JOHANNES KEPLER
UNIVERSITY LINZ

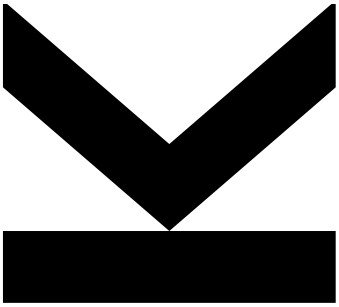
Submitted by
Christina Humer

Submitted at
Institute of Computational Perception

Supervisor
a.Univ.-Prof. Dr. Josef Scharinger

June 2020

Early Detection of Spruce Bark Beetles using Semantic Segmentation and Image Classification



Master Thesis
to obtain the academic degree of
Diplom-Ingenieurin
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

The fight against the "eight-toothed spruce bark beetle" (Ips Typographus) is a crucial task in forest management. Since the conditions for spruce bark beetles are becoming better due to climate change and monocultural farming, it is necessary to develop more efficient techniques to fight their spreading.

There is some recent research in the area of bark beetle or dead wood detection, however, all of the projects have slightly different goals and experiments with different approaches. Using recent research from semantic image segmentation and image classification with methods from deep learning, a novel approach to detect and identify infestation of spruce trees by the eight-toothed spruce bark beetle was developed in this thesis.

More concretely, this thesis covers the localization of sick spruce trees from a bird eye's perspective and the identification of infested spruce trees directly from the trunk, with the help of RGB images taken by drone and smartphone cameras.

The development of a mobile application allows users to perform inference in real-time and therefore leads to a more efficient and fast removal of infested trees.

With only 35 annotated images available for the segmentation task and 80 annotated images for the image classification task, it was necessary to use techniques like transfer learning or data augmentation to artificially enlarge the dataset.

The solutions presented in this project and the suggestions for future work should guide further research in this area and could be viewed as an advanced baseline for such work.

Contents

1 Introduction

The spread of the "eight-toothed spruce bark beetle" (Ips Typographus) in Austrian forests leads to an extreme loss of forest areas previously covered by spruce trees [Hoch and Perny, 2019].

As stated in [Hoch and Perny, 2019] and [Jakoby et al., 2015], it is proven that there is a correlation between increasing temperature due to climate change and the spread of bark beetles. Dry summers weaken the defense system of spruce trees against invaders because they lack the ability to produce resin due to the limited amount of water available to them. Furthermore, warm spring months enable bark beetles to begin their reproduction cycle early in the year and therefore more generations are produced throughout the year.

Especially forests with monocultural farming are affected by a fast spread of bark beetles since it is very easy for them to find a new habitat for breeding i.e. they do not have to search for long to find new uninhabited spruce trees, but they can just take the neighbor tree.

In this thesis domain specific knowledge about spruce bark beetles is leveraged to produce a machine learning system that is able to detect infested trees and therefore support the early detection of spruce bark beetles. This is done in a two step procedure where the first step is the localization of sick trees and the latter step is the identification of infested trees.

For developing the machine learning models, image processing techniques like classification and semantic segmentation are used. The experiments of either part of the project are compared to appropriate baselines, where one of them is taken from the experiments introduced in [Humer, 2020].

Due to the significance of early infestation detection, an app was developed that provides basic functionalities for real-time classification of images.

The models were created using the python machine learning API "Keras"[1]. Furthermore, "TensorFlow Lite" was used to create a version of the model that can be deployed on mobile devices as it is needed for the "Real-Time Classification" part of this project.

Although, the research possibilities were limited by low computational resources and little amount of training data, reasonable results were found. Techniques like transfer learning and data augmentation were a small remedy for the latter of these problems, but the computational limitations could not be fully resolved and might suggest that there is still a lot of potential to enhance the results of this project.

[1]See online: https://keras.io/

2 Related Work

Research in the direction covered in this thesis is sparse. To the best of out knowledge there is no research in the direction of detecting bark beetles based on images of the trunk as it was done in the "Classification" part of this thesis.

There was, however, research on bark beetle detection similar to the "Localization" part of this project by [Safonova et al., 2019]. Their focus was the detection of a special bark beetle species in Taiga and Boreal forests with images taken by a UAV, similarly to the approach described in this project. To segment the image they first developed an algorithm that identifies potential regions that contain a tree. By applying several transformation steps to the image and the use of a threshold, they manage to segregate the trees from the background. Each tree region is then classified into one of four categories that indicate the health status of the tree in this region.

Although the evaluation of the system gave significant results, it must be noted that only the classification part was evaluated, the tree extraction part, however, was not evaluated. Therefore, the overall performance of their system is probably lower than the paper suggests and cannot be directly compared to the results of this project.

The project introduced by [Jiang et al., 2019] focused on the detection of standing and fallen trees. With only one VHR CIR (very high-resolution color-infrared) image and the help of semantic image segmentation they were able to achieve quite good results. Although the techniques used in [Jiang et al., 2019] seem similar to the techniques used in this project, the task of the two projects is quite different. [Jiang et al., 2019] only covers the category of dead trees and does not cover any other health states of the trees.

3 Terms and Methods

In the following subsections, the used techniques, methods and other information that contributed to solve the project task are described.

3.1 Eight-Toothed Spruce Bark Beetle - Ips Typographus

The most common spruce bark beetle is called "eight-toothed spruce bark bark beetle" or in Latin "Ips Typographus". The start of an infestation by such beetles is just underneath the crown, where the bark is already thicker, and spreads to the bottom of the trunk during the infestation process [Landwirtschaftskammer, 2011].
The bark beetle infestation process can be divided into the following steps as provided by [Landwirtschaftskammer, 2011]:

1. Beetles find a new habitat inside a trunk. Signs for this stage are sightings of boreholes and boring dust around the holes or at the bottom of the tree as well as fresh resin flow. These signs can be seen in Figure 1.

2. Later, the beetles already plant their brood underneath the bark. Signs for this stage could be missing bark, lost needles lying around the tree or discolored needles on the tree as it can be seen in Figure 2. At this stage, some beetles already left the trunk to find new habitable trees to infest.

3. The last stage is signaled by a dead tree. Those trees are easy to spot, since all needles are lost and often there are large areas of bark missing as shown in Figure 3. At this stage most beetles left the trunk and other trees in close proximity are likely to show signs of being infested.

(a) The borehole and boring dust around the hole can be seen in this image.

(b) Here one can see the borehole and resin flow around the hole.

(c) The infestation can not be seen from above. The crown of the tree looks healthy.

Figure 1: Example of a Spruce Tree recently infested.

(a) In this picture the missing bark signals an advanced stage of a bark beetle infestation.

(b) The needles are discolored and the infestation can be seen from above.

Figure 2: Advanced stage of a bark beetle infestation.

(a) This tree has an extensive loss of bark.

(b) The tree lost most of its needles.

Figure 3: Signs of a dead tree.

As mentioned in [Landwirtschaftskammer, 2011], it is essential to find infested trees as early as possible and remove them immediately after they were found to minimize the infestation risk of surrounding trees. Therefore, a forest needs to be inspected frequently in search for the signs stated above.

3.2 Project Structure

This section explains the necessary steps to solve the project task. The fact that the project should be applicable for large forest areas, where not every single tree can be scanned for bark beetle infestations must be considered as well. Therefore, it is reasonable to split the project into two main steps: localization and identification.

3.2.1 Localization

This part of the project is concerned with the search for sick forest areas. As mentioned before, a bark beetle infestation causes the needles to change their color since they do not get enough water anymore. This happens in the second stage of the infestation, where some bark beetles already left the infested tree to find a new habitat.
During the last stage of the infestation, the tree looses most of its needles and the rest of the bark beetles already left the tree.
Both of these last stages allow identification from above. Therefore, it is possible to localize infested areas from a bird's-eye perspective.
It is also important to note that a tree with discolored needles or a tree that is already dead does not necessarily have to be infested by bark beetles. There are several possibilities, which could have caused this anomaly of appearance. Due to this fact, those trees are not referred to as "infested" but they are more appropriately called "sick" or "dead".
Considering the uncertainty associated with this part of the project and the

possibility that bark beetles already infested other trees in close proximity, the second part of the project is needed.

3.2.2 Identification

After localizing possible infested areas, the trees around this location need to be scanned to identify those trees that have to be removed.
Using RGB images of the trunk, a method will be developed in this project that is able to classify the images into "healthy" and "infested" by scanning the images for signs of bark beetle infestation.
In this step of the project, especially the signs of the early stage of the infestation are of importance i.e. boreholes, boring dust and resin flow. During this part it is desirable to get rid of all infested trees, as soon as possible, to prevent further spread of bark beetles.

3.3 Data Acquisition

The most viable solution for acquiring images from a bird's-eye perspective of a forest is the use of an unmanned aerial vehicle (UAV or more commonly: drone) with an RGB camera. Using a DJI Mavic 2 Zoom[2], 35 images were acquired, which show forests from above.
For the second part of the project, pictures of trunks with and without boreholes were collected. In total 44 pictures of infested trunks and 36 pictures of healthy trunks were taken with a smartphone camera.

3.4 Transfer Learning

A large amount of data is essential for deep learning tasks in order to be able to train a general, yet complex network. If there is only an insufficient

[2]See online: https://www.dji.com/at/mavic-2

amount of data available, training the network results in overfitting, which means that the network is able to fit the training data perfectly, but it is unable to generalize in a way that it also fits new unseen data.

Transfer learning enables us to reuse the architecture and the weights of an already trained model and fit it to a new custom task, which does not only help in making the network as general as possible, but also in speeding up the whole training process, since fewer layers need to be trained.

The use of transfer learning in image processing tasks works exceptionally well because of the similarity of lower level features between different models. The reason is that convolutional neural networks (CNNs) usually learn very general features in lower levels of the network, which apply to many other image processing models too. Layers which are on a higher level in the network usually produce task specific features.

In Figure 4 the features and filters of the bottom three layers of a convolutional neural network are visualized [Zeiler and Fergus, 2014]. It can be seen that those low level filters are specialised in discovering edges and other simple structures, which is applicable in all kinds of other tasks as well.

Experiments in [Yosinski et al., 2014] showed that the amount of layers which can be transferred depends on the similarity of the source and target task. They also showed that the performance of a network is boosted even if the two tasks are distant, if the right amount of layers is transferred.

For training a transferred model there are usually two approaches. The first one is to freeze the transferred weights, which means that they are not updated during training. The second approach is fine-tuning, which means that the transferred weights are updated during training - usually with a smaller learning rate than the remaining network. Freezing transferred weights is useful for similar tasks where only a little amount of data is available. If the tasks are dissimilar, it makes sense to fine-tune the transferred weights.

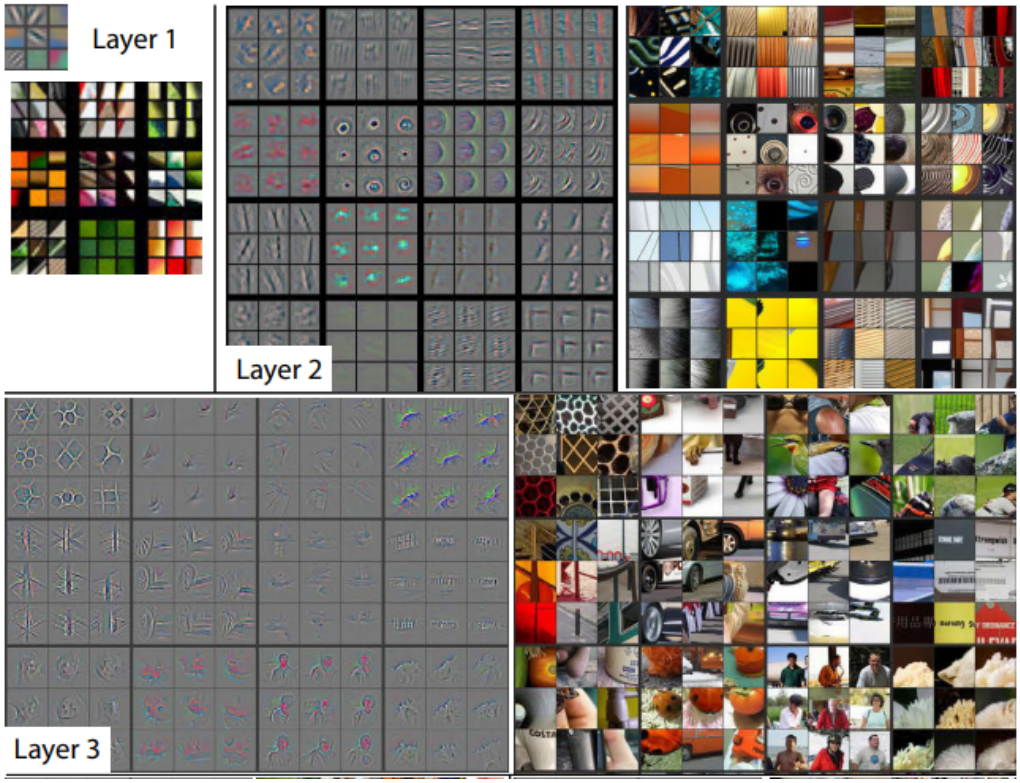


Figure 4: Visualizing the bottom layers of a CNN. [Zeiler and Fergus, 2014]

This however, requires a moderate amount of training data in order for the network to still be able to generalize.

3.5 Data Augmentation

Training a complex deep learning model requires a vast amount of data to make it accurate and robust. A common technique to artificially enlarge the dataset is data augmentation, which performs label preserving transformations of the input data.

For this project the data was augmented in terms of rotation, shifts, zoom, flips and brightness. The fill mode was set to "reflect", which means that points outside of the image range are reflected points from the image.[3] This

[3]See online: https://keras.io/preprocessing/image/

is used for example when zooming out from an image and instead of filling the rest of the image with black pixels the pixels of the image are reflected to fill up the empty space.

3.6 Image Classification

The task of image classification is a well researched area with a lot of progress over the last couple of years. Since the introduction of convolutional neural networks, a lot of different architectures were developed with CNNs as the key technique for their success. Benchmark datasets like "ImageNet" [Deng et al., 2009] allow the comparison of the performance of the different networks as it is done in [Sultana et al., 2018].
Due to the vast research in this area there are lots of insights to learn from and it eases the development and adaptation of a network that solves a new task. Moreover, most researchers even make their trained networks publicly available, which makes it easy to apply transfer learning.
In Figure 5 an example output for an image classification task can be seen at the top.
In this project, image classification was used to classify trees into "infested" and "healthy". With just 80 images in total, techniques like transfer learning and data augmentation were used to overcome the problem of overfitting. Several networks were tested using different architectures and weights from VGG16 [Simonyan and Zisserman, 2015], MobileNetV2 [Sandler et al., 2018], InceptionV3 [Szegedy et al., 2016] and DenseNet121 [Huang et al., 2017].

[4]Adapted From: https://d2l.ai//chapter$_c$*omputer*−*vision*/*semantic*−*segmentation*−*and* − *dataset.html*

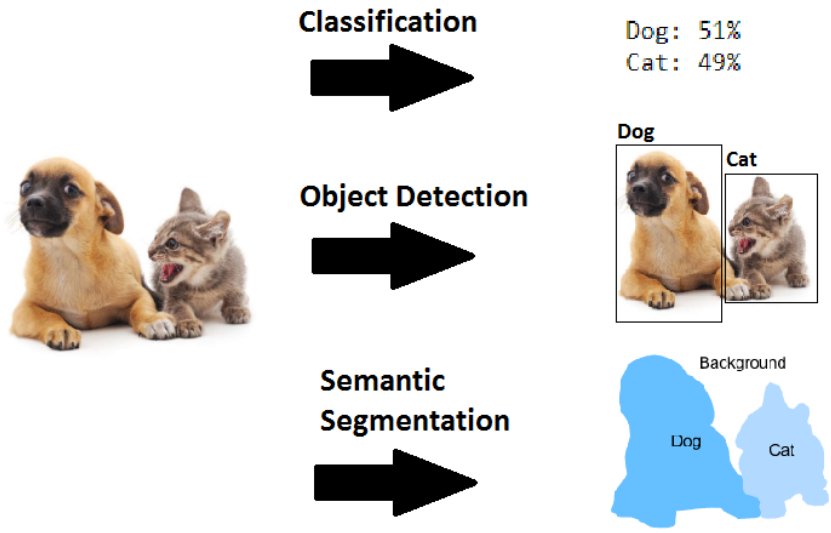


Figure 5: Comparison of the three main image processing tasks. [Humer, 2019][4]

3.7 Object Detection

The task of detecting objects in images is a discrete task like image classification, with the advantage that several areas can be proposed and classified independently. In Figure 5 the example in the middle shows an object detection task, which gives one bounding box around a dog and one bounding box around a cat and labels them accordingly.

One difficulty of object detection is the need of a second network, which proposes these bounding boxes. The proposed boxes are then inserted into an image classifier, which calculates the label probabilities. Another disadvantage of object detection is that the network only delivers discrete results i.e. objects are encapsulated by a rectangle and not their own shape.

3.8 Semantic Image Segmentation

Like image classification and object detection, semantic image segmentation is one of the main tasks in image processing. In semantic segmentation the

advantage of continuous labeling arises, which means that objects are labeled according to their true shapes. This is possible due to pixel-wise classification i.e. the network predicts a class for each pixel of the input image.
As it can be seen at the bottom of Figure 5 not only the shapes of the animals are labeled in a continuous way, but also background labels are provided.
The foundation for end-to-end training for semantic segmentation tasks was built with the development of "Fully Convolutional Networks" [Long et al., 2015]. After that, a lot of research was performed to find several approaches to improve solving semantic segmentation tasks. Important developments in this direction were the concept of "Encoder-Decoder Architectures" (e.g. [Noh et al., 2015], [Ronneberger et al., 2015], [Badrinarayanan et al., 2017]) and the concept of "Spatial Pyramid Pooling" (e.g. [He et al., 2014], [Zhao et al., 2017], [Chen et al., 2018]).

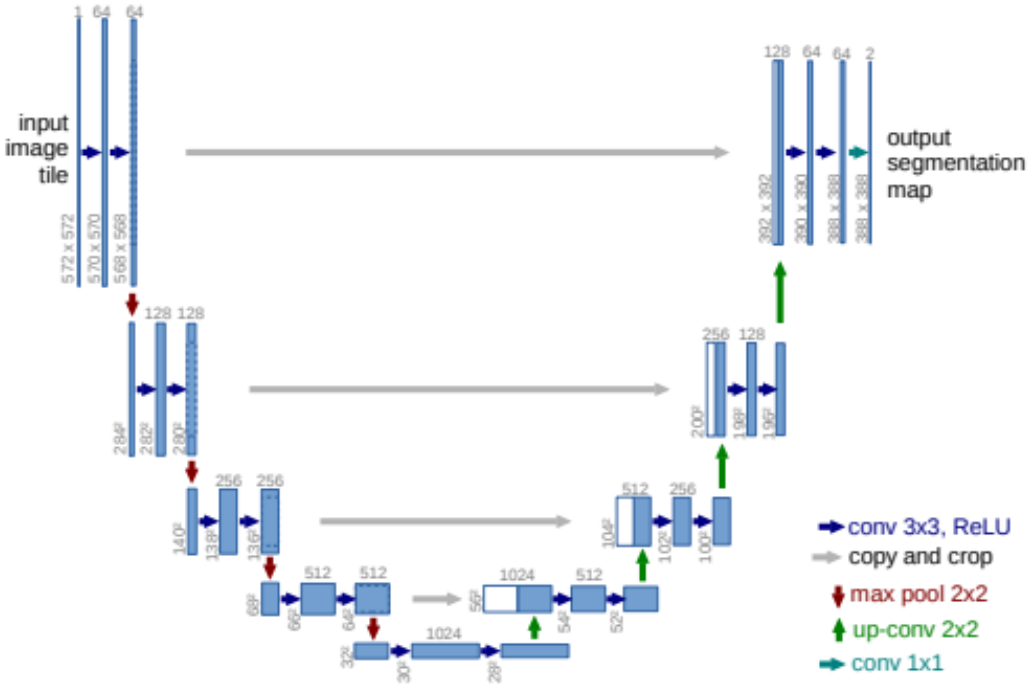


Figure 6: U-Net architecture as proposed by [Ronneberger et al., 2015]

The development of the U-Net architecture [Ronneberger et al., 2015] as it can be seen in Figure 6, showed that it is possible to train a semantic image segmentation network with only a small amount of training data (they used only 30 images with data augmentation). The concept of U-Net is to combine local and global features to make a precise and accurate pixel-wise classification. This is achieved by the special architecture which U-Net provides as it is shown in Figure 6.

The encoding (downsampling) path extracts global features over several steps. Each step delivers features which are a bit more specific than those of the previous step. This is done by alternating between convolutional and pooling layers.

The output of the encoder is inserted into the decoder (upsampling path) which is an alternation between convolutional layers and deconvolutional layers. At each of the upsampling steps, the output of the corresponding downsampling step is concatenated to the input, which introduces a skip connection.

Due to these skip connections, the network is able to combine complex features which give global information about the input, and low level features which deliver local, detailed information about the image.

The size of the predicted output corresponds to the input size to enable pixel-wise classification. Furthermore, there is one output channel for each label that needs to be predicted i.e. the output size of the network is WIDTH x HEIGHT x number of labels. The result delivers information about the probability of a label for each pixel. Taking the pixel-wise "argmax" (channel index of the highest value for each pixel) delivers a classification map of the input.

In this project experiments based on the architecture of U-Net were conducted. Due to the little amount of images (only 35 images were available),

the dataset was artificially enlarged with data augmentation. Furthermore, the experiments involve the use of transfer learning by using the architecture and weights from DenseNet121 [Huang et al., 2017].
As a baseline for this semantic image segmentation task the solution proposed by [Humer, 2020] is taken, which introduces some basic adaptation of U-Net and tries to solve the same task as it is covered in this thesis.

3.9 Data Annotation

The annotation of data is done manually. In the following subsections it is described how the labels for both parts of the project were generated.

3.9.1 Annotation for Semantic Segmentation

The masks for the localization part of the project were annotated using the image processing tool "Gimp"[5]. Each pixel was marked with a color which represents the class it belongs to. The labels and corresponding color codes are chosen as follows:

- "healthy": 0x000000 (Black)
- "sick": 0xff0000 (Red)
- "dead": 0xffffff (White)
- "unlabelled": 0x00ff00 (Lime)

As mentioned before, the label "healthy" means that the tree looks healthy from above. However, it does not mean that the tree is definitely healthy. The tree could still be sick or infested by bark beetles, but it is not yet recognizable from above. The label "sick" means that the tree is either infested by bark

[5]See online: https://www.gimp.org/

Figure 7: Example of an input image and its corresponding mask.

beetles or is sick due to some other reason, which causes the tree's needles to discolor. A "dead" tree already lost most of its needles due to some sickness or infestation. The class "unlabelled" is used for anything else that is not covered by the labels mentioned above. This covers for example background, deciduous trees and paths through the forest.

An example of an image and the corresponding mask, which are used for training the network, can be seen in Figure 7.

Due to the fact that an image processing tool like "Gimp" is not ideal for labeling such fine grained image data, the generated masks are not as precise as desirable.

3.9.2 Annotation for Classification

For annotating the images of the second part of the project, the data was manually sorted into an "infested" and a "healthy" folder. This way, the data can be nicely processed by Keras' "Data Generator".

3.10 Real-Time Classification

After localizing sick areas of a forest, the process of finding infested trees is shifted to the potential spots. Due to the time-critical nature of the problem task, it is good practice to remove an infested tree as soon as possible, such that the probability of spreading the infestation is kept small. To be able to do that, an application must be developed that processes the images, which were captured by the drone, right away.

The development of a real-time classification mobile app is the most obvious choice because a smartphone is used for directing the drone anyway. Due to limitations of resources, the mobile app was only developed for Android devices.

The official "DJI GO 4" app[6] is not open source, however, DJI provides an API[7] which allows to communicate with a drone and gives developers the possibility to build their own app with custom features.

In addition to the basic features of the standard "DJI GO 4" app, the customized app should allow users to select images, which will be automatically downloaded and classified. This way, the wood worker is immediately notified if a tree is infested.

[6]See online: https://www.dji.com/at/downloads/djiapp/dji-go-4

[7]See online: https://developer.dji.com/

4 Implementation

As previously mentioned, both networks - classification part and localization part - make use of commonly known deep learning practices and techniques like convolutional layers, pooling layers, transfer learning and data augmentation. The architectures and components that were used for each of the two main parts of the project are described in the following subsections.
Furthermore, the implementation of the Android application, which enables real-time classification of trees, will be described and a user guide will be provided.

4.1 Localization

The basic U-Net architecture as described in [Ronneberger et al., 2015] was adapted for this part of the project.
The goal of the development of the semantic image segmentation network in this project is to achieve a better performance than the proposed network in [Humer, 2020]. Based on the recommendations of future work mentioned in [Humer, 2020], experiments with transferring the encoding path from a pretrained imagenet classifier were conducted in this project, where the encoding path is the pretrained DenseNet121 classification network.
Furthermore, experiments with three different decoding paths were conducted and compared to each other, where each network is a bit more complex than the previous one. Each decoder consists of five upsampling blocks as it can be seen in 8.
The basic architecture of the semantic segmentation model is shown in Figure 8. The decoding path is an abstract placeholder, which is specified in more detail in the corresponding subsections. Note that the last upsampling block does not use a skip connection. This is due to the fact that DenseNet121

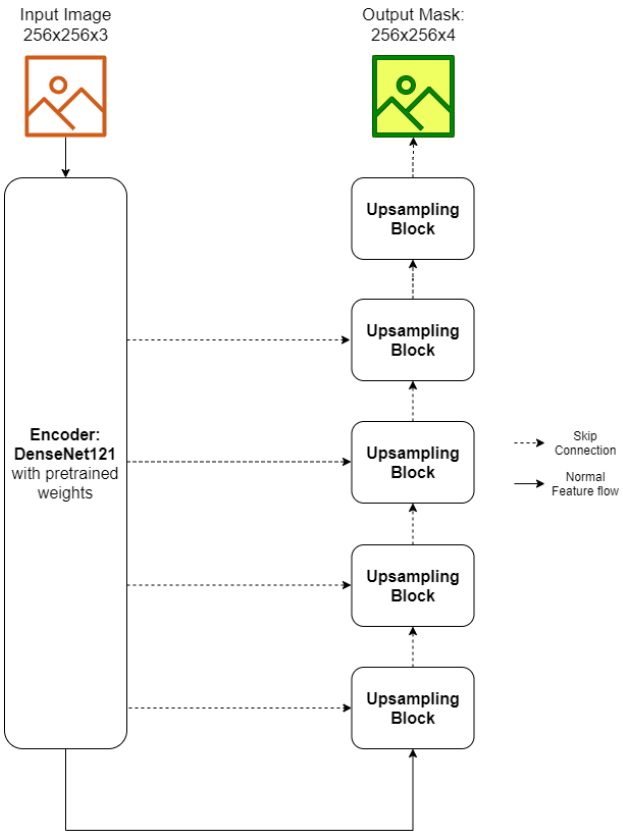


Figure 8: Basic architecture that is used for the semantic segmentation part of this project. The decoding path consists of five upsampling blocks, which are abstract placeholders and will be described later.

downscales the image already at the beginning of the classifier, which would leave us with a skip connection to the input image. Therefore, the last upsampling block does not merge with the corresponding downsampling of the encoding path.

4.1.1 Input and Output

The spatial resolution of the network is 256x256, which requires some downscaling of the data provided by the drone camera. Due to limits in computational capacity of the available machine, this is a necessary trade-off in order to be able to train the network.

The number of input channels is three, which covers the spectrum of RGB images. The number of output channels however, is four, which corresponds to the number of classes that need to be identified.

4.1.2 Encoding Path

The encoding path is responsible for the step-wise extraction of features from the image. Each step of the encoding path captures more and more complex features, starting from local information of the input and ending in contextual information that covers the whole image.
There are several techniques how transfer learning is applicable in semantic image segmentation as summarized in [Humer, 2019]. In this project transfer learning is used by replacing the encoding path with a classifier that was pretrained on the imagenet benchmark dataset. [Humer, 2019] mentions that the architecture which is used in DenseNet [Huang et al., 2017] is especially well suited for semantic segmentation tasks due to efficient parameter usage and a natural use of skip connections within the encoding model, which is able to carry features from the bottom of the network to the top.
In this project the weights and architecture of the DenseNet121 network [Huang et al., 2017] are reused, which downsamples the input image to a shape of 8x8x1024. This is an iterative process consisting of five dense blocks and downsampling steps.
A single dense block consists of several layers, where each layer input is concatenated with the outputs of the previous layers as shown in Figure 11. The abstract "Layer" term refers to a combination of convolutional layer, batch normalization and rectified linear unit ("ReLU") activation function.
Within a dense block the feature dimensions remain the same. A pooling layer follows the dense blocks, which halves the size of the feature maps. The concatenation of feature maps results in a linear increase of channels,

which is tractable because the size of the feature maps is shrinking.
Refer to [Huang et al., 2017] for a more detailed description of the implementation of DenseNet121.

4.1.3 Decoding Path - Simple

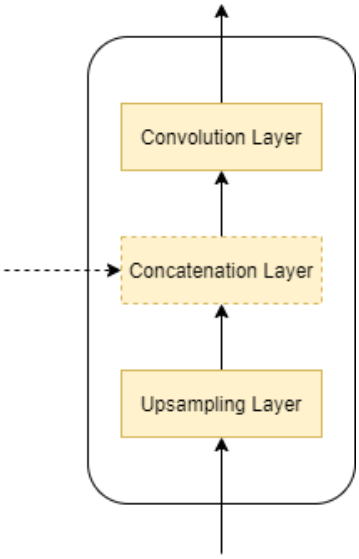


Figure 9: Architecture of a "Simple Upsampling Block". The dashed line indicates that the layer is not used in the last step of the upsampling path. The dashed arrow indicates the skip connection.

The first decoder has a very basic upsampling structure consisting of a single upsampling, concatenation and convolution layer at each upsampling step as it can be seen in Figure 9. The upsampling layer doubles the size of the feature maps by duplicating each entry of the input. The concatenation layer concatenates the output of the corresponding downsampling path to the result of the upsampling layer. The convolutional layer calculates new features based on the concatenated input with a 3x3 filter.
As mentioned before, the size of the feature map is doubled at each step, however, the channel size in this simple approach is a constant value of 32.

4.1.4 Decoding Path - U-Net

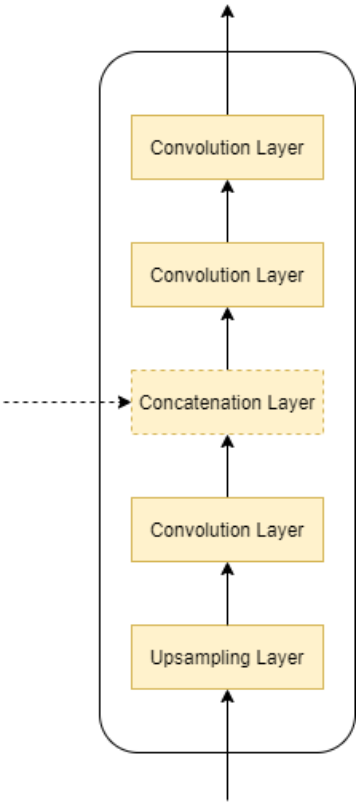


Figure 10: Architecture of an upsampling block as it is used in [Humer, 2020]. The dashed line indicates that the layer is not used in the last step of the upsampling path. The dashed arrow indicates the skip connection.

The second approach is an adaptation of the U-Net decoder as it was proposed by [Humer, 2020]. This decoder features two more convolutional layers followed by a dropout layer, which serve as a bridge between the encoding and decoding paths.

Furthermore, each decoding step consists of one upsampling layer, which doubles the size of the feature map, directly followed by a convolutional layer with a 2x2 filter, which calculates an interpolation of the upsampled features. After that, the corresponding features from the downsampling path are concatenated with the upsampled features and processed by two more

convolutional layers with a filter size of 3x3. The used architecture is illustrated in Figure 10.
The non-linearity needed for neural networks is introduced by a "ReLU" activation function after each convolutional layer.

4.1.5 Decoding Path - DenseNet

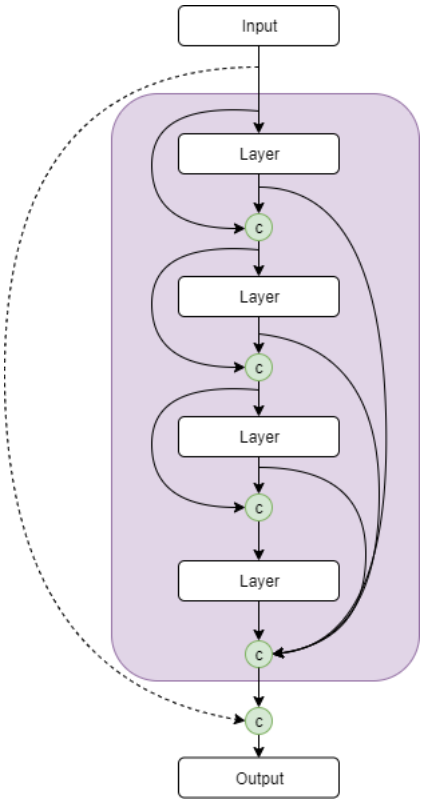


Figure 11: Architecture of a Dense Block. Adapted from [Humer, 2019] and [Jégou et al., 2017].

The idea of the "Fully Connected DenseNet" was introduced by [Jégou et al., 2017] where they proposed the use of dense blocks in an encoder-decoder architecture. To ensure that the parameters in the upsampling path do not become intractable, the skip connection between the input of a dense block and its output are omitted in the upsampling path. If it was

not omitted, not only the size of the feature map would grow, but also the amount of channels - which usually becomes smaller during the upsampling process - would grow and produce a large computational overhead. In Figure 11 the omitted connection in the upsampling path is represented as dashed line.
The skip connections between encoding and decoding path are inserted after the corresponding dense block in the encoding path and before the dense block in the decoding path.

4.1.6 Output Layer

For each of the decoding approaches, the same output layer was used, which consists of a convolutional layer with a channel size of four (one channel for each label) and a 1x1 filter. The activation function is a pixelwise softmax, which delivers probabilities for each pixel that indicates, how likely that pixel belongs to a certain class.
The "Adam"-optimizer [Kingma and Ba, 2015] was chosen for training and the "Categorical Crossentropy" loss function.

4.2 Classification (Identification)

The task of the classifier in this project is to determine whether or not a tree is infested by bark beetles. To achieve this, the use of a deep convolutional neural network is beneficial. Its purpose is to extract high level features, which are further processed by a feed forward classifier that results in probabilities of the tree's infestation status i.e. "healthy" or "infested". Further details will be given in the following subsections.
The basic architecture of the classification network is illustrated in Figure 12. It shows the input as an image and the two main parts of which the classifier consists. The two outputs show the probabilities of "healthy" and

Figure 12: Basic architecture of the classification network.

"infested", which sum up to one.

Since there is no comparable research in the area of this part of the project, the used baseline is a classifier that always predicts the most common class, which is "infested" and is represented by 55% of the training data.

4.2.1 CNN Feature Extractor

Due to the lack of data that can be used to train the model, data augmentation and transfer learning are used. The experiments with transfer learning cover the use of architectures and weights from VGG16 [Simonyan and Zisserman, 2015], MobileNetV2 [Sandler et al., 2018], InceptionV3 [Szegedy et al., 2016] and DenseNet121 [Huang et al., 2017]. The

used weights were previously trained on the imagenet dataset, which consists of various types of images (i.e. pictures of animal, plants, food etc.). More detailed information about the dataset can be found on the official website: http://image-net.org/.
Comparing the imagenet dataset and the dataset used in this task, it seems that the two tasks are quite distant from each other. However, transfer learning should still be beneficial as it was mentioned in Section 3.4. Due to this and the fact that a complex model could not be trained properly with the limited amount of data available, the choice of using transfer learning is obvious.
Nevertheless, the models cannot be taken as is due to the fact that they were trained for a different task (i.e. the original classifiers have an output of 1000 different labels; in this project only two labels are needed). Therefore, only the convolutional part is reused and an adaptive feed forward classifier is added on top.

4.2.2 Feed Forward Classifier

The extracted features provided by the convolutional part are inserted into a "Global Max Pooling" layer, which basically just takes the maximum value of each channel. Due to this layer, the input size of the network is flexible, since only feed forward layers need to know their input size beforehand and this layer delivers a fixed size output.
The number of channels (and therefore also the input size of the feed forward layer) is 1024. Following the pooling layer is a dropout layer (with a dropout probability of 0.4), which supports the network to become more general. The final feed forward layer has an output size of two.
The "AdaDelta"-optimizer [Zeiler, 2012] was chosen for training the network and "Categorical Crossentropy" was used as loss function.

4.2.3 Extension: Detect Trunk

Since the pictures of a trunk are taken from a certain distance to the tree, it is inevitable that the images, which need to be classified, also contain a certain amount of background. Because of this and the lack of data, it might happen that the classification model is prone to overfit to things it discovers in the background. To solve this problem, it might be helpful to segregate the corresponding trunk from its background and only forward images of the trunk without background to the feature extractor.

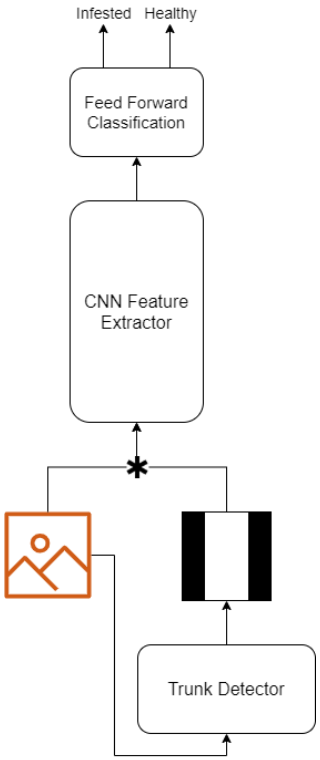


Figure 13: Extended architecture of the classification network. The image is preprocessed using the mask that is produced by the trunk detection network. In the figure the preprocessing step is abstractly indicated by an asterisk.

For handling this problem, we make use of semantic image segmentation again to produce a mask, which shows the relevant parts of the image. This mask can then be used to trim the left and right side of an image until only

the trunk is left. An alternative approach would be to just multiply the mask with the original image, which results in a background of zeros and therefore contains no information. The abstract preprocessing procedure can be seen in Figure 13.

The advantage of the first approach is that the size of the image would be reduced. The drawback is that the size is only reduced left and right and therefore it would result in a weird ratio, which varies for each picture. To handle this, the image could be split into squares by using the width and dividing the trunk into several images of the same height. This leaves us with the problem that the images are not properly annotated anymore in case of "infested" because it may happen that not every sub-image shows signs for a bark beetle infestation. Hence, if using this approach, each sub-image needs to be relabeled, or the input size of the network needs to be set to a fixed size, which would result in heavily distorted images.

The advantage of the second approach is that the image will neither be distorted, nor needs to be reassessed. However, there will be a lot of unnecessary overhead since it may occur that large parts of the image are set to zero and are therefore useless for the calculations. Furthermore, the second approach is very prone to introduce noise itself, if the trunk detection network does not deliver perfect results i.e. it is possible that also trunk pixels are just set to zero due to some misclassification of the trunk detection network.

In this thesis two approaches are compared. On the one hand there are experiments with images that are multiplied with the resulting mask, and on the other hand there are experiments where the images are truncated with the help of the resulting mask.

The trunk detection architecture used is derived from [Humer, 2020] with an adjusted class number of two.

4.3 Real-Time Classification

The development of a mobile application is supported by DJI's "Mobile Software Development Kit"[8] (Mobile SDK), which enables communication between the mobile device and the corresponding drone. On top of the "Mobile SDK" DJI developed the "UX SDK"[9], which eases the development of an application by providing lots of components that are commonly needed (e.g. camera live-stream, show current state of the drone, drone settings, image capturing etc).

In order to be able to use the provided SDK, there needs to be an online registration of the application as described in the SDK documentation[10].

The development of the Android application is based on the "Media Manager Application"[11] sample project provided by DJI. The app is written in Java[12] with the help of the Android Studio[13] development environment.

To perform inference with a deep learning model on a mobile device, the "TensorFlow Lite" framework[14] is used within the Android application. To use "TensorFlow Lite", the model in question must first be converted to a "TensorFlow Lite" model, which results in a .tflite file that can be loaded by the "TensorFlow Lite" interpreter. This can be done in Python as shown in [Abadi et al., 2015].

The most important parts of the application architecture are shown in the following subsections.

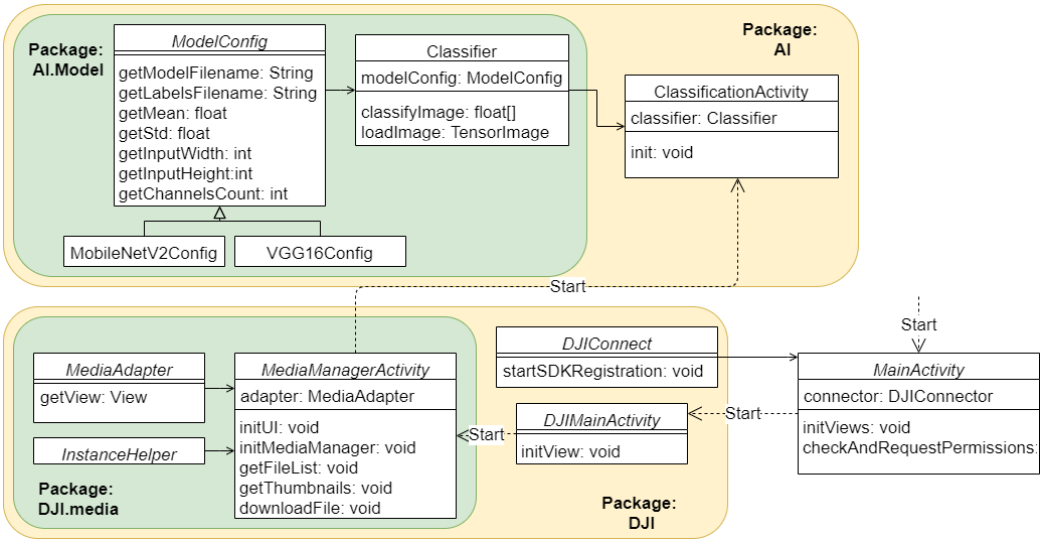


Figure 14: Most important classes and methods of the application and their dependencies.

4.3.1 Application Architecture

The basic structure of the application is divided into an "AI" package and a "DJI" package as shown in Figure 14. The "AI" package contains everything that is needed to perform inference on the previously trained model and show the results. The "DJI" package takes care of everything that is related to successfully establishing communication between the Android device and the drone.

Outside of both packages lies the "MainActivity", which is the first activity that is shown after the app is started. The "MainActivity" is responsible for handling the permissions that are required to run the application e.g.

[8]See online: https://developer.dji.com/mobile-sdk/
[9]See online: https://developer.dji.com/ux-sdk/
[10]See online: https://developer.dji.com/mobile-sdk/documentation
[11]See online: https://developer.dji.com/mobile-sdk/documentation/android-tutorials/MediaManagerDemo.html
[12]See online: https://www.java.com/
[13]See online: https://developer.android.com/
[14]See online: https://www.tensorflow.org/lite

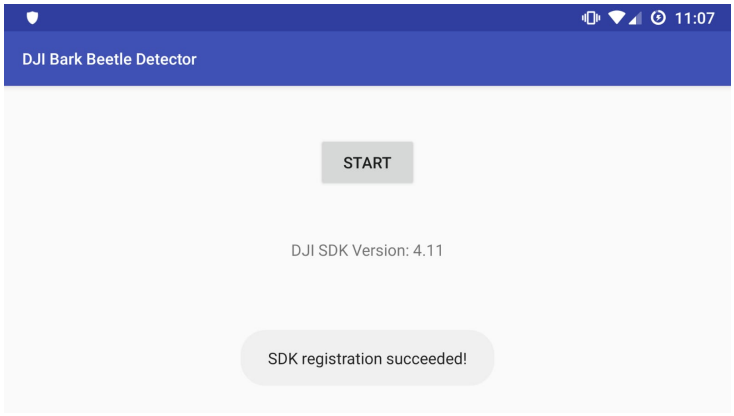


Figure 15: The screenshot shows the layout of the "MainActivity".

Figure 16: The image shows the layout of the "DJIMainActivity" with an example image.

location, media and internet access. This means that it has to check that all permissions were previously accepted by the user or prompts the user to accept them, if they have not done it previously. After that, the "MainActivity" uses the "DJIConnect" class to perform the SDK registration and connect the mobile device to the drone. The layout of the "MainActivity" is shown in Figure 15.

After the permissions were accepted and the SDK registration was successful, the user can start the "DJIMainActivity" by clicking the "START" button as shown in Figure 15. This activity shows a live stream and the status of

Figure 17: The image shows the layout of the "MediaManagerActivity" giving some example images.

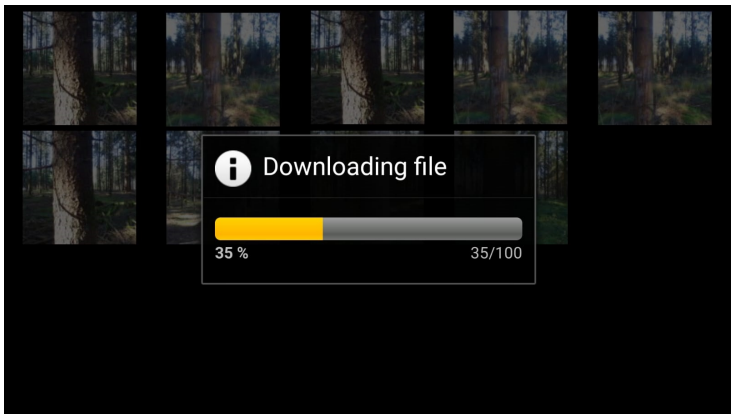


Figure 18: The image shows the layout of the "MediaManagerActivity" when an example image is downloaded.

the drone as shown in Figure 16. It also allows to change settings e.g. switch to manual or auto focus. Furthermore, the activity allows the user to take pictures, which can be used for the inference part later on.

After the images were taken, the user can click on the media manager symbol to start the "MediaManagerActivity". This activity downloads thumbnails and file names of the images that are currently saved on the drone and shows them to the user. The "MediaAdapter" class maps the downloaded thumbnails to a predefined representation in a view like it is shown in Figure 17.

Figure 19: The screenshot shows the layout of the "ClassificationActivity" with an example image.

The "InstanceHelper" class is used for providing easier interaction with the drone.

When the user clicks on a preview image, first the full size image is downloaded and then the "ClassificationActivity" is started. With the help of the "Classifier" class this activity processes the downloaded image and shows the results to the user. The download progress is illustrated in Figure 18 and the layout of the "ClassificationActivity" is shown in Figure 19. The "Classifier" class instantiates a tensorflow lite interpreter and loads the trained model using the information provided by the "ModelConfig" object, which was instantiated by the "ClassificationActivity".

When the "classifyImage" method is called, the desired image is loaded with the given path. The image is also resized and normalized according to the settings of the "ModelConfig" object it received. Finally, the interpreter runs the inference on the provided image and returns the probabilities of the picture showing signs of infestation or not.

The "ModelConfig" class specifies get-methods, which provide specific information about a certain model. This gives some flexibility for the model properties i.e. which kind of model should be used and what are the settings for this model. The "MobileNetV2Config" and "VGG16Config" are

two classes that inherit from "ModelConfig" and specify properties of two different models e.g. the path of the model file or the width and height of the input layer.
The example output shown in Figure 19 is produced by an untrained test model and therefore does not show accurate predictions as they are produced by the trained models that are described in Section 5.2. The bar at the top of the image indicates the probability that the image of the trunk shows signs of a bark beetle infestation or not. If a model is sure that the tree is healthy, most of the color bar will be green, otherwise it would be red. The exact probabilities are stated at the left and right borders of the color bar.
Due to the fact that the main focus of this project lies on the development of a deep learning model that distinguishes between healthy and infested trees, the Android application only covers basic functionality and no field tests were conducted.

4.3.2 Important Code Snippets

In this section the most important parts of the code are summarized and explained in a chronological order, i.e., from the start of the application until the inference produced for an image.
The registration of the SDK is performed in the "DJIConnect" class, which uses an "DJISDKManager" instance to register the application. This is done in parallel by starting a new thread as shown in the following code fragment:

```
AsyncTask.execute(new Runnable() {
    @Override
    public void run() {
        DJISDKManager.getInstance().registerApp(
            applicationContext, registrationCallback);
    }});
```

After the SDK registration was successful the "DJISDKManager" instance is used again to establish a connection to the drone:

```
DJISDKManager.getInstance().startConnectionToProduct();
```

Within the "DJIMainActivity" there is a lot of code responsible for the visualization of the livestream, that is sent by the drone, and all kinds of settings that the user is able to adjust within this activity.
The "initMediaManager" method within the "MediaManagerActivity" is responsible for loading the preview images for the gallery that is shown in this activity. This is done with the help of a DJI "MediaManager" instance, which first loads a snapshot of the internal storage files of the drone and orders them from new to old as shown in the following code snippet:

```
mediaFileList = mMediaManager
    .getInternalStorageFileListSnapshot();
Collections.sort(mediaFileList, new Comparator<MediaFile>() {
    @Override
    public int compare(MediaFile lhs, MediaFile rhs) {
        if (lhs.getTimeCreated() < rhs.getTimeCreated()) {
            return 1;
        } else if (lhs.getTimeCreated() > rhs.getTimeCreated())
        {
            return -1;
        }
        return 0;
    }
});
```

After the list is retrieved, the "getThumbnails" method is called, which is responsible for downloading the preview images. The following code is exe-

cuted for each item in the "MediaFile" list:

```
FetchMediaTask task = new FetchMediaTask(mediaFile,
    FetchMediaTaskContent.THUMBNAIL,
    taskCallback);
```

The preview images are presented in a "GridView" using the "MediaAdapter" class. The method "getView" is called for each image and returns a view that represents the thumbnail in an "ImageView":

```
MediaFile mediaFile = mediaFileList.get(position);
ImageView imageView;
if (convertView == null) {
    imageView = new ImageView(getApplicationContext());
    imageView.setLayoutParams(
        new GridView.LayoutParams(300, 300));
    imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
} else {
    imageView = (ImageView) convertView;
}
imageView.setOnClickListener(ImgOnClickListener);
imageView.setTag(mediaFile);

imageView.setImageBitmap(mediaFile.getThumbnail());
return imageView;
```

After clicking on a preview image, the "downloadFile" method is executed, which starts the download and forwards the user to the "ClassificationActivity" after the download finished successfully.
The "ClassificationActivity" loads the downloaded image and shows it to the user. It also creates an instance of the "Classifier" class with a class instance that inherits from the "ModelConfig" class as parameter. This gives more

flexibility on choosing a certain model. In a future version of this application the user might even be able to decide which model to use. An example for creating a "Classifier" instance is shown in the following code snippet:

```
classifier = new Classifier(this, -1, new MobileNetV2Config());
```

After calling the "classifyImage" method of the "Classifier" instance, the results are presented to the user.
By creating an instance of the "Classifier" class, the constructor is responsible for loading the tensorflow lite model and initializing an interpreter for this model:

```
tfliteModel = FileUtil
    .loadMappedFile(activity, modelConfig.getModelFilename());
...
tflite = new Interpreter(tfliteModel, tfliteOptions);
```

Furthermore, a buffer for the output probabilities needs to be created:

```
outputProbabilityBuffer = TensorBuffer
    .createFixedSize(probabilityShape, probabilityDataType);
```

The "classifyImage" method first executes the "loadImage" method, which loads the image and then processes it by resizing it to the dimensions of the input layer and performing normalization according to the "ModelConfig" settings:

```
inputImageBuffer.load(bitmap);
bitmap.getHeight());
ImageProcessor imageProcessor =
```

```
        new ImageProcessor.Builder()
                .add(new ResizeOp(
                    modelConfig.getInputWidth(),
                    modelConfig.getInputHeight(),
                    ResizeOp.ResizeMethod.BILINEAR))
                .add(new NormalizeOp(
                    modelConfig.getMean(),
                    modelConfig.getStd()))
                .build();
return imageProcessor.process(inputImageBuffer);
```

After successfully loading the image, the interpreter starts the inference:

```
tflite.run(
    inputImageBuffer.getBuffer(),
    outputProbabilityBuffer.getBuffer().rewind()
    );
```

After the interpreter ran the inference, the method returns the results and the "ClassificationActivity" presents them to the user.

5 Results

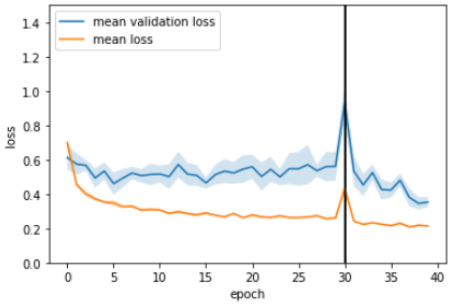
The results of the experiments for the two project parts are presented in the following subsections.

5.1 Localization

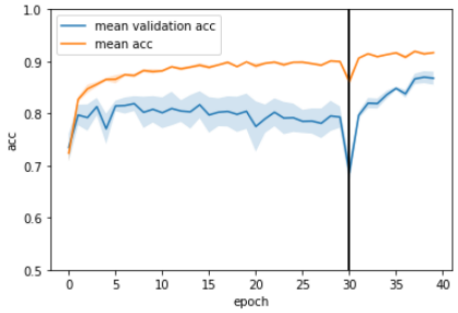
This section gives insight into the results of the experiments conducted to solve the localization task, which was explained in Section 4.1.
Due to the limited amount of data that is available, k-fold cross validation is used to train and validate the networks, which results in a lower evaluation bias. The total amount of 35 images was split randomly into five folds of seven unique images.
In all experiments the training covered 30 epochs where only the weights of the decoder were trained and the weights of the pretrained base model were frozen, and ten more epochs of fine-tuning the models. For the first part, each epoch covered 616 randomly augmented images, which are processed by a batchsize of 7 and 88 training steps. For validation only 22 steps are taken, which results in a total of 154 randomly augmented images.
For the second part, a batchsize of 7, 44 training steps and 11 validation steps were used, which is a total of 308 and 77 randomly augmented images respectively.
The learning rate for the first part of the training was set to 1e-4 and for the fine-tuning to 1e-5.

5.1.1 Decoding Path - Simple

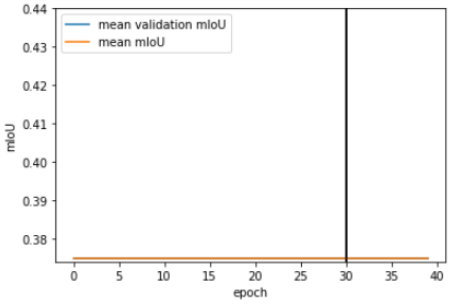
The simple decoding path is a lower bound on the complexity that a decoder can have. The experiments give insight on whether the segmentation task can be solved by such a minimalist decoder or not. A simple network is



(a) Train and validation loss.



(b) Train and validation accuracy.



(c) Train and validation mean intersection over union.

Figure 20: The plots illustrate the training histories of the "Simple Decoder" approach with different metrics. the vertical line at epoch 30 indicates the change of the training settings, i.e., fine-tuning of pre-trained weights with a lower learning rate and less train and validation steps.

usually preferable to a more complex one due to shorter training times and less overfitting possibilities, as long as the task is solved well.

The training histories of the "Simple Decoder" network are shown in Figure 20. The subplots show the average metric over all folds in a dark hue and its standard deviation in a light hue around the mean, with the validation metric in blue and the train metric in orange.

Plot (a) in Figure 20 shows that the training loss decreases in a steady way, whereas the validation loss seems to fluctuate without getting any better in the first part of the training. During the second part of the training (i.e., fine-tuning starting at epoch 30) the validation loss starts to decrease and

converge to the training loss. The same phenomenon can be seen in plot (b), which shows the train and validation accuracy.

The training histories seem to indicate that the decoding path on its own is not quite able to generalize properly. When starting fine-tuning, the whole network is trained, which seems to do the trick and prompt the network to generalize. This seems at first unintuitive, since usually a network is prone to overfitting, if it is too complex and does not have enough training data. However, one could argue that the pre-trained base model might be too distant to the problem task at hand and therefore the network is not able to train anything useful.

Plot (c) shows the mean intersection over union, which is usually a more suited way to evaluate a semantic segmentation task. It seems, however, that the metric fails to trace these small improvements. At this point it is important to note that the mean intersection over union measures and averages the overlaps of prediction and true areas for each label, which means that labels that are sparse (e.g. dead tree areas) are as important as labels that are dense (e.g. healthy tree areas or background areas). Due to this fact, the mIoU metric will probably only start to improve when the accuracy increases above 90% (which is a rough estimate). Before that, the changes are too small to be shown properly in the mean intersection over union.

Also, note that Figure 20 (a) and (b) show a peak at epoch 30. This indicates that the optimizer was switched to have a smaller learning rate and the new optimizer did not inherit the state of the previous optimizer. Since, the optimizer already recovers in the following epoch, this should not have too much impact.

Table 1 shows the best results of each fold and their mean and standard deviation. The mean validation accuracy is 87.2% with a standard deviation of 1. The mean validation loss is 0.329 with a standard deviation of 0.023.

	Train Loss	Val. Loss	Train Acc.	Val. Acc.	Train mIoU	Val. mIoU
Fold 0	0.211	0.323	91.8%	87.3%	0.375	0.375
Fold 1	0.204	0.349	92.1%	86.5%	0.375	0.375
Fold 2	0.207	0.321	92%	87.1%	0.375	0.375
Fold 3	0.21	0.295	91.9%	89%	0.375	0.375
Fold 4	0.204	0.359	92.1%	86%	0.375	0.375
Mean	0.207	0.329	92%	87.2%	0.375	0.375
Std.	0.003	0.023	0.1%	1%	0	0

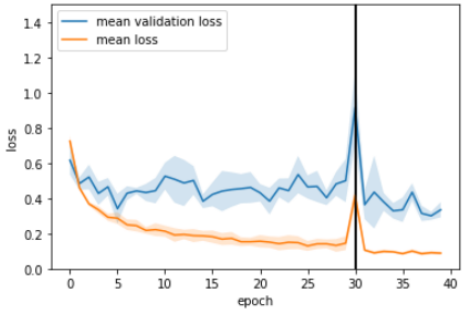
Table 1: The table shows the best results for each fold and metric using the "Simple Decoder" approach. At the bottom of the table the mean and standard deviation of each metric over all folds can be seen.

The results are similar to the baseline proposed in [Humer, 2020], however, it should be noted that there is still a lot of potential in fine-tuning the "Simple Decoder" approach, which will probably lead to much better results, whereas the baseline network already seems to have converged more or less.

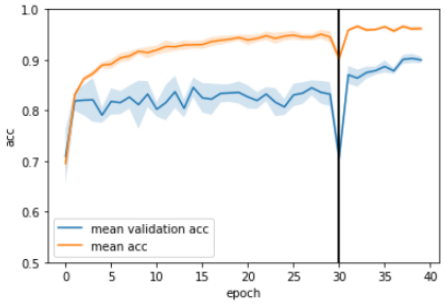
5.1.2 Decoding Path - U-Net

The "U-Net Decoder" approach gives a more sophisticated architecture and has already been proven to be able to tackle this task in [Humer, 2020]. The replacement of the encoding path by a pre-trained network should therefore enhance the baseline architecture and yield better results and a better generalization of the network.

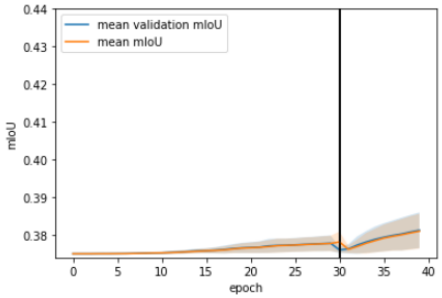
Figure 21 shows the plots for training/validation loss, accuracy and mean intersection over union like in Figure 20. Similar to the behaviour of the "Simple Decoder" approach, the training for this network shows only improvements for the validation loss and accuracy, during fine-tuning. Also, note that plot (c) shows the history of train and validation mIoU, which both improve similarly. This behaviour might suggest that the network seems to be able to improve in a way that influences the mean intersection over union



(a) Train and validation loss.



(b) Train and validation accuracy.



(c) Train and validation mean intersection over union.

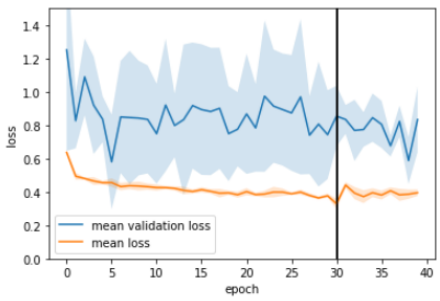
Figure 21: The plots illustrate the training histories of the "U-Net Decoder" approach with different metrics. The vertical line at epoch 30 indicates the change of the training settings, i.e., fine-tuning of pre-trained weights with a lower learning rate and less train and validation steps.

in a better way than it did for the "Simple Decoder" approach.
Similar to the phenomenon in the previous experiment, the validation loss and accuracy start to converge to the train loss and accuracy as soon as the fine-tuning part of the training starts. The mean intersection over union, however, improves very similarly in both, train and validation, which could be explained by the nature of the measure, as it was described in the previous experiment. Due to the fact that the labels within the dataset are highly unbalanced, it is beneficial to consider the mIoU metric to be important, since it treats each class with an equal amount of importance, no matter if it is sparse or dense.

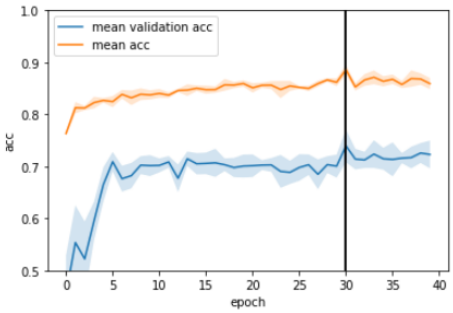
	Train Loss	Val. Loss	Train Acc.	Val. Acc.	Train mIoU	Val. mIoU
Fold 0	0.095	0.235	96.3%	90.5%	0.38	0.38
Fold 1	0.091	0.264	96.4%	90.6%	0.389	0.39
Fold 2	0.079	0.313	97%	89.8%	0.383	0.383
Fold 3	0.074	0.247	97.1%	91.7%	0.378	0.378
Fold 4	0.079	0.261	96.7%	90.4%	0.377	0.377
Mean	0.084	0.264	96.7%	90.6%	0.381	0.381
Std.	0.008	0.027	0.3%	0.6%	0.004	0.004

Table 2: The table shows the best results for each fold and metric using the "U-Net Decoder" approach. At the bottom of the table the mean and standard deviation of each metric over all folds can be seen.

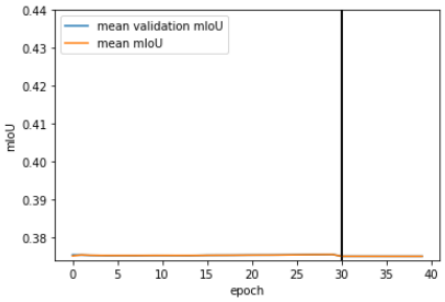
The best results of each metric and fold can be seen in Table 2, as well as the average and standard deviation over the folds. Note that the mean validation accuracy is 90.6%, the validation loss is 0.264 and the validation mean intersection over union is 0.381, which exceed the results of the baseline. It should also be noted that, similar to the previous approach, there is still a lot of potential in the fine-tuning part of this network. This means that the validation results will probably converge to the training results even more, if it would be trained for more epochs.



(a) Train and validation loss.



(b) Train and validation accuracy.



(c) Train and validation mean intersection over union.

Figure 22: The plots illustrate the training histories of the "DenseNet Decoder" approach with different metrics. the vertical line at epoch 30 indicates the change of the training settings, i.e., fine-tuning of pre-trained weights with a lower learning rate and less train and validation steps.

5.1.3 Decoding Path - DenseNet

The most complex architecture used in the experiments uses dense blocks in the decoding path. [Jégou et al., 2017] argues that the architecture of dense blocks is well suited for segmentation tasks, since it naturally introduces skip connections. Due to the complexity of the network, it is hard to find a sufficient amount of training epochs and suitable network settings.

Figure 22 shows the training and validation loss, accuracy and mean intersection over union. The train accuracy and loss behave similar to the previous experiments as well as the validation accuracy. The validation loss, however,

fluctuates heavily during the first training phase and seems to stabilize a bit during fine-tuning. Evaluation on the validation set is not satisfactory, since it does not seem to converge to the results of the training set. Also, the mean intersection over union does not show significant improvement.

	Train Loss	Val. Loss	Train Acc.	Val. Acc.	Train mIoU	Val. mIoU
Fold 0	0.365	0.392	87.4%	74.3%	0.376	0.376
Fold 1	0.355	0.354	87.8%	77.4%	0.375	0.375
Fold 2	0.291	0.51	89.6%	74.8%	0.375	0.376
Fold 3	0.324	0.351	88.7%	77.3%	0.376	0.376
Fold 4	0.299	0.458	89.6%	75.4%	0.376	0.376
Mean	0.327	0.413	88.6%	75.8%	0.376	0.376
Std.	0.029	0.062	0.9%	1.3%	0.0002	0.0002

Table 3: The table shows the best results for each fold and metric using the "DenseNet Decoder" approach. At the bottom of the table the mean and standard deviation of each metric over all folds can be seen.

With a mean validation loss of 0.413 and a mean validation accuracy of 75.8%, as it is shown in Table 3, the "DenseNet Decoder" approach was not able to outperform the baseline with the chosen settings. Nevertheless, the architecture itself is promising and further experiments should be conducted to generate a well performing network. Due to the complexity of this network, it might already be helpful to introduce more training steps, especially more fine-tuning epochs.

5.1.4 Comparison

Table 4 gives an overview over the mean results of all approaches, which makes it easier to compare their performance.
With the proposed training settings, the "U-Net Decoder" approach is able to outperform all other approaches. Most important to note is the increased performance of the mean intersection over union measure, which also consid-

	Train Loss	Val. Loss	Train Acc.	Val. Acc.	Train mIoU	Val. mIoU
Baseline	0.267	0.326	89.2%	87%	0.375	0.375
Simple D.	0.207	0.329	92%	87.2%	0.375	0.375
U-Net D.	**0.084**	**0.264**	**96.7%**	**90.6%**	**0.381**	**0.381**
DenseNet D.	0.327	0.413	88.6%	75.8%	0.376	0.376

Table 4: The table shows the mean results for each semantic segmentation approach and the baseline based on the experiments from [Humer, 2020].

ers the highly unbalanced distribution of labels in the dataset.

The goal of exceeding the proposed baseline was achieved, however, it is important to note that the potential of the proposed networks is not reached yet and further training - especially fine-tuning - will most probably result in much better evaluation performance.

The training histories, which were shown in Figure 20, 21 and 22 suggest that the pre-trained encoder task is too distant from the task at hand, and therefore requires the fine-tuning step. Also, it seems that the fine-tuning part is not so prone to overfitting, as it was assumed earlier due to the limited amount of data. This is probably due to the nature of semantic segmentation tasks, which is more complex than classification and they should therefore not be directly compared to each other.

The proposed networks could probably be improved by using a bigger input size, which is currently limited by the resources available. Also, some direct connection between input and output might be beneficial to enable the network to make more fine-grained predictions.

5.2 Classification

Due to limited computing resources, the models were only trained and evaluated with a random train-validation-test-split, and not with k-fold cross validation, which would be preferable with such a low amount of data. The train set consists of 40 images and the test and validation sets each consists of 20 images. The validation set is required to compare different network settings and optimize hyperparameters. To evaluate the model in the end, another unseen dataset (test dataset) is required, which was not used in the previous steps.

All classification networks were trained over the course of 20 epochs. The training data amount of 40 images was artificially enlarged to 960 by generating 24 random augmentations of each image. The same artificial data enlargement was performed on the validation set which results in 480 validation images.

Several experiments with different base models and input sizes were conducted and evaluated using the enlarged validation dataset.

5.2.1 Input Size: 512x512

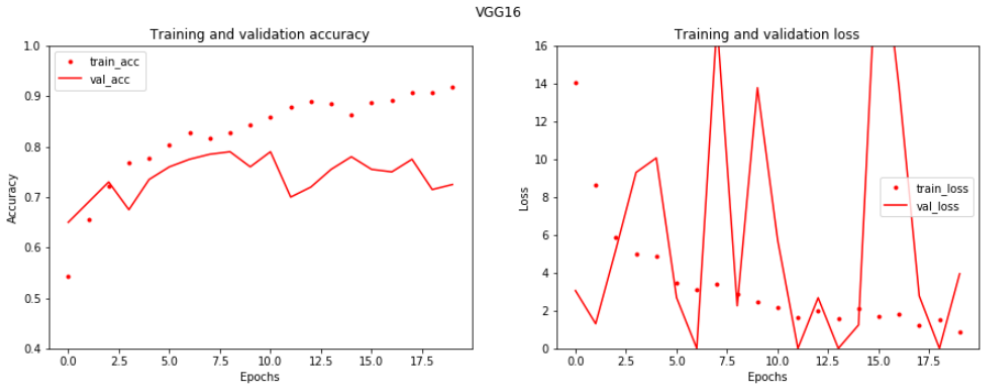


Figure 23: The plot shows the training history of the classifier with the pretrained VGG16 base model and an input size of 512x512.

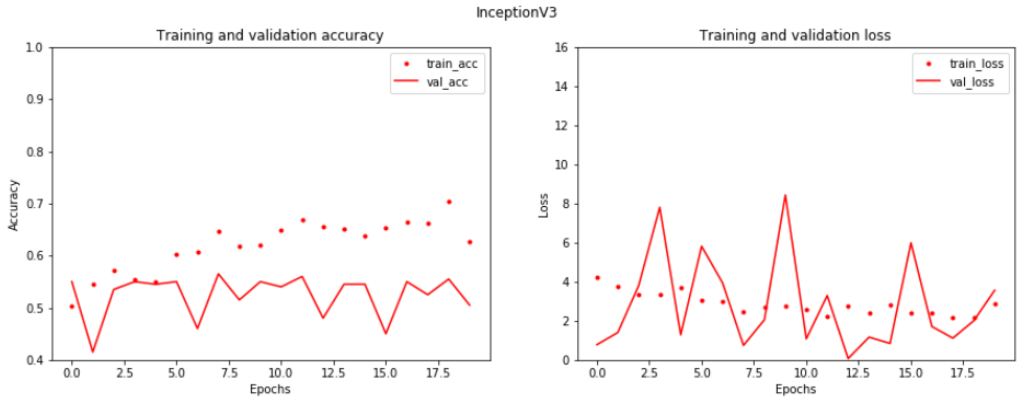


Figure 24: The plot shows the training history of the classifier with the pretrained InceptionV3 base model and an input size of 512x512.

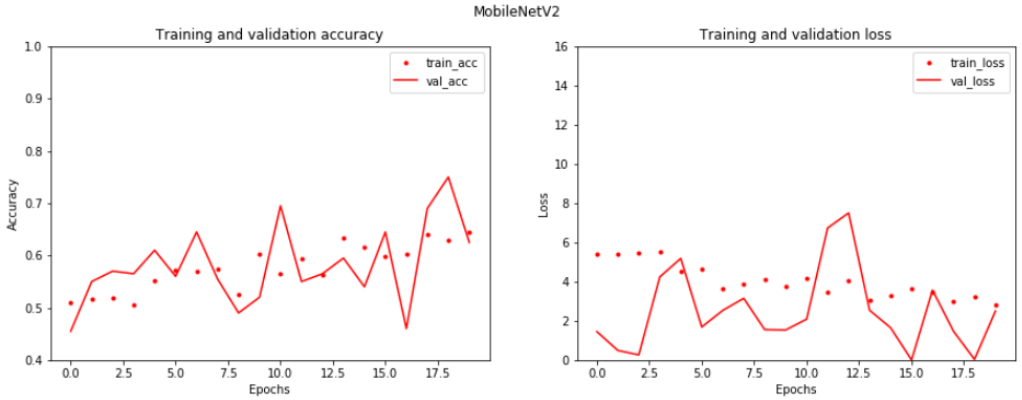


Figure 25: The plot shows the training history of the classifier with the pretrained MobileNetV2 base model and an input size of 512x512.

Experiments with an input size of 512x512 were conducted to check, whether the network is able to determine the signs of a bark beetle infestation - which are usually relatively small, especially on a low resolution input image. Smaller image resolutions are favorable because the network has less features to process and is therefore much faster.

With a batch size of 8, the training process needs 120 steps within one epoch to process the available data. For validation, 60 steps are required to process the available data.

The training histories of the classifiers - as shown in Figures 23, 24, 25 and 26

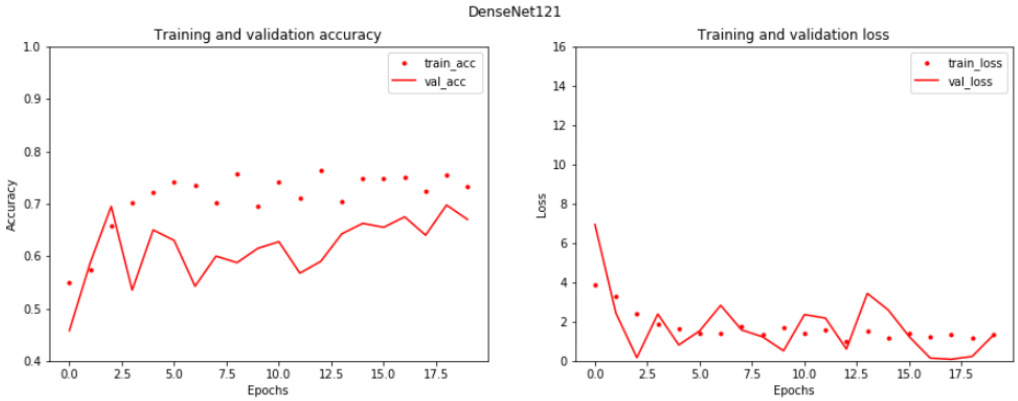


Figure 26: The plot shows the training history of the classifier with the pretrained DenseNet121 base model and an input size of 512x512.

- generally show a stable increase of training accuracy and a stable decrease of training loss. Validation loss and accuracy however, seem very unstable. The plot on the right-hand side of Figure 23 shows extreme fluctuations of validation loss during the training of a classifier with the VGG16 base model. On the left-hand side of Figures 23 and 26, which correspond to VGG16 and DenseNet121, the validation accuracy seems quite stable compared to the validation accuracy of the other two models. The accuracy history of the VGG16 model also shows that the network starts to overfit to the training data around the seventh epoch where train and validation accuracy seem to diverge.

In Figure 24 it can be seen that the network with the InceptionV3 base model does not seem to learn anything useful when looking at the validation accuracy and loss, whereas the other models seem to generalize at least to some degree.

In case of the classifier with the MobileNetV2 base model that is shown in Figure 25, it can be seen that the train and validation accuracy grow a bit, but the validation accuracy fluctuates heavily. Also, the best validation accuracy is better than the best training accuracy, which seems strange. This

could be caused by variations in the datasets and solved by using k-fold cross validation, which delivers a more unbiased evaluation.
In general it can be said that the DenseNet121 classifier seems to be the most stable model in terms of validation loss and accuracy, whereas the VGG16 classifier delivers the best validation accuracy, but is prone to overfit.

Base Model	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy
Baseline	7.253	7.253	55%	55%
VGG16	0.847	0.0	**91.9%**	**79%**
InceptionV3	2.152	0.066	70.4%	56.5%
MobileNetV2	2.804	0.001	64.4%	75%
DenseNet121	0.975	0.079	76.4%	69.7%

Table 5: The table states the best results for each of the techniques with an input size of 512x512.

Table 5 shows the metrics for the best result for each base model. All of the experiments were able to exceed the baseline classifier. Due to the unstable behaviour of validation loss, models will not be compared by this measure for now. When measuring performance by validation accuracy, the VGG16 classifier performs better than all other classifiers with 79% followed by MobileNetV2 with 75%.
The difference between train and validation accuracy is exceptionally high for InceptionV3 and VGG16 with almost 14 and 13 percentage points respectively. This suggests that the two models are prone to overfit to the training data. MobileNetV2 and DenseNet121 yield a difference of 10.6 and 6.7 percentage points.
These results suggest that the models are able to learn something useful, although the input resolution is quite small. The next subsection compares the results of the models when using a higher input resolution.

5.2.2 Input Size: 768x768

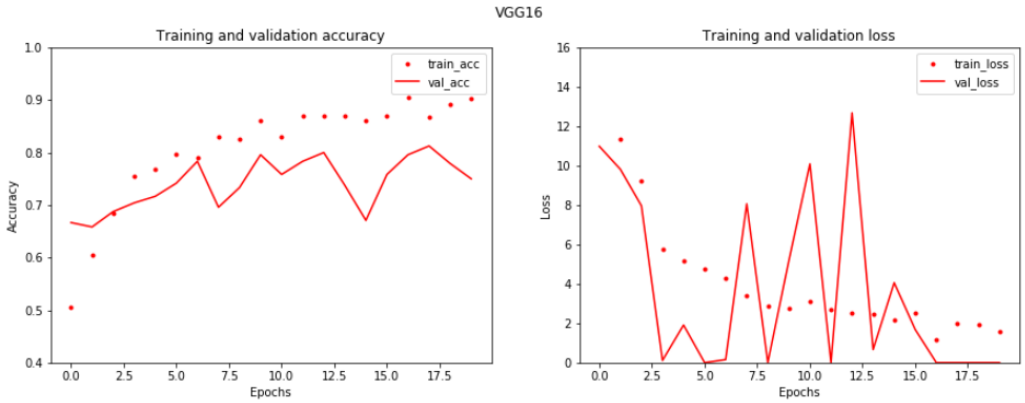


Figure 27: The plot shows the training history of the classifier with the pretrained VGG16 base model and an input size of 768x768.

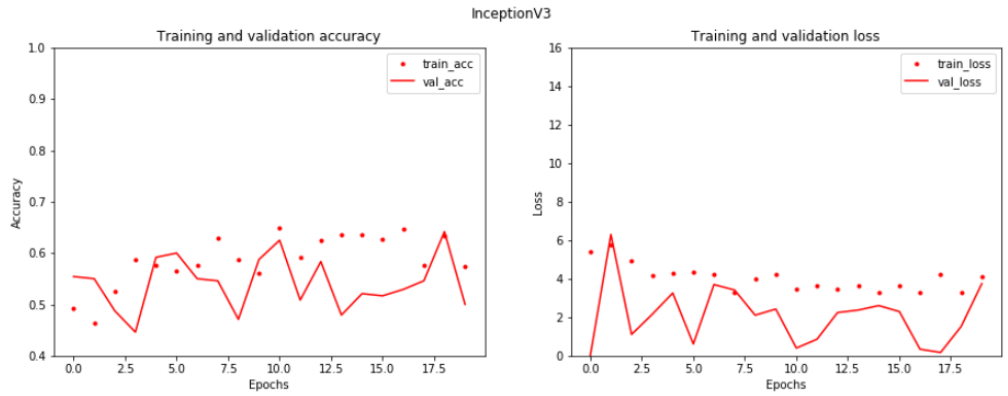


Figure 28: The plot shows the training history of the classifier with the pretrained InceptionV3 base model and an input size of 768x768.

Although the classifiers were able to retrieve some relevant information from 512x512 input data, the performance is not as good as it is desired. With a 768x768 input size, the data becomes more complex and should help against the overfitting problem of the VGG16 classifier. Also, a higher resolution input means more information (relevant and irrelevant) and may therefore need more epochs to converge, but may also yield better results in general. A bigger input, also means that more memory is required and therefore a

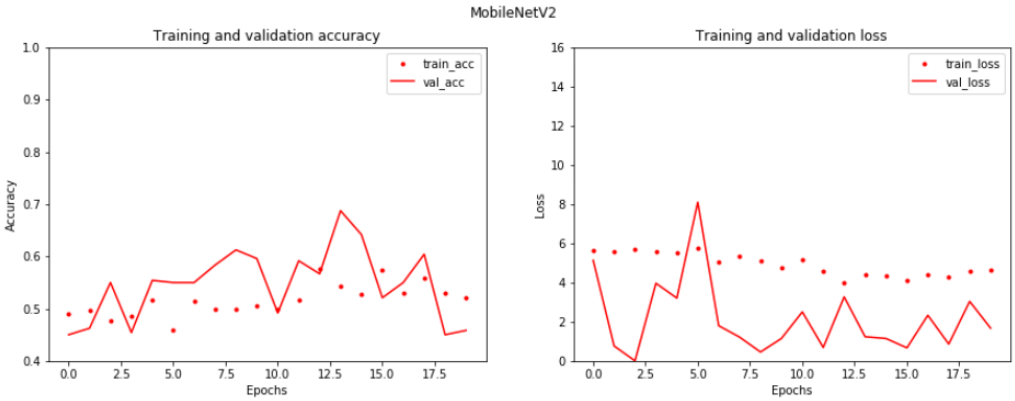


Figure 29: The plot shows the training history of the classifier with the pretrained MobileNetV2 base model and an input size of 768x768.

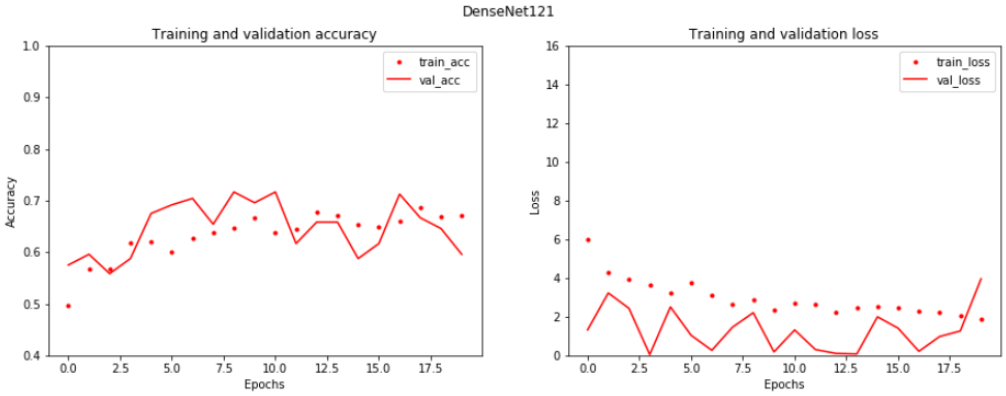


Figure 30: The plot shows the training history of the classifier with the pretrained DenseNet121 base model and an input size of 768x768.

smaller batch size is required in order to train the network without running into memory issues. For the following experiments, a batch size of 4 is used and the number of training and validation steps are doubled compared to the experiment settings from before (240 training steps and 120 validation steps). This results again in a total of 960 training images and 480 validation images.

The training and validation histories shown in Figures 27, 28, 29 and 30 reveal similar results as in the previous experiments. In general it seems that

the fluctuations of the validation metrics are smaller. The VGG16 and InceptionV3 classifiers show an improvement regarding their previous overfitting problem. Also, the InceptionV3 model seems to learn a little bit where it previously could not extract anything useful when comparing the plots in Figure 24 and Figure 28.
The DenseNet121 classifier seems again to be the most stable one with regards to validation accuracy and loss.

Base Model	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy
Baseline	7.253	7.253	55%	55%
VGG16	1.172	0.0	**90.4%**	**81.3%**
InceptionV3	3.275	0.011	65%	64.2%
MobileNetV2	3.984	4.822	57.7%	68.8%
DenseNet121	0.845	0.038	68.8%	71.7%

Table 6: The table states the best results for each of the techniques with an input size of 768x768.

Table 6 shows the metrics for the best result for each base model. Again, all of the experiments were able to exceed the baseline classifier and almost every model performed better than the respective model of the previous experiment with regards to the validation accuracy. Only the MobileNetV2 classifier seems to perform worse with a difference of about six percentage points in validation accuracy and ten percentage points in train accuracy.
Due to the unstable behaviour of validation loss, models will not be compared by this measure. When measuring performance by validation accuracy, the VGG16 classifier again performs better than all other classifiers with 81.3% followed by DenseNet121 71.7%.
The difference between train and validation accuracy is exceptionally high for MobileNetV2 with about 11 percentage points. This suggests that the model is prone to overfit to the training data. VGG16 yields a difference of around 9 percentage points and DenseNet121 yields a difference of almost 3 percentage

points, which is an improvement compared to the previous experiments. The lowest difference with 0.8 % was produced by the InceptionV3 classifier.
Although the improvements seem small, it is clear that the a higher input resolution counters the problem of overfitting, which is very important with such a small amount of data available for this project.

5.3 Classification Extension: Detect Trunk

Due to the limitations in training data, it is important to preprocess the data in a way that reduces the overfitting possibilities of the networks. This means that noise from the input data should be removed as much as possible such that the model is able to focus on the differences that are important, i.e., the signs of a bark beetle infestation.
The most noisy features of an image are background-pixels, since their variance is big, but they do not carry important information for classification. Sometimes, half the image consists of background, which makes it especially hard for a model to extract something useful.
As mentioned in Section 4.2.3, a semantic image segmentation network was implemented to segregate the trunk from the background using the architecture from [Humer, 2020].
Models with different input sizes were trained and evaluated using the same train-validation-test split as it was used in Section 5.2. The corresponding masks, which label the image as "background" or "trunk", were created manually in the same way as in the "Localization" part of this project.
All models were trained over the course of 90 epochs. Due to computational limitations, the models were not trained over 90 epochs at once, but they were trained in junks of either 10 or 20 epochs, which only influences the training due to the fact that the optimizer state is reset after each of the junks.

The training histories in Figures 31, 33 and 35 show evaluations on train and validation data based on the metrics "accuracy", "loss" and "mean intersection over union (mIoU)".

5.3.1 Input Size: 128x128

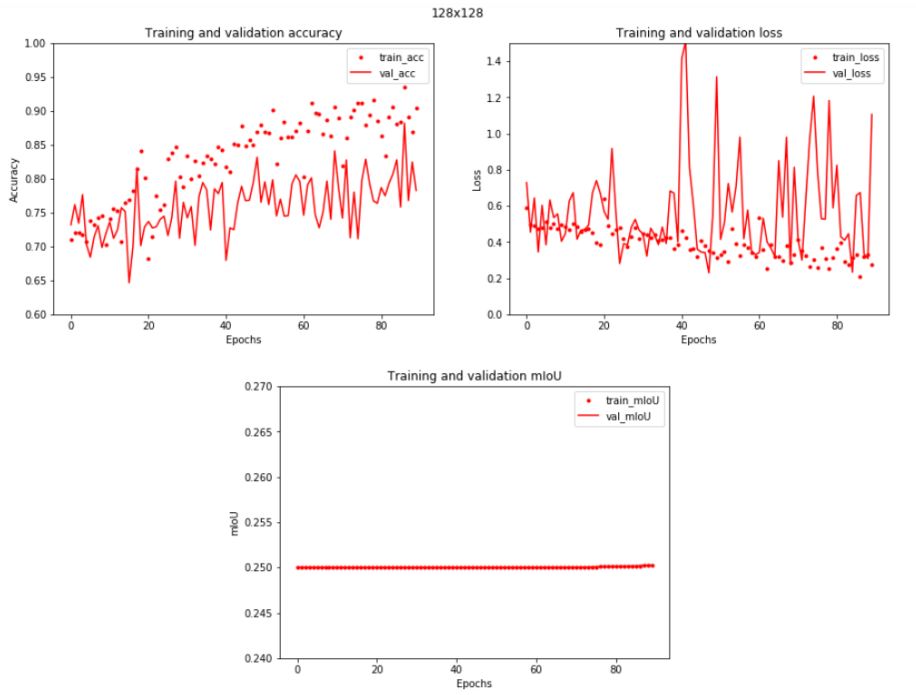


Figure 31: The plot shows the training history of the trunk detection network with an input size of 128x128.

In the first experiment the input size of the model was set to 128x128. Due to the fact that the segmentation task is not as complex and fine grained as it was in the "localization" part of this project, it is possible that such a small input size is sufficient to solve the task. As mentioned before, smaller images speed up the training significantly.
The training history, which is depicted in Figure 31 shows slight improvement for each measure. The validation accuracy shows signs of overfitting after training reached epoch 20 and the validation loss starts to fluctuate heavily

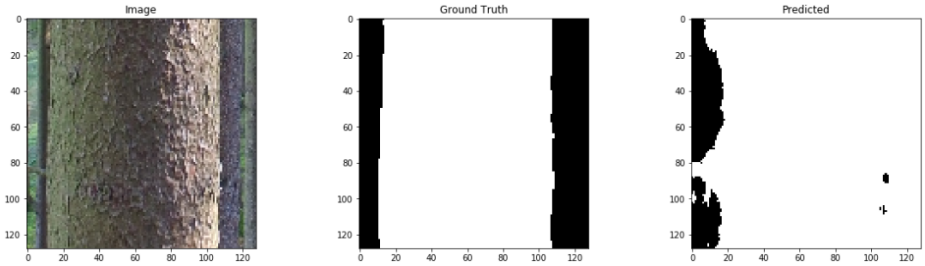


Figure 32: The figure shows an example input image on the left, the annotated ground truth in the middle and the mask prediction of the network with a 128x128 input size on the right-hand side.

after epoch 40. The mean intersection over union starts to improve slightly after epoch 80.

Overall, it seems like the network can learn something from the small input, but not enough to converge properly.

Figure 32 shows an example of detecting a trunk using the small input size network. It can be seen that the trunk cannot be properly segregated from the background.

5.3.2 Input Size: 256x256

In the second experiment the input size of the model was set to 256x256, which gives the model much more information to derive from and therefore results in longer training, but should also result in a more accurate mask.

The training history in Figure 33 shows nice improvement in all measures. Compared to the previous experiment, the network does not seem to overfit at all and also the mean intersection over union measure starts to improve after epoch 50. The plots show weird peaks at epoch 40, 60 and 70, which happen due to the interruptions of the training process, which reset the current state of the optimizer. However, the optimizer recovers soon after and is able to continue the training properly.

The network is able to deliver nice results with the increased input resolu-

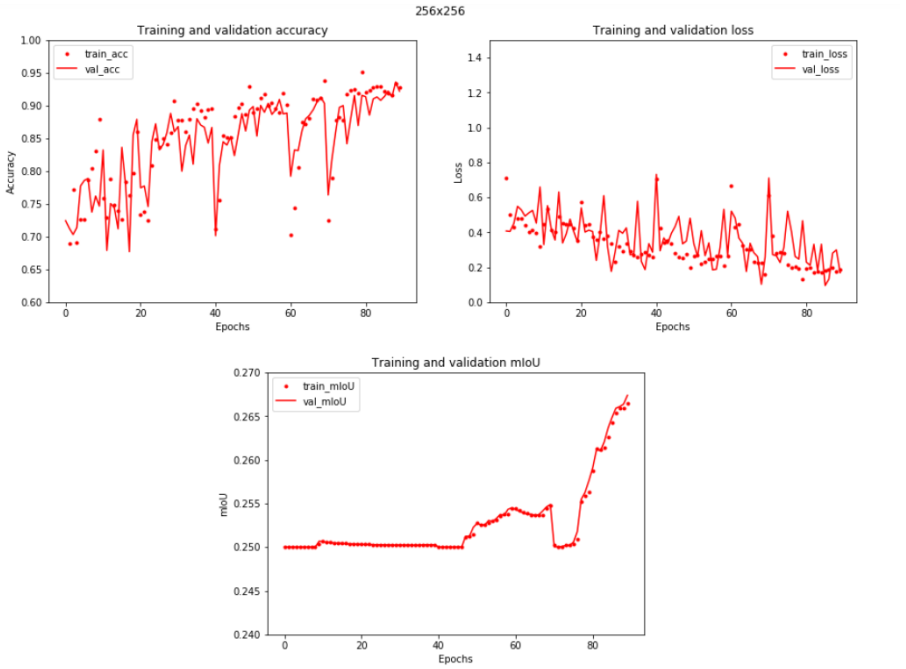


Figure 33: The plot shows the training history of the trunk detection network with an input size of 256x256.

tion, which tells us that the previous resolution was, in fact, too small. Figure 34 shows an example of detecting a trunk using the high resolution input data. This network manages to segregate tree and background reasonably well for this example. Note that the network is confused by the change of light in the bottom right corner of the trunk and missclassifies this area as background.

5.3.3 Input Size: Trade-off

The third experiment combines the advantages of the previous models, i.e. reduce training time, speed up convergence and still provide accurate results. This is accomplished by setting the input size to 128x128 at first and after 40 epochs the trained weights are stored. Then a model with an input size of 256x256 is initialized with the previously stored weights and trained for

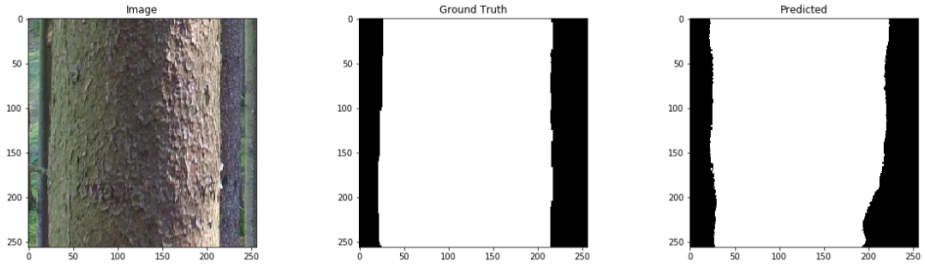


Figure 34: The figure shows an example input image on the left, the annotated ground truth in the middle and the mask prediction of the network with a 256x256 input size on the right-hand side.

another 50 epochs.

The training history in Figure 35, again, shows nice improvement in all metrics. It seems like the pretrained weights of the model with input size 128x128 gives the model with the bigger input size a headstart, since the mean intersection over union measure starts to improve at epoch 50 like in the previous experiment. The evaluation confirms that a speed up in training time is achievable by training the network on low image resolution first and on higher resolution input at a later stage, and still deliver comparable results. Figure 36 shows an example of detecting a trunk using the trade-off network. This network manages to segregate tree and background very well for this example, even for the light spot on the bottom right corner of the trunk.

Input Size	Train Loss	Val. Loss	Train Acc.	Val. Acc.	Train mIoU	Val. mIoU
128x128	0.209	0.229	93.45%	88.19%	0.2502	0.2502
256x256	0.131	0.095	95.16%	93.61%	**0.2664**	**0.2674**
Trade-off	0.107	0.096	**95.79%**	**95.19%**	0.261	0.2614

Table 7: The table shows the best metrics for models that were trained on different input sizes.

Table 7 shows the best results for each metric and network. Again, it can be seen that, with an input size of 128x128, the model is not able to deliver desirable results. It is also notable that the model trained on both input sizes

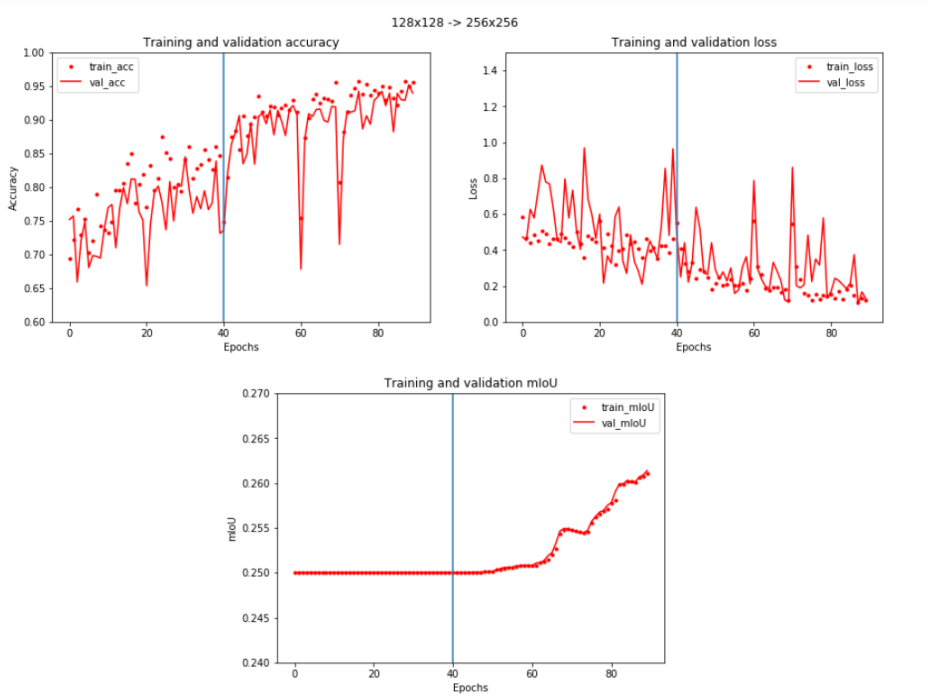


Figure 35: The plot shows the training history of the trunk detection network with an input size of 128x128 until epoch 40 and an input size of 256x256 until epoch 90. The blue line indicates the switch of input sizes.

yields a better validation accuracy, but a smaller validation mIoU compared to the model that was only trained on the higher input resolution. Both of the latter networks do not seem to overfit heavily, however, the "trade-off" network shows smaller differences between train and validation metrics than the model with an input resolution of 256x256.

5.3.4 Trunk Classification

The resulting mask, which was produced by one of the previously introduced segmentation networks, can be used to preprocess the data that is further processed by the classification network. Experiments in this chapter are conducted with the VGG16 and DenseNet121 classifiers, which seemed most promising during the previous experiments.

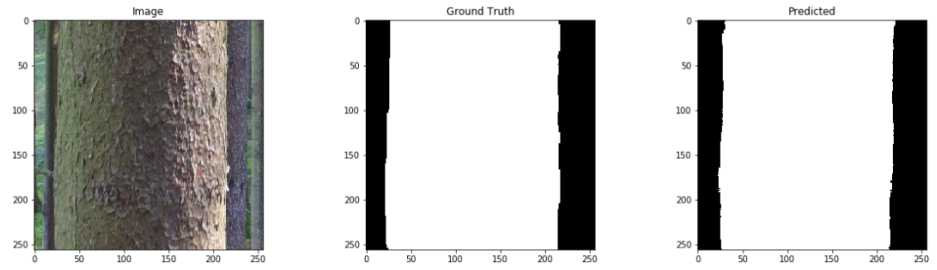


Figure 36: The figure shows an example input image on the left, the annotated ground truth in the middle and the mask prediction of the trade-off network on the right-hand side.

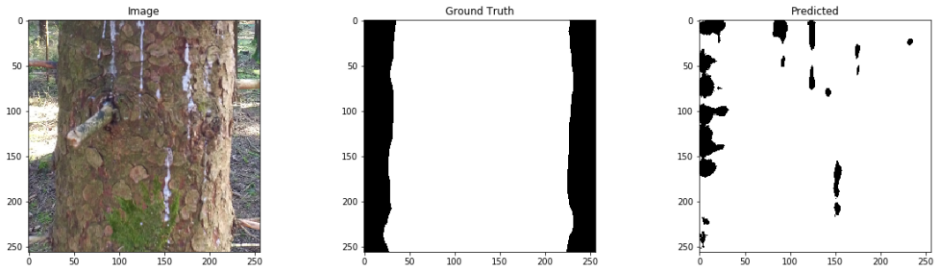


Figure 37: The plot shows a failed trunk detection, which misclassifies critical regions of the trunk.

There are several ways to perform the preprocessing using the generated masks, two of the possible approaches are covered in this project.
Following the simple approach, the resulting masks are multiplied with the corresponding images and the classifiers are trained on the masked images. Before multiplying, the masks need to be set to zero or one, depending on whether their value is above or below some threshold, which is set to 0.5 here. The desired result would be a black background and an unmodified trunk. However, in order for this approach to work as desired, the trunk detection network that produces the mask needs to perform very well, otherwise, it will introduce noise due to trunk pixels being set to zero as it can be seen in Figure 37.
Figure 38 and 39 show the training history of the two networks given the

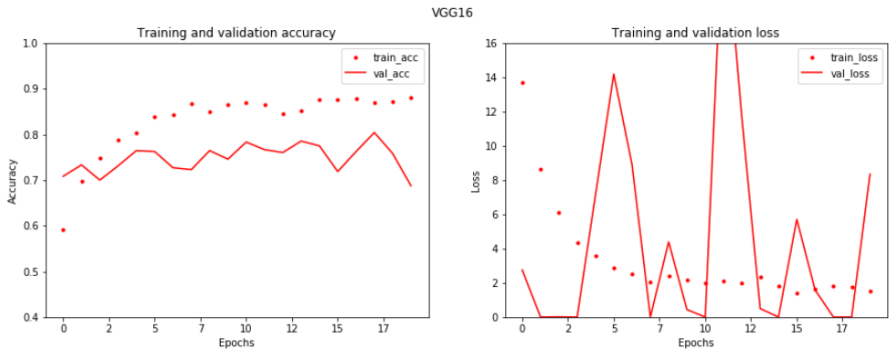


Figure 38: The plots show the train and validation accuracy (left) and the train and validation loss (right) of the VGG16 classifier with masked inputs.

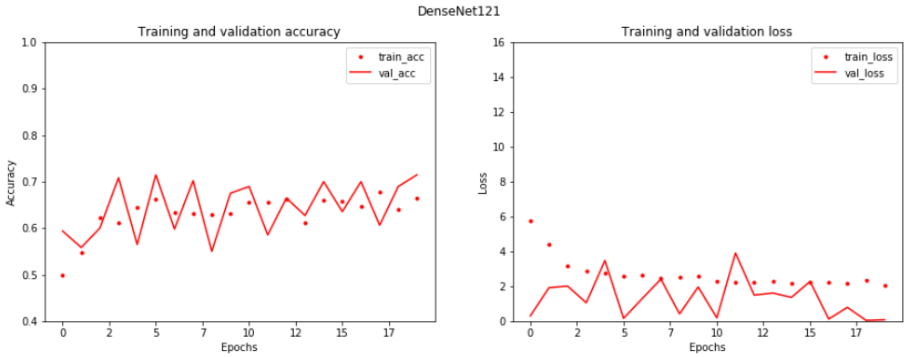


Figure 39: The plots show the train and validation accuracy (left) and the train and validation loss (right) of the DenseNet classifier with masked inputs.

inputs that were multiplied by the trunk detection network masks. The training histories do not seem to improve i.e. the training loss of the VGG16 network is still fluctuating heavily and the network seems to be overfitting quite early. For the DenseNet121 classifier the validation loss seems to fluctuate a bit more than before.

Since the first approach does not seem to improve anything, another experiment was conducted, which uses the produced mask to calculate the probability of each column in an input image to contain a trunk. Since input images are supposed to contain a standing tree, only the columns of an image are considered, i.e., the produced mask is averaged over its row values. The

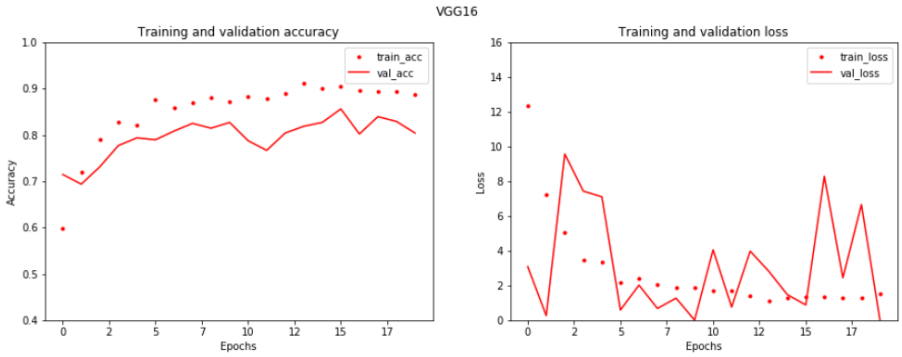


Figure 40: The plots show the train and validation accuracy (left) and the train and validation loss (right) of the VGG16 classifier with cropped inputs.

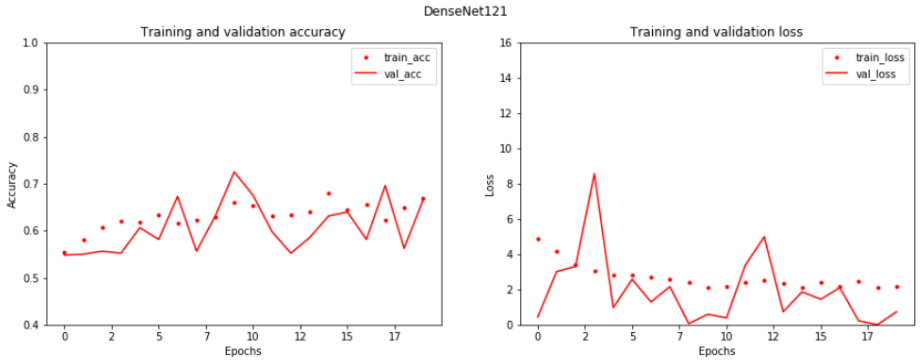


Figure 41: The plots show the train and validation accuracy (left) and the train and validation loss (right) of the DenseNet classifier with cropped inputs.

resulting array has the size of the image width and contains the probabilities for each column to contain a part of the trunk.

The next step is the definition of some threshold, with which the array is divided into "contains trunk" and "does not contain trunk". This results in an array with the values "true" and "false", which can be used to drop those columns in the original image that are not likely to contain a part of the trunk. For this project a quite small threshold of 0.2 was used because a larger threshold would make it more likely that important parts of an image are removed as well.

The drawback of this approach is that each input image will have a different image size after the preprocessing. This can result in distortion of the classifier input, since the size of the network is currently fixed to 768x768 and the images need to be interpolated to fit the resolution.
Figure 40 and 41 visualize the training history of the VGG16 and DenseNet121 classifiers with truncated images. In case of the VGG16 classifier, it seems that the validation loss shows reduced fluctuations. It should also be noted that the training and validation accuracies are much closer to each other, which suggests less overfitting than previously observed.
The DenseNet121 classifier with the new inputs does not show too much difference to the previous approach with regards to the training history.

Base Model	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy
Baseline	7.253	7.253	55%	55%
VGG16	1.172	0.0	**90.4%**	**81.3%**
DenseNet121	0.845	0.038	68.8%	71.7%
Masked VGG16	1.379	0.0	88%	80.4%
Masked DenseNet121	2.064	0.715	67.8%	71.5%
Cropped VGG16	1.082	0.0	**91.1%**	**85.6%**
Cropped DenseNet121	2.104	0.0	67.9%	72.5%

Table 8: The table compares the best metrics of the classifiers without preprocessed images, the classifiers with the simple preprocessing approach and the classifiers with cropped images.

When comparing the best results of each input data approach and classifier, as it is summarized in Table 8, it can be seen that the train and validation accuracy of the VGG16 classifier with truncated images exceeds the results of the VGG16 classifier without preprocessing by 0.7 and 4.3 percentage points respectively. It is important to note that the difference in train and validation accuracy decreased, which means that the preprocessing step helps the VGG16 classifier to generalize better and overfit less.
Also, note that the DenseNet121 classifier does not seem to be affected by

the preprocessed input as much, which seems unintuitive. One explanation could be that the DenseNet121 classifier needs more training epochs in order to get better results and show some change.

5.4 Classification: Test-Set Evaluation

For the final evaluation of the best classification network, a set of 20 images is taken, all of which have never been used in any of the previous experiments. This test-set was randomly selected during the train-validation-test split that was conducted earlier and should therefore be able to represent results for future use of the network.
The network in question is the classifier with the VGG16 pretrained base model, as explained in Section 4.2, using input data that was preprocessed with the help of the trunk detection network from Section 5.3.3.

Metric	Value
Loss	7.7e-6
Accuracy	0.9
Precision	0.8181
Recall	1.0
F1-Score	0.9

Table 9: The table shows the test-set evaluation results of the VGG16 classifier trained on truncated input images.

Although the test-set is representative for future, unseen data, it is important to note that the evaluation results are only valid to a certain degree due to the limited amount of 20 test images, i.e., the real performance of the network could be within a certain variance of the evaluation results.
The results shown in Table 9 look quite promising. The precision score suggests that there are certain images of infested trees that are classified as "healthy", which is problematic. The recall score, however, ensures that no healthy trees are mistaken for being infested, which means that it is unlikely

to remove a healthy tree by mistake. This is also confirmed when looking at the confusion matrix in Figure 42.

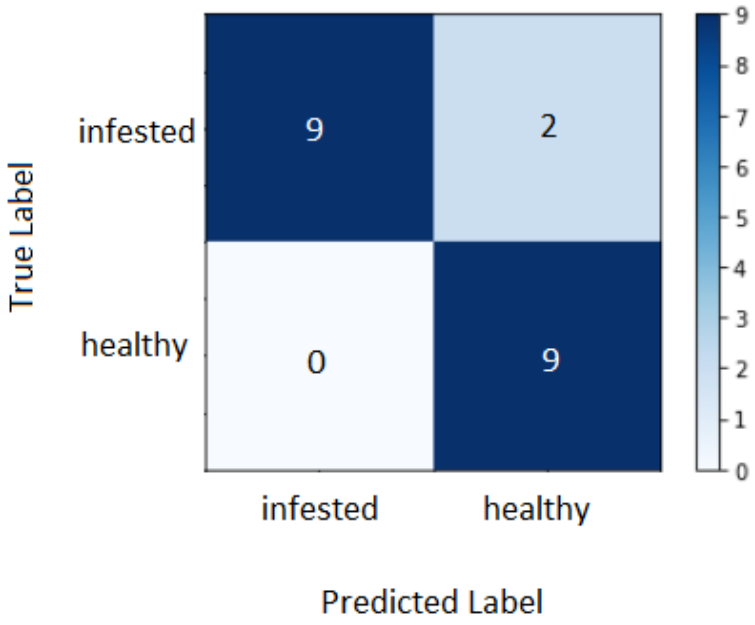


Figure 42: The figure shows the confusion matrix for the evaluation of the chosen classification network.

6 Conclusion

The development of deep learning models to support the early detection of spruce bark beetles is interesting and important to slow down the deforestation of spruce trees. Image processing techniques like semantic image segmentation and image classification were successfully demonstrated in this project to develop a system that is able to localize sick trees and furthermore identify trees infested by the "Eight-Toothed Spruce Bark Beetle" (Ips Typographus).

The experiments in this project were limited by a lack of resources and a small number of training images, the results, however, were reasonable. Also, due to those limitations, the ideas that were introduced in the project still have a lot of potential for improvement.

The best results for the "Localization" part of this project were delivered by the architecture that used the pre-trained DenseNet121 model in the encoding path and the decoding path as proposed in [Humer, 2020]. The pixel-wise validation accuracy achieved 90.6% and was therefore able to exceed the given baseline.

In the "Classification" part of the project, the experiments showed that an increase of the input resolution improves the validation accuracy from 79% to 81.3% and - more importantly - reduces overfitting when using the VGG16 base model.

With the introduction of a semantic segmentation model that is able to detect trunks, which is used as a preprocessing step for the classifier, the validation accuracy increased to 85.6%. The final evaluation of this model on the unseen test-set resulted in 90% accuracy with the VGG16 base model.

7 Future Work

Further research along the line of this thesis would be of great benefit to forest management. The additional research could improve the identification of sick or infested trees and would therefore limit the spread of spruce bark beetles.

Limitations of the presented work are of computational nature. Also, a larger amount of training data would result in a more stable training process and provide models that are more general. Having more training data enables the use of more complex networks or the use of fine-tuning in the "Classification" part of the project.

A higher quality of training data would probably also result in better model performance. If, for example, the masks that are used in the "Localization" part were more fine-grained and precise, the model would be able to improve its learning vastly.

A bigger image input size could improve generalization and deliver better results, but would also make the task much more complex and more resources would be needed to train such a model. Especially in the "Classification" part a detailed input image could be beneficial due to the fact that the signs of a bark beetle infestation are usually small. To enable the "Classification" model to improve finding such areas, also the use of semantic image segmentation would be a valid solution, since the model receives more information from the human "expert" about the areas that need to be identified. This, however, requires a lot of work, since there must be masks created that are precise and do not mislead the model.

Experiments with different pre-trained models or different network architectures could be beneficial to enhance the performance of the models.

Furthermore, visualization techniques could be used to gain more insight into the problem domain and to better understand what the models actually

learned about the task with the given data. There could also be a visualization within the mobile application that shows the user those spots that the model assumed to be signs of bark beetle infestation.
Since the mobile application developed during the course of this project only provides basic functionality, it would be desirable to enhance the app functionalities and actually try out the whole system in a field test, where the best model is used and the "Trunk Detection" is plugged in as a pre-processing step before the classification.
In the long run it might also be better to perform the inference part of the system directly on the drone and only send the results to the mobile device.

List of Figures

List of Tables

References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems - Converter Python API guide. Software available from tensorflow.org.

[Badrinarayanan et al., 2017] Badrinarayanan, V., Kendall, A., and Cipolla, R. (2017). Segnet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(12):2481–2495.

[Chen et al., 2018] Chen, L., Zhu, Y., Papandreou, G., Schroff, F., and Adam, H. (2018). Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VII*, pages 833–851.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Li, F. (2009). Imagenet: A large-scale hierarchical Image Database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pages 248–255.

[He et al., 2014] He, K., Zhang, X., Ren, S., and Sun, J. (2014). Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition.

In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part III*, pages 346–361.

[Hoch and Perny, 2019] Hoch, G. and Perny, B. (2019). Die anhaltende Borkenkäfer-Kalamität in Österreich. *BFW-Praxisinformation*, 49:18–21.

[Huang et al., 2017] Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2017). Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269.

[Humer, 2019] Humer, C. (2019). Semantic Image Segmentation and Transfer Learning. Seminar Report, Institute of Computational Perception, JKU Linz, Austria, 2019.

[Humer, 2020] Humer, C. (2020). Detection of sick Spruce Trees using Semantic Segmentation. Project Report, Institute of Computational Perception, JKU Linz, Austria, 2020.

[Jakoby et al., 2015] Jakoby, O., Wermelinger, B., Stadelmann, G., and Lischke, H. (2015). Borkenkäfer im Klimawandel – Modellierung des künftigen Befallsrisikos durch den Buchdrucker (Ips Typographus). *Eidg. Forschungsanstalt WSL, Birmensdorf, Switzerland, 45 S.*

[Jégou et al., 2017] Jégou, S., Drozdzal, M., Vázquez, D., Romero, A., and Bengio, Y. (2017). The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 1175–1183.

[Jiang et al., 2019] Jiang, S., Yao, W., and Heurich, M. (2019). Dead Wood Detection based on Semantic Segmentation of VHR Aerial CIR Imagery

using optimized FCN-Densenet. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W16:127–133.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[Landwirtschaftskammer, 2011] Landwirtschaftskammer (2011). *Merkblatt Borkenkäfer*. Landwirtschaftskammer Österreich. `https://ooe.lko.at/media.php?filename=download%3D%2F2016.01.27%2F145389236853413.pdf&rn=Merkblatt_Borkenk%E4fer_-_Version_O%D6_-_21.1.2016.pdf`.

[Long et al., 2015] Long, J., Shelhamer, E., and Darrell, T. (2015). Fully Convolutional Networks for Semantic Segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 3431–3440.

[Noh et al., 2015] Noh, H., Hong, S., and Han, B. (2015). Learning Deconvolution Network for Semantic Segmentation. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1520–1528.

[Ronneberger et al., 2015] Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, pages 234–241.

[Safonova et al., 2019] Safonova, A., Tabik, S., Alcaraz-Segura, D., Rubtsov, A., Maglinets, Y., and Herrera, F. (2019). Detection of Fir Trees (*Abies sibirica*) Damaged by the Bark Beetle in Unmanned Aerial Vehicle Images with Deep Learning. *Remote Sensing*, 11(6):643.

[Sandler et al., 2018] Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. (2018). Mobilenetv2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4510–4520.

[Simonyan and Zisserman, 2015] Simonyan, K. and Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[Sultana et al., 2018] Sultana, F., Sufian, A., and Dutta, P. (2018). Advancements in Image Classification using Convolutional Neural Network. *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, pages 122–129.

[Szegedy et al., 2016] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826.

[Yosinski et al., 2014] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How Transferable are Features in Deep Neural Networks? In *Advances in Neural Information Processing Systems 27: Annual Confer-*

ence on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada, pages 3320–3328.

[Zeiler, 2012] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. *CoRR*, abs/1212.5701.

[Zeiler and Fergus, 2014] Zeiler, M. D. and Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*, pages 818–833.

[Zhao et al., 2017] Zhao, H., Shi, J., Qi, X., Wang, X., and Jia, J. (2017). Pyramid Scene Parsing Network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6230–6239.