



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME
UND DATA MINING



Bachelor Thesis
in Computer Science

Improving Seedling Detection by Predicting Height Information

Alexander Dobra

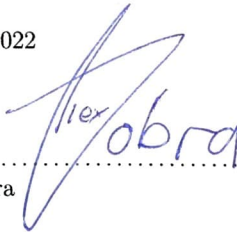
Aufgabensteller: Prof. Dr. Matthias Schubert
Betreuer: Maximilian Bernhard
Abgabedatum: 24.01.2022

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

This paper was not previously presented to another examination board and has not been published.

Munich, 24.01.2022

A handwritten signature in blue ink, appearing to read 'Alex Dobra', is written over a horizontal dotted line.

Alexander Dobra

Abstract

As a result of oil and gas explorations seismic lines threaten the environment in the boreal forest of Alberta, Canada. In order to monitor tree regeneration in disturbed forest regions for more sustainable management cost efficient yet effective ways of field surveys are studied. Imagery obtained through drones are a potential solution as they produce large quantities of data, which have proven to be reliable when analysed with artificial intelligence. As technology advances a large decrease in cost of field surveys and increase in reliability can be expected. We explored the possibility of increasing seedling detection performance with the addition of height information to drone imagery. For this research three convolutional neural networks (CNNs) with different architectures were used. For the first model using only drone imagery to detect seedlings a faster region-CNN (Faster R-CNN) was used. For the second model for the estimation of height information from drone imagery a modified Feature Pyramid Network (FPN) was used. For the third model using both drone imagery and height information to detect seedlings a modified Faster R-CNN called a Pre-Region Proposal Network (Pre-RPN) was used. We evaluated the effects of real and more importantly estimated height information on seedling detection performance. Unfortunately, one of the literatures on which the Pre-RPN and therefore most important model was based on was disproven during our research. Rather than the addition of height information it was flawed code that lead to their assumed improvement in detection performance. For adequate results rectified code was used. Additionally, the model for monocular depth estimation had good predictions for the trainset but not for the validation or the testset as semantic similarities between the images and ground truth depth seemed to be scarce. Ground truth depth generally seemed to be of low quality in our dataset. Our results indicate that the addition of either ground truth height information or estimated height information, obtained through our monocular depth estimation model, to drone imagery does not lead to an increased performance when using a Pre-RPN model for detecting conifer seedlings along recovering seismic lines.

Contents

Introduction	iv
1 Related work	1
1.1 Object Detection	1
1.2 Object Detection with RGB-Depth Information	1
1.3 Monocular Depth Estimation	2
2 Methodology	3
2.1 Object Detection	3
2.1.1 Vanilla Model	3
2.1.2 Pre-Region Proposal Network	3
2.2 Monocular Depth Estimation Network	4
3 Experimental results	8
3.1 Setup	8
3.1.1 Settings	8
3.1.2 Dataset	9
3.1.3 Code	11
3.2 Experiments	11
3.2.1 Monocular Depth Estimation	12
3.2.2 Fake Channel instead of Depth	14
3.2.3 Two Fake Channels	15
3.2.4 Number of Classes	16
3.2.5 Retesting with Modified Models	17
3.2.6 Inspecting Labels	19
3.2.7 Visualising Bounding Boxes	20
3.2.8 Package	22
3.2.9 Object Detection with Monocular Depth Estimation	23
4 Conclusion	25
A Appendix	26
A.1 Model Architecture and Weights	26

<i>CONTENTS</i>	iii
A.1.1 MDE Network	26
A.1.2 Pre-RPN	31
A.2 Object Detection Backbone	37

Introduction

The oil and gas industry has grown significantly throughout the boreal and arctic ecosystems of North America. A major feature of the ecological footprint of oil and gas exploration is seismic lines: narrow corridors used to transport and deploy geophysical survey equipment. They are 5-10 m wide and often tens of kilometres long [11]. Lee and Boutin [13] noted 2006 that in northeastern Alberta, Canada, the mean density of conventional seismic lines was estimated to be $1.5 \frac{km}{km^2}$, and in some regions even as high as $10 \frac{km}{km^2}$. They also noted that after 35 year, only 8.2% of seismic lines across all forest types in Canada’s western Boreal Plains had recovered to greater than 50% cover of woody vegetation. Over time the Caribou population has been negatively affected by the resulting fragmentation of forests as they allow for easier access for their predators [1]. Now the Canadian government demand that responsible companies reforest the affected forest areas [17]. Two of the criteria by which their advancement in reforestation is measured by is the size and number of seedlings along seismic lines [11]. Use of drones and subsequent object detection on the recorded imagery could lead to significant reductions in cost and time of monitoring landscape management [11].

The detection of seedlings is challenging due to their smaller profile and high resemblance to other vegetation from a top down view [11]. Use of CNN-based object detection methods on drone imagery for automatically detecting conifer seedlings along recovering seismic lines was deemed a promising alternative to traditional monitoring programs that rely on costly field surveys that involve identifying and counting seedlings on the ground [8]. Additionally adding height information to drone imagery supposedly leads to an increased performance in detection when analysed using a Pre-RPN: a novel Faster R-CNN with ResNet-FPN backbone in which height is added in-between the FPN and Region Proposal Network (RPN) [11].

As far as we know, the use of estimated height information to improve object detection on seedlings has not yet been attempted. In this paper we test if the addition of height information received through monocular depth estimation with a modified FPN still leads to a meaningful improvement in seedling detection utilising a Pre-RPN compared to applying a conventional CNN-based object detection method for processing of only drone imagery.

Chapter 1

Related work

This Thesis is predominantly based on three literature. In the following sections their contribution will be clarified. All of them are related to each other as they all use data from the Boreal Forest of Alberta, Canada. Additionally, two of them focus on conifer seedling detection. Other literature was used but most of them support one of the three main literature.

1.1 Object Detection

In thesis [8] three different CNN architectures were used to analyse drone imagery of seismic lines containing conifer seedlings. Effects of training-set size, season, seedling size and spatial resolution on the detection performance was evaluated. The evaluation metric used was MAP@0.5IoU: The mean average precision (MAP) at an intersection over union threshold of 0.5. It concluded that detection of conifer seedlings in regenerating sites through use of artificial intelligence on drone imagery achieves good results. They found that two-stage networks perform better than single shot networks, with the ResNet-101 Faster R-CNN model pretrained on the COCO dataset performing best with a mean average precision of 81%. The ResNet-50 Faster R-CNN model had a mean average precision of 66%. By using a pretrained network, the size of the training dataset can be reduced to a couple hundred seedlings without any significant loss of accuracy (cit. [8]). Combining data from different seasons yielded the best results. Best performance for winter images was a MAP of 65%.

1.2 Object Detection with RGB-Depth Information

In thesis [11] ways to effectively integrate elevation data to improve the performance of the FPN-Faster R-CNN object detection algorithm have been researched. The direct addition of height information as a fourth channel showed no improvement, but integration after the backbone network and before the RPN supposedly led to considerable improvements, even on longer training regimes.

1.3 Monocular Depth Estimation

The thesis [5] investigated the performance of recently developed methods for monocular depth estimation on aerial images in the remote sensing application case. The capabilities of a modified FPN and generative adversarial networks were tested. It aimed to predict depth maps of at least reasonable quality, given an aerial image as input. The prediction of crisp boundaries in depth maps is also a topic that has been questioned. Furthermore, the effects of various loss functions, normalisation and interpolation were explored. A local MinMax normalisation on individual samples was deemed best. Additionally, Nearest-neighbour interpolation was used on data to deal with missing depth values. A composite loss function consisting of tree loss functions, which will be explained in more detail later, had the best performance. A modified FPN was chosen for monocular depth estimation as it had proven to give good predictions when testing it on the NYU Depth V2 Dataset with the gradient loss as a loss function [19]. The gradient of depth maps was obtained by a Sobel filter [19].

Chapter 2

Methodology

For this paper we defined three models: a baseline Faster R-CNN model (Vanilla), the Pre-RPN and the Monocular Depth Estimation Network (MDE Network).

2.1 Object Detection

For Object detection we require two models: The Vanilla model and the Pre-RPN. The Vanilla model did not receive any height information and was used to compare results with the Pre-RPN models which used real or estimated height information. Both models predict bounding boxes and their corresponding scores for conifer seedlings. We set two classes: one for background and one for seedlings. Models, training process and evaluation method are identical to those in [11], since employing the same process increases the chances of reproducing results similar to theirs, which makes cross-checking easier.

2.1.1 Vanilla Model

For this model we utilised the implementation of the Faster R-CNN model with a ResNet50-FPN backbone in the TorchVision [16] python library as the same model was chosen in [11] for its reduced number of parameters, allowing for quicker testing of models and faster hyperparameter tuning. It comprises of four main components: The Backbone, the Region Proposal Network (RPN), Region of Interest Pooling (RoI-Pooling) and the Region of Interest Heads (RoI-Heads) [11]. The Backbone consists of a FPN which utilises a ResNet-50 for feature extraction. Final predictions will be given by the Box Predictor, a fifth and final component.

2.1.2 Pre-Region Proposal Network

The Pre-RPN is created by modifying the Vanilla model. As mentioned in the introduction it is a Faster R-CNN in which height data is added in-between the FPN and RPN. The FPN backbone outputs five feature maps of dimensions 256x200x200, 256x100x100, 256x50x50,

256x25x25 and 256x13x13. This can be seen in more detail in Appendix A.2 or the original work of FPNs for object detection: [15]. Height information was concatenated to each of the five feature map outputs of the FPN resulting in five feature map dimensions of 257x200x200, 257x100x100, 257x50x50, 257x25x25 and 257x13x13. Before concatenation the AdaptiveAvgPool2d function from PyTorch was used to reshape height information to fit individual feature sizes. Additionally, weights in both the RPN and ROI Heads required extension because the new feature maps were forwarded to the RPN and ROI Heads. In the RPN, the weights of the first convolutional layer were extended from 256x256x3x3 to 256x257x3x3. In the ROI Heads, the weights of the first fully connected layer were extended from 1024x12544 to 1024x12593. New weights were initialised with Glorot initialisation [9]. Weights and convolutional layers of the Pre-RPN can be seen in detail in Appendix A.1.2.

Norm Clipping

L2 norm was clipped at a value of 1 in both Vanilla and Pre-RPN. Same was done in [11] as occasional instances of exploding gradients were encountered.

Evaluation Metric

Model performance was evaluated with the Mean Average Precision (MAP) at an IoU of 0.5, calculated with the Pycocotools library. The library approximates the MAP by calculating the precision values for 101 different recall scores [11] and is detailed more clearly in [4].

Transfer learning

The TorchVision library allows us to use fine-tuning models for transfer learning, models which were pretrained on the COCO object detection dataset [18]. For this thesis we experimented with and without pretraining as pretrained models often perform better on other data.

2.2 Monocular Depth Estimation Network

For monocular depth estimation an FPN with ResNet101 as backbone was used as described in [19]. A similar network was used in thesis [5] on which most of the topic of depth estimation in this work is based on. It was called an MDE Network and we will name ours the same throughout this thesis. It is visualised in Figure 2.2 and the detailed listing of the layers is available in Appendix A.1.1. It takes an 3x256x256 image input and outputs a single corresponding height prediction of size 1x256x256. An FPN is an effective model for monocular depth estimation because of its ability to extract features and semantics at different scales [15]. Additionally, residual networks with skip-connections are a powerful tool for feature extraction because they are easier to optimise, and can gain

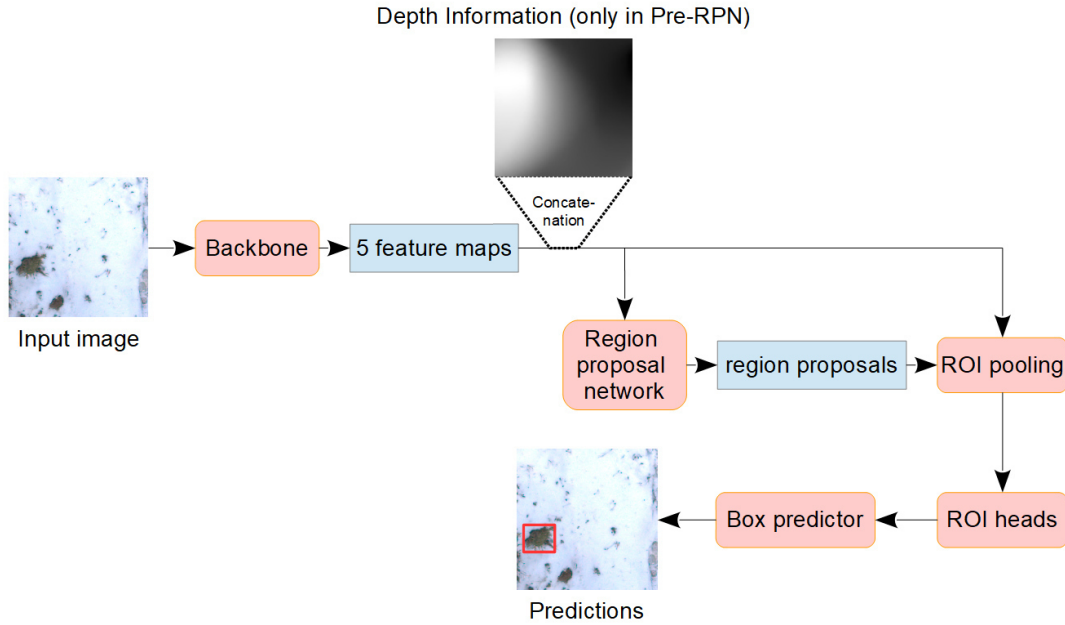


Figure 2.1: Simplified build-up of the object detection models

accuracy from considerably increased depth [10]. The construction of a FPN involves a bottom-up pathway, a top-down pathway and four lateral connections [15]. We can observe that the bottom-up pathway consists of the 4 ResNet101 layer blocks before which we have a 7x7 convolution followed by Batch Normalisation, ReLu and Max Pooling operations. Lateral connections apply 1x1 convolutions on outputs of individual ResNet Blocks. In the top-down pathway we apply up-sampling with bilinear interpolation followed by an additive skip-connection with lateral connections and then a 3x3 convolution. This is done three times. At the end for feature processing to receive the correct output size we utilize a 2x2 transposed convolution followed by two consecutive 3x3 convolutions. ReLU is used as activation function in the last two convolutional layers. Depending on our experiment the ResNet can be either pretrained on the COCO dataset or not. Optionally, we can leave out the topmost block of both the top-down and bottom-up path resulting in a smaller MDE model, which is identical to the one used in thesis [5], with three lateral connections, three ResNet blocks and two additive connections. This would result in considerably less trainable parameters, which might be useful for experimenting as training of features will be quicker even though potentially less deep.

Data normalisation

Ground truth depth maps need to be restricted to values between 0 and 1 without losing information on relative differences between individual values before training the network. For this process normalisation is key. We choose to use local MinMax normalisation (equa-

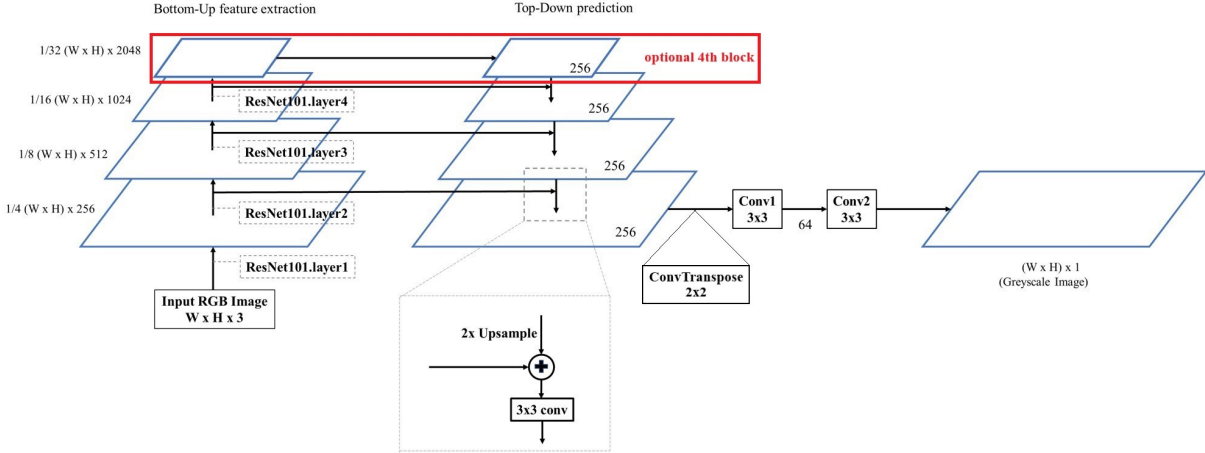


Figure 2.2: MDE Network architecture [19]

tion 2.1) as the same normalisation method was applied in [5] with the reasoning that values might become too small in many samples if we use global normalisation. The minimum and maximum values of individual samples from one another can vary dramatically depending of their respective region [5]. Samples of flat fields might only have very small height values while samples with trees might only have very big height values. As global normalisation takes minimum and maximum values over the entire dataset rather than within individual samples it has the benefit of retaining original depth values [5]. But this comes at the cost of less discriminative enough depth values and as a result training might be compromised [5]. Local normalisation circumvents this problem by normalising based on maximum and minimum of individual samples to retain relative depth differences within a bigger range of $[0, 1]$. Additionally, [5] mentioned that absolute performance metrics towards ground truth values is possible with local normalisation for model evaluation purposes. Input images were not normalised to smaller values as [5] used them full scaled with RGB values [6].

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (2.1)$$

Data interpolation

Data interpolation is used to fill out missing values within data as sparse data can compromise training. In our case we wished to replace missing depth values if any were found. [5] used nearest-neighbour interpolation, while polynomial methods (such as Linear or Cubic) were dismissed because of their smooth output, which contrasts their target of predicting sharp boundaries in depth maps. Additionally, they found that MDE Networks with interpolation consistently performed slightly better than those without. Therefore, we chose to use nearest-neighbour interpolation.

Loss function

Same composite loss function as in [5] was used. This composite loss function proved to perform second best among the individual and composite loss functions they tested in relation to Root Mean Squared Error and Mean Absolute Error:

$$L_{MDE} = \lambda_1 * Loss_{L1_weighted} + \lambda_2 * Loss_{Gradient} + \lambda_3 * Loss_{Normal} \quad (2.2)$$

$$Loss_{L1_weighted} = \alpha * L1_{edge} + (1 - \alpha) * L1_{non-edge} \quad (2.3)$$

$$Loss_{Gradient} = \frac{1}{n} \sum_{i=1}^n \|\nabla y - \nabla \hat{y}\|_1 \quad (2.4)$$

$$Loss_{Normal} = \frac{1}{n} \sum_{i=1}^n \left(1 - \frac{\langle n_i^{\hat{y}}, n_i^y \rangle}{\|n_i^{\hat{y}}\| \|n_i^y\|} \right) \quad (2.5)$$

The composite loss function in 2.2 consists of three parts:

1. Weighted L1 loss (Equation 2.3): This loss tries to weight edge pixels higher than non-edge pixels. The weighting coefficient α was set to 0.6. L1 loss is defined as a mean absolute error: $\frac{1}{n} \sum_{i=1}^n |y - \hat{y}|$.
2. Gradient loss (Equation 2.4): This loss calculates the mean of the L1 norm of the difference between y and \hat{y} , which are gradients of target and prediction depth maps, respectively, calculated with Sobel filter.
3. Surface normal loss (Equation 2.5): This loss computes the mean of the differences between 1 and the cosine distance between two normals of the depth maps represented as surfaces.

A weighting of $\lambda_1 = 0.2$, $\lambda_2 = 1.0$ and $\lambda_3 = 0.1$ was used for the composite loss function to make each loss components impact in the backpropagation process about equal [5]. For the calculation of equation 2.3 we require an edge detection algorithm, since we do not have any ground truth data for edges. We chose Laplacian operator with a threshold of 0.7 as a method of edge detection as it was applied on image data in [5] after it had shown better results in an empirical investigation compared to the Canny method, another edge detection algorithm.

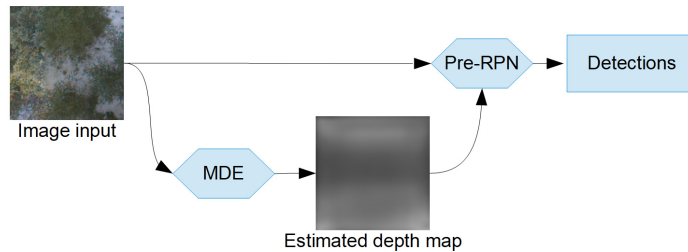


Figure 2.3: Input of Pre-RPN with estimated height information

Chapter 3

Experimental results

3.1 Setup

Before we start with experiments, we require the proper settings and a dataset.

3.1.1 Settings

Pre-RPN and Vanilla

Same training setting as in [11] were used for the Pre-RPN and Vanilla models:

Setting	Value
Learning rate	0.002
Momentum	0.9
Learning rate decay (each epoch)	0.0001
Batch size	5
L2 Norm clipping	1.0

Table 3.1: Object detection models settings

Just like in [11] Stochastic Gradient Descent was chosen as an optimiser.

MDE Network

The training settings for the MDE Network can be viewed in table 3.2. Just like in [5] Adam optimiser was chosen. They used a learning rate decay of 0.95, which was applied once every epoch. We did not use learning rate decay on top of the already existing weight decay since limited testing showed that a learning rate decay of 0.95 every epoch did not improve performance. This might be since the dataset used in [5] was significantly larger by a factor of at least 10, therefore requiring significantly less epochs. Additionally, they used a batch size of 46 and a training time of 48 hours. Our environment had troubles

Setting	Value
Learning rate	0.00001
Weight decay (L2 penalty)	0.00004
Approx. training time	12 hours
Batch size	25
Betas	(0.9, 0.999)
Eps	1E-08

Table 3.2: MDE Network settings

with batch sizes in a similar range therefore requiring a smaller batch size. Longer training times usually performed better, but not by a significant amount. To save time a shorter training regime of 12 hours was enforced.

3.1.2 Dataset

The study area analysed in this work was the boreal forest of north east Alberta, Canada. The dataset used during this study was received preprocessed. It contained 5303 256x256 pixel 3 channel RGB image tiles in TIFF format, 5303 256*256 pixel single channel images containing height data in TIFF format and 538 XML files containing seedling annotations for 538 corresponding images. 21 XML files did not contain any labels for seedlings and were deleted leaving us with 517 XML files. RGB Images were taken from a DJI Mavic Pro Drone over 25m sections of three seismic lines hereby referred to as Lane 460, Lane 464 and Lane 466 (cit. [11]). All images were taken in October 2017 (leaf off) as a height of 30m with a Ground Sampling Distance (GSD) of 0.75cm (cit. [11]). The individual drone images were rectified into a single image for each lane and then split into 256x256 pixel images (cit. [11]). Seedlings that were split across multiple images were considered as two separate seedlings (cit. [11]). As a result of the orientation of Lane 466, Images outside of the recorded area were removed by filtering out tiles containing more than 5% pure white (cit. [11]).

Height data for the lanes was extracted from a photogrammetric Digital Surface Model (DSM) of the larger research area taken by a fixed wing plane also in October 2017 (cit. [11]). For Lane 464 higher resolution data was available. The generation of ground truth bounding boxes was performed manually with the software Labelling [14], with early annotations serving as an initial training to precompute suggestions for further manual annotation (cit. [11]). Many images did not contain any seedlings at all. We refer to these as negative images or negative samples, and vice versa images with seedlings are referred to as positive images or positive samples.

For training and testing of networks 4 different subdatasets were created (see table 3.3). Among the 186 XML files with labels for Lane 464 13 had no corresponding high resolution

data available leaving 173 files for the high resolution height subdataset. All subdatasets for Lane 464 use the same labels to better differentiate results between testing high resolution height data (HD) and lower resolution height data (SD). For Vanilla and Pre-RPN models 5 random subsets each using all data of a given subdataset were created. For MDE models only one random subset for each given subdataset was given, as training them required significantly more time. For each test a train-validation-test split of 0.72:0.08:0.2 was used.

subset	Lane	resolution	+/-	images	annotations
1	460/464/466	SD	0	513	1250
2	460/464/466	SD	all	3865	1250
3	464	SD	0	173	537
4	464	HD	0	173	537

Table 3.3: Summary of the four subdatasets and ground truth seedling detections in each set, (+/-) denotes how many negative samples were used of the original dataset

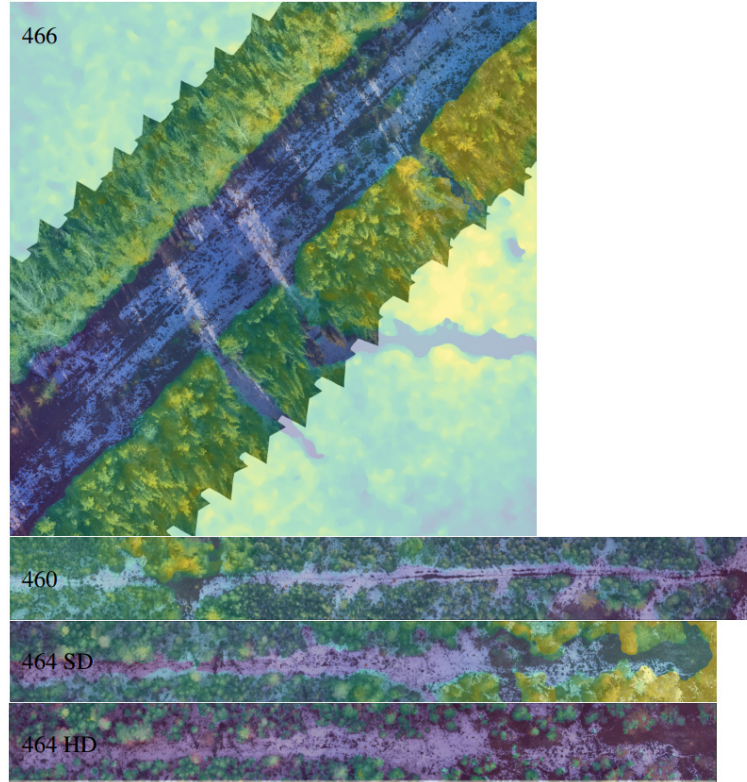


Figure 3.1: The lanes used in this data analysis. All lanes show the RGB recorded image data overlaid with height data from low (purple) to high (yellow). Lane 466 clearly shows the boundaries of the recorded image data, with only height data available beyond the boundaries. (cit. & source [11])

3.1.3 Code

For training and testing of Vanilla models and Pre-RPN models code from [11], which is available on GitHub [12], was used. Reproducing results in [11] without knowledge or usage of the GitHub code proved to be improbable as flawed code resulted in false results. During training of all models, a deep copy of the model was saved as a *bestmodel* when the current epoch had a better validation map (for Vanilla, Pre-RPN models) or validation loss (for MDE models) than every epoch before. Models were tested on the *bestmodel*. This approach was chosen as it proved to achieve consistently better results during limited testing in all models.

3.2 Experiments

As we want to develop a network which takes in drone imagery and estimated height information we require a Pre-RPN that can make use of height information and a MDE Network that can provide reasonable depth estimations. Each Pre-RPN with MDE Network combination is streamlined in a way that the latter part uses estimated depth for both training and testing. This is logical as differences between real and estimated height information could lead to worse results in models which use estimated depth for testing but real depth for training. For training of all models, the same seed was used to reduce the possibility of outliers. Depending on which kind of model was trained a different number of test runs were completed and for object detection four different subexperiments were conducted:

For MDE models:

Every MDE Network was trained only one time per setting as a very long training time is required.

For Pre-RPN and Vanilla models:

Each model was tested 10 times for each setting. The utilised subdataset was randomised into 5 different subsets. Each of the resulting subsets was used for training the current setting once with and once without pretrained backbone. All models had 5 trainable backbone layers, since models with trainable backbone layers consistently had better results during limited testing. Best model was chosen as representative for each setting in tables.

Subexperiment 1:

Training on fewer epochs: 35 and 100 epochs, with and without pretraining respectively. Subset 1 was used.

Subexperiment 2:

Training on more epochs: 150 epochs with and without pretraining. Subset 1 was used.

Subexperiment 3:

Training on subset 2 in which all images, including those without labels, are used as the effect of negative samples will be investigated.

Subexperiment 4:

Training on different height data resolutions: pretrained Pre-RPN models were trained over 35 epochs on either HD data or SD data of Lane 464. Subsets 3 and 4 were used.

3.2.1 Monocular Depth Estimation

Before we can use estimated depth data, we require a well-trained MDE model. We tested 3-layer and 4-layer MDE models, each with and without pretraining as can be seen in table 3.4. The best model was saved for later usage in Pre-RPN experiments.

Layers	Pretraining	Subset	Resolution	Loss
3	False	1	SD	0.000756
3	True	1	SD	0.000793
4	False	1	SD	0.000777
4	True	1	SD	0.000809

Table 3.4

Overall MDE Networks with 3 Layers performed better than 4-Layered MDE Networks. This might be because of less trainable parameters. Additionally, MDE Networks without pretraining had an advantage, indicating that pretraining is not suitable for our dataset.

HD and SD data of Lane 464

We trained 3-layer MDE Networks without pretraining on other datasets of Lane 464 resulting in a total of 3 saved MDE Networks which were used in later experiments. Each model was named in table for easier reference.

Subset	Resolution	Loss	MDE Model
1	SD	0.000756	MDE_SD_1
3	SD	0.002217	MDE_SD_3
4	HD	0.002627	MDE_HD_4

Table 3.5

We are able to observe that MDE_SD_3 and MDE_HD_4 have a worse loss than MDE_SD_1. This indicates that use of more data potentially leads to better results as subset 3 and 4 are smaller than subset 1. Additionally, MDE_HD_4 has a worse loss than MDE_SD_3, possibly

the cause of significantly more detailed HD data in subset 4. We only used subsets that had positive samples for MDE Networks as we believed them to be of grater importance for seedling detection, because possible similarities in semantics between ground truth height and seedlings might give a higher chance of better performance.

Visualising estimations

Since we only have loss as a measure of performance, we cannot be certain that height estimations are indeed good. We visualised depth estimations of MDE Network MDE_SD_1 for the trainset in Figure 3.2, validationset in Figure 3.3 and testset in Figure 3.4.

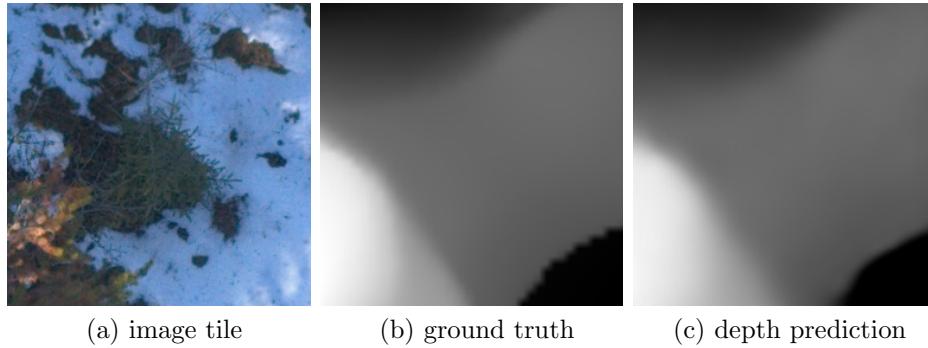


Figure 3.2: drone image, ground truth and prediction of a sample in the train set

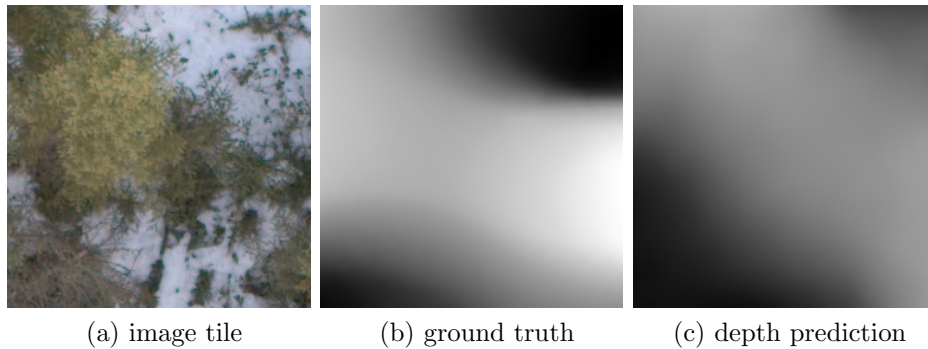


Figure 3.3: drone image, ground truth and prediction of a sample in the validation set

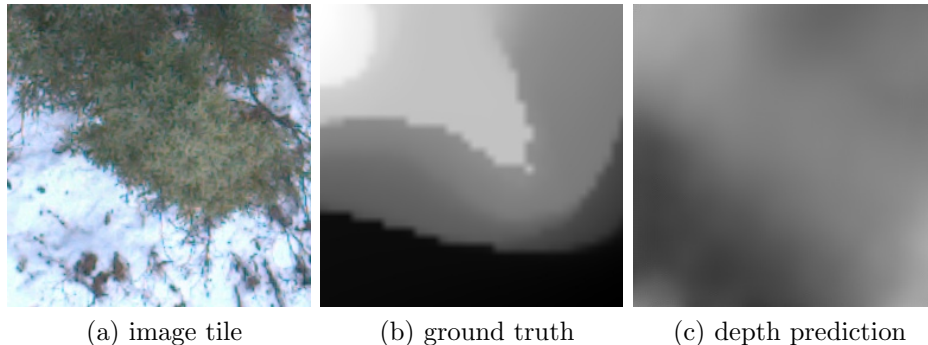


Figure 3.4: drone image, ground truth and prediction of a sample in the test set

Predictions of samples in the trainset are of good quality, but predictions for validationset and testset unfortunately are not. No overfitting should have occurred since the bestmodel is chosen by the best validation performance during testing. We can easily see that image tiles and their respective ground truth do not seem to have much semantic information in common, when analysed through human eyes. This does not mean that no semantic information is detectable after feature extraction through CNNs, but it could explain the big discrepancy between ground truth and depth prediction in the validation and testset.

3.2.2 Fake Channel instead of Depth

In this experiment fake channels were used to certify that results in [11] are legitimate. We replaced depth images with tensors of zeros in order to effectively lose any semantic information on depth. This can be done easily using `torch.zeros(1, 256, 256)` in PyTorch to replace all batch elements containing depth information. Both training and testing are done on real aerial images and fake depth. Results in [11] are included in table 3.6 for comparison.

Results in table 3.6 indicate that Pre-RPN Models perform better even if no semantic information is contained in depth images. This implies that depth images are not the factor that leads to improvement of the Pre-RPN models, but rather the setup of the model itself towards aerial images. There are several possibilities for this result, including:

- At least one mistake in code:
 - Map calculated wrong
 - Model setup wrong
 - Data loading wrong
- Code is correct, but there was a mistake in interpretation of results
- Code is correct, but prior best trained models were preloaded for testing

Subexp	Model	Pretraining	Subset	Epochs	ZeroDepth	[11]
1	Pre-RPN	False	1	100	0.683	0.612
1	Vanilla	False	1	100	0.335	0.289
1	Pre-RPN	True	1	35	0.739	0.704
1	Vanilla	True	1	35	0.592	0.579
2	Pre-RPN	False	1	150	0.652	0.652
2	Vanilla	False	1	150	0.339	0.324
2	Pre-RPN	True	1	150	0.742	0.718
2	Vanilla	True	1	150	0.566	0.605
3	Pre-RPN	False	2	35	0.608	0.537
3	Vanilla	True	2	35	0.538	0.475

Table 3.6: Experiments with single fake channel

3.2.3 Two Fake Channels

In this experiment we tried to achieve a low MAP deliberately by using fake channels for both aerial images and depth images. By doing so we can reject the possibility of saved models being used in code used by [11] to achieve a high MAP during testing. More importantly, we can also test the effect that real depth images have on the Pre-RPN models, if they are the only semantic information available.

Subexp	Model	Pretraining	Subset	Epochs	ZeroDepth	RealDepth	[11]
1	Pre-RPN	False	1	100	0.020	0.013	0.612
1	Vanilla	False	1	100	0.006	0.010	0.289
1	Pre-RPN	True	1	35	0.017	0.012	0.704
1	Vanilla	True	1	35	0.012	0.012	0.579
2	Pre-RPN	False	1	150	0.013	0.019	0.652
2	Vanilla	False	1	150	0.005	0.014	0.324
2	Pre-RPN	True	1	150	0.014	0.019	0.718
2	Vanilla	True	1	150	0.004	0.013	0.605
3	Pre-RPN	True	2	35	0.000	0.000	0.537
3	Vanilla	True	2	35	0.000	0.000	0.475

Table 3.7

The results in all settings are extremely low. We can conclude that no pre-saved models were loaded. Additionally, using real height information does not seem to have any impact on the results. Overall, Pre-RPNs are insignificantly better independent from height information.

3.2.4 Number of Classes

In all models we want to detect one class, which is seedlings. Using fine-tuning models from the PyTorch library we would require the initialisation of two classes, one for background and one for seedlings. The initialisation of number of classes was included in all Vanilla models, but none of the Pre-RPN models of [11]. This might have been done deliberately to increase MAP of Pre-RPN models or by mistake. Class initialisation is performed by changing the number of out features of individual layers in the box predictor. In [11]’s GitHub code [12] following initialisation of box_predictor was applied for Vanilla models:

```
from torch.nn import Linear
```

```
NUMCLASSES = 2
```

```
model = model() #PSEUDOCODE: make Vanilla model with 91 classes
```

```
input_size = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor.cls_score = Linear(input_size, NUMCLASSES)
```

In the Pre-RPN model the initialisation was left out completely. As a result, we would have following initialisation of the layers within the box predictor in both models:

For Vanilla model:

```
(box_predictor): FastRCNNPredictor(
  (cls_score): Linear(in_features=1024, out_features=2, bias=True)
  (bbox_pred): Linear(in_features=1024, out_features=364, bias=True)
)
```

For Pre-RPN model:

```
(box_predictor): FastRCNNPredictor(
  (cls_score): Linear(in_features=1024, out_features=91, bias=True)
  (bbox_pred): Linear(in_features=1024, out_features=364, bias=True)
)
```

We can observe that there is a different number of out features between both cls_score layers. Additionally, there is a mismatch between the number of out features of the cls_score and bbox_pred layer for the Vanilla model. We say mismatch, as each cls_score out feature requires 4 bbox_pred out features, as each bounding box prediction has four predicted coordinates. This mismatch could potentially lead to misleading results when it is used for validating both models against each other. Typically, initialisation is done as follows:

```
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
```

```
NUMCLASSES = 2
```

```
model = model() #PSEUDOCODE: make a model with 91 classes
```

```
input_size = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(input_size, NUMCLASSES)
```

This initialisation in which both out features match is standard according to PyTorch source code documentation for all Faster R-CNN finetuning-models [7]. This would lead to the following initialisation of the layers within the box predictor in both models:

For Vanilla & Pre-RPN model:

```
(box_predictor): FastRCNNPredictor(
  (cls_score): Linear(in_features=1024, out_features=2, bias=True)
  (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
)
```

At first the possibility was considered that this initialisation was not used by [11] during testing as it would be incorrect, but a following inspection of the commit history and documentation in GitHub [12] indicated that this assumption is false.

3.2.5 Retesting with Modified Models

We tested all object detection models with modifications to the box predictor by changing the number of out features of its individual layers and comparing results between the modifications. For two tests we applied the same initialisation as [11] as this would make crosschecking of results easier. For the last test we applied standard initialisation.

out features for cls_score=2 and for bbox_pred=364:

Subexp	Model	Pretraining	Subset	Epochs	ZeroDepth	RealDepth	[11]
1	Pre-RPN	False	1	100	0.400	0.470	0.612
1	Vanilla	False	1	100	0.427	0.439	0.289
1	Pre-RPN	True	1	35	0.643	0.570	0.704
1	Vanilla	True	1	35	0.585	0.614	0.579
2	Pre-RPN	False	1	150	0.465	0.474	0.652
2	Vanilla	False	1	150	0.447	0.453	0.324
2	Pre-RPN	True	1	150	0.594	0.571	0.718
2	Vanilla	True	1	150	0.601	0.583	0.605
3	Pre-RPN	True	2	35	0.575	0.535	0.537
3	Vanilla	True	2	35	0.515	0.548	0.475
4	Pre-RPN	True	3	35	0.636	0.602	—
4	Pre-RPN	True	4	35	0.627	0.614	—

Table 3.8

out features for cls_score=91 and for bbox_pred=364:

This is the presetting for Faster-RCNN fine-tuning models in PyTorch.

Subexp	Model	Pretraining	Subset	Epochs	ZeroDepth	RealDepth	[11]
1	Pre-RPN	False	1	100	0.665	0.647	0.612
1	Vanilla	False	1	100	0.659	0.676	0.289
1	Pre-RPN	True	1	35	0.733	0.747	0.704
1	Vanilla	True	1	35	0.719	0.717	0.579
2	Pre-RPN	False	1	150	0.670	0.651	0.652
2	Vanilla	False	1	150	0.648	0.657	0.324
2	Pre-RPN	True	1	150	0.730	0.725	0.718
2	Vanilla	True	1	150	0.724	0.720	0.605
3	Pre-RPN	True	2	35	0.589	0.598	0.537
3	Vanilla	True	2	35	0.589	0.606	0.475
4	Pre-RPN	True	3	35	0.777	0.747	—
4	Pre-RPN	True	4	35	0.776	0.774	—

Table 3.9

out features for cls_score=2 and for bbox_pred=8:

Subexp	Model	Pretraining	Subset	Epochs	ZeroDepth	RealDepth	[11]
1	Pre-RPN	False	1	100	0.652	0.679	0.612
1	Vanilla	False	1	100	0.670	0.698	0.289
1	Pre-RPN	True	1	35	0.730	0.731	0.704
1	Vanilla	True	1	35	0.723	0.724	0.579
2	Pre-RPN	False	1	150	0.672	0.660	0.652
2	Vanilla	False	1	150	0.666	0.634	0.324
2	Pre-RPN	True	1	150	0.705	0.725	0.718
2	Vanilla	True	1	150	0.722	0.731	0.605
3	Pre-RPN	True	2	35	0.614	0.614	0.537
3	Vanilla	True	2	35	0.621	0.610	0.475
4	Pre-RPN	True	3	35	0.751	0.792	—
4	Pre-RPN	True	4	35	0.770	0.791	—

Table 3.10

We can discern that models with standard initialisation for two and 91 classes performed significantly better than the models with the first initialisation independent from height information. This indicates that false initialisation did indeed lead to misleading results in [11].

3.2.6 Inspecting Labels

Labels of first batch of training set have been viewed at the beginning of 1st and 10th epoch on three different settings of the box predictor. Furthermore, labels of validation set after training in 1st and 10th epoch have been probed. We used a pretrained Pre-RPN. Arrays that only contained ones were shortened by counting the number of ones.

Setting 1: out features for cls_score=2 and for bbox_pred=8

Labels for first batch in trainset:

```
Epoch 1: [[100 ones], [100 ones], [100 ones], [100 ones], [100 ones]]
Epoch 10: [[7 ones], [5 ones], [5 ones], [9 ones], [4 ones]]
```

Labels for first batch in validationset:

```
Epoch 1: [[15 ones], [100 ones], [61 ones], [100 ones], [81 ones]]
Epoch 10: [[1 one], [20 ones], [6 ones], [17 ones], [14 ones]]
```

Setting 2: out features for cls_score=2 and for bbox_pred=364

Labels for first batch in trainset:

```
Epoch 1: [[4 ones], [6 ones], [6 ones], [3 ones], [5 ones]]
Epoch 10: [[5 ones], [2 ones], [2 ones], [3 ones], [2 ones]]
```

Labels for first batch in validationset:

```
Epoch 1: [[5 ones], [8 ones], [7 ones], [5 ones], [7 ones]]
EPOCH 10: [[4 ones], [5 ones], [5 ones], [2 ones], [2 ones]]
```

Setting 3: out features for cls_score=91 and for bbox_pred=364

Labels for first batch in trainset:

```
Epoch 1: [[],
           [17, 65],
           [10, 1, 38, 88, 10, 38, 35, 16, 38],
           [17, 1, 65],
           []]
EPOCH 10: [[9 ones], [4 ones], [15 ones], [3 ones], [10 ones]]
```

Labels for first batch in validationset:

```
Epoch 1: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 64, 1],
           [31 ones], [13 ones], [20 ones], [30 ones]]
Epoch 10: [[3 ones], [2 ones], [2 ones], [2 ones], [5 ones]]
```

We can observe that in setting 3 we have many labels that are not seedlings in first epochs. This is to be expected since the box predictor was initialised for 91 classes rather than 2. But after 10 epochs label predictions in both the train and validation set were only seedlings. This indicates that features of other objects, brought by pretraining, are overwritten as the model trains with a dataset that only contains seedlings. Through label-checking it was determined that all predicted labels in the testdataset were ones and therefore seedlings after training for 35 epochs. Between the models with initialisation for two classes setting 2 had significantly less label predictions in first epochs while setting 1 had more. Additionally, setting 1 had 100 label predictions for every image in the first batch of the trainset in the first epoch. This suggests that pretraining might indeed be more prevalent in setting 2, even though this setting inevitable performed worse overall as can be seen in prior experiment.

3.2.7 Visualising Bounding Boxes

During training of a pretrained Pre-RPN over 35 epochs we visualised images of the validation set with their respective ground truth bounding boxes and bounding box predictions independent of scores and labels. Two samples were taken after first epoch and last epoch to analyse learning behaviour of models over time when applying different initialisation of the box predictor.

out features for `cls_score=2` and for `bbox_pred=8`:

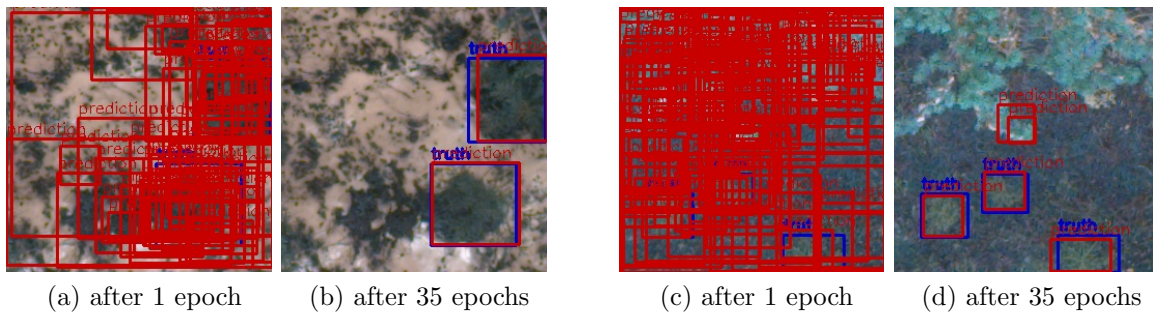


Figure 3.5: ground truth (blue) and predictions (red) for two images of the validation set

out features for cls_score=2 and for bbox_pred=364:

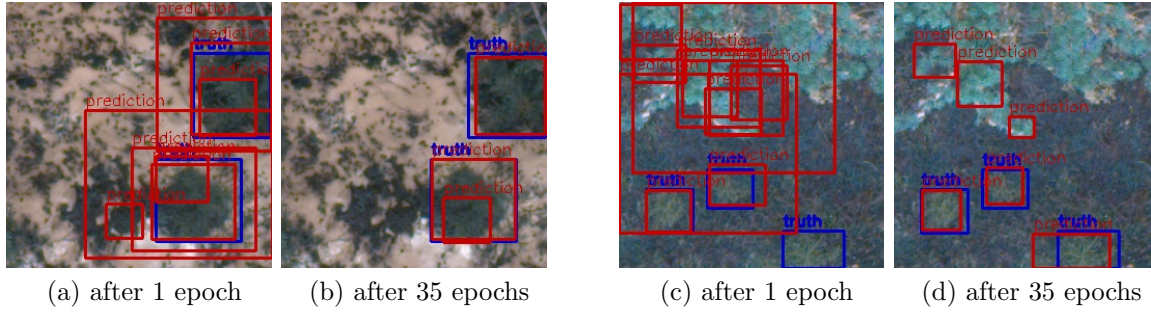


Figure 3.6: ground truth (blue) and predictions (red) for two images of the validation set

out features for cls_score=91 and for bbox_pred=364:

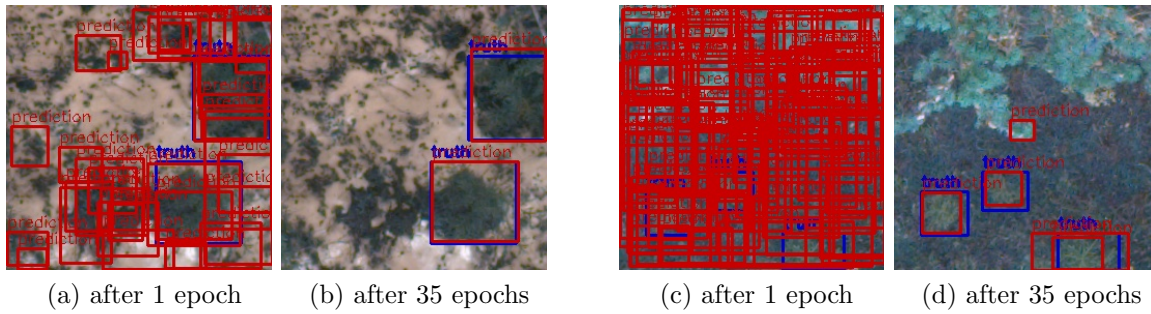


Figure 3.7: ground truth (blue) and predictions (red) for two images of the validation set

We can see that during training of all models bounding box predictions become less over time and more accurate. After the first epoch most bounding box predictions are near regions of ground truth bounding boxes (blue), this is particularly the case for models in Figure 3.5 (a) and Figure 3.6 (a). Contrastive, most bounding box predictions for the second image in Figure 3.6 (c) are on a branch of a nearby tree. Additionally, in Figure 3.7 (a) & (b) we can see that at the beginning other objects (e.g. soil, moss) have bounding boxes around them but none after 35 epochs. All models predicted part of a branch as a seedling in the second image even after 35 epochs: prediction of other vegetation as seedlings, leading to false positive bounding box predictions, was not uncommon. During visualisation we were also able to find one case (Figure 3.8) in which one model predicted a seedling correctly, but the ground truth label was missing, indicating that samples are not accurately labelled: the possibility of human error during labelling of data for supervised learning tasks is not unusual, but in this case problematic since small improvements in map might have a higher chance of being luck rather than a real improvement. Models which predict false positive bounding boxes in the test set correctly could inevitable have

a worse MAP than less accurate models that did not predict unlabelled seedlings in the same train-validation-test split. Furthermore, it is difficult to set bounding boxes around seedling with absolute accuracy. Depending on the randomised subset, changes in map of more than 10% were frequent.

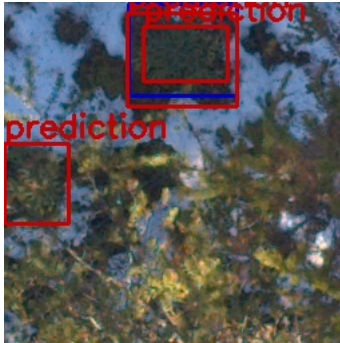


Figure 3.8: ground truth (blue) and predictions (red) for positive sample of the validation set with one missing label

3.2.8 Package

We wanted to know if COCO MAP is calculated wrong in code of [12]. For this purpose we utilised another package to evaluate COCO MAP: the mean_average_precision python package [2]. It allows the calculation of different MAP calculations (e.g. VOC PASCAL mAP, COCO mAP) in a more simplified manner. Results with the package are denoted in table as *PacMap* and those with COCOevaluation tools as in [12] as *CocoMap*. *Cls* and *Bbox*, are notations used for the number of out features in *cls_score* and *bbox_pred* of the box predictor, respectively.

Cls	Bbox	Model	Pretraining	Subset	Epochs	PacMap	CocoMap	[11]
2	2	Pre-RPN	True	1	35	0.733	0.731	—
2	2	Vanilla	True	1	35	0.724	0.724	—
2	91	Pre-RPN	True	1	35	0.625	0.600	—
2	91	Vanilla	True	1	35	0.617	0.629	0.579
91	91	Pre-RPN	True	1	35	0.747	0.745	0.704
91	91	Vanilla	True	1	35	0.718	0.712	—

Table 3.11: Results of two MAP evaluation packages with different box predictor settings.

We can conclude that MAP calculation was indeed correct in [12] as MAPs are extremely similar.

3.2.9 Object Detection with Monocular Depth Estimation

Finally, we train and test Pre-RPN models with monocular depth estimation with following MDE Networks: MDE_SD_1, MDE_SD_3 and MDE_HD_4. All models are trained on the same dataset as the corresponding MDE Network since we want to know if lane specific estimated depth information on Lane 464 could lead to improvements. Even though this examination is not useful for the study of detecting seedlings on other lanes it can give us a clue to whether and how much height information in our dataset has value for object detection. For all object detection models the layers of the box predictor were initialised correctly for two classes.

Subexp	MDE Network	Model	Pretraining	Subset	Epochs	Test MAP
1	MDE_SD_1	Pre-RPN	False	1	100	0.666
1	MDE_SD_1	Vanilla	False	1	100	0.674
1	MDE_SD_1	Pre-RPN	True	1	35	0.733
1	MDE_SD_1	Vanilla	True	1	35	0.716
2	MDE_SD_1	Pre-RPN	False	1	150	0.659
2	MDE_SD_1	Vanilla	False	1	150	0.669
2	MDE_SD_1	Pre-RPN	True	1	150	0.718
2	MDE_SD_1	Vanilla	True	1	150	0.725
1	MDE_SD_3	Pre-RPN	False	3	100	0.704
1	MDE_SD_3	Vanilla	False	3	100	0.776
1	MDE_SD_3	Pre-RPN	True	3	35	0.772
1	MDE_SD_3	Vanilla	True	3	35	0.727
2	MDE_SD_3	Pre-RPN	False	3	150	0.750
2	MDE_SD_3	Vanilla	False	3	150	0.731
2	MDE_SD_3	Pre-RPN	True	3	150	0.788
2	MDE_SD_3	Vanilla	True	3	150	0.751
1	MDE_HD_4	Pre-RPN	False	4	100	0.719
1	MDE_HD_4	Vanilla	False	4	100	0.739
1	MDE_HD_4	Pre-RPN	True	4	35	0.783
1	MDE_HD_4	Vanilla	True	4	35	0.762
2	MDE_HD_4	Pre-RPN	False	4	150	0.715
2	MDE_HD_4	Vanilla	False	4	150	0.736
2	MDE_HD_4	Pre-RPN	True	4	150	0.777
2	MDE_HD_4	Vanilla	True	4	150	0.741

Table 3.12: Final results

For subset 1 the best Vanilla model MAP was 0.725 and best Pre-RPN MAP was 0.733. For data on Lane 464 the best Vanilla model MAP was 0.776 and the best Pre-RPN MAP was 0.788. In both cases the Pre-RPN is slightly better, by 0.008 and 0.012, respectively.

Considering the fact that the labelled dataset is not entirely reliable and that Pre-RPN models are often slightly better than Vanilla models independent from height information this increase can not necessarily be considered an improvement achieved through the addition of estimated height information.

Chapter 4

Conclusion

During this study no meaningful improvement in seedling detection with estimated height information was possible. Furthermore, we disproved the possibility of Pre-RPN models being able to improve seedling detection significantly with real height information of our dataset. We were able to identify that wrong initialisation of the pox predictor in both the Pre-RPN and Vanilla models in code [12] led to misleading results in thesis [11]. As to why models with standard initialisation of box predictor for 91 classes also performed well we were not able to identify accurately. The MDE Networks had good predictions for the trainset but not for the validation or the testset as semantic similarities between the images and ground truth depth seemed to be scarce. Additionally, input images were not normalised for the MDE Networks which could have potentially hindered a more stable learning progress. Normalisation usually ensures that all the input and target variables are of order unity, in which case we expect that the network weights should also be of order unity resulting in a more stable learning process than input or target values of dissimilar size [3]. We were not able to test the effect of normalisation of input images in MDE Networks during this thesis. All MDE Networks were trained only on positive samples. Training on a larger dataset and additionally negative samples might lead to better depth predictions, but it is unlikely to improve seedling detection. As technology advances, it should be possible to acquire higher quality depth data cost efficiently, increasing the chances of improving seedling detection with real and possibly even estimated height information.

Appendix A

Appendix

A.1 Model Architecture and Weights

A.1.1 MDE Network

A 4 Layered MDE Network architecture using a ResNet101 backbone is provided below with specification of its layers.

```
MDE(  
  (layer0): Sequential(  
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)  
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (2): ReLU(inplace=True)  
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)  
  )  
  (layer1): Sequential(  
    (0): Sequential(  
      (0): Bottleneck(  
        (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)  
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)  
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (downsample): Sequential(  
          (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)  
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        )  
      )  
    )  
    (1): Bottleneck(  
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)  
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)  
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (relu): ReLU(inplace=True)  
    )  
    (2): Bottleneck(  
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)  
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```



```

        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
    )
)
)
(layer2): Sequential(
  (0): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
)
)
(layer3): Sequential(
  (0): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
)

```

[illegible]

[illegible]

[illegible]

```

        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
    )
)
)
(layer4): Sequential(
  (0): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
)
)
(toplayer): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
(smooth1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(smooth2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(smooth3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(latlayer1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
(latlayer2): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
(latlayer3): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
(upconv1): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(4, 4))
(predict1): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(relu1): ReLU()
(predict2): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(relu2): ReLU()
)

```

A.1.2 Pre-RPN

Pre-RPN Network architecture with specification of its layers and corresponding weights is provided below.

```

FasterRCNNPreRpn(
  (transform): GeneralizedRCNNTransform(

```

```

        Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        Resize(min_size=(800,), max_size=1333, mode='bilinear')
    )
    (backbone): BackboneWithFPN(
      (body): IntermediateLayerGetter(
        (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
        (bn1): FrozenBatchNorm2d(64, eps=1e-05)
        (relu): ReLU(inplace=True)
        (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
        (layer1): Sequential(
          (0): Bottleneck(
            (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (bn1): FrozenBatchNorm2d(64, eps=1e-05)
            (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn2): FrozenBatchNorm2d(64, eps=1e-05)
            (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (bn3): FrozenBatchNorm2d(256, eps=1e-05)
            (relu): ReLU(inplace=True)
            (downsample): Sequential(
              (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): FrozenBatchNorm2d(256, eps=1e-05)
            )
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=1e-05)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=1e-05)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=1e-05)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=1e-05)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=1e-05)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=1e-05)
          (relu): ReLU(inplace=True)
        )
      )
    )
    (layer2): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(128, eps=1e-05)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(128, eps=1e-05)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(512, eps=1e-05)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): FrozenBatchNorm2d(512, eps=1e-05)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(128, eps=1e-05)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(128, eps=1e-05)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(512, eps=1e-05)
        (relu): ReLU(inplace=True)
      )
    )
  )

```

```

(2): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): FrozenBatchNorm2d(128, eps=1e-05)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): FrozenBatchNorm2d(128, eps=1e-05)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): FrozenBatchNorm2d(512, eps=1e-05)
  (relu): ReLU(inplace=True)
)
(3): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): FrozenBatchNorm2d(128, eps=1e-05)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): FrozenBatchNorm2d(128, eps=1e-05)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): FrozenBatchNorm2d(512, eps=1e-05)
  (relu): ReLU(inplace=True)
)
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256, eps=1e-05)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256, eps=1e-05)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024, eps=1e-05)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): FrozenBatchNorm2d(1024, eps=1e-05)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256, eps=1e-05)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256, eps=1e-05)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024, eps=1e-05)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256, eps=1e-05)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256, eps=1e-05)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024, eps=1e-05)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256, eps=1e-05)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256, eps=1e-05)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024, eps=1e-05)
    (relu): ReLU(inplace=True)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256, eps=1e-05)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256, eps=1e-05)
  )
)

```

```

        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=1e-05)
        (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=1e-05)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=1e-05)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=1e-05)
        (relu): ReLU(inplace=True)
    )
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(512, eps=1e-05)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(512, eps=1e-05)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(2048, eps=1e-05)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): FrozenBatchNorm2d(2048, eps=1e-05)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(512, eps=1e-05)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(512, eps=1e-05)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(2048, eps=1e-05)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(512, eps=1e-05)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(512, eps=1e-05)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(2048, eps=1e-05)
    (relu): ReLU(inplace=True)
  )
)
)
(fpn): FeaturePyramidNetwork(
  (inner_blocks): ModuleList(
    (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (2): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
    (3): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
  )
  (layer_blocks): ModuleList(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (extra_blocks): LastLevelMaxPool()
)
)
(rpnp): RegionProposalNetwork(

```



```

(anchor_generator): AnchorGenerator()
(head): RPNHead(
  (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
  (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
)
)
(roi_heads): RoIHeadsVanilla(
  (box_roi_pool): MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'], output_size=(7, 7), sampling_ratio=2)
  (box_head): TwoMLPHead(
    (fc6): Linear(in_features=12544, out_features=1024, bias=True)
    (fc7): Linear(in_features=1024, out_features=1024, bias=True)
  )
  (box_predictor): FastRCNNPredictor(
    (cls_score): Linear(in_features=1024, out_features=2, bias=True)
    (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
  )
)
)
)

```

And following are the corresponding weights:

```

backbone.body.conv1.weight torch.Size([64, 3, 7, 7])
backbone.body.layer1.0.conv1.weight torch.Size([64, 64, 1, 1])
backbone.body.layer1.0.conv2.weight torch.Size([64, 64, 3, 3])
backbone.body.layer1.0.conv3.weight torch.Size([256, 64, 1, 1])
backbone.body.layer1.0.downsample.0.weight torch.Size([256, 64, 1, 1])
backbone.body.layer1.1.conv1.weight torch.Size([64, 256, 1, 1])
backbone.body.layer1.1.conv2.weight torch.Size([64, 64, 3, 3])
backbone.body.layer1.1.conv3.weight torch.Size([256, 64, 1, 1])
backbone.body.layer1.2.conv1.weight torch.Size([64, 256, 1, 1])
backbone.body.layer1.2.conv2.weight torch.Size([64, 64, 3, 3])
backbone.body.layer1.2.conv3.weight torch.Size([256, 64, 1, 1])
backbone.body.layer2.0.conv1.weight torch.Size([128, 256, 1, 1])
backbone.body.layer2.0.conv2.weight torch.Size([128, 128, 3, 3])
backbone.body.layer2.0.conv3.weight torch.Size([512, 128, 1, 1])
backbone.body.layer2.0.downsample.0.weight torch.Size([512, 256, 1, 1])
backbone.body.layer2.1.conv1.weight torch.Size([128, 512, 1, 1])
backbone.body.layer2.1.conv2.weight torch.Size([128, 128, 3, 3])
backbone.body.layer2.1.conv3.weight torch.Size([512, 128, 1, 1])
backbone.body.layer2.2.conv1.weight torch.Size([128, 512, 1, 1])
backbone.body.layer2.2.conv2.weight torch.Size([128, 128, 3, 3])
backbone.body.layer2.2.conv3.weight torch.Size([512, 128, 1, 1])
backbone.body.layer2.3.conv1.weight torch.Size([128, 512, 1, 1])
backbone.body.layer2.3.conv2.weight torch.Size([128, 128, 3, 3])
backbone.body.layer2.3.conv3.weight torch.Size([512, 128, 1, 1])
backbone.body.layer3.0.conv1.weight torch.Size([256, 512, 1, 1])
backbone.body.layer3.0.conv2.weight torch.Size([256, 256, 3, 3])
backbone.body.layer3.0.conv3.weight torch.Size([1024, 256, 1, 1])
backbone.body.layer3.0.downsample.0.weight torch.Size([1024, 512, 1, 1])
backbone.body.layer3.1.conv1.weight torch.Size([256, 1024, 1, 1])
backbone.body.layer3.1.conv2.weight torch.Size([256, 256, 3, 3])
backbone.body.layer3.1.conv3.weight torch.Size([1024, 256, 1, 1])
backbone.body.layer3.2.conv1.weight torch.Size([256, 1024, 1, 1])
backbone.body.layer3.2.conv2.weight torch.Size([256, 256, 3, 3])
backbone.body.layer3.2.conv3.weight torch.Size([1024, 256, 1, 1])
backbone.body.layer3.3.conv1.weight torch.Size([256, 1024, 1, 1])
backbone.body.layer3.3.conv2.weight torch.Size([256, 256, 3, 3])
backbone.body.layer3.3.conv3.weight torch.Size([1024, 256, 1, 1])
backbone.body.layer3.4.conv1.weight torch.Size([256, 1024, 1, 1])
backbone.body.layer3.4.conv2.weight torch.Size([256, 256, 3, 3])
backbone.body.layer3.4.conv3.weight torch.Size([1024, 256, 1, 1])
backbone.body.layer3.5.conv1.weight torch.Size([256, 1024, 1, 1])
backbone.body.layer3.5.conv2.weight torch.Size([256, 256, 3, 3])
backbone.body.layer3.5.conv3.weight torch.Size([1024, 256, 1, 1])

```

```

backbone.body.layer4.0.conv1.weight torch.Size([512, 1024, 1, 1])
backbone.body.layer4.0.conv2.weight torch.Size([512, 512, 3, 3])
backbone.body.layer4.0.conv3.weight torch.Size([2048, 512, 1, 1])
backbone.body.layer4.0.downsample.0.weight torch.Size([2048, 1024, 1, 1])
backbone.body.layer4.1.conv1.weight torch.Size([512, 2048, 1, 1])
backbone.body.layer4.1.conv2.weight torch.Size([512, 512, 3, 3])
backbone.body.layer4.1.conv3.weight torch.Size([2048, 512, 1, 1])
backbone.body.layer4.2.conv1.weight torch.Size([512, 2048, 1, 1])
backbone.body.layer4.2.conv2.weight torch.Size([512, 512, 3, 3])
backbone.body.layer4.2.conv3.weight torch.Size([2048, 512, 1, 1])
backbone.fpn.inner_blocks.0.weight torch.Size([256, 256, 1, 1])
backbone.fpn.inner_blocks.0.bias torch.Size([256])
backbone.fpn.inner_blocks.1.weight torch.Size([256, 512, 1, 1])
backbone.fpn.inner_blocks.1.bias torch.Size([256])
backbone.fpn.inner_blocks.2.weight torch.Size([256, 1024, 1, 1])
backbone.fpn.inner_blocks.2.bias torch.Size([256])
backbone.fpn.inner_blocks.3.weight torch.Size([256, 2048, 1, 1])
backbone.fpn.inner_blocks.3.bias torch.Size([256])
backbone.fpn.layer_blocks.0.weight torch.Size([256, 256, 3, 3])
backbone.fpn.layer_blocks.0.bias torch.Size([256])
backbone.fpn.layer_blocks.1.weight torch.Size([256, 256, 3, 3])
backbone.fpn.layer_blocks.1.bias torch.Size([256])
backbone.fpn.layer_blocks.2.weight torch.Size([256, 256, 3, 3])
backbone.fpn.layer_blocks.2.bias torch.Size([256])
backbone.fpn.layer_blocks.3.weight torch.Size([256, 256, 3, 3])
backbone.fpn.layer_blocks.3.bias torch.Size([256])
rpn.head.conv.weight torch.Size([256, 257, 3, 3])
rpn.head.conv.bias torch.Size([256])
rpn.head.cls_logits.weight torch.Size([3, 256, 1, 1])
rpn.head.cls_logits.bias torch.Size([3])
rpn.head.bbox_pred.weight torch.Size([12, 256, 1, 1])
rpn.head.bbox_pred.bias torch.Size([12])
roi_heads.box_head.fc6.weight torch.Size([1024, 12593])
roi_heads.box_head.fc6.bias torch.Size([1024])
roi_heads.box_head.fc7.weight torch.Size([1024, 1024])
roi_heads.box_head.fc7.bias torch.Size([1024])
roi_heads.box_predictor.cls_score.weight torch.Size([2, 1024])
roi_heads.box_predictor.cls_score.bias torch.Size([2])
roi_heads.box_predictor.bbox_pred.weight torch.Size([8, 1024])
roi_heads.box_predictor.bbox_pred.bias torch.Size([8])

```


Bibliography

- [1] J. Beguin, E.J.B. McIntire, D. Fortin, S.G. Cumming, F. Raulier, P. Racine and C. Dussault: *Explaining Geographic Gradients in Winter Selection of Landscapes by Boreal Caribou with Implications under Global Changes in Eastern Canada*, volume 8. Public Library of Science, 2013.
- [2] S. Belousov: "<https://pypi.org/project/mean-average-precision/>", *MAP python package*.
- [3] C. Bishop, P. Bishop, G. Hinton and Oxford University Press: *Neural Networks for Pattern Recognition*. Advanced Texts in Econometrics. Clarendon Press, 1995.
- [4] COCO API: "<https://github.com/cocodataset/cocoapi>", 2015.
- [5] D. Davletshina: *Monocular Depth Estimation for Aerial Images*, *Master Thesis*, 2020.
- [6] D. Davletshina: *Monocular Depth Estimation for Aerial Images*, "<https://github.com/DianaDI/mde>, *GitHub*", 2020.
- [7] Faster R-CNN Source Code: "https://pytorch.org/vision/stable/_modules/torchvision/models/detection/faster_rcnn.html".
- [8] M. Fromm, M. Schubert, G. Castilla, J. Linke and G. McDermid: *Automated Detection of Conifer Seedlings in Drone Imagery Using Convolutional Neural Networks*, volume 11, 2019.
- [9] X. Glorot and Y. Bengio: *Understanding the difficulty of training deep feedforward neural networks*, In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*. "<http://proceedings.mlr.press/v9/glorot10a.html>", 2010.
- [10] K. He, X. Zhang, S. Ren and J. Sun: *Deep Residual Learning for Image Recognition*, In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [11] J. Jooste: *Conifer Seedling Detection in UAV-Imagery with RGB-Depth Information*, 2020.

- [12] J. Jooste: *Conifer Seedling Detection in UAV-Imagery with RGB-Depth Information*, "https://github.com/JasonJooste/seedlings_height/", GitHub, 2021.
- [13] P. Lee and S. Boutin: *Persistence and developmental transition of wide seismic lines in the western Boreal Plains of Canada*. In *Journal of Environmental Management*, volume 78, 2006.
- [14] T.T. Lin: *Labelimg*, "<https://github.com/tzutalin/labelImg/>", 2021.
- [15] T.Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie: *Feature Pyramid Networks for Object Detection*. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [16] S. Marcel and Y. Rodriguez: *Torchvision the Machine-Vision Package of Torch*. In *Proceedings of the 18th ACM International Conference on Multimedia*. MM '10. Association for Computing Machinery, New York, NY, USA, 2010.
- [17] Z. Preiffer: *Cumulative effects, caribou and national energy board regulation*. In *Proceeding presented at the 36th annual meeting of the International Association for Impact Assessment, Achi-Nagoya*, 2016.
- [18] PyTorch Documentation: *Models and pre-trained weights*, "<https://pytorch.org/vision/main/models.html>".
- [19] Z. Zekuan: "<https://github.com/wolverinn/Depth-Estimation-PyTorch/>", 2020.