Hochschule Rhein-Waal
Rhine-Waal University of Applied Sciences
Faculty of Communication and Environment

Professor Dr.-Ing. Rolf Becker
Professor Dr. Daniela Lud

Object Detection based on Convolutional Neural Networks of *Senecio jacobaea* for Weed Control

Bachelor Thesis

by
Jonas Moritz Zender

Hochschule Rhein-Waal
Rhine-Waal University of Applied Sciences
Faculty of Communication and Environment

Professor Dr.-Ing. Rolf Becker
Professor Dr. Daniela Lud

Object Detection based on Convolutional Neural Networks of *Senecio jacobaea* for Weed Control

A Thesis Submitted in
Partial Fullfillment of the
Requirements of the Degree of

Bachelor of Science
in
Environment and Energy

by
Jonas Moritz Zender
Matriculation number: 21125

Submission date:
14th of September, 2021

Abstract

Senecio jacobaea massively spread in Germany during recent years causing substantial economic damages to farmers due to its hepatotoxic pyrrolizidine alkaloids leading to liver failure of cattle and horses. Regulation of *Senecio jacobaea* in environmentally protected areas is often done by systematic mowing or digging the plants out manually since broad application of herbicides is not possible. Temporal and financial expenses of farmers could be drastically reduced by methods of precision agriculture utilizing spray drones to apply small amounts of herbicide directly onto the plant at an early growth stage where the specimens are more responsive to the herbicide. This requires the automatic detection of *Senecio jacobaea* in images collected by the drone which is the topic of this project. A dataset containing 2128 images with 5723 specimens of *Senecio jacobaea* was collected, annotated, and used to train a pretrained SSD-MobileNet-v1 which is a single-stage object detector. The model yielded a mean average precision (mAP) of 21.93% showing that it is possible to detect *Senecio jacobaea* from background vegetation of similar appearance. It was found that halving the training dataset size reduced the model performance by a factor of 7.61 and that small specimens were detected much less reliably than larger ones, indicating that plant size plays a major role for the model performance. This suggests that the model could be significantly improved by increasing the dataset size and by enlarging the specimens in the model input by using cropped or sliced images.

Keywords: Deep Learning, Computer Vision, Single-Stage Object Detection, *Senecio jacobaea*, Weed Control

Contents

1 Introduction **1**

2 Theoretical Background **3**
2.1 Image Data . . . 3
2.2 Discrete Convolutions . . . 5
2.3 Structure of a CNN . . . 6
2.4 Training and Using a CNN . . . 8
2.5 SSD-MobileNet-v1 . . . 10

3 Materials and Methods **11**
3.1 *Senecio jacobaea* Dataset . . . 11
3.1.1 Collecting Data . . . 11
3.1.2 Labelling the Data . . . 13
3.1.3 Creating a Dataset in PASCAL VOC Format . . . 14
3.1.3.1 Training, Validation and Test Set . . . 14
3.1.3.2 The PASCAL VOC Dataset Format . . . 15
3.1.3.3 Automatizing the Process . . . 16
3.1.4 Dataset Visualization . . . 17
3.2 Training Toolchain . . . 19
3.2.1 Preparing the Training . . . 19
3.2.2 Training the Model . . . 20
3.2.3 Evaluating Logging Files . . . 21
3.2.4 Model Export . . . 22
3.3 Training on Modified Datasets . . . 23
3.3.1 Dataset 2: Influence of Object Size . . . 23
3.3.2 Dataset 3: Influence of Dataset Size . . . 26
3.4 Inference on Test Datasets . . . 27
3.4.1 Creating the Test Datasets . . . 27
3.4.2 Inference . . . 27
3.5 Model Evaluation . . . 28
3.5.1 Evaluation Metrics . . . 28
3.5.1.1 Intersection over Union (IoU) . . . 29
3.5.1.2 Precision and Recall . . . 30
3.5.1.3 Precision × Recall Curve . . . 31
3.5.1.4 Average Precision and mean Average Precision . . . 32
3.5.2 Toolbox for Object Detection Metrics . . . 32
3.5.3 Visualizing Model Outputs . . . 33

4 Results **35**
4.1 Model 1: Original Dataset . . . 35
4.2 Model 2: Influence of Object Size . . . 37
4.3 Model 3: Influence of Dataset Size . . . 39
4.4 Samples from the Test Dataset . . . 40

5 Discussion **43**
5.1 Discussion of Sample Images . . . 43
5.1.1 Distorted Bounding Boxes . . . 43
5.1.2 Small Bounding Boxes . . . 43
5.1.3 False Predictions . . . 44
5.1.3.1 False Positive Detections . . . 44
5.1.3.2 False Negative Detections . . . 45
5.1.4 Duplicate Bounding Boxes . . . 46
5.2 Problems with the Dataset . . . 47
5.2.1 Missing Annotations . . . 47
5.2.2 Size Classes . . . 48
5.2.3 Composition of the Dataset . . . 49
5.2.4 Illumination of *Senecio jacobaea* . . . 49
5.3 How could the model be improved . . . 49
5.4 Conclusion . . . 51

List of Abbreviations

ANN	Artificial Neural Network
AP	Average Precision
CNN	Convolutional Neural Network
CSV	Comma-Separated Values
CVAT	Computer Vision Annotation Tool
DL	Deep Learning
FN	False Negative
FP	False Positive
FPS	Frames Per Second
FSOD	Fully-Supervised Object Detection
GPS	Global Positioning System
GUI	Graphical User Interface
IoU	Intersection over Union
mAP	mean Average Precision
MS COCO	Microsoft Common Objects in Context
NDVI	Normalized Difference Vegetation Index
NJX	NIVIDA Jetson Xavier NX
NMS	Non-Maximum Suppression
ONNX	Open Neural Network Exchange
OpenCV	Open Source Computer Vision Library
PA	Pyrrolizidine Alkaloid
PASCAL VOC	PASCAL Visual Object Classes
R-CNN	Region based Convolutional Neural Network
ReLU	Rectified Linear Unit
SDK	Software Development Kit
SGD	Stochastic Gradient Descent
SSD	Single Shot MultiBox Detector
TN	True Negative
TP	True Positive
UAV	Unmanned Aerial Vehicle
XML	Extensible Markup Language
YOLO	You Only Look Once

Listings

3.1 Extensible Markup Language (XML) ground truth label for image 436 of the leu_210601_n_acb4 sub-dataset. 14
3.2 Python script for combining different sub-datasets into a single dataset in PASCAL Visual Object Classes (PASCAL VOC) format which can be used for training a model. 17
3.3 Python script for visualising the dataset. 18
3.4 Command for running the Docker container with the folder mounted. 20
3.5 Command for training SSD-MobileNet-v1 on a custom dataset in PASCAL VOC format. 20
3.6 Python script for reading the log file containing the logging information of the training process. 21
3.7 Command for converting a model from .pth to .onnx format. 23
3.8 Python script for dividing the Senecio class into multiple subclasses based on bounding box size. 24
3.9 Command for extracting the test dataset from a whole dataset in PASCAL VOC format. 27
3.10 Command for applying custom object detection models on test images and storing the outputs as text files. 28
3.11 Command for running the ImageViewer utility for visualizing bounding boxes on images. 33

List of Figures

2.1 a) Monochromatic image of the number ‘5’ represented as grayscale. c) 2D tensor containing the intensity values of the pixels. b) image overlain with the tensor. 4
2.2 a) Image of *Senecio jacobaea*. b) red, c) green, d) blue colour channel of the same image. 4
2.3 a) 512 $\times$ 384 grayscale image of *Senecio jacobaea* and surrounding vegetation. b) 3 $\times$ 3 horizontal Sobel operator kernel. c) 510 $\times$ 382 output image from convolving the input image from a) with the kernel from b). 6
2.4 LeNet-5 architecture. Own illustration, based on LeCun et al. [1998]. Conv1 and Conv2 are convolutional layers. Pool1 and Pool2 are max pooling layers. FC1 and FC2 are fully connected layers. Notations: 6@28 $\times$ 28 refers to 6 feature maps of size 28 $\times$ 28 pixels 7

3.1 Image 436 of the `leu_210601_n_acb4` sub-dataset. The corresponding bounding box enclosing the *Senecio jacobaea* specimen, encoded by the XML file shown in listing 3.1, is displayed in cyan. 15
3.2 Directory structure for a dataset in PASCAL VOC dataset format. . 16
3.3 Heatmap showing the spatial distribution of all 5723 specimen of *Senecio jacobaea* in the 2128 images of the dataset. x and y refer to the zero-based pixel coordinates of the images of size 1024 $\times$ 768 pixels. 18
3.4 Histogram of the distribution of bounding box sizes in the dataset. The right tail is cut off since there are only very few outliers larger than 150000 $pixel^2$. 19
3.5 Part of the `jetson-inference` repository which is relevant for the project. Irrelevant folders and files are not shown here. Names enclosed with `<>` have to be replaced with the actual names used. . . . 20
3.6 Line plot showing the validation loss (classification, regression, total) of the model during the training process. 22
3.7 Histogram of the distribution of bounding box sizes in the dataset split into three classes of different bounding box size intervals. 23
3.8 Line plot showing the validation loss (classification, regression, total) of the model during the training process on the second dataset. . . . 25
3.9 Line plot showing the validation loss (classification, regression, total) of the model during the training process on the third dataset. 26
3.10 Directory tree of the test datasets. 27
3.11 Intersection over Union (IoU). Own illustration, based on Padilla et al. [2021]. 29
3.12 (Interpolated) Precision $\times$ Recall curve of table 3.6. 32
3.13 GUI of the object detection metrics toolbox provided by [Padilla et al., 2021]. 33

4.1 a) precision $\times$ recall curve, b) interpolated precision $\times$ recall curve of model 1 on the test dataset with $\tau = 0.2$ and an IoU of 0.50. . . . 36

4.2 a) precision × recall curve, b) interpolated precision × recall curve of model 2 for class Senecio_s on the test dataset ($\tau = 0.2$; IoU = 0.50). 38
4.3 a) precision × recall curve, b) interpolated precision × recall curve of model 2 for class Senecio_m on the test dataset ($\tau = 0.2$; IoU = 0.50). 38
4.4 a) precision × recall curve, b) interpolated precision × recall curve of model 2 for class Senecio_l on the test dataset ($\tau = 0.2$; IoU = 0.50). 39
4.5 a) precision × recall curve, b) interpolated precision × recall curve of model 3 on the test dataset with $\tau = 0.3$ and an IoU of 0.50. . . . 40
4.6 Predictions (red) of a) model 1, b) model 2, and c) model 3 on image kam_210515_n_acb4_0023. Ground truth labels are in cyan. 41
4.7 Predictions (red) of a) model 1, b) model 2, and c) model 3 on image kam_210515_n_acb4_0039. Ground truth labels are in cyan. 41
4.8 Predictions (red) of a) model 1, b) model 2, and c) model 3 on image kle_210516_n_acb4_0184. Ground truth labels are in cyan. 42
4.9 Predictions (red) of a) model 1, b) model 2, and c) model 3 on image leu_210601_n_acb4_0307. Ground truth labels are in cyan. 42
4.10 Predictions (red) of a) model 1, b) model 2, and c) model 3 on image moe_210527_a_acb4_0102. Ground truth labels are in cyan. 42

5.1 Image kle_210516_n_acb4_0184 (1024 × 768 pixel^2) cut into four slices (656 × 492 pixel^2) overlapping each other in the center. 50

List of Tables

3.1 Camera settings used for creating the *Senecio jacobaea* dataset. . . . 12
3.2 Metadata of the different sub-datasets. 'Imgs' refers to the number of images and 'Insts' refers to the number of class instances in those images (i.e. the number of *Senecio jacobaea* specimens.) 13
3.3 Best performing model checkpoints according to the total validation loss. 22
3.4 Best performing model checkpoints according to the total validation loss for the second training. 25
3.5 Best performing model checkpoints according to the total validation loss for the third training. 26
3.6 Example for evaluating outputs of an object detection model. The example dataset has 8 ground truth bounding boxes. 31

4.1 Overview over the three different models which were evaluated and their properties. 35
4.2 Results of model 1 on the test dataset. 35
4.3 Results of model 2 on the test dataset. 37
4.4 Results of model 3 on the test dataset. 39

1 Introduction

Senecio jacobaea is one of three species of the *Senecio* genus that are native in Germany and has spread massively in ruderal areas and extensively used pastures during the last years causing conflicts between agricultural land-use and environmental protection [Deutscher Verband für Landschaftspflege e.V., 2017]. Per specimen, *Senecio jacobaea* produces up to 100,000 seeds of which up to 80% are germinable and can persist up to 15 years in the soil making it difficult to regulate the species once enough seeds are accumulated in the soil of an area of extensive growth [Suter and Lüscher, 2017].

Senecio jacobaea contains Pyrrolizidine Alkaloids (PAs) proven to be hepatotoxic to humans but also livestock, especially cows and horses [Lampen, 2017] causing substantial economical damages to farmers. PAs were also much more likely to be found in quantities harmful to health in honey produced in areas populated by *Senecio jacobaea* [Neumann and Huckauf, 2015]. Therefore, efficient ways of regulating *Senecio jacobaea* have to be found with a special focus on environmentally protected areas where severe interventions like extensive usage of herbicides is not possible. As of now, specimens have to be plugged and dug out tediously by hand or the area has to be systematically mowed and the forage has to be disposed [Zehm, 2017].

An alternative to these methods might be found in the field of precision agriculture where Unmanned Aerial Vehicles (UAVs) have been used to automatically spray herbicides on crop fields reducing costs and risks for farmers [Mogili and Deepak, 2018]. The vision that inspired this project is the idea of a two step weed control method utilizing Deep Learning (DL) techniques for finding *Senecio jacobaea*. One UAV should fly across a field at a higher altitude and collect images using a downward-pointed camera. In these images, possible candidates of *Senecio jacobaea* should be detected and their Global Positioning System (GPS) coordinates should be mapped. Afterwards, another UAV, equipped with a sprinkling system, should fly to the mapped coordinates and validate whether the detected candidates are really *Senecio jacobaea* and if so spray a tiny amount of herbicide directly onto the center of the plant. This would likely be most efficient in spring when *Senecio jacobaea* specimens are still in an early growth stage since it was shown that plants in later growth stages are much less responsive to the same dose of herbicide [Kieloch and Domaradzki, 2011]. Another option would be to combine the approach with a field robot to apply the herbicide. Such an approach could drastically reduce the

temporal and financial expenses of farmers while being much less damaging to the ecosystem than mass spraying of herbicides. It could thus also pose a viable solution for removing *Senecio jacobaea* in protected areas where regulation methods should be least invasive.

This project focused on the problem of detecting *Senecio jacobaea* in images which would likely be the bottleneck for such a weed control system. The object detection was done using an SSD-MobileNet-v1 [Franklin et al., 2016] which is a type of DL model, pretrained on the Microsoft Common Objects in Context (MS COCO) dataset with 91 classes. Object detection refers to finding multiple instances of different object classes in an image and localizing them by the means of a bounding box enclosing them which could be used to map the coordinates of *Senecio jacobaea* specimens. For implementing the model an NVIDIA Jetson Xavier NX [NVIDIA, 2021], which is an embedded computer designed for DL applications, was used since it can be mounted onto a UAV and perform object detection in real-time.

Specifically, the work attempts to answer the question whether it is possible to detect *Senecio jacobaea* at 1m height in images also containing background vegetation of similar appearance in terms of colour, structure or leaves. Furthermore, it was investigated how to set up a working toolchain, from the collection and annotation of raw data, training of the model, to evaluating the finished result. Additionally, the goal was to find out which factors influence the performance of the model and what the limitations of such a model are, to gain more insight of how to approach this problem and yield better results.

2 Theoretical Background

A Convolutional Neural Network (CNN) is a type of DL model which is widely used for computer vision tasks. DL refers to a subset of machine learning models which utilize Artificial Neural Networks (ANNs) which map input data to a certain output by passing them through a number of computational layers which operate on the data and pass the modified results to the next layer. CNNs are a subclass of ANNs which utilize discrete convolutions in their network architecture to extract features of increasing complexity [Yamashita et al., 2018]. The use of convolutions makes them especially useful for processing data which is organized into a grid structure such as images. The output of such a network depends on its architecture; in general, there are networks for classification, object detection, and object segmentation. In classification, an object class is assigned to an image as a whole, while in object detection, multiple objects can be found in the same image. Lastly, object segmentation refers to assigning classes to individual individual regions of the input image. To be able to solve such a task, a CNN has to be trained on a training dataset consisting of several thousand up to millions of labelled images [Russakovsky et al., 2014]. During the training, the network predicts a certain output for the training data which is then compared with the ground truth labels of the data. The larger the difference between the two, the higher the error of the network. The parameters of the network are then adjusted, step by step, as to decrease that error in a process called backpropagation. After successfully training the model, its parameters should be fit well to the given task.

The following subsections give a short overview of images as a data type, discrete convolutions, CNNs, and how the training and application of such models works.

2.1 Image Data

There are different types of image formats which result in different digital representations of images such as monochromatic or RGB images. Generally, an image is represented as a 2- or 3-dimensional array of intensity values. The first two dimensions are the height and width of the image, while the third dimension represents a colour channel. In the context of CNNs those arrays are called tensors. A monochromatic image consists of only a single channel and can be represented by a 2-dimensional tensor where each element represents a pixel in the image and the

value of the element corresponds to the brightness of the pixel.

Figure 2.1a shows a monochromatic image of the number '5' represented as a grayscale image. Figure 2.1c shows the 2D tensor representing the image and 2.1b shows the image overlain with the tensor.

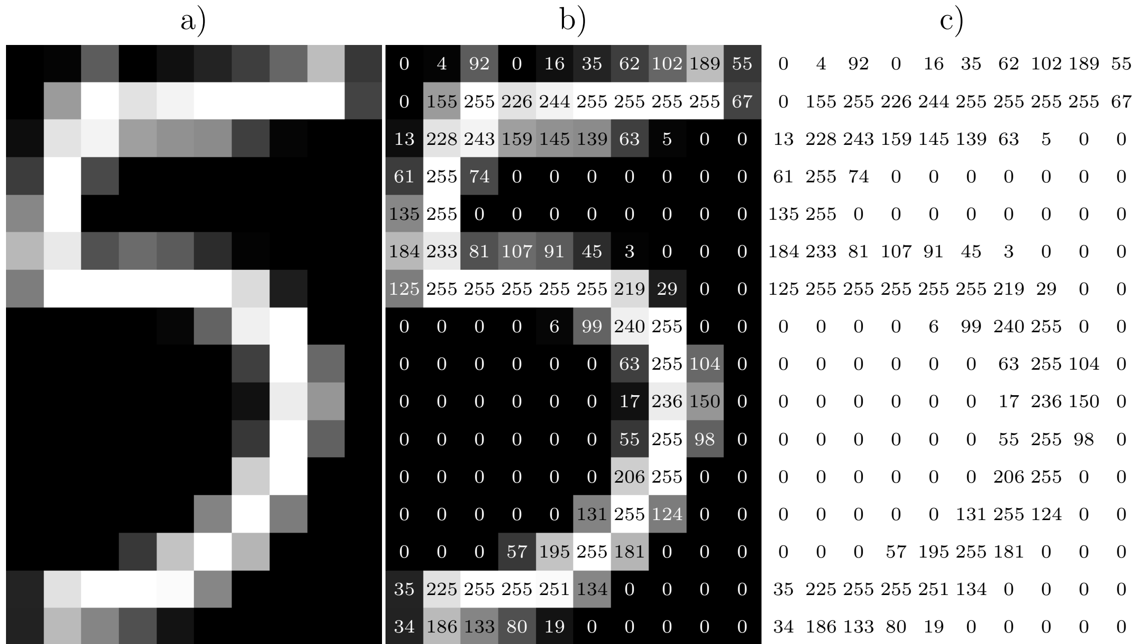

Figure 2.1: a) Monochromatic image of the number '5' represented as grayscale. c) 2D tensor containing the intensity values of the pixels. b) image overlain with the tensor.

However, the images of *Senecio jacobaea* used in this project are in RGB colour space. That means each image can be represented as a 3D tensor with a depth of 3 because the image consists of 3 colour channels for red, green and blue, respectively. The values in each channel represent the intensity of the colour of the channel for each pixel. The intensity values can be any integer in the range of $[0, 255]$.

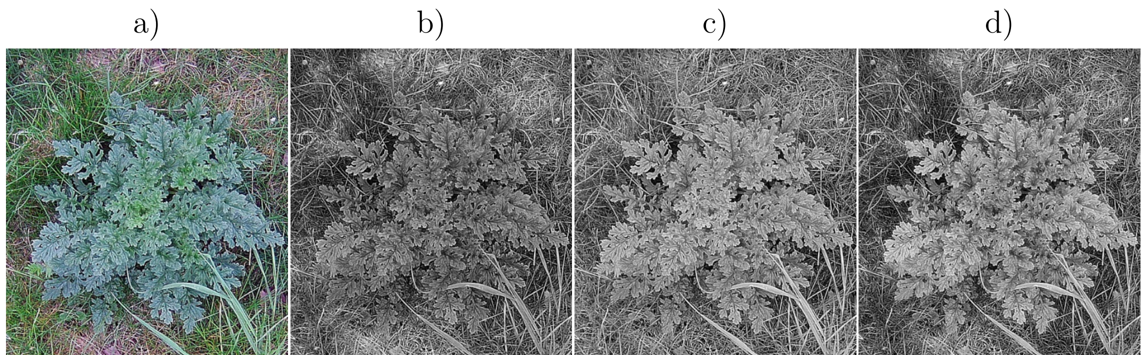

Figure 2.2: a) Image of *Senecio jacobaea*. b) red, c) green, d) blue colour channel of the same image.

Figure 2.2a shows an image of a specimen of *Senecio jacobaea*; b, c, and d show the red, green, and blue colour channels of the image, respectively. The example

shows how different channels can highlight different features even in the input image itself. This becomes especially evident in figure 2.2d where the specimen is quite contrasting to the background vegetation which indicates that *Senecio jacobaea* has a more pronounced blueish tint than most other vegetation.

2.2 Discrete Convolutions

A CNN is based on convolutional layers which utilize convolutions to extract features from input feature maps to produce new output feature maps of higher complexity. A convolution is a mathematical operation which can be used for processing and manipulating functions or signals. For continuous signals, a convolution is given by an integral; for discrete signals, it is given by a discrete sum. A grayscale image can be considered a discrete version of a continuous 2D function $g[x, y]$ where x and y are the discrete indices of the pixels in the image and $g[x, y]$ is the intensity of the pixels. The indices are ordered from top to bottom and from left to right of the image. Outside of the defined area of the image, the values are considered to be 0. A convolution of such an image can be represented by the sum

$$f[x, y] = g[x, y] * h[x, y] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} g[k, l] \cdot h[x - k, y - l] \tag{2.1}$$

The original image (input tensor) $g[x, y]$ is convolved ($*$) with another, usually much smaller, image $h[x, y]$, called a kernel, to yield the manipulated image (output tensor) $f[x, y]$. The pixel intensity values in the kernel are weights which are learnable parameters of a CNN. That means the values are adjusted during training to increase the accuracy of the model.

During the convolution, the kernel is mirrored about both axes and shifted to the x and y coordinates. At every point $[x, y]$ each element of the input image is multiplied with the corresponding element of the kernel and the results are summed up which yields the new intensity value for the output image f at the pixel $[x, y]$. This is done for all possible $[x, y]$ coordinates, i.e. every point where the kernel overlaps a defined area of the input image.

To conclude, a convolution can be thought of as mirroring the Kernel and sliding it across the input feature map from left to right, from top to bottom. At each position an elementwise multiplication between the kernel weights and the elements of the input feature map, overlapped by the kernel, is performed and the results are summed up to obtain a new value for that position [Dumoulin and Visin, 2018]. Figure 2.3 shows an example of how convolving an image with a kernel can extract features. Here, a horizontal Sobel operator, which is used for detecting horizontal edges, is used for convolution. In the context of CNNs, a) would be the input tensor, b) would be the filter, and c) would be the output tensor with one feature map.

Kernel sizes larger than 1×1 decrease the output feature map size, since it reduces

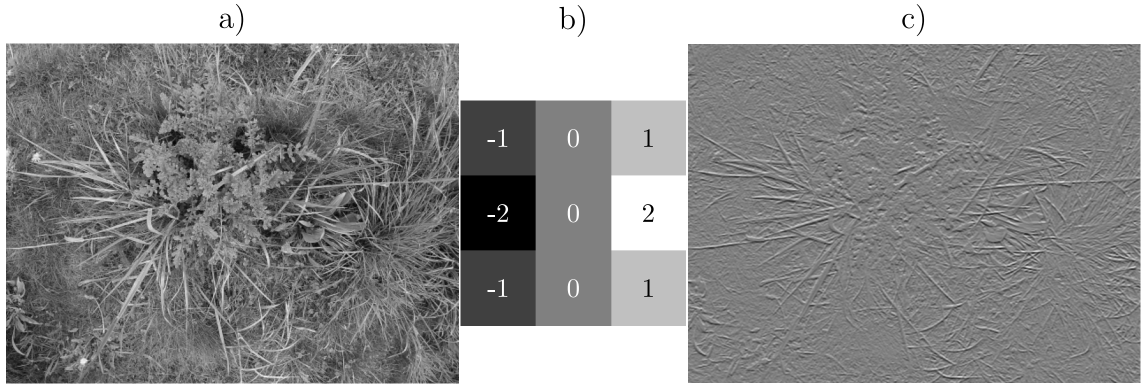

Figure 2.3: a) 512×384 grayscale image of *Senecio jacobaea* and surrounding vegetation. b) 3×3 horizontal Sobel operator kernel. c) 510×382 output image from convolving the input image from a) with the kernel from b).

the number of possible positions for the kernel without overlapping undefined areas of the input feature map (See figure 2.3). The input feature map can therefore be padded to conserve the original size. Increasing the step size or stride of the kernel (i.e. skipping positions) further reduces the output feature map size. According to [Dumoulin and Visin, 2018], the output feature map size can be calculated with

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \tag{2.2}$$

where o is the output feature map size, i is the input feature map size, p is the padding, k is the kernel size and s is the stride.

In a CNN, the input tensor usually consists of multiple channels (feature maps), i.e. is 3-dimensional. For 2D convolutions, the kernel or filter has as many channels (with distinct weights in each channel) as the input tensor. Each channel of the input tensor is convolved with the corresponding channel of the filter. Finally, all resulting channels are summed up element-wise to yield a single new feature map. To obtain n feature maps for the output tensor, the input tensor has to be convolved with n filters.

2.3 Structure of a CNN

As mentioned before, a CNN consists of several layers. The first layer is the input layer which takes in an image of a fixed size as its input. The last layer is the output layer which returns a certain output, e.g. a class label, depending on the type of network. The layers in between are called hidden layers and consist of different building blocks like convolutional layers, pooling layers, or fully connected layers [Yamashita et al., 2018]. In a feed-forward neural network, the output of the previous layer is used as input in the next layer. So, the input is passed from one layer to the next in forward direction through the network. The general structure of

a simple CNN can be seen in figure 2.4 which shows the architecture of the famous LeNet-5 proposed by LeCun et al. [1998], which was designed to distinguish handwritten digits in monochromatic images of size 32×32 pixels. It is a decent example for understanding the structure of CNNs as it is comprehensible due to its small size, yet still contains the most important building blocks that modern day CNNs are composed of.

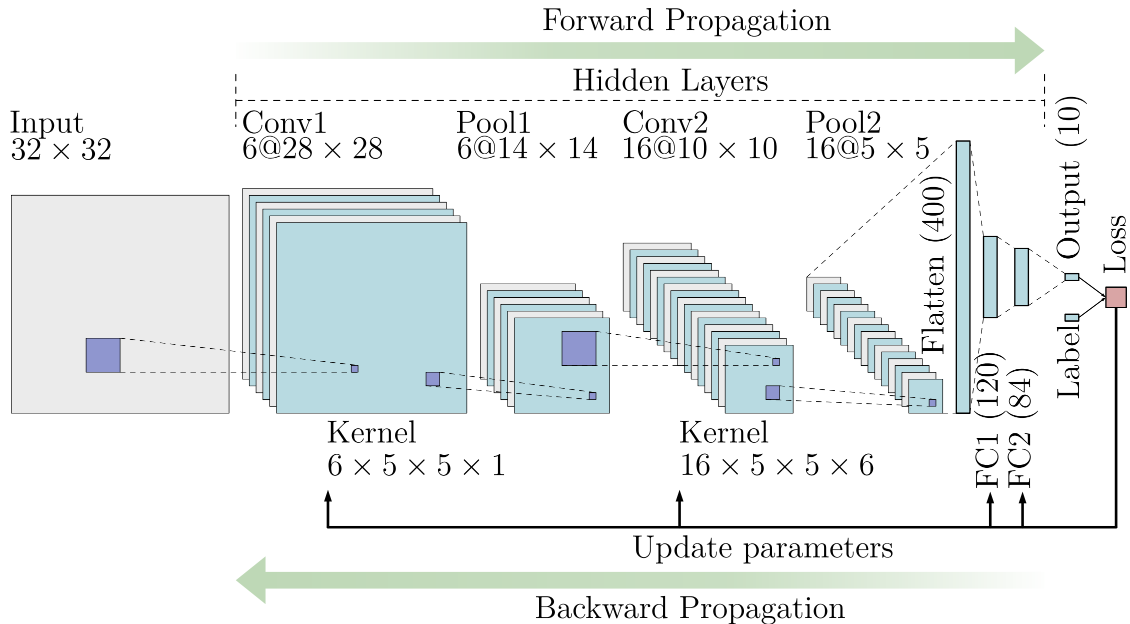

Figure 2.4: LeNet-5 architecture. Own illustration, based on LeCun et al. [1998]. Conv1 and Conv2 are convolutional layers. Pool1 and Pool2 are max pooling layers. FC1 and FC2 are fully connected layers. Notations: $6@28 \times 28$ refers to 6 feature maps of size 28×28 pixels

A convolutional layer (Conv1 and Conv2 in figure 2.4) is a combination of linear and nonlinear operations. The linear operation is a convolution operation as described in the previous subsection. It is done by applying n different filters to the layer input to produce n different feature maps as the layer output. The weights in the filters are the learnable parameters of the network. As convolutions are linear operations, it is important to introduce a non-linearity by applying another nonlinear operation since otherwise, the network could only detect linear relationships which would defeat the purpose of the hidden layers. Therefore, a so-called activation function is applied to each element of the output feature maps before they are passed as input to the next layer. There are different activation functions such as the hyperbolic tangent, the sigmoid, or the Rectified Linear Unit (ReLU). SSD-MobileNet-v1 which is the CNN used in this project utilizes the ReLU function [Howard et al., 2017] which is given by

$$f(x) = \max(0, x) \tag{2.3}$$

In LeNet-5 (see figure 2.4), the kernel sizes in both convolutional layers are 5×5 (width $\times$ height), therefore, the output feature maps are reduced by 4 pixels in height and width as explained by equation 2.2.

Another important building block is the pooling layer (Pool1 and Pool2 in figure 2.4). In a CNN, the height and width of the feature maps typically get smaller and smaller with each subsequent layer. This is desired since larger feature maps are computationally more expensive and if the model is to be deployed in an embedded system with limited computing power, computational efficiency is important for the performance of the application. Therefore, feature maps are often downsampled with pooling layers to further reduce the size of the feature maps. Like in a convolutional layer, a kernel is slid across the input feature map. The kernel is usually of size 2×2 and a stride of 2 is used such that it skips every second position in the feature map. The difference is, that the kernel does not contain learnable parameters and no elementwise multiplication is performed. In max pooling, the value returned for each position is the maximum value of the input feature map within the area overlapped by the kernel; the other values of the input are omitted. There are also other pooling methods such as min pooling which takes the minimum value, or average pooling which calculates the average of the values. A pooling layer with a kernel of size 2×2 and stride of 2 can thus reduce the height and width of the input feature map by 50% and the area by 75%, which makes calculations in subsequent layers faster [Yamashita et al., 2018]. How a pooling layer reduces the feature map size by a factor of 2 can also be seen in figure 2.4.

In CNNs for classification tasks such as MobileNet, the last layers are typically fully connected layers (FC1 and FC2 in figure 2.4), also referred to as dense layers. That means the output of the last convolutional layer is flattened to a 1-dimensional vector and used as input for one or more dense layers. In a dense layer each input is connected to each output and a learnable weight is associated with each connection. A dense layer thus calculates linear combinations of the previous layers outputs. Therefore, like with convolutional layers, a nonlinear activation function like ReLU is applied to each output before passing it to the next layer. The final output is for example a vector with the probabilities of each class.

2.4 Training and Using a CNN

When a CNN is used for inference, the process is called forward propagation because the input data is propagated through the network in forward direction to yield some predicted output.

As mentioned before, training a model refers to optimizing its parameters (e.g. kernels, weights) to minimize the error that the model makes. The process is done by a backpropagation algorithm which utilizes a loss function and an optimizer like

Stochastic Gradient Descent (SGD). It is called backpropagation because during the optimization process, the propagation occurs in the backward direction starting from the output layer and ending in the first hidden layer.

The loss function L, also called cost function, is a measure of the error that the model makes. As an input it uses the output predicted by the model and the ground truth labels of the training dataset. A high loss means that model output and ground truth labels do not match well and that the model accuracy is low. Therefore, during training, the goal is to reduce the loss of the model. The type of loss function depends on the type of output that the model produces.

According to Yamashita et al. [2018], gradient descent is a commonly used optimization algorithm. The loss function is a function of all learnable parameters of the model. The gradient of the loss function points in the direction of steepest rate of increase of the loss function. It is a vector containing the partial derivatives of the loss function with respect to each parameter. As the goal is to reduce the loss, each parameter has to be tweaked in the negative direction of the gradient. Thus, the sign of each partial derivative gives the opposite direction of how to tweak the parameter and its magnitude is a measure by how much it has to be tweaked. Therefore, gradient descent refers to finding a (local) minimum of the loss function by updating its parameters as to decrease the loss as fast as possible. To not overshoot such a minimum of the loss function, the learning steps are multiplied with a small factor (e.g. 0.01) called learning rate. A single update of a parameter can thus be expressed with

$$w := w - \alpha \cdot \frac{\partial L}{\partial w} \tag{2.4}$$

where w represents each learnable parameter, α is the learning rate, and L is the loss function. In gradient descent, the gradient of the loss function would be calculated for each image in the training dataset. Afterwards, the values of the partial derivatives for each parameter would be averaged, then multiplied with the learning rate and subtracted from the corresponding parameter to yield its new value. However, it would be extremely time consuming to calculate the output for each image of the training dataset for every optimization step. Therefore, a more commonly used optimizer is the SGD in which the training dataset is split into several batches. Instead of using the whole dataset for an optimization step, the model is updated with each batch making training much more time efficient.

Model parameters which are not learnable and which have to be set manually before training, such as activation functions, loss functions, optimizers, batch sizes, or learning rates are referred to as hyperparameters of the model [Yamashita et al., 2018].

2.5 SSD-MobileNet-v1

The CNN used for this project is the SSD-MobileNet-v1 which is an object detection model provided by NVIDIA on GitHub [Franklin et al., 2016]. The network combines the two separate models MobileNet [Howard et al., 2017] and Single Shot MultiBox Detector (SSD) [Liu et al., 2016] to a single new model.

MobileNet is a CNN for image classification which uses depthwise separable convolutional layers instead of standard convolutional layers. Depthwise separable convolution splits the convolution into two steps, a depthwise convolution which is used for extracting features and a pointwise convolution which calculates a linear combination to obtain the final feature map as output. This greatly reduces the number of calculations necessary as well as the number of trainable network parameters [Howard et al., 2017] which makes it very well suited for systems with limited computing power such as the NVIDIA Jetson Xavier NX.

The fully connected layers of MobileNet used for classification are cut off and the convolutional base of the network is used as input for SSD. So, MobileNet is used in the beginning for extracting features and SSD is used afterwards for extracting higher order features and for the object detection itself.

In general, there are two different kinds of object detection models: multi-stage and single-stage detectors [Sumit et al., 2020]. Multi-stage models split the image into subsections and perform a classification on these subsections in different scales to find objects in the image. That means, for a single object detection, the network has to perform many evaluations. In contrast to that, single-stage object detection models only need a single evaluation of the model as they directly map pixel values to bounding boxes and class probabilities. This lets them outperform multi-stage methods like Region based Convolutional Neural Network (R-CNN) [Redmon et al., 2015]. SSD is such a single-stage object detection model. Therefore, the combination of the lightweight MobileNet with the single-stage detector SSD was chosen to be used in this project since the computational efficiency is important when working with embedded computers like the NVIDIA Jetson Xavier NX which is able to run this model in real-time.

3 Materials and Methods

3.1 *Senecio jacobaea* Dataset

To train SSD-MobileNet-v1 to detect *Senecio jacobaea*, a labelled dataset is necessary. The dataset should be as large as possible for the model to learn the features of *Senecio jacobaea* and to prevent the model from being overfit. Overfitting refers to the model learning irrelevant noise specific to the training dataset by training too much on the same data or on a dataset that is too small. The model would then memorize the images from the training dataset and which would decrease its accuracy on unseen data. A larger dataset thus helps the model to better generalize the common features of the different classes and prevent overfitting [Yamashita et al., 2018].

Besides collecting a larger dataset, there are also other possibilities to prevent overfitting. One option is to artificially increase the dataset size using data augmentation. Data augmentation refers to modifying the images with random transformations such as translation, rotation or mirroring and adding those modified images to the dataset.

Another possibility is to use transfer learning. When using transfer learning, the weights in the model are not initialized with random values (i.e. not trained from scratch) but instead pretrained model parameters are used. The assumption is that generic features learned from another large dataset are also valid for apparently disparate smaller datasets and can thus be shared [Yamashita et al., 2018]. Since the size of the dataset which could be obtained in this project is limited, a version of SSD-MobileNet-v1 pretrained on the MS COCO dataset containing 330000 images of 91 object classes [Lin et al., 2015] was used. For the format of the *Senecio jacobaea* dataset the PASCAL VOC [Everingham et al., 2010] dataset format was chosen.

3.1.1 Collecting Data

The dataset should be representative of what the model would see during application afterwards. That means the images should be taken at different times of day, under different weather and lighting conditions. The locations and the surrounding vegetation should be diverse and the specimens of *Senecio jacobaea* should be in different growth phases. If these conditions are met and the dataset is as large as possible, it helps the model to generalize the features of the plants better and make

it more robust during inference later on.

Since the idea is to use the model on an NVIDIA Jetson Nano or NVIDIA Jetson Xavier NX mounted onto a drone to detect *Senecio jacobaea* from a height of about 1 to 5 meters, the images should meet the same conditions. Therefore, each image was taken at a height of approximately 1 meter above ground referring to the center of the image. All images were taken with an Acaso Brave 4 action camera which was used as there were already images of *Senecio jacobaea* available, made with the same camera which could later on be combined to enlarge the dataset. Furthermore, all images were taken with the same camera settings to keep the dataset consistent. The details can be found in table 3.1.

Table 3.1: Camera settings used for creating the *Senecio jacobaea* dataset.

Parameter	Setting Selected
Camera Model	Akaso Brave 4 Action Cam
Image Type	JPEG
Resolution	5120 × 3830
Aspect Ratio	4:3
Exposure Time	1/849 second
Aperture Value	1.60 EV (f/1.7)
ISO Speed Rating	50
Flash Fired	flash did not fire
Metering Mode	Average
Exposure Program	normal program
Focal Length	2.7 mm

The images were taken in the period of the 15th of May 2021 to the 1st of June 2021. To identify the different sub-datasets, they were given unique identifiers containing information on location, date, time of day and camera model. For example the ID `kam_210515_n_acb4` refers to the images taken in Kamp-Lintfort on the 15th of May 2021 at noon with the Akaso Brave 4 action cam. The images from the sub-datasets were additionally given unique four digit numbers separated from the ID by an underscore. For the project, seven sub-datasets of varying sizes were created at different locations, dates and time of day. The metadata for these sub-datasets can be found in table 3.2.

After collecting the images, they were renamed as mentioned before and resized to 1024 × 768 pixels, keeping the 4:3 aspect ratio, using Open Source Computer Vision Library (OpenCV) [Bradski, 2000] in Python. The Python scripts used in this project can be retrieved directly from the GitLab repository of this project [Zender, 2021]. The resizing was done to make the handling of the data easier and faster. Since any input data is resized to 300 × 300 pixels [Franklin et al., 2016] before it is used in the network, resizing the images does not lower the resolution for the input data of the network and thus does not negatively affect its performance.

Table 3.2: Metadata of the different sub-datasets. 'Imgs' refers to the number of images and 'Insts' refers to the number of class instances in those images (i.e. the number of *Senecio jacobaea* specimens.)

ID / Date / Time	Location	Type	Weather	Imgs	Insts
`kam_210515_n_acb4` 15.05.21 12:00-14:30	Kamp-Lintfort: Stephanswäldchen, LaGa entrance	park	mostly cloudy, rainy	144	201
`kle_210516_n_acb4` 16.05.21 11:30-14:30	Kleve: Forstgarten, Joseph-Beuys Allee	park	cloudy, later sunny	469	1242
`moe_210518_n_acb4` 18.05.21 11:30-13:30	Moers: Schlosspark, Grafschafter Kampfbahn	park, demolition area	mostly sunny, cloudy	85	209
`moe_210527_a_acb4` 27.05.21 16:30-17:00	Moers: Grafschafter Kampfbahn	demolition area	cloudy, rainy	112	381
`kam_210529_a_acb4` 29.05.21 14:00-15:00	Kamp-Lintfort: Stephanswäldchen, LaGa entrance	park	sunny	90	151
`mil_210529_e_acb4` 29.05.21 17:00-18:30	Rheinberg Millingen: Heidestraße 15	meadow, pasture	sunny	260	517
`leu_210601_n_acb4` 01.06.21 12:00-15:30	Leucht: Strohweg, Stappweg, Bierweg	forest, fields	sunny	968	3022

3.1.2 Labelling the Data

The output of the network is a fixed number of predictions where each prediction has a class, a confidence value and a bounding box. The confidence value is the probability that the detection is valid. Usually, a threshold is set which cuts off all predictions which have a confidence that is too low. The bounding box encloses the object predicted by the model and it is given by the bounding box coordinates. Specifically, the model will return the x and y coordinates of the top left corner and the bottom right corner of the bounding box.

To train the model, each sample from the dataset needs a ground truth label which has to contain the class and bounding box coordinates of every object that the model should find in that image. The labelling was done using the free and open source Computer Vision Annotation Tool (CVAT) developed by Intel [Sekachev et al., 2020]. CVAT can be used online to draw the bounding boxes directly on the image and export them in different formats, however, it should be noted that the limit for uploaded data is 500Mb. Therefore, CVAT was installed locally and for each sub-dataset a separate task was created. The finished labels for each sub-dataset were exported in PASCAL VOC dataset format in which each image has its own XML file containing the label. Listing 3.1 shows an example for such a label file

for image 436 of the Leucht sub-dataset. The file contains the file-name (l. 3) and dimensions (l. 10-11) of the image it belongs to. Each bounding box is defined in an `object` tag where `xmin`, `ymin`, `xmax`, and `ymax` (l. 19-22) are the coordinates of the top left and bottom right corner of the bounding box, respectively. The class name is given in the `name` tag (l. 16).

Listing 3.1: XML ground truth label for image 436 of the `leu_210601_n_acb4` sub-dataset.

```xml
<annotation>
  <folder>leu</folder>
  <filename>leu_210601_n_acb4_0436.JPG</filename>
  <source>
    <database>Unknown</database>
    <annotation>Unknown</annotation>
    <image>Unknown</image>
  </source>
  <size>
    <width>1024</width>
    <height>768</height>
    <depth></depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>Senecio</name>
    <occluded>0</occluded>
    <bndbox>
      <xmin>308.88</xmin>
      <ymin>230.42</ymin>
      <xmax>738.71</xmax>
      <ymax>739.41</ymax>
    </bndbox>
  </object>
</annotation>
```

Figure 3.1 shows the image with the bounding box encoded in the label file. In total, for the 7 sub-datasets listed in table 3.2, 5723 instances of the `Senecio` class were labelled.

3.1.3 Creating a Dataset in PASCAL VOC Format

After creating the labels for all sub-datasets shown in table 3.2 using CVAT and exporting them, the collected data and labels have to be reorganized into a single dataset that can be used for training. Furthermore, the dataset has to be split into a training, validation, and test dataset [Yamashita et al., 2018].

3.1.3.1 Training, Validation and Test Set

The training dataset is used to optimize the model during training through back-propagation. The features of the training data are the features that the model learns.

Figure 3.1: Image 436 of the `leu_210601_n_acb4` sub-dataset. The corresponding bounding box enclosing the *Senecio jacobaea* specimen, encoded by the XML file shown in listing 3.1, is displayed in cyan.

The validation dataset is used during the training process to monitor the progress and performance of the model. During training the model iterates over the training data multiple times to optimize the model parameters. One iteration over the whole training dataset is referred to as an epoch. After completing an epoch, the model is evaluated on the validation dataset by performing forward propagation on each validation sample and calculating the average loss for the whole validation dataset. The loss can then be used to track the training progress and for deciding if the model should be trained further or not.

The test dataset is not needed until the training is finished. It is then used to evaluate the most promising model checkpoints (i.e. the ones with the lowest validation loss) by using it on data it has never seen before. The model which performs best on the test dataset is then usually used in practice later on.

There are different common ratios of how to split the dataset into training, validation and test dataset. In this project, 80% were used for training, 10% for validation, and the remaining 10% for testing, which in comparison to other ratios yielded the best results for datasets of similar size [Prashanth et al., 2020].

3.1.3.2 The PASCAL VOC Dataset Format

To enable the model to process the data, it has to be organized into to a standardized file and directory structure which is called the PASCAL VOC dataset format. Figure 3.2 shows the directory tree that has to be used.

`Annotations` must contain the labels as XML files while `JPEGImages` contains the corresponding images. `default.txt` and `train.txt` are identical and define the training dataset; i.e. the name of each sample from the training dataset written

into a new line. Analogous to that, `test.txt` and `val.txt` define which samples belong to the test and validation dataset, respectively. Lastly, `labels.txt` must contain the names of all the classes from the dataset as they are written in the XML labels.

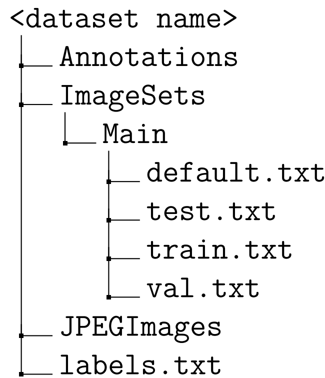

Figure 3.2: Directory structure for a dataset in PASCAL VOC dataset format.

3.1.3.3 Automatizing the Process

Since converting the dataset manually is a tedious and error-prone task, it was automatized using a custom Python class `DataSetCreator` which is available on the project GitLab page [Zender, 2021]. Listing 3.2 shows how to use it inside the cloned repository.

After importing the class from the Python module a `DataSetCreator` object can be instantiated by passing in a name for the dataset and a path where it should be saved (l. 13). The `loadSamples()` method can be used to copy the images and their labels into the dataset folder by passing in a list of sub-dataset names or tuples containing the paths to images and their labels (l. 14). `createLabeLtxt()` is used to create the `labels.txt` file by passing in a list of class names which are used in the dataset (l. 15). These have to be spelled exactly as in the XML labels. Here, only a single class `Senecio` is used. Finally, with `split()` the dataset can be split up into training, validation, and test dataset by passing in the proportions that each of them have; i.e. 80% training, 10% validation, 10% testing (l. 16). The method will populate the four text files defining which image belongs to which set as described in section 3.1.3.2. If `shuffle` is set to True, then the samples will be shuffled randomly before being distributed to the three subsets which should be done to ensure each of them contains samples from each sub-dataset. This is important, because training, validation and test datasets must all be representative for the dataset as a whole. For shuffling, Pythons `random` module was used which produces pseudo-random numbers for shuffling the images. Pseudo-random number generation refers to producing numbers which appear as if random, but are reproducible since they are calculated by an algorithm. The algorithm needs a starting value called a seed for which, by default, the system time is used. By choosing a constant value for the seed, the

results can be reproduced if the seed is known. This is important later on, for creating new, modified datasets which should have the same training, validation, and test sets to be able to compare the results.

Listing 3.2: Python script for combining different sub-datasets into a single dataset in PASCAL VOC format which can be used for training a model.

```
from custom_scripts.DataSetCreator import DataSetCreator

# available sub-datasets
dname_list = ['kam_210515_n_acb4',
              'kle_210516_n_acb4',
              'moe_210518_n_acb4',
              'moe_210527_a_acb4',
              'kam_210529_a_acb4',
              'mil_210529_e_acb4',
              'leu_210601_n_acb4']

# create new dataset
dataset = DataSetCreator('Senecio_dataset', path='data/datasets')
dataset.loadSamples(dname_list)
dataset.createLabeltxt(['Senecio'])
dataset.split(train=80, val=10, test=10, shuffle=True, seed=42)
```

3.1.4 Dataset Visualization

Since it is difficult to understand the properties of a dataset just by observing the raw data, a Python class was written to visualize the dataset by reading in the XML label files. The goal was to understand how the specimens of *Senecio jacobaea* are spatially distributed in the images which could be an important factor since specimens located towards the edges of the images are more distorted compared to specimens located in the center. This was done by creating a grid to lay on top of the images using a NumPy array [Harris et al., 2020]. Then the center of each bounding box was calculated and the value of the grid cell containing the center point was incremented by one. In other words, each element in the array is a counter which is incremented by one for each bounding box centred at that element. The resulting array is visualized in figure 3.3 as a heatmap which shows that the vast majority of plants are located in the center of the images. The other specimens are evenly distributed over the remaining area.

Another goal was to visualize the distribution of the bounding box sizes. This is important information when evaluating the performance of the finished model since small specimens can generally be expected to be harder to detect. Figure 3.4 shows a histogram of the absolute frequencies of *Senecio jacobaea* for different bounding box size bins. In the figure, the right tail of the distribution is cut off since there were only very few outliers in those size ranges (30 outliers of up to 313000 pixel^2). The histogram reveals that the distribution is right-skewed and as such smaller specimens are much more common than large specimen.

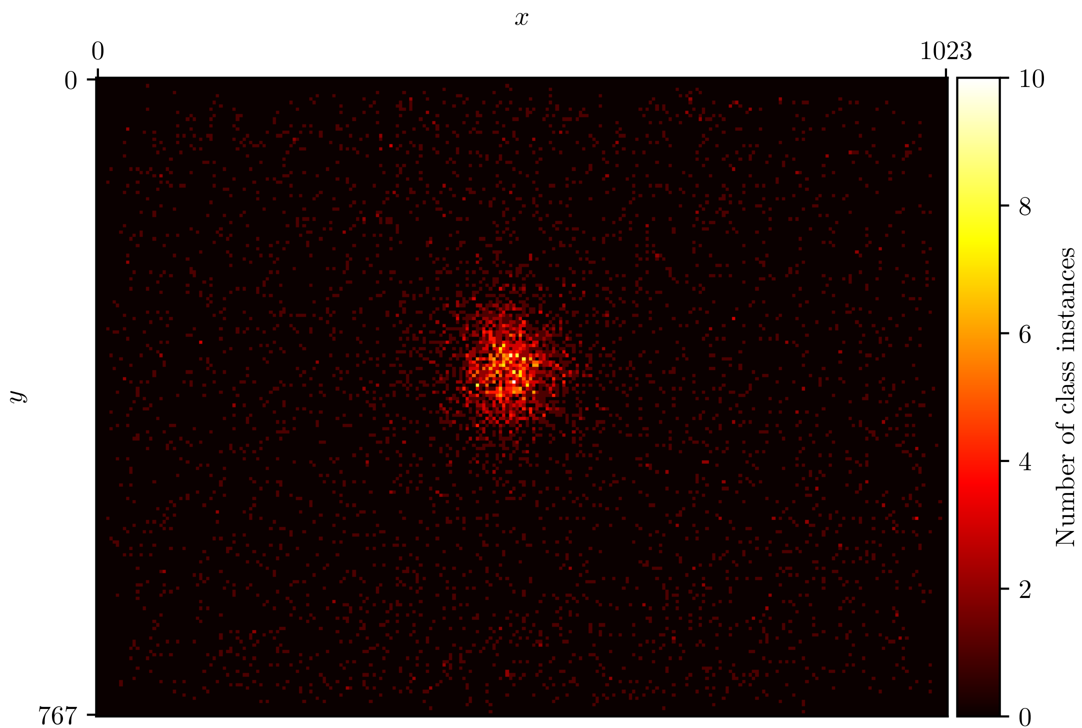

Figure 3.3: Heatmap showing the spatial distribution of all 5723 specimen of *Senecio jacobaea* in the 2128 images of the dataset. x and y refer to the zero-based pixel coordinates of the images of size 1024×768 pixels.

Listing 3.3 shows how to use the Python script [Zender, 2021] to visualize the dataset. After importing the class (l. 1) and defining the path to the labels (l. 3), a `DataSetReader` instance is created (l. 4). The XML files are loaded to a Python dictionary (l. 5) and the bounding box statistics are printed (l. 6). Then the counter-array of size 256×192 is created (l. 8) and plotted as a heatmap (l. 9). Lastly, the histogram of the bounding box size distribution is plotted (l. 11). The histogram bins and their values can afterwards be accessed by the `bins` and `counts` variables.

Listing 3.3: Python script for visualising the dataset.

```python
from custom_scripts.DataSetReader import DataSetReader

label_path = 'data/datasets/Senecio_dataset/Annotations'
dataset = DataSetReader(label_path)
dataset.loadLabels()
dataset.printStats(2)
# Heatmap
dataset.distrMatrix(dst=[int(val/4) for val in (1024,768)])
dataset.plotHeatMap()
# Histogram
fig,ax,counts,bins,bars = dataset.plotBBoxHist()
```

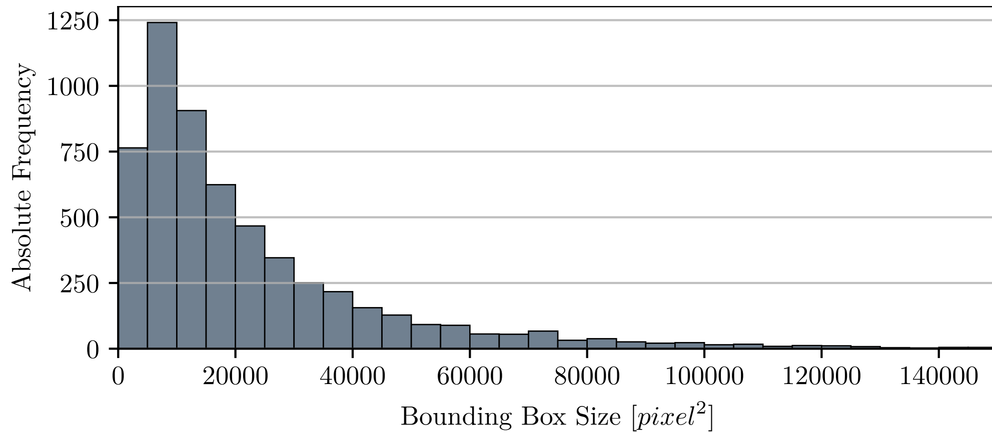

Figure 3.4: Histogram of the distribution of bounding box sizes in the dataset. The right tail is cut off since there are only very few outliers larger than 150000 $pixel^2$.

3.2 Training Toolchain

The training and inference of the model was done using an NIVIDA Jetson Xavier NX (NJX) Developer Kit which includes an NJX module attached to a reference carrier board which is designed for developing and testing software, specifically in the field of DL and computer vision. NVIDIA provides the JetPack Software Development Kit (SDK) which can be flashed onto a microSD card and used on the NJX. The SDK provides a Linux operating system and several software packages and libraries for DL and computer vision.

3.2.1 Preparing the Training

After flashing the SD card image, the jetson-inference GitHub repository was cloned [Franklin et al., 2016]. The repository includes a Docker image that can be used to build a Docker container which contains built-in models and necessary Python modules like PyTorch for running the scripts for training and inference of these models. The container can be navigated using a Linux terminal. Figure 3.5 shows the part of the directory tree which is relevant to this project.

The dataset created in section 3.1.3.3 was copied to the `data` folder and a new directory in the `models` folder was created for saving the checkpoints of the model during training.

Furthermore, the GitLab repository of this project [Zender, 2021] must be cloned to reproduce the results. The folder `senecio_jacobaea/mount` contains all necessary Python scripts for training and inference of SSD-MobileNet-v1.

Afterwards, the container was started and the `mount` folder was mounted into the container. Mounted folders are available inside the container after starting it and changes made inside the folder are persistent even when the container is shut down later on. The container can be started with the folder mounted into it as

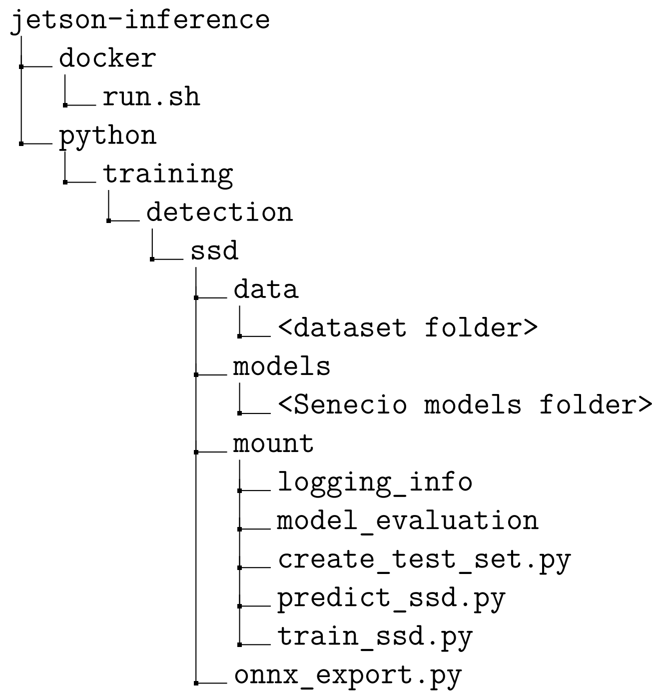

Figure 3.5: Part of the jetson-inference repository which is relevant for the project. Irrelevant folders and files are not shown here. Names enclosed with <> have to be replaced with the actual names used.

shown in listing 3.4 where <host path> must be replaced with the path to the senecio_jacobaea/mount folder.

Listing 3.4: Command for running the Docker container with the folder mounted.

```
1 /jetson-inference$ docker/run.sh --volume <host path>:
    python/training/detection/ssd/mount
```

3.2.2 Training the Model

The NVIDIA repository includes a script for training SSD-MobileNet-v1, pretrained on the MS COCO dataset, using PyTorch. However, the information on the training progress, such as the validation loss after each epoch, is only printed to the terminal. Therefore, the training script was modified using Pythons logging module to automatically create a log file in the logging_info folder in which all logging information is stored. This log file can later be used to visualize and evaluate the training progress of the model. The modified script is included in the mounted folder. The training was then started as shown in listing 3.5.

Listing 3.5: Command for training SSD-MobileNet-v1 on a custom dataset in PASCAL VOC format.

```
1 /mount$ python3 train_ssd.py --dataset-type=voc --data=data/<
    dataset folder> --model-dir=models/<Senecio models folder>
    --batch-size=2 --workers=1 --epochs=100
```

The paths to the dataset and the model output folder were specified according to figure 3.5 and the dataset-type was set to the PASCAL VOC dataset format.

Furthermore, the batch size was set to 2 which means that the 1702 images from the training dataset were split up into 851 batches containing 2 images each. The model performs backpropagation on both images and after completing a batch, the model parameters are updated according to equation 2.4. So, for each epoch, there are 851 optimization steps. The learning rate was left to its default value of 0.01. The number of epochs was set to 100 which needed 4:45 hours to complete.

3.2.3 Evaluating Logging Files

After the training script finishes, the log file of the training can be used to visualize the training process which helps to identify the model checkpoints which performed best on the validation dataset. To automatize the task, another Python class was used which can be found in the GitLab repository of the project. Listing 3.6 shows how to use it for analysing the log file.

Listing 3.6: Python script for reading the log file containing the logging information of the training process.

```python
from custom_scripts.LogReader import LogReader

log = LogReader(path='training_logs/logging_info_0016.log')
log.readLog()
log.linePlot()
log.sort().head(3)
```

After importing the class (l. 1), a `LogReader` object can be instantiated by passing in the path to the log file to analyse (l. 3). The `readLog()` method (l. 4) reads in the log file and creates a DataFrame out of it using the Pandas library [pandas development team, 2020; Wes McKinney, 2010]. `linePlot()` (l. 5) creates a line plot showing the validation loss throughout the training process. `sort()` (l. 6) returns the DataFrame sorted for the total validation loss in ascending order; i.e. the best performing model at the top.

Figure 3.6 shows a line plot of the validation loss during the training process. After completing an epoch, the model is evaluated on the validation dataset consisting of 214 images, then the average loss is calculated (validation loss), and a checkpoint of the model is stored to the model directory. In this case, the loss is calculated with two different loss functions, one for classification and one for regression. In classification tasks the output is discrete and the input has to be classified into one of those discrete categories whereas in regression tasks, the output is continuous [Muthukumar et al., 2020]. The classification loss refers to the error that the model makes concerning the assignment of the classes; i.e. does it assign the correct class to a detected object. In this case it only distinguishes between the `Senecio` class and a background class that it assigns to everything else. The regression loss refers to the error that the model makes concerning the bounding box coordinates. The total validation loss is the sum of classification and regression loss.

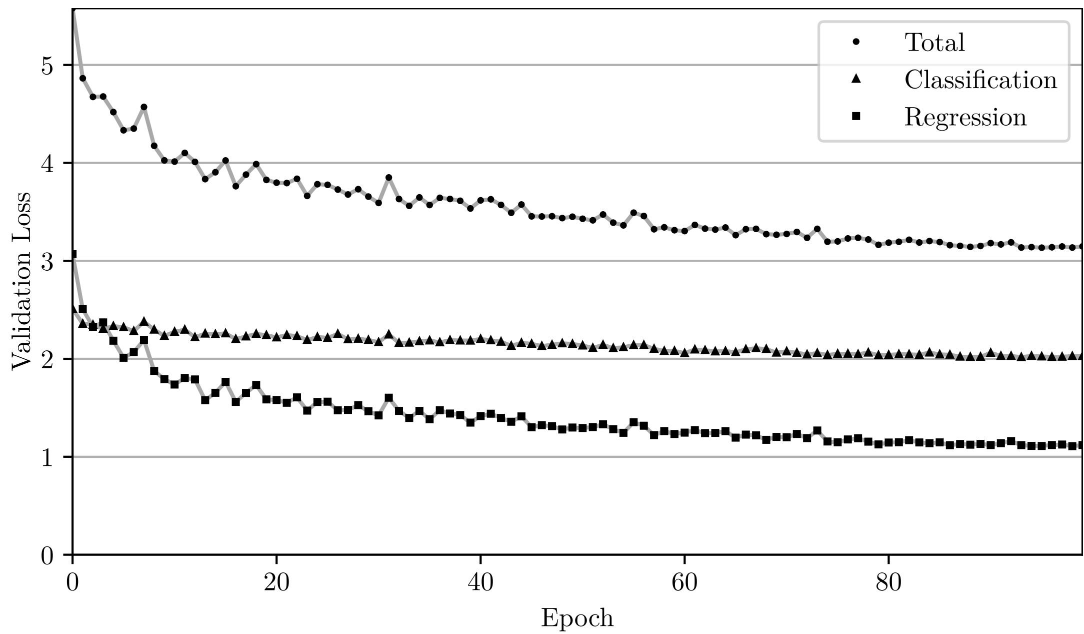

Figure 3.6: Line plot showing the validation loss (classification, regression, total) of the model during the training process.

The graph shows that the classification loss decreases, albeit slowly, over time. The regression loss initially decreases rapidly and then starts to approach a constant value. In the end, the total validation loss starts to stagnate around 3.13 which shows that continuing the training is unlikely to further improve the model. The graph further allows to monitor any under- or overfitting of the model. During the first epochs, the model is still underfit since the validation loss continues to decrease. If, at some point, the validation loss would start to increase again, it would signal that the model is overfit, which is not the case here. The three most promising model checkpoints (95, 98 & 93) are the ones with the lowest total validation loss as shown in table 3.3.

Table 3.3: Best performing model checkpoints according to the total validation loss.

Epoch	Total Validation Loss	Classification Loss	Regression Loss
95	3.1342	2.0239	1.1103
98	3.1356	2.0274	1.1082
93	3.1359	2.0173	1.1187

3.2.4 Model Export

After completing an epoch and calculating the validation error, the model checkpoint is saved as a PyTorch file (`.pth`) to the model directory. Under the `jetson.inference` module, NVIDIA provides the `detectNet` class for inference of object detection models. However, the model needs to be in Open Neural Network Exchange (ONNX) format to be used with `detectNet`.

The best model checkpoint (epoch 95) and the *labels.txt* file were therefore copied to a new sub-folder. Note that the *labels.txt* file from the model directory must be used (not from the dataset), since PyTorch adds a `BACKGROUND` class to the beginning of the file; if this class is omitted, all predictions with the model will be erroneous. The conversion from PyTorch to ONNX format was done using the Python script `onnx_export.py`, provided by NVIDIA [Franklin et al., 2016]. How to use it from the terminal is shown in listing 3.7.

Listing 3.7: Command for converting a model from `.pth` to `.onnx` format.

```
1 /ssd$ python3 onnx_export.py --model-dir=<model directory>
```

3.3 Training on Modified Datasets

This first model was trained on the original dataset. Before evaluating it, two more datasets were created by modifying the original dataset and the model was trained on each of them individually. The goal was to gain more insight later on into what factors influence the performance of the model. Specifically to answer the questions how the size of the *Senecio jacobaea* specimens influences the model performance and secondly if a larger dataset improves the accuracy of the model.

3.3.1 Dataset 2: Influence of Object Size

To find out how the size of the objects influences the results, the `Senecio` class from dataset 1 was split into three different size classes, a small `Senecio_s`, a medium-sized `Senecio_m`, and a large `Senecio_l` class. To which of these three classes a specimen belongs was determined by calculating its bounding box size and setting certain size thresholds for each class (see figure 3.7).

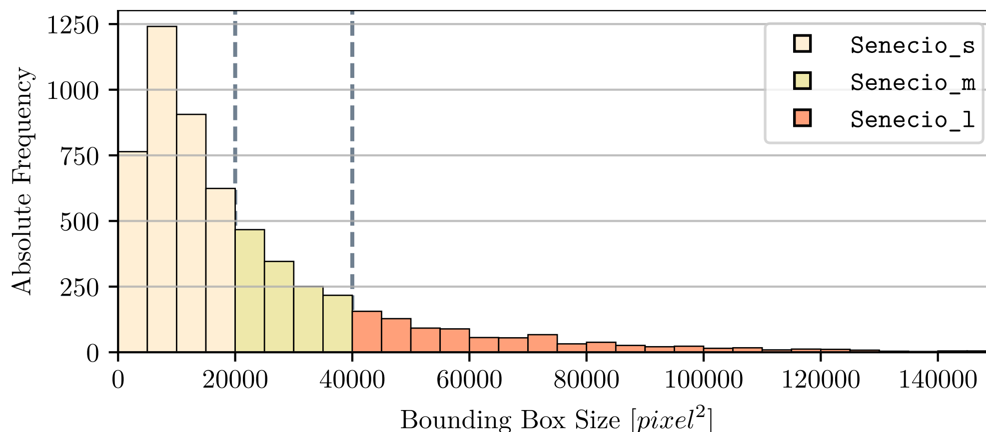

Figure 3.7: Histogram of the distribution of bounding box sizes in the dataset split into three classes of different bounding box size intervals.

The underlying idea was that, since all images were taken at the same height of 1 meter above ground and are thus scaled consistently, the bounding box size in the

image directly correlates to the size of the plant in the real world. A small bounding box would thus correspond to a small plant which, generally, should correspond to an early growth stage of *Senecio jacobaea*. As a specimen of *Senecio jacobaea* grows, the number, size, and shape of its leaves change and consequently also the nature of the features, that the model has to identify, changes. Therefore, splitting the `Senecio` class into three subclasses, which share more uniform features, could increase the accuracy of the model since it would be able to specialize for different growth stages. Furthermore, it would reveal how the size of a specimen affects the probability of the model correctly identifying it.

The boundaries for the size classes were set at 20000 and 40000 pixel^2 as shown in figure 3.7. The advantage of using fixed bounding box size thresholds for the classes is that the labels can be modified automatically instead of doing it manually in CVAT. This was done using another Python script which is shown in listing 3.8. After importing the class (l. 1), a list of available sub-datasets (same as the first dataset) was defined (l. 3-9). A `ClassDivider` [Zender, 2021] object was then instantiated and passed the list of sub-datasets to use and a path where the modified labels should be stored (l. 11,12). Afterwards, the original labels were copied to that folder (l. 14) and the class was divided into the three size classes using the defined boundaries (l. 15,16). The script then reads each XML file and determines the size of each bounding box. Then it checks to which interval the bounding box belongs, changes the class name accordingly and saves the XML file. The intervals for the three size classes are

- `Senecio_s`: $[0, 20000]$ pixel^2
- `Senecio_m`: $(20000, 40000]$ pixel^2
- `Senecio_l`: $(40000, \infty)$ pixel^2

Listing 3.8: Python script for dividing the `Senecio` class into multiple subclasses based on bounding box size.

```
1  from custom_scripts.ClassDivider import ClassDivider
2
3  dname_list = ['kam_210515_n_acb4',
4                'kle_210516_n_acb4',
5                'moe_210518_n_acb4',
6                'moe_210527_a_acb4',
7                'kam_210529_a_acb4',
8                'mil_210529_e_acb4',
9                'leu_210601_n_acb4']
10
11 dataset = ClassDivider(dataset_ids=dname_list,
12                        mod_dir='senecio_v002')
13
14 dataset.copyLabels()
15 dataset.divide(class_dividers=[20000, 40000],
16                class_labels=['Senecio_s', 'Senecio_m', 'Senecio_l'])
```

The script also returns the number of class instances for each class which were 3535 of `Senecio_s`, 1281 of `Senecio_m`, and 907 of `Senecio_l`. These modified labels were then used to create a second dataset which was done as described in section 3.1.3.3. The only difference was, that instead of passing in the IDs of the sub-datasets directly, tuples with the paths to images and the modified labels were used and the `labels.txt` file had to be adjusted. Furthermore, the same random seed was used as in the creation of the first dataset. Thus, the training, validation, and test sets of the new dataset contained the same images as in the first dataset such that the results later on can be compared.

The new dataset was then used to train the model as described in the training toolchain (section 3.2) and the log file was evaluated as can be seen in figure 3.8.

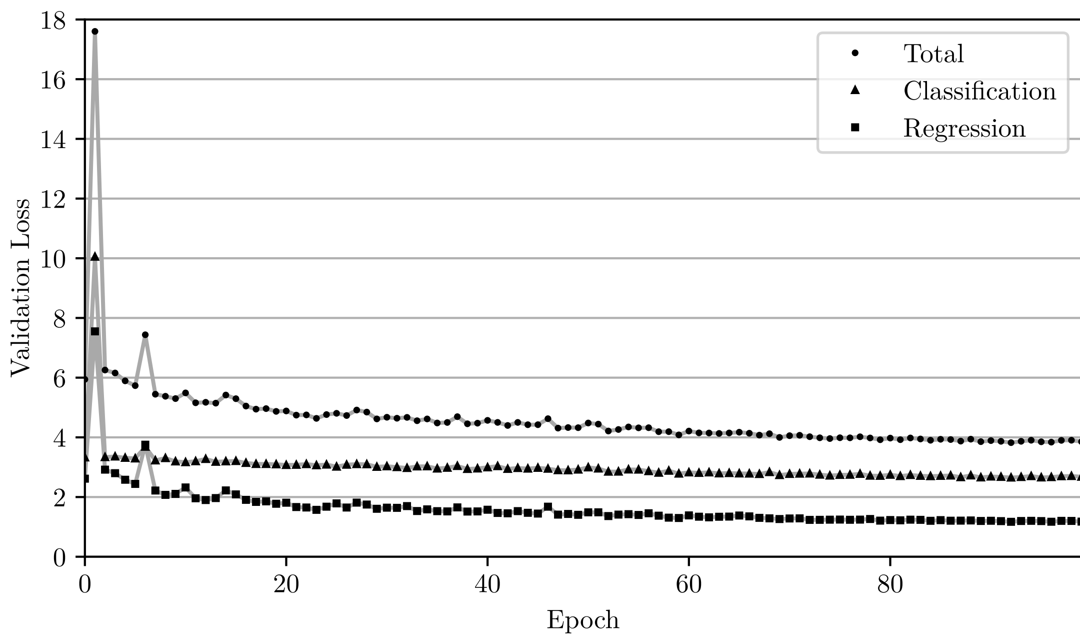

Figure 3.8: Line plot showing the validation loss (classification, regression, total) of the model during the training process on the second dataset.

The plot is similar to the one of the first model with the exception of the two extraordinarily pronounced spikes in the validation error after epoch 1 and 6. The best performing model checkpoints are shown in table 3.4. The regression loss is almost identical to the first model but the classification loss is on average 0.636 higher among the three best models. The model checkpoint of epoch 92 was exported to ONNX format for further evaluation.

Table 3.4: Best performing model checkpoints according to the total validation loss for the second training.

Epoch	Total Validation Loss	Classification Loss	Regression Loss
92	3.8227	2.6524	1.1703
96	3.8398	2.6672	1.1726
95	3.8512	2.6563	1.1949

3.3.2 Dataset 3: Influence of Dataset Size

A third dataset was created to answer the question whether collecting more data to increase the dataset size would be likely to increase the model accuracy as well. To test this hypothesis, the first dataset was copied and the training dataset was reduced by 50%. This was done manually by deleting half of the entries in the `train.txt` and `default.txt` files which define the images used for training. Afterwards, the training was done as described in the training toolchain (section 3.2).

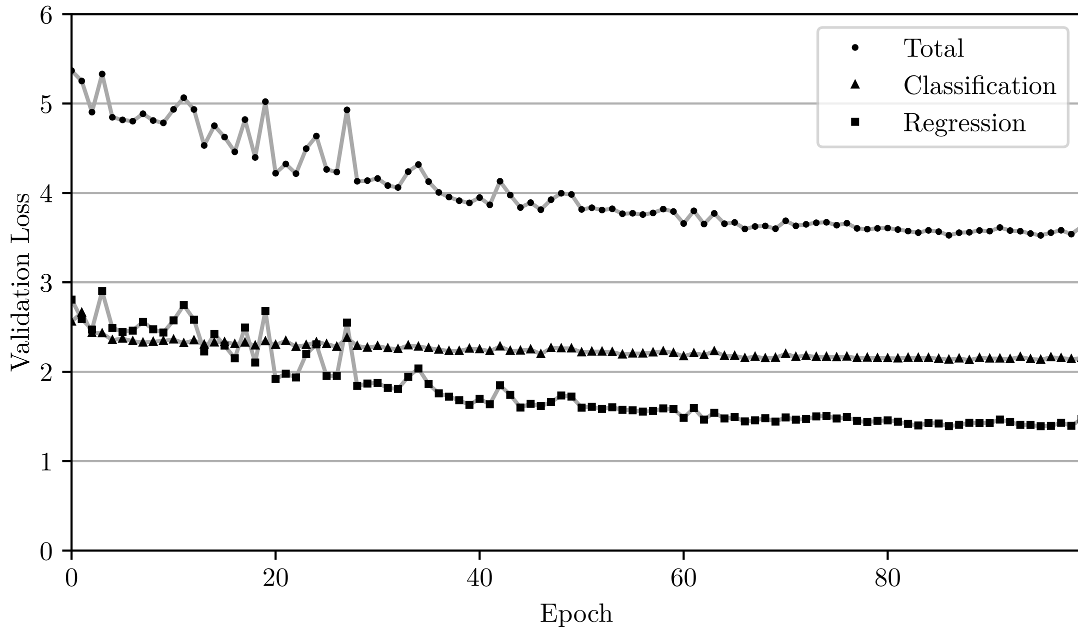

Figure 3.9: Line plot showing the validation loss (classification, regression, total) of the model during the training process on the third dataset.

Figure 3.9 shows the curve of the validation loss during the training process. For the most part it appears similar to the curve of the first model. The difference is that the curve does not decrease as stably but the validation loss fluctuates more from epoch to epoch, especially within the first half of the training. In the second half, the curve becomes more stable and approaches a constant value which is about 0.4 higher than the loss of the first model.

Table 3.5: Best performing model checkpoints according to the total validation loss for the third training.

Epoch	Total Validation Loss	Classification Loss	Regression Loss
95	3.5249	2.1347	1.3902
86	3.5259	2.1353	1.3906
98	3.5387	2.1419	1.3969

Table 3.5 shows the three best performing model checkpoints of the third training session. The classification loss is at a similar level as for the first model but the regression loss is about 0.28 higher for the third model. For further evaluation, the checkpoint of epoch 95 was exported to ONNX format.

3.4 Inference on Test Datasets

After finishing the training and exporting the best checkpoints, the performance of each model was evaluated. For that purpose, the models have to be applied on the test dataset [Yamashita et al., 2018]. The test dataset consists of completely new data that the model has not seen before and is defined by the `test.txt` file in the dataset folder (see figure 3.2).

3.4.1 Creating the Test Datasets

For inference with the model and subsequent performance evaluation, the images and XML labels of the test dataset were extracted from the main dataset using a Python script which can be applied as shown in listing 3.9.

Listing 3.9: Command for extracting the test dataset from a whole dataset in PASCAL VOC format.

```
1 /mount$ python3 create_test_set.py --dataset_path='<input path>'
    --output_folder='<output path>'
```

The script reads the `test.txt` file and extracts only images and labels of the test dataset into a directory structure as shown in figure 3.10.

```
<test dataset name>
├── images
├── labels
├── predictions
└── outputs
```

Figure 3.10: Directory tree of the test datasets.

`images` contains the images and `labels` contains the ground truth labels of the test dataset. `predictions` is used for storing the predictions of the model during inference and `outputs` is used to store any data on the model performance during evaluation later on. The script was used on each of the three datasets (original dataset, dataset with size classes, original dataset with 50% training data) to evaluate the three corresponding models individually on their own test dataset.

3.4.2 Inference

For the evaluation of the models, the toolbox provided by Padilla et al. [2021] was used which requires the predictions of the model to be in a certain format. Specifically, each image has to have its own text file in which each line represents a prediction of the model of the form

`<class name> <confidence> <left> <top> <right> <bottom>`

containing the class name, the confidence score and the absolute pixel coordinates of the top-left and bottom-right corners of the predicted bounding box.

NVIDIA provides the `detectNet` class which allows to use object detection models in ONNX format for inference in custom Python scripts and is included in the `jetson.inference` module of the GitHub repository [Franklin et al., 2016]. The class was used for applying the models on their respective test datasets and output the results in the correct format as described above. Listing 3.10 shows how to use the script for inference on the test datasets created in the previous section.

Listing 3.10: Command for applying custom object detection models on test images and storing the outputs as text files.

```
/mount$ python3 predict_ssd.py --model_path='<path to ONNX file>'
    --labels_path='<path to labels.txt>'  --image_path='<path to
    test images>' --predictions_path='<path to predictions folder>'
    --threshold=0.5
```

Note that the `labels.txt` containing the class labels must include the `BACKGROUND` class or the results will be erroneous. Furthermore, `detectNet` is implemented with NVIDIA's TensorRT which is an SDK that optimizes computational graphs of DL models to increase the inference performance. Therefore, when first executing the script, it may take a few minutes for the model optimization process to finish. Afterwards the optimizations are loaded from cache and the process becomes much faster.

The `threshold` argument refers to the confidence score threshold. Every detection of the model that has a confidence below the threshold is omitted. Since the best confidence threshold is dependent on the model itself and on the intended use, there is no universal value which should be used. Which threshold is best for application can be determined during the evaluation step. However, since the toolbox used for evaluation cannot handle a too high number of bounding boxes, the script was executed multiple times at 20%, 30%, 40%, and 50% confidence threshold. If the number of bounding boxes is too high for the toolbox for a given threshold, the next higher threshold can be used.

3.5 Model Evaluation

There is a number of different metrics which can be used to evaluate the performance of a CNN. While classification tasks can be evaluated with comparatively simple measures such as the F1 score [Lam et al., 2021], evaluating object detection models requires more sophisticated metrics since not only the classification, but also the bounding box parameters of a detected object have to be considered.

3.5.1 Evaluation Metrics

The metrics used for object detection models usually depend on the benchmark dataset used for evaluation. For the PASCAL VOC dataset, the Average Precision (AP)

of each class and the corresponding mean Average Precision (mAP) for all classes is used, evaluated at an Intersection over Union (IoU) of 0.50 [Padilla et al., 2021]. Since, the PASCAL VOC dataset format was used in this project, the results were evaluated by those metrics as well. The AP has the advantage that it evaluates the model against different confidence thresholds. The following gives a brief overview over the necessary concepts and the evaluation process by example.

Each class is evaluated independently by comparing how well the detected bounding boxes (txt files) represent the ground truth bounding boxes (XML files). When comparing ground truth values and predicted outcomes, one distinguishes four different cases:

- True Positive (TP): the model correctly detected *Senecio jacobaea*
- False Positive (FP): the model detected *Senecio jacobaea* where there is none in reality
- True Negative (TN): the model correctly detected that there is no *Senecio jacobaea*
- False Negative (FN): the model did not detect *Senecio jacobaea* when in reality there is a specimen

3.5.1.1 Intersection over Union (IoU)

Whether a detection is considered correct or not depends on the IoU which is the ratio of the intersecting area to the area of the union of ground truth bounding box B_{gt} and predicted bounding box B_p [Padilla et al., 2021] and is thus given by equation 3.1 which is illustrated in figure 3.11.

$$J(B_p, B_{gt}) = \text{IoU} = \frac{\text{area}(B_p \cap B_{gt})}{\text{area}(B_p \cup B_{gt})} \tag{3.1}$$

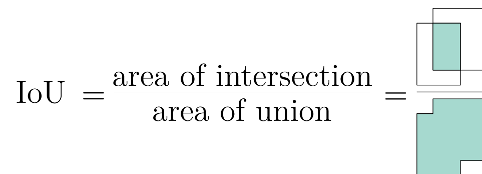

Figure 3.11: Intersection over Union (IoU). Own illustration, based on Padilla et al. [2021].

For the evaluation, an IoU threshold of 0.50 was used, which is the standard value [Padilla et al., 2021]. Therefore, predictions with an IoU higher than 0.50 for one of the ground truth labels of the same class are considered TPs, while detections with an IoU below 0.5 are considered FPs. An IoU of 1.0 would occur when both bounding boxes are perfectly covering each other, while an IoU of 0.0 would occur when there is no overlap between the bounding boxes at all.

3.5.1.2 Precision and Recall

As mentioned in the previous section, each detected object has a confidence value associated with it and during inference, a confidence threshold τ was used which filters out all predictions below the threshold. SSD-MobileNet-v1 outputs 3000 bounding boxes per image, most of which are false and have a low confidence value. The goal of the confidence threshold τ is to filter out all those false, low-confidence detections. What the threshold should be is dependent on the model itself and on the intended use, so it must be found experimentally using the test dataset.

When setting τ too high, the model would rarely make a false detection but it would miss a lot of objects in the image. On the other hand, when setting τ too low, it would find almost all objects but it would also make many false predictions.

This is described by the concepts of precision and recall. Precision refers to the percentage of correct positive predictions; i.e. out of all predictions, how many are true. Recall refers to the percentage of found ground truth boxes; i.e. out of all real objects, how many were detected [Padilla et al., 2021; Lam et al., 2021]. Since the number of TPs, FPs, TNs, and FNs is dependent on τ, precision and recall are also functions of τ. When considering a dataset with G ground truths objects and a model which outputs N detections of which S are true, precision and recall are given by equations 3.2 and 3.3, respectively.

$$Pr(\tau) = \frac{\sum_{n=1}^{S} \mathrm{TP}_n(\tau)}{\sum_{n=1}^{S} \mathrm{TP}_n(\tau) + \sum_{n=1}^{N-S} \mathrm{FP}_n(\tau)} = \frac{\sum_{n=1}^{S} \mathrm{TP}_n(\tau)}{\text{all detections}(\tau)} \tag{3.2}$$

$$Rc(\tau) = \frac{\sum_{n=1}^{S} \mathrm{TP}_n(\tau)}{\sum_{n=1}^{S} \mathrm{TP}_n(\tau) + \sum_{n=1}^{G-S} \mathrm{FN}_n(\tau)} = \frac{\sum_{n=1}^{S} \mathrm{TP}_n(\tau)}{\text{all ground truths}} \tag{3.3}$$

Note that $\mathrm{TP}(\tau)$ and $\mathrm{FP}(\tau)$ are decreasing functions of τ since a larger τ reduces both the number of TPs and FPs as they are filtered out by the threshold. On the other hand, $\mathrm{FN}(\tau)$ is an increasing function of τ as a higher threshold leads to more missed ground truth objects. Furthermore, $\sum TP(\tau) + \sum FN(\tau)$ is just the number of ground truth objects and thus a constant which is not dependent on τ. Hence, the recall $Rc(\tau)$ is a decreasing function of τ: the lower the threshold, the higher the recall as more detections are considered. The precision however can decrease or increase with increasing τ [Padilla et al., 2021].

An ideal model should find all ground truth objects (recall of 1.0) while not making any false predictions (precision of 1.0.).

3.5.1.3 Precision × Recall Curve

The Average Precision can then be calculated from the precision × recall curve ($Pr \times Rc$). Table 3.6 shows an example for how to process and evaluate the outputs of an object detection model with respect to the ground truth values.

Table 3.6: Example for evaluating outputs of an object detection model. The example dataset has 8 ground truth bounding boxes.

BBox	τ	IoU	IoU>0.5?	$\sum$TP	$\sum$FP	$Pr(\tau)$	$Rc(\tau)$	$Pr_{interp.}(\tau)$
B	99%	0.80	True	1	0	1.000	0.125	1.000
G	95%	0.93	True	2	0	1.000	0.250	1.000
D	94%	0.71	True	3	0	1.000	0.375	1.000
C	84%	0.44	False	3	1	0.75	0.375	0.800
A	76%	0.65	True	4	1	0.800	0.500	0.800
F	72%	0.00	False	4	2	0.667	0.500	0.714
E	67%	0.53	True	5	2	0.714	0.625	0.714

Each row represents a bounding box predicted by the model and the predictions are ordered by their confidence τ. Afterwards, the IoU with the corresponding ground truth bounding box is calculated and it is checked whether the IoU is above the threshold of 0.5. If that is the case, the prediction is considered TP, otherwise FP. The rows $\sum$TP and $\sum$FP represent the cumulative sum of TPs and FPs. For each row, precision and recall are calculated using equations 3.2 and 3.3 (the number of ground truth values is 8 in this example). Since the recall is a decreasing function of τ, it continues to increase with each row (with decreasing τ). On the other hand, the precision increases with each new TP and decreases with each new FP. Therefore, $Pr \times Rc$ shows a zig-zag pattern. The last column is the interpolated precision. For a given row, the interpolated precision is the maximum precision of that row and all rows below. The interpolated precision thus flattens the zig-zag pattern of the curve.

A table like 3.6 can help to find the right confidence threshold for the application of the model later on. For example, if the minimum precision during application should be 80%, a confidence threshold of 76% should be used. Or, if it is more important to find most of the objects rather than making always correct predictions, a confidence threshold of 67% should be used. Here, some of the precision is traded off in order to obtain a higher recall. Which of the two is more important would depend on the intended use.

Figure 3.12 shows the precision × recall curve of table 3.6. The black points are the precision and recall values from the table which show the typical zig-zag pattern. The red line is the interpolated precision × recall curve.

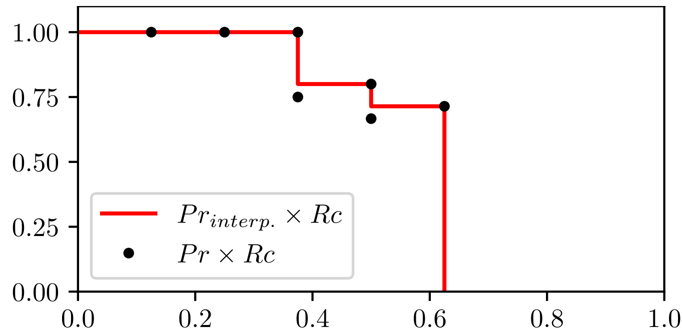

Figure 3.12: (Interpolated) Precision × Recall curve of table 3.6.

3.5.1.4 Average Precision and mean Average Precision

The Average Precision is the area under the interpolated $Pr \times Rc$ curve as shown in figure 3.12. Like precision and recall, the AP is always within 0 and 1. An AP of 1.0 means that the model found all ground truth objects without making any false predictions; i.e. both recall and precision are 1. An AP of 0.0 means that the model did not find any of the ground truth objects; i.e. both recall and precision are 0. For the example above, the AP was 0.5643 or 56.43%.

There are different ways of evaluating the area under the curve. One method is to sample the $Pr_{interp.} \times Rc$ curve at 11 equidistant points and calculate their mean value (N-point interpolation). Another method, which is the one that was used in this project, is the all-point interpolation which considers every point on the curve.

As mentioned before, the AP is calculated individually for each class. To measure the performance of a model as a whole, the mean Average Precision is used which simply calculates the mean AP of all classes [Padilla et al., 2021].

3.5.2 Toolbox for Object Detection Metrics

Padilla et al. [2021] provide an open-source toolbox for object detection metrics which allows to evaluate model performances based on ground truth labels and model outputs in a variety of commonly used formats. The GitHub repository of their project includes a more detailed description of how to install and run the toolbox.

Figure 3.13 shows the Graphical User Interface of the toolbox. In 1) and 2), the paths to the ground truth labels and the images of the test dataset were specified. In 3) the dataset format was set to PASCAL VOC format. In 4) the path to the model predictions was given and in 5) the format of the coordinates was chosen appropriately. In 6) the AP per class and the mAP were chosen as metrics to calculate at an IoU of 0.5. In 7), the output folder, where the evaluation results were saved, was specified. Finally, the evaluation was started in 8).

This process was done for each of the three models at the lowest possible confidence threshold. For model 1 and 2, the model predictions with 20% confidence threshold

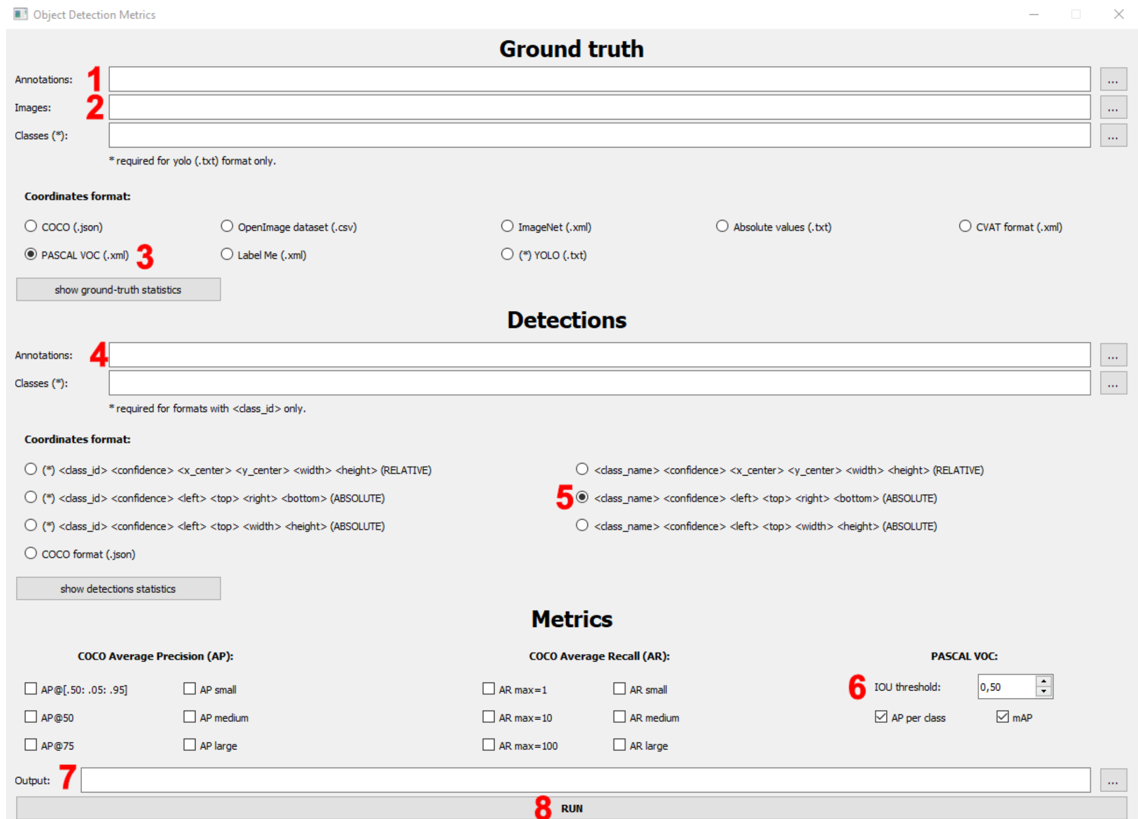

Figure 3.13: GUI of the object detection metrics toolbox provided by Padilla et al. [2021].

could be used, while for model 3 30% had to be used due to the large number of bounding boxes at 20% threshold which the toolbox was unable to process.

By default, the toolbox only prints the values of the requested metrics and saves plots of the precision × recall curves. Since the toolbox is an open-source software written in Python, the source-code could be slightly modified to save raw data as shown in table 3.6 as a Comma-Separated Values (CSV) file and some metadata in a text file. The raw data was then used to create and analyse the precision × recall curves in Python which is shown in the results.

3.5.3 Visualizing Model Outputs

For discussing the outcomes of the three models and finding possible problems which could be addressed in future works, a Python tool was written for plotting the test dataset images with the predicted and ground truth bounding boxes using OpenCV. The script can be found on the project repository [Zender, 2021] and can be run from the command line as shown in listing 3.11.

Listing 3.11: Command for running the ImageViewer utility for visualizing bounding boxes on images.

```
/custom_scripts$ python ImageViewer.py -i '<image path>' -l '<path to ground truth labels>' -p '<path to predicted bounding boxes>' -o '<output path>'
```

The command opens an OpenCV window which can be navigated using the keyboard. `a` and `d` can be used to skip through the images of the test dataset. `p` and `l` toggle the visibility of predicted and ground truth bounding boxes, respectively. `t` toggles the visibility of the class label. `s` saves the image as displayed to the output folder and `ESC` quits the program. The script was used to create the figures in the results section.

4 Results

The results are presented separately for each of the three models shown in table 4.1.

Table 4.1: Overview over the three different models which were evaluated and their properties.

Model	Epoch	Classes	Trained on	τ	IoU
1	95	`Senecio`	1702 images	20%	0.50
2	92	`Senecio_s`, `Senecio_m`, `Senecio_l`	1702 images	20%	0.50
3	95	`Senecio`	851 images	30%	0.50

4.1 Model 1: Original Dataset

Model 1 was trained on the original dataset consisting of 1702 training images and a single class `Senecio`. For evaluation, a confidence threshold of 20% was used. Table 4.2 shows the results of model 1 on the test dataset.

Table 4.2: Results of model 1 on the test dataset.

Parameter	Value
Total Positives (Ground Truth)	564 instances in 214 images
Total Positives (Predictions)	599 instances in 193 images
Total True Positives	198
Total False Positives	401
Average Area of BBoxes (Ground Truth)	23823.78
Average Area of BBoxes (Predictions)	52032.08
Precision	0.33055
Recall	0.35106
AP/mAP	0.21928

The model found 35.11% of the ground truth objects and 33.06% of its predictions were correct. Even though, model 1 did not distinguish between different size classes as model 2 does, it seems to be better at detecting larger specimens of *Senecio jacobaea*. This is shown by comparing the average areas of ground truth and predicted bounding boxes. While the average are of ground truth bounding boxes is only 23824 pixel^2, the average area of predicted bounding boxes was much larger at 52032 pixel^2. Overall, the model achieved an AP of 21.93% which is also the models mAP since it only has a single class `Senecio`.

Figure 4.1 shows the precision × recall curve (a) and the interpolated precision × recall curve (b) used for calculating the AP which is given by the area of the grey-shaded region under the interpolated precision × recall curve in b).

Since the detections are ordered by confidence scores before calculating precision and recall, the leftmost parts of the curve are associated with the highest confidence. The confidence score of the detections thus decreases with increasing recall. The lower the confidence score, the higher the probability that the detection is false. Therefore, the interpolated precision decreases with decreasing confidence and increasing recall. This explains the shape of the precision × recall curve. Setting a confidence threshold greater than 0 essentially cuts off the right part of the curve. However, since most predictions below the confidence threshold are false, the precision would rapidly drop while the recall would only slowly increase. The curve would thus get very flat and the area under the (interpolated) precision × recall curve would only marginally increase. Therefore, setting a confidence threshold of 20% or 30% does not significantly alter the resulting curves and APs.

It should be noted that the metrics like precision, recall, AP, and mAP not only depend on the confidence score but also on the used IoU threshold. For other dataset formats like MS COCO, those metrics are usually also calculated at different IoU thresholds [Padilla et al., 2021]. As explained, the IoU measures how well predicted and ground truth bounding boxes overlap and the IoU threshold determines whether a prediction is considered TP or FP. Choosing a higher, more restrictive IoU thus results in more FPs, consequently in a lower precision and recall, and therefore a lower AP and mAP, and vice versa. Since using an IoU of 0.50 with the PASCAL VOC dataset is the standard method, it was used here as well.

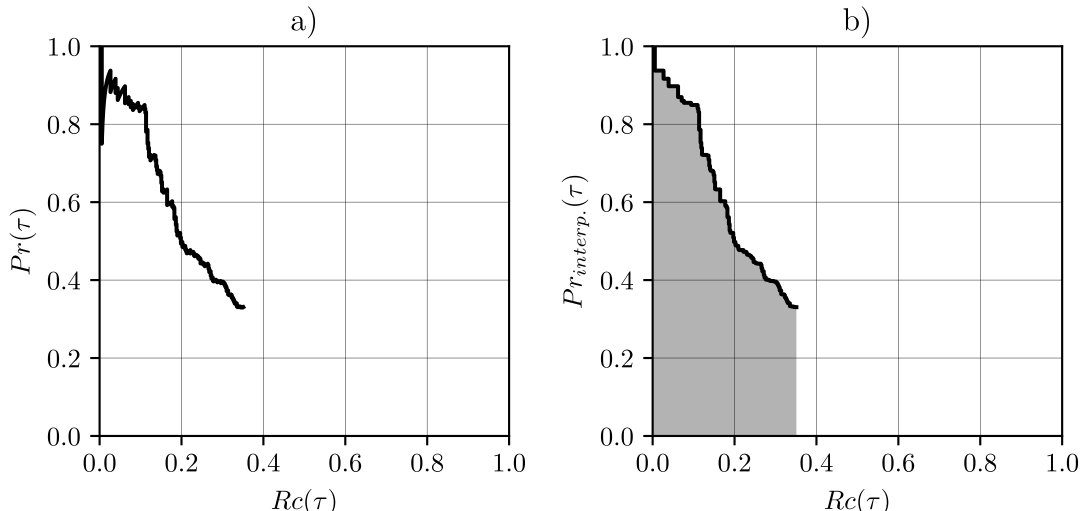

Figure 4.1: a) precision × recall curve, b) interpolated precision × recall curve of model 1 on the test dataset with $\tau = 0.2$ and an IoU of 0.50.

4.2 Model 2: Influence of Object Size

The training, validation and test dataset of model 2 were the same as with model 1 but the ground truth labels consisted of three different size classes. `Senecio_s` refers to all specimens of an area of 20000 pixel^2 or less. `Senecio_m` refers to all specimens between 20000 and 40000 pixel^2, and `Senecio_l` refers to all specimens larger than that. In total, the test dataset contained 564 ground truth objects in 214 images for all three classes combined. The model found 313 objects in 149 images of which 102 were TPs.

Table 4.3: Results of model 2 on the test dataset.

Parameter	Senecio_s	Senecio_m	Senecio_l
Instances in Complete Dataset	3535	1281	907
Total Positives (Ground Truth)	344	132	88
Total Positives (Predictions)	38	82	193
Total True Positives	13	36	53
Total False Positives	25	46	140
Precision	0.34211	0.43902	0.27461
Recall	0.03779	0.27273	0.60227
Average Precision (AP)	0.01995	0.17688	0.38626
mean Average Precision (mAP)		0.19436	

Table 4.3 shows the results for all three object classes of model 2 on the test dataset. Figure 4.2, 4.3 and 4.4 show the precision × recall curves (a) and the interpolated precision × recall curves (b) of model 2 for the classes `Senecio_s`, `Senecio_m` and `Senecio_l`, respectively.

Even though `Senecio_s` had almost 4 times as many samples in the complete dataset as `Senecio_l` and almost 3 times as many as `Senecio_m`, it had the least amount of detections while evaluating the model on the test dataset. As the amount of training data on small specimens was significantly larger than that of medium-sized and large specimens, the model should be optimized best for finding small *Senecio jacobaea*. In the test dataset, the number of small, medium-sized, and large specimens is comparable to the distribution in the complete dataset. Therefore, it could be expected that the model would find small specimens most consistently, which is not the case. It not only had an extraordinarily low recall, finding only 3.78% of the ground truth objects but also had the least total amount of predictions for `Senecio_s`. Even though the precision was close to the mean of all three classes, the low recall resulted in an AP of only 2% which is by far the lowest of the three classes.

The values for the medium-sized `Senecio_m` are in between of the two other classes apart from the precision. For `Senecio_m` the model had a significantly higher recall, finding 27.27% of all medium-sized ground truth specimens. Furthermore, the model showed the highest precision for medium-sized specimens since 43.90% of the

predicted objects were TPs. Overall, the AP for this class was 17.69% which is just below the mean.

Senecio_l, representing the large specimens, had the lowest number of samples in the training and test dataset but had the highest number of predictions by the model. The model found the majority of ground truth objects at a recall of 60.23% which is by far the highest value of any class in any of the three models. However, compared to the small and medium-sized classes, it also made the most false predictions resulting in a precision of only 27.46% which is the lowest of the three classes. Nevertheless, Senecio_l achieved the highest AP at 38.63%.

The mAP, the mean of the AP for all three classes, was 19.44% which is 2.49% less than that of model 1.

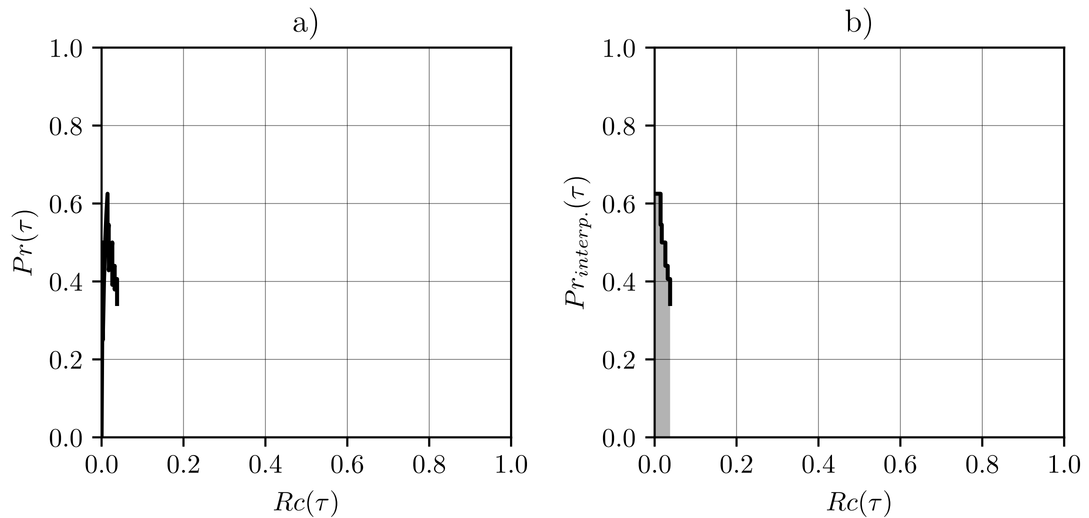

Figure 4.2: a) precision × recall curve, b) interpolated precision × recall curve of model 2 for class Senecio_s on the test dataset ($\tau = 0.2$; IoU = 0.50).

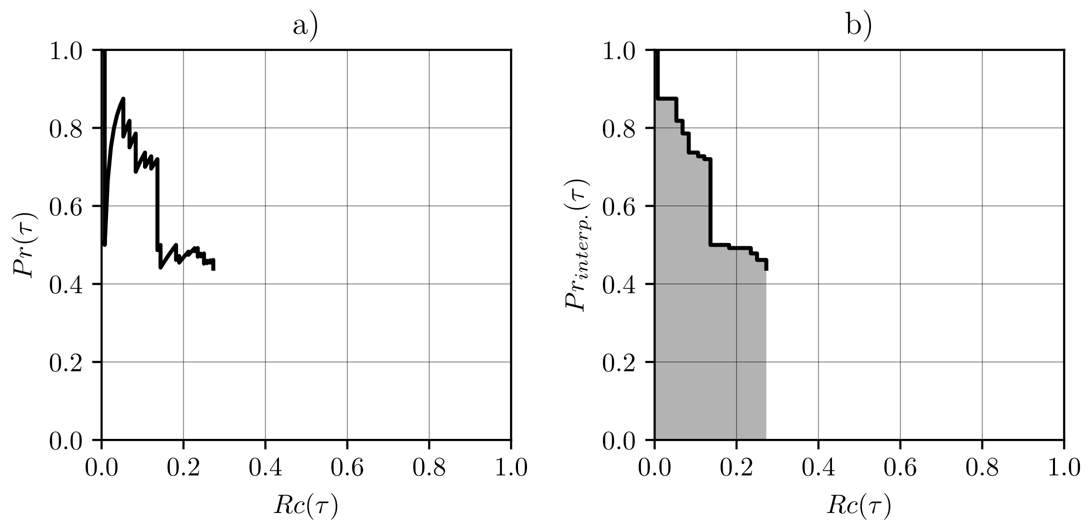

Figure 4.3: a) precision × recall curve, b) interpolated precision × recall curve of model 2 for class Senecio_m on the test dataset ($\tau = 0.2$; IoU = 0.50).

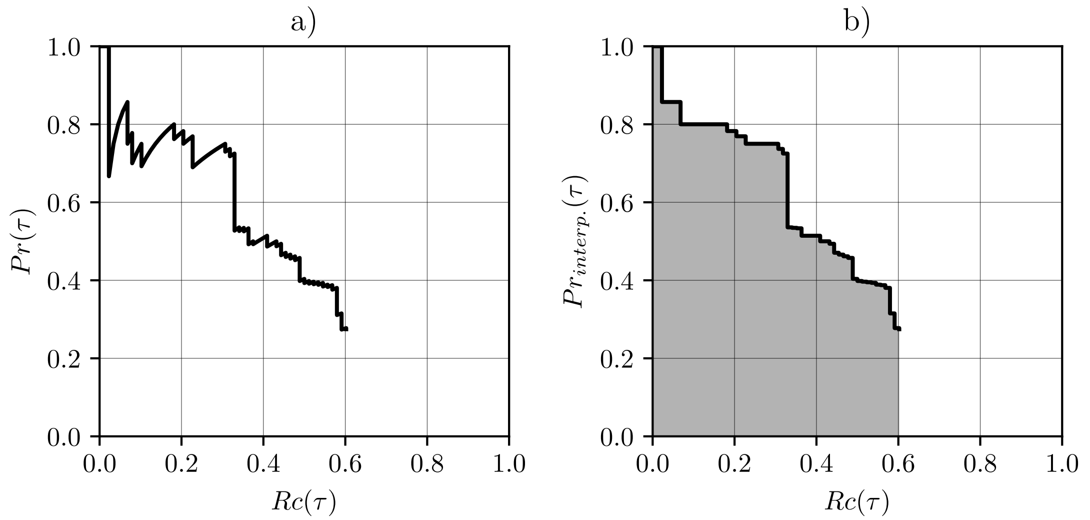

Figure 4.4: a) precision × recall curve, b) interpolated precision × recall curve of model 2 for class `Senecio_1` on the test dataset ($\tau = 0.2$; IoU = 0.50).

4.3 Model 3: Influence of Dataset Size

Model 3 was trained on the same dataset and the same `Senecio` class as model 1. The only difference was that the size of the training dataset was reduced by 50% to find out how much of an influence the dataset size has on the performance of the model. Furthermore, for the evaluation part, the confidence threshold was set to 30% as opposed to the 20% of models 1 and 2. At 20% confidence, the model predicted 241884 objects which was too much for the toolbox to handle. However, the vast majority of these predictions would have been FPs since there are only 564 ground truth objects in the test dataset resulting in a precision of less than 0.25%. Therefore, the area under the precision × recall curve would have been only marginally higher than with the 30% threshold.

Table 4.4: Results of model 3 on the test dataset.

Parameter	Value
Total Positives (Ground Truth)	564 instances in 214 images
Total Positives (Predictions)	51 instances in 35 images
Total True Positives	24
Total False Positives	27
Average Area of BBoxes (Ground Truth)	23823.78
Average Area of BBoxes (Predictions)	42084.12
Precision	0.47059
Recall	0.04255
AP/mAP	0.02876

Table 4.4 shows the results of model 3 on the test dataset. 47.06% of the predictions were correct but only 4.26% of the ground truth objects were found which is much less than in model 1. However, the final precision and recall cannot really be

compared with the other models due to the different confidence threshold, resulting in the curve being cut at a different position.

Figure 4.5 shows the normal (a) and interpolated (b) precision × recall curve of model 3 for the test dataset. When comparing the curves of model 1 and model 3, it can be seen that the curve of model 3 is much steeper; i.e. the precision of model 3 decreases much more rapidly with decreasing confidence and increasing recall. This is also shown by the very low mAP of 2.88% compared to model 1 with an mAP of 21.93%. Reducing the training dataset size by a factor of 2, reduced the mAP by a factor of 7.61.

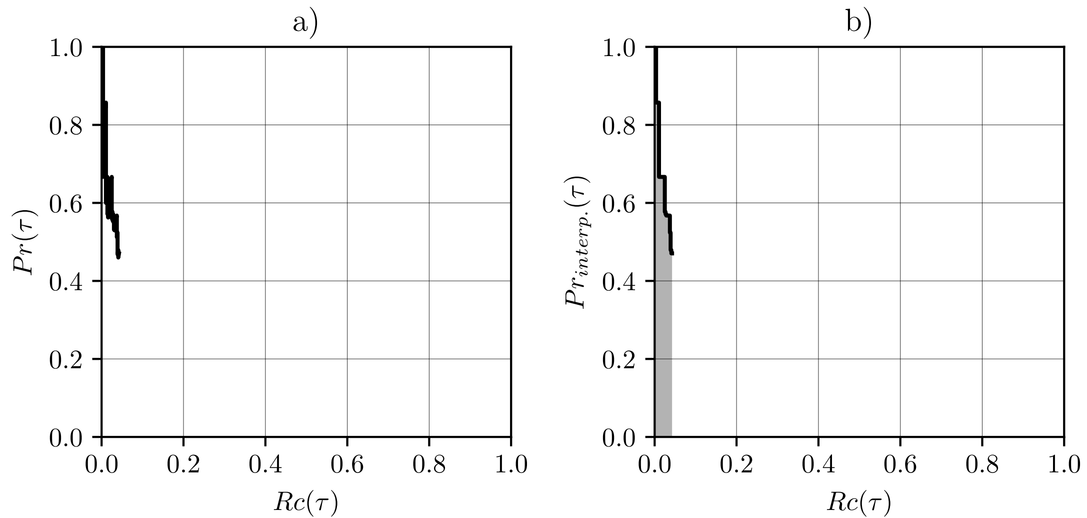

Figure 4.5: a) precision × recall curve, b) interpolated precision × recall curve of model 3 on the test dataset with $\tau = 0.3$ and an IoU of 0.50.

4.4 Samples from the Test Dataset

To see how the three models perform on the same image, five samples from the test dataset were collected using the `ImageViewer.py` script from section 3.5.3. For model 1 and 2 the 20% confidence threshold was used. For model 3 a 30% confidence threshold was used since otherwise the sample images would be filled with low-confidence bounding box predictions.

It should be noted that the five samples are not necessarily representative for the performance of the models for the whole dataset, which is better described by the precision × recall curves. The images were selected because they represent the diversity of the dataset well and highlight interesting outcomes of the different models.

Figure 4.6 shows the predictions of model 1 (a), model 2 (b), and model 3 (c) as well as the ground truth labels for image `kam_210515_n_acb4_0023`. Predicted bounding boxes are red, while ground truth bounding boxes are cyan. The image features a single large specimen of *Senecio jacobaea* which all three models managed

to detect. Model 1 and 2 overestimated the size of the bounding box while model 3 underestimated it. Furthermore, both model 1 and 2 predicted the specimen twice.

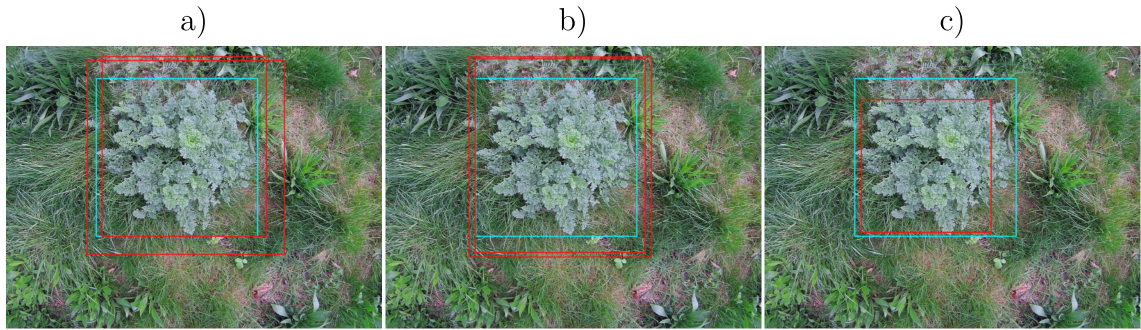

Figure 4.6: Predictions (red) of a) model 1, b) model 2, and c) model 3 on image `kam_210515_n_acb4_0023`. Ground truth labels are in cyan.

Figure 4.7 shows the model predictions for image `kam_210515_n_acb4_0039`. The image contains two medium-sized and one large specimen which are partly obscured by other vegetation or cut off at the image edge. While model 3 was not able to detect any specimens, model 2 detected two of them. Model 1 was the only one that found all three specimens. However, it predicted one of them twice and also falsely recognized a patch of grass as *Senecio jacobaea*.

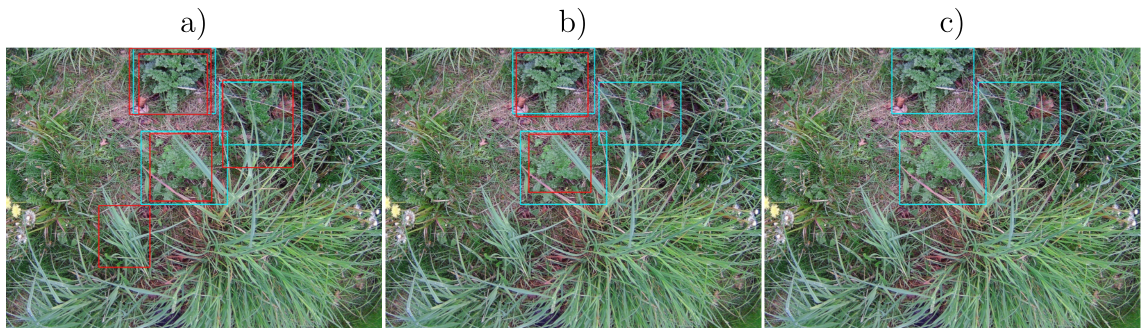

Figure 4.7: Predictions (red) of a) model 1, b) model 2, and c) model 3 on image `kam_210515_n_acb4_0039`. Ground truth labels are in cyan.

Figure 4.8 shows the model predictions for image `kle_210516_n_acb4_0184`. The image contains 17 small specimens of *Senecio jacobaea*. Despite the large number of ground truth objects, there were barely any predictions. Model 3 again did not find any objects. Model 2 did not find any specimen but falsely identified a patch of bare soil as *Senecio jacobaea*. Model 1 made the same false prediction but correctly identified the largest of the specimens twice and was thus the only model to make any correct prediction.

a) b) c)

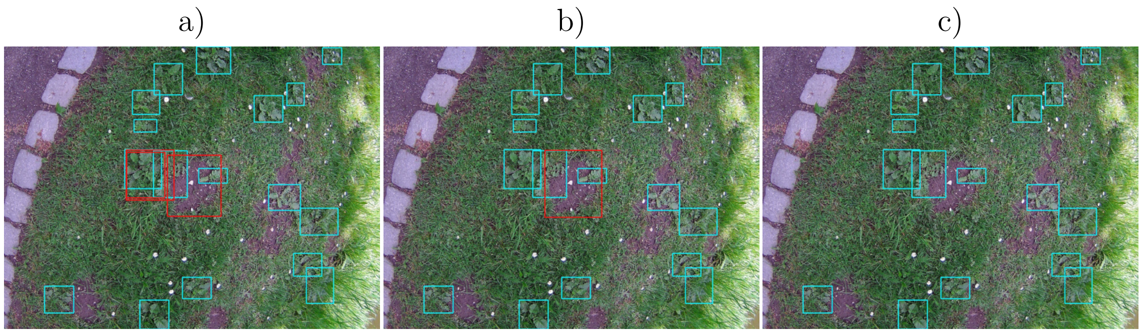

Figure 4.8: Predictions (red) of a) model 1, b) model 2, and c) model 3 on image kle_210516_n_acb4_0184. Ground truth labels are in cyan.

Figure 4.9 shows the model predictions for image leu_210601_n_acb4_0307 which contains a single medium-sized specimen. None of the models correctly identified the *Senecio jacobaea* specimen but all made different sets of false predictions. Model 1 and 2 made two, and model 3 made three false predictions mistaking *Urtica dioica* for *Senecio jacobaea*. Additionally, model 1 made three, and model 2 one false prediction for a specimen of *Rumex obtusifolius*.

a) b) c)

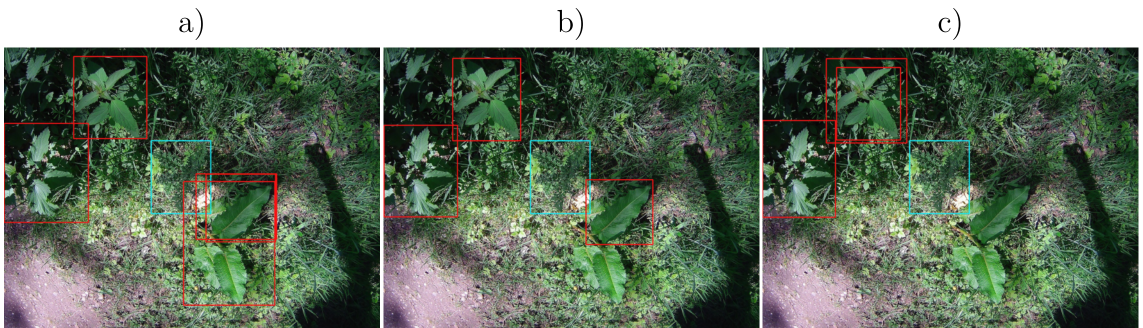

Figure 4.9: Predictions (red) of a) model 1, b) model 2, and c) model 3 on image leu_210601_n_acb4_0307. Ground truth labels are in cyan.

Figure 4.10 shows the model predictions for image moe_210527_a_acb4_0102 containing one large specimen. All three models found the specimen, model 1 predicting it twice. However, model 2 falsely identified one, and model 1 three specimens of *Cirsium vulgare* as *Senecio jacobaea*.

a) b) c)

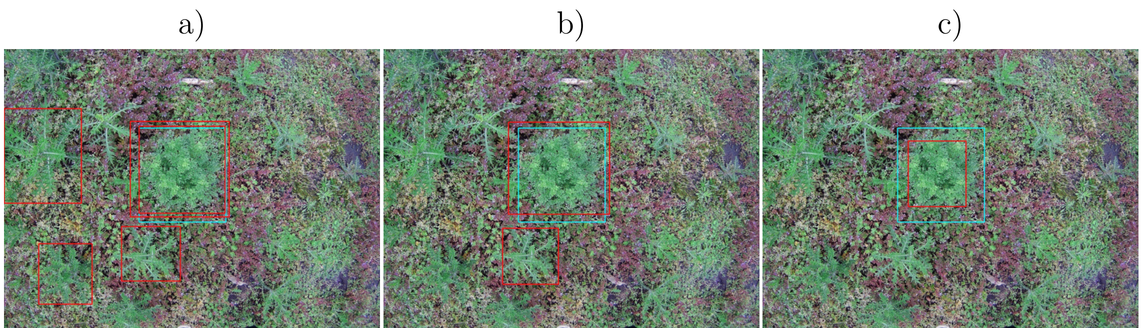

Figure 4.10: Predictions (red) of a) model 1, b) model 2, and c) model 3 on image moe_210527_a_acb4_0102. Ground truth labels are in cyan.

5 Discussion

5.1 Discussion of Sample Images

The sample images in the results section highlight some interesting problems which are worth to be discussed in more detail.

5.1.1 Distorted Bounding Boxes

During application, the model should be used to detect possible candidates for *Senecio jacobaea* in aerial images of a greater height (e.g. 5 meters). Each possible candidate would be enclosed in a bounding box from which the GPS coordinates could be extracted if the camera points directly downwards, the GPS coordinates of the camera are known, and the orientation of the drone is known.

If those conditions are met, the position of the specimen can be found by calculating the center point of the bounding box like it was done in the dataset visualization in section 3.1.4. In some of the test samples in section 4.4, the bounding boxes were distorted compared to the ground truth labels and did not cover the entire specimen or included unnecessary surrounding vegetation. While for certain tasks, such errors of the model might be an issue, it would likely not be a problem for this application since only the center point is relevant for calculating the position. If height and width of the bounding box are distorted equally on both sides (figure 4.6a and b, 4.10c), the center point would still be correct. Even if the distortion is slightly off on one side (figure 4.6c, 4.7, 4.10a and b) of the bounding box, it would still not be a problem as long as the center point of ground truth and predicted bounding boxes remain close to each other, which was the case for most TPs in the test dataset.

5.1.2 Small Bounding Boxes

When observing the sample images (figure 4.8), it becomes evident that the model clearly has problems detecting smaller specimens. This is further illustrated in table 4.3 showing the evaluation results of model 2. The smallest size class `Senecio_s` had by far the most instances in the whole dataset which means that the model should have learned to recognize that class the best. The test dataset also consisted of `Senecio_s` by more than 60%. However, not only the recall for that class was significantly lower than for the other classes but also the absolute number of pre-

dictions. The influence of the bounding box size also shows in model 1 and 3 when comparing the average sizes of predicted and ground truth bounding boxes (table 4.2 and 4.4). In both cases, the size of predicted bounding boxes was significantly higher indicating that mostly the large specimens were detected.

Compared to the two larger size classes, there was clearly no lack of training data for `Senecio_s`. Therefore, it can be concluded that the problem of the model not finding smaller specimens is directly related to the size of the objects. For a human eye, the small specimens were still easily recognizable during the labelling process (figure 4.8) at an image resolution of 1024×768 pixel. However, images are resized to 300×300 pixel before the model evaluates them [Franklin et al., 2016; Liu et al., 2016]. During this image compression, lots of information are lost and features like edges which are easily recognizable at a high resolution might be no longer visible. For detecting small objects in images, current object detection methods which are based on CNNs usually show a significantly lower performance, since small objects tend to lack characteristic shapes and textures [Stojnić et al., 2021]. Furthermore, the classification and regression of the bounding boxes is done using the inputs from the last layers. Since the size of the feature maps usually gets lower from layer to layer, these final layers have a much lower resolution than the input layer. A small specimen of *Senecio jacobaea* might then be represented by only a few or even a single pixel which makes it hard or even impossible to perform classification and bounding box regression for that object [Bosquet et al., 2018].

The only property of the specimen by which it can be identified at such a low resolution might be its color. This becomes problematic when the background colour is too similar which is likely the case here. As the surrounding vegetation is coloured in similar shades of green, the only property by which the small specimens can be found is lost causing the models to perform so poorly on smaller *Senecio jacobaea*.

5.1.3 False Predictions

During inference on the test dataset all three models made false predictions by either misidentifying other plants or background for *Senecio jacobaea* (FPs) or by not detecting an existing specimen (FNs).

5.1.3.1 False Positive Detections

In some cases the models made a false prediction for plants that are very similar in their appearance (figure 4.10a and b) due to their rosette-like growth form such as spear thistles (*Cirsium vulgare*) which were not only difficult for the models to distinguish but were sometimes even challenging during the labelling process due to their similar colour, growth form, and size. Since part of the shapes and textures that distinguish the two is lost during the resizing to the model input size, those plants commonly led to FPs. Other similar looking plants present in the dataset

such as catsear (*Hypochaeris radicata*) or common dandelion (*Taraxacum officinale*) were usually not as problematic due to their smaller size which made them harder for the models to find and misidentify.

In other cases (figure 4.9, 4.7a), the models misidentified plants that should be much easier to distinguish from *Senecio jacobaea* such as stinging nettle (*Urtica dioica*) or broad-leaved dock (*Rumex obtusifolius*), among others. However, sometimes the models falsely identified objects which have barely anything in common with *Senecio jacobaea* such as patches of bare soil (figure 4.8a and b) or tree trunks.

This raises the question what it is that the models are actually looking for when detecting *Senecio jacobaea*. During the labelling process, the specimens were identified by a variety of features. Smaller specimens could usually be recognized by their lyre-shaped leaves while larger specimens tended to have more, cabbage-like leaves. Good indicators in general were the rosette-like structure and the colour which usually had a blueish tint compared to the other vegetation. This is how a human would identify an object in an image. However, the predictions of the models might be made on a completely different basis.

Although there have been attempts at visualizing the features that are identified in feature maps [Zeiler and Fergus, 2014], deep learning is still considered a black box whose decisions cannot be properly explained [Yamashita et al., 2018]. How fundamentally different a CNN sees and evaluates an image compared to a human can be illustrated by adversarial examples. An adversarial example is an image which is manipulated in a subtle way by adding a small amount of noise, which is not visible to the human eye. However, the CNN changes its originally correct prediction and predicts a wrong class at a high confidence [Goodfellow et al., 2015]. This shows how different the vision of a CNN is compared to a human, which also makes it very difficult to explain why the model would falsely identify a patch of soil or a tree trunk as *Senecio jacobaea*.

The purpose of the model would be to find possible candidates of *Senecio jacobaea* and extract their GPS coordinates from aerial images captured on a UAV. Afterwards, another UAV or field robot could directly move towards those candidates and collect an image from a closer distance to evaluate at a higher precision whether the candidate actually is *Senecio jacobaea* or not. That means that all positive predictions would be validated using another more precise model trained on closer images. Thus, False Positive predictions would be sorted out in the confirmation step and would not cause any harm.

5.1.3.2 False Negative Detections

In other occasions, the models failed to identify a specimen of *Senecio jacobaea* resulting in a False Negative prediction. This might be caused due to the specimen being too small (figure 4.8), being partly obscured by other vegetation or being cut

off at the image edge (figure 4.7), or due to low contrast to the background (figure 4.9).

In contrast to FPs, FNs would not be validated again by the second UAV or field robot since their locations would not be mapped as they would not be detected by the first UAV. A false negative prediction of the model would thus have worse consequences as the error could not be corrected. An FP would just result in a small amount of additional work, while an FN would directly impair the quality of such a weed control project as the plant would not be found and removed.

Therefore, if such a model would be applied in a weed control project for *Senecio jacobaea*, it would be better to trade off a bit of the precision in order to increase the recall. It would be more important to find all specimens than to make only correct predictions since all predictions would be validated again before taking action. To achieve the desired trade-off between precision and recall, the confidence threshold must be set accordingly by analysing the precision × recall curve of the model.

5.1.4 Duplicate Bounding Boxes

In some instances, the model detected the same specimen more than once (figure 4.6a and b, 4.7a, 4.8a, 4.10a). These redundant bounding boxes are problematic during the evaluation of the model as well as during a possible application.

When using the model for weed control, each plant should only be detected once to ensure that it also only receives a single treatment to remove it. Otherwise, unnecessary amounts of herbicides would be used potentially causing harm to the ecosystem. Redundant bounding boxes should therefore be filtered out before calculating the GPS coordinates of the different candidates.

However, redundant bounding boxes already become problematic during the evaluation step. The toolbox used in this project [Padilla et al., 2021] directly uses the model outputs and ground truth labels for evaluating the model performance. Whether a detection is considered true of false is evaluated for each bounding box individually without taking other detections into account. The evaluation therefore does not include a filtering step in which redundant bounding boxes are sorted out. When considering each detection individually, only the class and the IoU of the predicted and ground truth bounding box determine whether it is considered true or false. In the examples from the test dataset mentioned above, the class was always predicted correctly and the IoU was always above the 0.5 threshold value used for evaluation. Therefore, all of the redundant bounding boxes would be considered true by the toolbox. This introduces an error to the precision and recall distorting the evaluation results. When observing the precision × recall curve, the model performs better on paper than it does in reality because specimens detected more than once are counted multiple times. That means, in reality, the models did not find as many of the specimens as indicated by their recall values. When observing the samples

from the test dataset, this seems to be especially problematic for model 1 which in general had the highest number of redundant bounding boxes.

Consequently, the duplicate bounding boxes should be filtered out of the model output to obtain a more realistic evaluation and to prevent repeated herbicide treatment of the same specimen. This can be done through a process called Non-Maximum Suppression (NMS) which is sometimes already included in multi-stage detectors like R-CNN [Padilla et al., 2021] but not in single-step detectors like SSD-MobileNet-v1. NMS is a post-processing algorithm which merges all detections referring to the same ground truth object [Hosang et al., 2017].

The algorithm processes each class individually. It finds the detection with the highest confidence score, assumes that it is true and then removes any detections which are likely belonging to the same ground truth object by calculating the IoU between the high-confidence detection and the remaining ones. If the IoU is above a certain threshold, i.e. the bounding boxes are rather similar, they are likely duplicates and the detection with the lower confidence value is removed. After the first filter step, the bounding box with the second highest confidence (which would belong to another object) is chosen and the process is repeated again to remove any of its duplicates. This is done until the list of detections is exhausted. Afterwards, the filtered model output can be used to evaluate it more realistically or to extract GPS coordinates of possible candidates of *Senecio jacobaea*.

It should be noted that the IoU threshold by which NMS filters the model output should be chosen with care since it may happen that two ground truth objects actually overlap substantially or are very closely together. If the threshold is set too low, only a single detection would remain even if the model originally found both objects. As a result, the recall would decrease as an actual, non-duplicate TP would be removed [Hosang et al., 2017]. The IoU for NMS must therefore be adjusted to suit the individual application.

5.2 Problems with the Dataset

Apart from the issues discussed previously based on the samples of the test dataset, there might be some general problems with the datasets used for training the models.

5.2.1 Missing Annotations

Even though, the dataset was created and labelled with great care, it is possible that there are errors in the dataset. In some cases, the specimens of *Senecio jacobaea* were difficult to identify for example due to similarities with other plant species, small size, or too low contrast with the surrounding vegetation. It can therefore not be guaranteed that every specimen in the dataset was found and some annotations

might be missing. A dataset containing incorrectly labelled data or data without a label decreases the quality of the trained model.

How strongly missing annotations affect the mAP of different object detection models was investigated by Xu et al. [2019]. The instance-level missing label rate M_r is the fraction of instance labels that are missing in a dataset. The study evaluated the performance of different Fully-Supervised Object Detection (FSOD) detectors like R-CNN, You Only Look Once (YOLO), or SSD which is the detector used in SSD-MobileNet-v1. The PASCAL VOC 2007 dataset was used to train the detectors with different missing label ratios M_r. The study concluded that FSOD detectors are considerably impinged by the quality of the dataset and that the mAP drops significantly as M_r increases. However, the significant drop in performance occurred for M_r larger than 0.5. For lower M_r, the mAP usually decreased only slightly [Xu et al., 2019]. Since the fraction of missed *Senecio jacobaea* specimens in the dataset is likely much lower then 50%, the effect of potentially missing labels on the mAP of the models can be assumed to be marginal.

5.2.2 Size Classes

A possible problem specific to the second model is the distribution of the dataset into three different size classes. As explained in section 3.3.1, the main idea was that, since all images were taken at the same height, the bounding box size corresponds to the growth phase of the specimen. As the shape and amount of leaves is typically dependent on the growth phase, dividing the `Senecio` class into sub-classes based on bounding box size would organize the specimens into groups with similar features making it easier for the model to learn them.

The distribution into the three size classes was done automatically by setting two size thresholds as shown in figure 3.7. This however raises the question how to set these boundaries to get the best result. Here, the thresholds were set more or less arbitrarily by observing the bounding box size distribution. The problem with using fixed threshold values for grouping the specimens is that they are grouped solely based on their size but not their features. Two specimens might be extremely similar in appearance while one belongs to `Senecio_s` and the other to `Senecio_m` because their bounding box size is close to the 20000 pixel^2 threshold. Furthermore, how the leaves of a specimen are spread (direction and angle to the ground) greatly influences the size of its bounding box which makes the method even more inaccurate.

The automatic grouping of data based on bounding box size was a very fast and easily adjustable method, which worked excellently for showing the influence of a specimens size on the models ability to find it. For optimizing the model performance, it would be better to group the data based on their features directly instead of their bounding box size. However, this would require much more work since the whole dataset would have to be relabelled manually.

5.2.3 Composition of the Dataset

Another issue might be that the different conditions under which the data was collected are not represented equally in the dataset as can be seen in table 3.2. The majority of images was taken at noon, during sunny weather in forest or park areas. In contrast to that, images taken in the morning or evening or in pastures and meadows are under-represented in the dataset. If the model is to face such conditions during application, it would have more difficulties to recognize *Senecio jacobaea*.

Therefore, for making the model more robust to various conditions, the dataset should represent the different circumstances more evenly.

5.2.4 Illumination of *Senecio jacobaea*

Lastly, differing lighting conditions could be problematic during application of the model since they already posed difficulties during the labelling process. The annotations for data collected on cloudy days or within shady areas were usually easier to create compared to the annotations of images with a lot of direct sunlight.

Areas which were not illuminated directly by the sun tended to be much more homogeneous in terms of brightness and showed less contrast. Especially the diffuse lighting during cloudy weather caused each surface to be illuminated more evenly. Due to this, it was easier to identify and follow edges in the image and distinguish different objects. Furthermore, the colour of *Senecio jacobaea* appeared to be much more homogeneous among those images.

Opposed to that, images taken under direct sunlight were often much more difficult to annotate. Since only the surfaces facing the sun were properly illuminated, the images were high in contrast and there was a lot of noise in the image due to shade of overlapping vegetation. Furthermore, the colour of *Senecio jacobaea* was much more inconsistent among the images taken under sunny conditions. Sometimes the leaves partly appeared as a dark green or even black when there was too much shade while in other occasions the leaves appeared white when they reflected a lot of sunlight back to the camera. These inconsistencies in colours and shapes might make it much more difficult for the model to properly learn the features of *Senecio jacobaea* and to identify it in the field.

Of course it would be convenient if the model would be robust to such inconsistencies and different circumstances but that might be difficult to achieve.

5.3 How could the model be improved

There are several things which could be done to improve the model. As mentioned before, DL projects require a lot of annotated data which was also evident when

comparing the performance of model 1 and model 3. Model 1 was trained on the default dataset and achieved a mAP of 21.93%. Model 3 was trained on the same dataset but the training dataset was reduced by half which resulted in a mAP of only 2.88%. Therefore, increasing the dataset size would likely yield a better performing model.

When increasing the dataset size, it should be done such that all possible circumstances are represented equally to make the model more robust to various conditions in the field.

Furthermore, a Non-Maximum Suppression should be applied to the output of the model. This would ensure that each object is only detected once by removing any duplicate bounding boxes. Thus, the model evaluation would be more precise and each specimen would be treated only once during a weed control project, sparing the ecosystem from unnecessary amounts of herbicides.

The problem of the small specimens not being detected could be addressed with different approaches. One possibility would be to cut each image into four slices overlapping each other in the middle and using each slice individually as a a new sample. Figure 5.1 shows image `kle_210516_n_acb4_0184` (1024×768 pixel^2) split up into four slices (656×492 pixel^2) which overlap each other in the center.

Figure 5.1: Image `kle_210516_n_acb4_0184` (1024×768 pixel^2) cut into four slices (656×492 pixel^2) overlapping each other in the center.

The input images are resized to 300×300 pixel^2 before being evaluated by the model. Using the default dataset, the whole image would be resized to 300×300 pixel^2 but when using image slices, each slice would be resized to 300×300 pixel^2. Therefore, small specimens would appear much larger in the model input and their distinct features would be conserved better. Model 2 showed that medium sized specimens were much easier to find than small specimens. Since slicing the images would essentially convert most of the `Senecio_s` specimens to `Senecio_m` specimens (in terms of specimen size in the model input), it can be expected that the model would perform at a higher precision and recall. On the other hand, when applying the model later on, it could only process a smaller area at a time or the same area at a lower Frames Per Second (FPS) since each image taken by the camera would have to be sliced and evaluated four times by the model. This method would thus trade some of the FPS for a higher mAP.

Another option would be to train and test other models using the same dataset. There are object detectors which work with a larger input image size which might be better for detecting small specimens. An example would be YOLOv4 whose backbone uses 512×512 pixel^2 images as an input [Bochkovskiy et al., 2020]. It would also be possible to use multi-stage object detectors like R-CNN but that might exceed the computational power of the NIVIDA Jetson Xavier NX. The disadvantage would be that the format of the dataset would have to be adjusted to the new model and a new training toolchain would have to be established.

Lastly, it may be an option to analyse the spectral signature of *Senecio jacobaea*. Figure 2.2 shows that *Senecio jacobaea* tends to have a more blueish colour than most of its surrounding vegetation. An additional multispectral camera could collect images of different wavelengths which could be used to calculate a spectral index that highlights leaves of a such a blueish tint. Spectral indices like the Normalized Difference Vegetation Index (NDVI) were already used successfully in precision agriculture projects to detect plants in the field [Mogili and Deepak, 2018]. If there was an area highlighted by the spectral index, the confidence of detections in the model output for that area could be increased before applying the NMS. So, if an area is more likely to contain *Senecio jacobaea* because the spectral index highlights leaves of a blueish tint there, the confidence for detections there should be artificially increased based on the value of the spectral index. This could be a way of improving the model output without changing the model itself.

5.4 Conclusion

It has been proven that it is possible to detect *Senecio jacobaea* in image data containing background vegetation of similar colour and appearance at a height of 1m above ground using SSD-MobileNet-v1. A toolchain was proposed which includes all

steps from collecting and annotating raw data, visualizing the dataset, training and inference of the model, to evaluating the final outputs. By modifying the original dataset, it was shown that the quality of the model highly depends on the size of the training dataset. Furthermore, the results demonstrated that the ability of the model to detect *Senecio jacobaea* was especially reliant on the size of the specimen where larger specimens had a much higher chance of being detected. Overall, this project can be seen as a prove of concept that object detection of *Senecio jacobaea* in the field is a solvable albeit challenging problem.

Bibliography

Bochkovskiy, A., Wang, C., and Liao, H. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. *CoRR*, abs/2004.10934.

Bosquet, B., Mucientes, M., and Brea, V. M. (2018). STDnet: A ConvNet for Small Target Detection. *Proceedings of the British Machine Vision Conference (BMVC)*.

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Deutscher Verband für Landschaftspflege e.V. (2017). Kreuzkräuter und Naturschutz. In *Tagungsband der internationalen Kreuzkraut-Fachtagung in Göttingen 2017*, Nr. 23 der DVL-Schriftenreihe Landschaft als Lebensraum.

Dumoulin, V. and Visin, F. (2018). A guide to convolution arithmetic for deep learning. *arXiv e-prints*.

Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. (2010). The PASCAL Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338.

Franklin, D., Yato, C., Kislan, T., Nguyen, D., Darbha, R., Tran, B., and Linderoth, M. (2016). Hello AI World. `https://github.com/dusty-nv/jetson-inference`.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and Harnessing Adversarial Examples. *arXiv e-prints*.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hosang, J. H., Benenson, R., and Schiele, B. (2017). Learning non-maximum suppression. *CoRR*, abs/1705.02950.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*.

Kieloch, R. and Domaradzki, K. (2011). The role of the growth stage of weeds in their response to reduced herbicide doses. *Acta Agrobotanica*, 64:259–266.

Lam, O. H. Y., Dogotari, M., Prüm, M., Vithlani, H. N., Roers, C., Melville, B., Zimmer, F., and Becker, R. (2021). An open source workflow for weed mapping in native grassland using unmanned aerial vehicle: using Rumex obtusifolius as a case study. *European Journal of Remote Sensing*, 54(sup1):71–88.

Lampen, A. (2017). Risikobewertung: Wie hoch ist die Gefährdung duch Pyrrolizidin-Alkaloide? In *Tagungsband der internationalen Kreuzkraut-Fachtagung in Göttingen 2017*, Nr. 23 der DVL-Schriftenreihe Landschaft als Lebensraum, pages 25–34.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proc. of the IEEE*.

Lin, T.-L., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. (2015). Microsoft COCO: Common Objects in Context. *arXiv e-prints*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. *arXiv e-prints*.

Mogili, U. R. and Deepak, B. B. V. L. (2018). Review on Application of Drone Systems in Precision Agriculture. In *International Conference on Robotics and Smart Manufacturing (RoSMa2018)*, 133, pages 502–509.

Muthukumar, V., Narang, A., Subramanian, V., Belkin, M., Hsu, D. J., and Sahai, A. (2020). Classification vs regression in overparameterized regimes: Does the loss function matter? *CoRR*, abs/2005.08054.

Neumann, H. and Huckauf, A. (2015). Jakobs-Kreuzkraut (Senecio jacobaea): eine Ursache für Pyrrolizidin-Alkaloide im Sommerhonig? *Journal für Verbraucherschutz und Lebensmittelsicherheit*, 11:105–115.

NVIDIA (2021). Jetson Xavier NX Developer Kit. `https://developer.nvidia.com/embedded/jetson-xavier-nx-devkit`.

Padilla, R., Passos, W. L., Dias, T. L. B., Netto, S. L., and da Silva, E. A. B. (2021). A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit. *Electronics*, 10(3).

pandas development team, T. (2020). pandas-dev/pandas: Pandas.

Prashanth, D. S., Mehta, R. V. K., and Sharma, N. (2020). Classification of Handwritten Devanagari Number – An analysis of Pattern Recognition Tool using Neural Network and CNN. *Procedia Computer Science*, 167:2445–2457. International Conference on Computational Intelligence and Data Science.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2015). You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C., and Li, F. (2014). ImageNet Large Scale Visual Recognition Challenge. *CoRR*, abs/1409.0575.

Sekachev, B., Manovich, N., Zhiltsov, M., Zhavoronkov, A., Kalinin, D., Hoff, B., TOsmanov, Kruchinin, D., Zankevich, A., DmitriySidnev, Markelov, M., Johannes222, Chenuet, M., a andre, telenachos, Melnikov, A., Kim, J., Ilouz, L., Glazov, N., Priya4607, Tehrani, R., Jeong, S., Skubriev, V., Yonekura, S., vugia truong, zliang7, lizhming, and Truong, T. (2020). opencv/cvat: v1.1.0.

Stojnić, V., Risojević, V., Muštra, M., Jovanović, V., Filipi, J., Kezić, N., and Babić, Z. (2021). A Method for Detection of Small Moving Objects in UAV Videos. *Remote Sensing*, 13(4).

Sumit, S., Watada, J., Roy, A., and Rambli, D. (2020). In object detection deep learning methods, YOLO shows supremum to Mask R-CNN. *Journal of Physics: Conference Series*, 1529:042086.

Suter, M. and Lüscher, A. (2017). Habitatpräferenzen von Jakobs- und Wasser-Kreuzkraut und Risikofaktoren für deren Auftreten. In *Tagungsband der internationalen Kreuzkraut-Fachtagung in Göttingen 2017*, Nr. 23 der DVL-Schriftenreihe Landschaft als Lebensraum, pages 9–17.

Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.

Xu, M., Bai, Y., and Ghanem, B. (2019). Missing Labels in Object Detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

Yamashita, R., Nishio, M., Do, R. K. G., and Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights Imaging*, 9:611–629.

Zehm, A. (2017). Auf welchen Flächen mit Relevanz für den Naturschutz sollen welche Kreuzkräuter reguliert werden? In *Tagungsband der internationalen Kreuzkraut-Fachtagung in Göttingen 2017*, Nr. 23 der DVL-Schriftenreihe Landschaft als Lebensraum, pages 19–22.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. In Fleet, D., Pajdla, T., Schiele, B., and Tuytelaars, T., editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham. Springer International Publishing.

Zender, J. (2021). Object Detection based on Convolutional Neural Networks of Senecio jacobaea for Weed Control. `https://git.hsrw.eu/21125/thesis_doc`.

Declaration of Authenticity

I, Jonas Moritz Zender, hereby declare that the work presented herein is my own work completed without the use of any aids other than those listed. Any material from other sources or works done by others has been given due acknowledgement and listed in the reference section. Sentences or parts of sentences quoted literally are marked as quotations; identification of other references with regard to the statement and scope of the work is quoted. The work presented herein has not been published or submitted elsewhere for assessment in the same or a similar form. I will retain a copy of this assignment until after the Board of Examiners has published the results, which I will make available on request.