT_HERSHEY_SIMPLEX, 7, (0, 0, 255),
    #                         thickness=10,
    #                         lineType=8, bottomLeftOrigin=False)
    # check_img = cv2.drawContours(check_img, [contour], -1, (0, 255, 0), 10)
    # custom_imshow("check", check_img)
    # cv2.imwrite("coordinates check img.jpg", check_img)
    return upper_left, upper_right, lower_right, lower_left


def distance(pt1, pt2):
    import math
    dx = pt1[0] - pt2[0]
    dy = pt1[1] - pt2[1]
    t = dx * dx + dy * dy
    return math.sqrt(t)


def get_square_sidelength(upper_left, upper_right, lower_right, lower_left):
    # get distances
    dist_x_top = distance(upper_left, upper_right)
    dist_x_bottom = distance(lower_left, lower_right)
    dx = (dist_x_bottom + dist_x_top) / 2  # average WIDTH

    dist_y_left = distance(lower_left, upper_left)
    dist_y_right = distance(lower_right, upper_right)
    dy = (dist_y_left + dist_y_right) / 2  # average HEIGHT

    dxy = int((dx + dy) / 2)  # average SIDELENGTH (cells are square-shaped)
    return dxy


def build_kernel(kernel_type, kernel_size):
    """Creates the specific kernel: MORPH_ELLIPSE, MORPH_CROSS or MORPH_RECT"""

    if kernel_type == cv2.MORPH_ELLIPSE:
        # We build a elliptical kernel
        return cv2.getStructuringElement(cv2.MORPH_ELLIPSE, kernel_size)
    elif kernel_type == cv2.MORPH_CROSS:
        # We build a cross-shape kernel
        return cv2.getStructuringElement(cv2.MORPH_CROSS, kernel_size)
    else:  # cv2.MORPH_RECT
        # We build a rectangular kernel:
        return cv2.getStructuringElement(cv2.MORPH_RECT, kernel_size)


def dilate(image, kernel_type, kernel_size):
    """Dilates the image with the specified kernel type and size"""

    kernel = build_kernel(kernel_type, kernel_size)
    dilation = cv2.dilate(image, kernel, iterations=1)
    return dilation


def draw_contour_points(img, cnts, color, do_squeeze=True):
    """Draw all points from a list of contours"""

    for cnt in cnts:
        if do_squeeze == True:
            cnt = np.squeeze(cnt)

        for i, p in enumerate(cnt):
            p = p.reshape(1, -1)[0]  # converts array to tuple
            if len(p) == 2:
                cv2.circle(img, p, 12, color, -1)
    return img
```

```
def draw_contour_outline(img, cnts, color, thickness=1):
   """Draws contours outlines of each contour"""

   for cnt in cnts:
      cv2.drawContours(img, [cnt], 0, color, thickness)


### 1.) GENERAL SETTINGS and LOADING the ORIGINAL image
## LOAD the ORIGINAL image
img_name = 'DSC_0650_G_pers_corrected_with_rim'
img_original = cv2.imread(img_name + '.JPG')
custom_imshow("original", img_original)
img_name = "DSC_0650_G" # short name for tiltes, headings and filenames

## GENERAL SETTINGS
SAVE = True
import os
path = img_name + " cropping cells"
if SAVE:
   try:
      os.mkdir(path)
   except:
      print("folder for cropping cell images already exists, program contin-
ues")
path = path + "/"
take_green_channel = False
G_thresh = 105

### 2.) BRIGHTNESS CORRECTION
gamma = 2.5  # change the value here to get different result
img_gamma = gammaCorrection(img_original, gamma=gamma)
custom_imshow("corrected with gamma =" + str(gamma), img_gamma)
if SAVE: cv2.imwrite(path + img_name + ' gamma correction 2_5.jpg', img_gamma)

img_gamma2_5_equalized = equalize_hist_color(img_gamma)
custom_imshow('gamma=2.5 and color histogram equalization',
img_gamma2_5_equalized)
if SAVE: cv2.imwrite(path + img_name + ' gamma correction 2_5 and color hist
equalized.jpg', img_gamma2_5_equalized)

# CHOOSE
img = img_gamma2_5_equalized


### 3.) GREEN COLOUR CHANNEL or  GRASCALING

### 3.A) Extracting the GREEN COLOUR CHANNEL
B = img[:, :, 0]  # splitting into colour channels
G = img[:, :, 1]
R = img[:, :, 2]
# img_without_red = img[:, :, 0]

if take_green_channel:
   custom_imshow('GREEN channel', G)
   cv2.imwrite(img_name + '_GREEN channel.jpg', G)
   # custom_imshow('RED channel', R)
   if SAVE: cv2.imwrite(img_name + '_RED channel.jpg', R)
   # custom_imshow('BLUE channel', B)
   if SAVE: cv2.imwrite(img_name + '_BLUE channel.jpg', B)
   img_name = img_name + "_G"
   simple_thresh = G_thresh
   img_gray = G

### 3.B) GRAYSCALING (alternative to 3.A)
img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

```
custom_imshow("grayscale after mB and dilate", img_gray)
if SAVE: cv2.imwrite(path + img_name + " grayscale after mB and dilate.jpg",
img_gray)

### 4.) INVERTING
img = cv2.bitwise_not(img_gray)
custom_imshow("inverted grayscale after mB and dilate", img)
if SAVE: cv2.imwrite(path + img_name + " inverted grayscale after mB and di-
late.jpg", img)

### 5.) SIMPLE THRESHOLDING
th, img_thresh_150 = cv2.threshold(img, 150, 255, cv2.THRESH_BINARY)
custom_imshow("img_thresh_150", img_thresh_150)
if SAVE: cv2.imwrite(path + img_name + " img_thresh_150.jpg", img_thresh_150)

# CHOOSE
img = img_thresh_150

### 6.)  CONTOUR DETECTION AND MORPHOLOGICAL OPERATIONS
kernel_averaging_10_10 = np.ones((10, 10), np.float32) / 100
kernel_averaging_5_5 = np.ones((5, 5), np.float32) / 25
kernel_size_3_3 = (3, 3)
kernel_size_5_5 = (5, 5)


def detect_contours(img, which='all', eps=0.05, iter=1, draw=False,
save=False, plots=False):
   if which == 'all':
      contours, hierarchy = cv2.findContours(img, cv2.RETR_LIST,
                                              cv2.CHAIN_APPROX_SIMPLE)   #### retrieve
ALL METHOD !
   elif which == 'external':
      contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL,
                                              cv2.CHAIN_APPROX_SIMPLE)   #### retrieve
EXTERNAL METHOD !
   nr_contours = len(contours)
   print("detected contours: '{}' ".format(nr_contours))
   color_image = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
   if draw == True:
      # Draw the outline of all detected contours:
      image_contours = color_image.copy()   # copy color image to show the re-
sults (be able to draw colored lines)
      draw_contour_outline(image_contours, contours, (0, 255, 0), 5)
      title = str(nr_contours) + " contours found in iteration " + str(iter)
      custom_imshow(title, image_contours)
      if SAVE: cv2.imwrite(path + img_name + " " + title + '.jpg', image_con-
tours)

   ### SORT & FILTER CONTOURS ###
   # SORT contours by AREA
   sorted_contours = sorted(contours, key=cv2.contourArea, reverse=True)

   # get contour lengths, areas and perimeters
   cont_lengths = np.zeros((len(contours) + 1,))
   cont_areas = np.zeros((len(contours) + 1,))
   cont_perimeters = np.zeros((len(contours) + 1,))
   for i, cont in enumerate(sorted_contours, 1):
      cont_lengths[i] = len(cont)
      cont_areas[i] = cv2.contourArea(cont)
      cont_perimeters[i] = cv2.arcLength(cont, True)

   if plots:
      # PLOT contour lengths vs. contour area
      plt.plot(cont_lengths)
      plt.ylabel('contour length')
      plt.xlabel('contour (by descending area)')
      title = "Contour lengths vs. area at iteration " + str(iter)
```

```
        plt.title(title)
        plt.show()

    # CONTOUR APPROXIMATIONS - idea:Amanda, Aline, source: opencv.org tutorial
py contour features
    img_contour_approximations = color_image.copy()
    cont_approx_lengths = np.zeros(len(contours) + 1)
    approx = []
    for i, cnt in enumerate(contours, 1):
        epsilon = eps * cv2.arcLength(cnt, True)
        current_approx = cv2.approxPolyDP(cnt, epsilon, True)
        if draw == True:
            cv2.drawContours(img_contour_approximations, [current_approx], 0, (0,
255, 0), 10)
            draw_contour_points(img_contour_approximations, current_approx, (255,
0, 0), do_squeeze=False)
        cont_approx_lengths[i] = len(current_approx)
        approx.append(current_approx)
    if draw == True:
        title = str(len(approx)) + " contour approximations epsilon " + str(eps)
+ " iteration " + str(iter)
        custom_imshow(title, img_contour_approximations)
        if SAVE: cv2.imwrite(path + img_name + " " + title + '.jpg', img_con-
tour_approximations)

    if plots == True:
        # PLOT contour lengths vs. contour area - APPROXIMATED contours
        sorted_approx = sorted(approx, key=cv2.contourArea, reverse=True)
        plt.plot(cont_approx_lengths)
        plt.ylabel('approximated contour length')
        plt.xlabel('contour (by descending area)')
        title = "Lengths of approximated contours vs. contour area epsilon " +
str(eps) + " iteration " + str(iter)
        plt.title(title)
        plt.show()

    # compute average values to be returned as metrics
    # total number of contours
    total_nr_contours = nr_contours
    nr_approx_contours = len(approx)

    # nr_corners_per_contour (for approximated contours)
    corners_per_contour = sum(cont_approx_lengths) / len(approx)

    return total_nr_contours, nr_approx_contours, corners_per_contour, approx


max_iter = 20

# initialize history arrays
hist_total_nr_contours = np.zeros((max_iter))
hist_nr_approx_contours = np.zeros((max_iter,))
hist_corners_per_contour = np.zeros((max_iter,))

hist_nr_of_cells = np.zeros((max_iter,))
hist_nr_4_cornered_contours = np.zeros((max_iter,))
hist_nr_parallelograms = np.zeros((max_iter,))

mB_kernel_size = 9
kernel_size = kernel_size_3_3
for i in range(max_iter):
    if i > 0:
        img = dilated_33_sp

    dilated_33 = dilate(img, cv2.MORPH_CROSS, kernel_size)

    dilated_33_sp = cv2.medianBlur(dilated_33, mB_kernel_size)
```

```
   detecting = True  # if true: do morphology and contours detection alongside
-> recommended to observe the decrease of contours
   if detecting == True:
      if i == 0:  # draw plots and show images at the FIRST iteration
         total_nr_contours, nr_approx_contours, corners_per_contour, contours
= detect_contours(dilated_33_sp,

which='external',

iter=i + 1,

draw=True,

save=True,

plots=False)
      elif i == (max_iter - 1):
         total_nr_contours, nr_approx_contours, corners_per_contour, contours
= detect_contours(dilated_33_sp,

which='external',

iter=i + 1,

draw=True,

save=False,

plots=False)
      else:  # draw plots and show images at the LAST iteration
         total_nr_contours, nr_approx_contours, corners_per_contour, contours
= detect_contours(dilated_33_sp,

which='external',

iter=i + 1,

draw=False,

save=False,

plots=False)

      # create arrays of the metrics
      hist_total_nr_contours[i] = total_nr_contours
      hist_nr_approx_contours[i] = nr_approx_contours
      hist_corners_per_contour[i] = corners_per_contour

      # ABORTION CRITERION = include  CELL CONTOUR CRITERIA ???:

      ### 6.) FILTERING CELL CONTOURS (get true cell contours only)
      if nr_approx_contours <= 250:

         ## 6.A.) Check CELL contour criteria
         # ------ 4 CORNER criterion
         cont_4_corners = []
         for cont in contours:  ### 4 CORNER criterion
            if len(cont) == 4:
               cont_4_corners.append(cont)
         nr_4_cornered_contours = len(cont_4_corners)
         hist_nr_4_cornered_contours[i] = nr_4_cornered_contours

         ## 6.B.) ------ SIDELENGTH criterion
         parallelograms = []
         for cont in cont_4_corners:
```

```
            upper_left, upper_right, lower_right, lower_left = get_cor-
ner_points(cont)
            dx, dy = get_rectangle_sidelengths(upper_left, upper_right,
lower_right, lower_left)  # get distances
            t = 0.4  # tolerance - relative difference permitted between
height and width -> 'rectangularity'
            if dx > dy * (1 - t) and dx < dy * (1 + t):
               parallelograms.append(cont)
         nr_parallelograms = len(parallelograms)
         hist_nr_parallelograms[i] = nr_parallelograms

         ## 6.C.) ------ AREA criterion
         cells = []
         # get image_size = module_size, and compute cell size
         shape = img_original.shape
         module_size = shape[0] * shape[1]
         nr_cells_per_module = 60
         cell_area = int(module_size / nr_cells_per_module)
         t_a = 0.4  # cell area tolerance
         for cont in parallelograms:
            contour_area = cv2.contourArea(cont)
            if contour_area > cell_area * (1 - t_a) and contour_area <
cell_area * (1 + t_a):
               prev_cell_contours = cells
               cells.append(cont)
         nr_cells = len(cells)
         hist_nr_of_cells[i] = nr_cells

         # COMBINATION of criteria => CELL CRITERION
         # when the number of detected cell contours stagnates, stop morpho-
logical loop
         # t_c = 0.05
         stop_iter = 0
         if i >= 1:
            if nr_cells < hist_nr_of_cells[i - 1]:  # only select, don't de-
stroy contours
               stop_iter = i  # also saved for title strings of plots/images
               cells = prev_cell_contours
               print("Quitting morphology loop at iteration " +
str(stop_iter))
               print("because the iteration " + str(stop_iter + 1) + " de-
stroyed cell contours.")
               break
         color_image = cv2.cvtColor(dilated_33_sp, cv2.COLOR_GRAY2RGB)
         color_image_copy = color_image.copy()
         for j, cont in enumerate(cells, 1):
            cv2.drawContours(color_image_copy, [cont], -1, (0, 255, 0), 20)
            draw_contour_points(color_image_copy, [cont], (255, 0, 0))
         nr = len(cells)
         title = str(nr) + ' contours fulfilling the contour criteria after '
+ str(stop_iter) + ' iterations'
         custom_imshow(title, color_image_copy)
         if SAVE: cv2.imwrite(path + img_name + " " + title + ".jpg",
color_image_copy)

plot_hist = False
if plot_hist == True:
   ### plot HISTORY ###
   plt.plot(hist_total_nr_contours)
   plt.ylabel('total nr of contours')
   plt.xlabel('iterations of morphological operation(s)')
   plt.title('History of average contour lengths vs. morphological ops itera-
tions')
   plt.show()

   plt.plot(hist_nr_approx_contours)
   plt.ylabel('total nr of APPROXIMATED contours')
```

```
    plt.xlabel('iterations of morphological operation(s)')
    plt.title('History of average APPROX-contour lengths vs. morphological ops
iterations')
    plt.show()

    plt.plot(hist_corners_per_contour)
    plt.ylabel('average nr of corners per contour')
    plt.xlabel('iterations of morphological operation(s)')
    plt.title('History of average nr of corners per contour vs. morphological
ops iterations')
    plt.show()

    plt.plot(hist_nr_4_cornered_contours)
    plt.ylabel('nr_4_cornered_contours')
    plt.xlabel('iterations of morphological operation(s)')
    plt.title('hist_nr_4_cornered_contours vs. morphological ops iterations')
    plt.show()

    plt.plot(hist_nr_parallelograms)
    plt.ylabel('hist_nr_parallelograms')
    plt.xlabel('iterations of morphological operation(s)')
    plt.title('hist_nr_parallelograms vs. morphological ops iterations')
    plt.show()

    plt.plot(hist_nr_of_cells)
    plt.ylabel('hist_nr_of_cells')
    plt.xlabel('iterations of morphological operation(s)')
    plt.title('hist_nr_of_cells vs. morphological ops iterations')
    plt.show()

### 8.) CROPPING EACH CELL by perspective transformation func-
tions
#
# CREATE FOLDERS to SAVE CELL IMAGES
saving_cell_imgs = True
if saving_cell_imgs:
    import os

    color_folder_path = img_name + " color cell images"
    try:
        os.mkdir(color_folder_path)
    except:
        print("folder for color cell images already exists, program continues")

    binary_folder_path = img_name + " binary cell images"
    try:
        os.mkdir(binary_folder_path)
    except:
        print("folder for binary cell images already exists, program continues")

# CROP CELL IMAGES one by one using the perspective correction functions
for cell_index, contour in enumerate(cells):
    ## --- get CORNER POINTS
    upper_left, upper_right, lower_right, lower_left = get_corner_points(con-
tour)

    ## --- get DISTANCES
    dx, dy = get_rectangle_sidelengths(upper_left, upper_right, lower_right,
lower_left)
    # transform the (slight) rectangle into a square -> slight rectangularity
due to imperfect module image sizing -> relation 6/10 cells height/width
    dxy = int((dx + dy) / 2)

    ## --- PERSPECTIVE TRANSFORM
    # upper left corner is taken as (0,0) in the new image's coordinate system
    input_points = np.float32([[upper_left], [upper_right], [lower_left],
[lower_right]])
```

```
   target_points = np.float32([[0, 0], [dxy, 0], [0, dxy], [dxy, dxy]])

   ## --- perspective transform on BINARY image
   img_bin_module = img_thresh_150.copy()
   matrix = cv2.getPerspectiveTransform(input_points, target_points)
   img_bin_out = cv2.warpPerspective(img_bin_module, matrix, (dxy, dxy))
   title = img_name + " binary cell " + str(cell_index)
   custom_imshow(title, img_bin_out)
   if saving_cell_imgs: cv2.imwrite(binary_folder_path + "/" + title + '.jpg', 
img_bin_out)

   ## --- perspective transform on COLOR image (original)
   img_color = img_original.copy()
   img_color_out = cv2.warpPerspective(img_color, matrix, (dxy, dxy))
   title = img_name + " color cell " + str(cell_index)
   custom_imshow(title, img_color_out)
   if saving_cell_imgs: cv2.imwrite(color_folder_path + "/" + title + '.jpg', 
img_color_out)
```

8.3 Code: Training a CNN on cell image classification

The code is based on code examples for image classification in [8].

```
import os
################## global variables ###########################
images_per_class = 35
classes_list = ["intact", "defect"]
main_dir = "C:/Users/timon/Documents/UVF_cells_classification"
original_dir = os.path.join(main_dir, 'original')
images_per_batch = 2
################################################################

### STEP 1: Splitting the cropped images into train, validation 
and test data
# import libraries and create access to the original data folder
import os, shutil

original_directories = []
for classname in classes_list:
   original_directories.append(os.path.join(original_dir, classname))  # list 
the paths of source directories

# create base directories for training, validation and test images
train_dir = os.path.join(main_dir, 'train')
validation_dir = os.path.join(main_dir, 'validation')
test_dir = os.path.join(main_dir, 'test')
os.mkdir(train_dir)
os.mkdir(validation_dir)
os.mkdir(test_dir)

# create a dictionary assigning class names to class indices:
classes_dict = {}
i = 0
for word in classes_list:
   classes_dict[word] = i
   i = i + 1

# create one directory for each class and dataset (train, test, validate)
train_directories = [' '] * len(classes_list) # initialization
validation_directories = [' '] * len(classes_list)
test_directories = [' '] * len(classes_list)
i = 0
```

```
# create directories to classes within train, val, test folders
for classname in classes_list:
   train_directories[i] = os.path.join(train_dir, classes_list[i])  # - TRAIN-
ING images
   os.mkdir(train_directories[i])
   validation_directories[i] = os.path.join(validation_dir, classes_list[i])
# - VALIDATION images
   os.mkdir(validation_directories[i])
   test_directories[i] = os.path.join(test_dir, classes_list[i])  # TESTING
images
   os.mkdir(test_directories[i])
   i = i + 1

# copy images to their destination folder
for classname in classes_list:
   # copy 60 % of the images to the TRAIN directories
   percentage_training = 0.65
   max_train = int(percentage_training * images_per_class) + 1
   fnames = [classname+" ({}).jpg".format(i) for i in range(1, max_train)]

   for fname in fnames:
      src = os.path.join(original_directories[classes_dict[classname]], fname)
# source path
      dst = os.path.join(train_directories[classes_dict[classname]], fname)  #
destination path
      shutil.copyfile(src, dst)  # copy

   # copy 20 % of the images to the VALIDATION directories
   max_val = int(0.85 * images_per_class) + 1
   fnames = [classname+" ({}).jpg".format(i) for i in range(max_train,
max_val)]

   for fname in fnames:
      src = os.path.join(original_directories[classes_dict[classname]], fname)
# source path
      dst = os.path.join(validation_directories[classes_dict[classname]],
fname)  # destination path
      shutil.copyfile(src, dst)  # copy

   # copy 15% to the TEST directories
   fnames = [classname+" ({}).jpg".format(i) for i in range(max_val, im-
ages_per_class)]

   for fname in fnames:
      src = os.path.join(original_directories[classes_dict[classname]], fname)
# source path
      dst = os.path.join(test_directories[classes_dict[classname]], fname)  #
destination path
      shutil.copyfile(src, dst)  # copy

   print("Images of " + classname + " successfully preprocessed.")

# testing whether all images are imported correctly
print("TRAINING")
for classname in classes_list:
   print("Number of image in folder <" + classname + "> = ",
len(os.listdir(train_directories[classes_dict[classname]])))
print("VALIDATION")
for classname in classes_list:
   print("Number of image in folder <" + classname + "> = ",
len(os.listdir(validation_directories[classes_dict[classname]])))
print("TESTING")
for classname in classes_list:
   print("Number of image in folder <" + classname + "> = ",
len(os.listdir(test_directories[classes_dict[classname]])))
```

```
### STEP 2: create Image Data Generators
# creating image data generators that apply data augmentation
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1. / 255,
                                   rotation_range=40,  # data generator with
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,  # data augmentation
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=False,
                                   )
train_generator = train_datagen.flow_from_directory(train_dir,  # configure
the train generator
                                                    target_size=(150, 150),
                                                    batch_size=images_per_batch,
                                                    class_mode='categorical')
validation_datagen = ImageDataGenerator(rescale=1. / 255)  # simple data gen-
erator
validation_generator = validation_datagen.flow_from_directory(validation_dir,
# configuring
                                                              target_size=(150, 150),
                                                              batch_size=images_per_batch,
                                                              class_mode='categorical')
test_datagen = ImageDataGenerator(rescale=1. / 255,
                                  dtype="float32")  # simple data generator
test_generator = test_datagen.flow_from_directory(test_dir,  # configuring
                                                  target_size=(150, 150),
                                                  batch_size=1,
                                                  class_mode='categorical'
                                                  )
print("Data generators successfully created.")


### STEP 3: Load & modify a pretrained neural network
from keras.applications import VGG16
from keras import optimizers
from keras import models
from keras import layers
import os

conv_base = VGG16(weights='imagenet',  # Initializing/loading pretrained
weights
              include_top=False,  # whether the classifier is to be included
              input_shape=(150, 150, 3))  # shape of the input image tensor

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(len(classes_list), activation='softmax'))

# freeze the convolutional base # optionally, not recommended here
#print("Number of weights before freezing weights: ", len(model.traina-
ble_weights))
#conv_base.trainable = False
#print("Number of trainable weights after freezing weights: ",
len(model.trainable_weights))


### STEP 4.) TRAIN and SAVE the CNN model
model.compile(loss='categorical_crossentropy',
           optimizer=optimizers.RMSprop(lr=1e-5),
           metrics=['acc'])
```

```
history = model.fit_generator(train_generator,
                              steps_per_epoch=int(max_train/images_per_batch),
                              epochs=10,
                              validation_data=validation_generator,
                              validation_steps=int(max_val-max_train)/im-
ages_per_batch)

### save the model
model.save('cell_image_classification_model.h5')


### STEP 5.)  Evaluate the training progress
# use the pyplot library to draw diagrams of the training progress and test
results
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_loss']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='training')
plt.plot(epochs, val_acc, 'r', label='validation')
plt.title('Correct classification rate training/validation')
plt.legend()
plt.xlabel('epochs')
plt.ylabel('correct classification rate')
plt.figure()
plt.plot(epochs, loss, 'bo', label='loss training')
plt.plot(epochs, val_loss, 'r', label='loss validation')
plt.title('Loss function value training/validation')
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss function value')
plt.show()

### STEP 6: Evaluate the model on test data
test_datagen = ImageDataGenerator(rescale=1. / 255)
test_generator = test_datagen.flow_from_directory(test_dir,
                                                  target_size=(150, 150),
                                                  batch_size=10,
                                                  class_mode='categorical')
test_loss, test_acc = model.evaluate_generator(test_generator, steps=int(im-
ages_per_class-max_val))
print('Correct classification rate on test data:', test_acc)
```

The code should yield an ouput similar to the following, as well as the graphs shown in Figure 42.

```
Images of intact successfully preprocessed.
Images of defect successfully preprocessed.
TRAINING
Number of image in folder <intact> =  22
Number of image in folder <defect> =  22
VALIDATION
Number of image in folder <intact> =  7
Number of image in folder <defect> =  7
TESTING
Number of image in folder <intact> =  5
Number of image in folder <defect> =  5
[…]
Found 44 images belonging to 2 classes.
```

```
Found 14 images belonging to 2 classes.
Found 10 images belonging to 2 classes.

Data generators successfully created.

Epoch 1/10
11/11 [==============================] - 6s 457ms/step - loss: 0.7319 - acc: 0.5000 -
val_loss: 0.7654 - val_acc: 0.3750

Epoch 2/10
11/11 [==============================] - 5s 438ms/step - loss: 0.6917 - acc: 0.5909 -
val_loss: 0.6396 - val_acc: 0.5000

Epoch 3/10
11/11 [==============================] - 5s 440ms/step - loss: 0.6420 - acc: 0.6364 -
val_loss: 0.7616 - val_acc: 0.2500

Epoch 4/10
11/11 [==============================] - 5s 473ms/step - loss: 0.6928 - acc: 0.4545 -
val_loss: 0.5259 - val_acc: 0.6250

Epoch 5/10
11/11 [==============================] - 5s 490ms/step - loss: 0.5525 - acc: 0.8182 -
val_loss: 0.8203 - val_acc: 0.3750

Epoch 6/10
11/11 [==============================] - 5s 474ms/step - loss: 0.5767 - acc: 0.6364 -
val_loss: 0.5103 - val_acc: 0.8750

Epoch 7/10
11/11 [==============================] - 5s 474ms/step - loss: 0.5659 - acc: 0.7273 -
val_loss: 0.3665 - val_acc: 0.8750

Epoch 8/10
11/11 [==============================] - 5s 470ms/step - loss: 0.5534 - acc: 0.7273 -
val_loss: 0.3211 - val_acc: 0.8750

Epoch 9/10
11/11 [==============================] - 5s 470ms/step - loss: 0.5768 - acc: 0.6364 -
val_loss: 0.2533 - val_acc: 1.0000

Epoch 10/10
11/11 [==============================] - 5s 468ms/step - loss: 0.3831 - acc: 0.9545 -
val_loss: 0.2822 - val_acc: 0.8750

Found 10 images belonging to 2 classes.

Correct classification rate on test data: 0.699999988079071
```

Declaration

I hereby declare that I created the present Bachelor thesis myself. Only explicitly named sources and auxiliary means were used. Literary property that was literally or correspondingly quoted here is designated as such. The present work has not been presented before, neither to the *HRW - Hochschule Ruhr West University of Applied Sciences* nor to any other scientific university.

Bilbao, 02/08/2022

place, date

signature