Institute for Computer Science VII
Robotics and Telematics

Master's thesis

Autonomous Terrain Classification Through Unsupervised Learning

Felix Zeltner

November 2016

First supervisor: Prof. Dr. Andreas Nüchter
Second supervisor: Dr. Anita Enmark

Abstract

A key component of autonomous outdoor navigation in unstructured environments is the classification of terrain. Recent development in the area of machine learning show promising results in the task of scene segmentation but are limited by the labels used during their supervised training.

In this work, we present and evaluate a flexible strategy for terrain classification based on three components: A deep convolutional neural network trained on colour, depth and infrared data which provides feature vectors for image segmentation, a set of exchangeable segmentation engines that operate in this feature space and a novel, air pressure based actuator responsible for distinguishing rigid obstacles from those that only appear as such. Through the use of unsupervised learning we eliminate the need for labeled training data and allow our system to adapt to previously unseen terrain classes.

We evaluate the performance of this classification scheme on a mobile robot platform in an environment containing vegetation and trees with a Kinect v2 sensor as low-cost depth camera. Our experiments show that the features generated by our neural network are currently not competitive with state of the art implementations and that our system is not yet ready for real world applications.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

One of the most challenging tasks in mobile robotics is the autonomous navigation of unknown terrain. A key element hereby are the sensor systems that let the robot perceive its environment. These sensor systems vary greatly in their complexity and versatility. Simple sensors include ultrasonic distance sensors and bumpers that can be used for coarse obstacle detection. More complex are 2D laser scanners, that can provide depth information for a cross-section of the environment. Of course, laser scanners can also be used to generate full 3D point clouds which can provide very detailed environment maps.

These types of sensors do however only provide depth information and in some cases values for the reflectivity of the surface the laser beam was returned from. Cameras on the other hand can produce high resolution colour data of the scene. Through the use of filters, images outside the visible spectrum can also be acquired. These images do however lack depth information.

One method by which three dimensional colour images can be produced is the stereo camera. These systems are actually compromised of two image sensors set up in a configuration much like the human eyes. By matching the two images from the left and right camera, depth information can be obtained from the parallax effect. An alternative to this passive ranging system is the use of an offset emitter to illuminate the scene with a known light pattern or to measure the time-of-flight.

Much harder than the acquisition of the data is its processing and interpretation. A 3D map of the surroundings may be sufficient for some applications, but for outdoor environments, just the geometry is often not enough. A purely geometry based approach lacks certain semantics that are needed to plan optimal paths. An example that illustrates this case is an environment with vegetation, where overgrown regions may look like solid obstacles to a laser scanner while the vehicle is actually capable of traversing them.

This requires the system to segment the scene into distinct regions that can be assigned to different object classes. This segmentation is traditionally based on various features derived from the sensor data such as colours, normal vectors, roughness values and many others, often without direct physical meaning. These features can then be used to train a classifier which can then assign labels to regions of the input data.

Recently, the use of neural networks for machine vision has gained popularity by often times significantly outperforming traditional systems for tasks such as image classification, segmentation and labeling. Machine learning is already in use by companies such as Google, Facebook and Microsoft to interpret text, voice and images on the internet. Autonomous driving is another area where neural networks and computer vision can be used to detect road markings, read traffic signs or identify pedestrians and other vehicles.

In the future, such systems could also see application in extreme environments such as planetary exploration with rovers. Autonomous navigation can reduce the need to plan movements by hand and in great detail. One could even imagine a rover seeking out subjects of interests entirely by itself.

1.2 Goals and Scientific Contribution

The goal of this work is to implement and evaluate such a terrain classification system. Our approach consists of three major components:

Feature generation The feature space for our terrain classifier will be produced by a deep convolutional neural network that we train on colour, depth and infrared images captured with a Kinect v2 sensor.

Segmentation We present multiple algorithms for segmenting the input images based on the feature vectors produced by the neural network. The implementations share a common interface and can therefore be exchanged with one another.

Classification To classify the encountered clusters as drivable or non-drivable, we implement a novel, air-pressure based sensing method to distinguish rigid obstacles from ones that can be navigated by our robot vehicle. This lets our system produce more optimal routes that can pass through high grass and similar objects that appear as obstacles in purely geometry based approaches.

We evaluate the individual components based on data captured in our test environment to determine if they improve upon the current state of the art in terrain classification.

Unlike previous methods, our approach focuses on performing the classification task without supervised training and without the use of handcrafted features. Most current implementations are based on deep convolutional networks which are trained end-to-end with labeled data which is costly and time-consuming to produce and further more limits the system to just the number of classes it was trained on.

Through the unsupervised training of our neural network we simplify the training process as it requires no such labeled data. In combination with our separated, flexible segmentation and classification systems, we can support any number of previously unencountered terrain classes and obstacles. This yields a very flexible terrain classification system that can automatically adjust to many different types of environments.

Chapter 2

Related Work

There have been many attempts to solve the problem of terrain classification. The approaches differ widely in terms of the employed sensors and inference methods. Sensors range from different vision based systems such as 2D and 3D LiDAR and cameras over vibration measurements to slip ratios of wheels.

In [14] the authors use LiDAR point clouds for terrain classification. They employ three handcrafted geometric features along with multiple echos from vegetation as input for their random forest classifier. With supervised learning this allows them to distinguish six classes of terrain reasonably well. Another use of LiDAR point clouds is described in [10]. Unlike the example above, the authors are using an unsupervised Gaussian Mixture Model to generate intermediate classes that are then grouped by a human expert into domain specific classes.

A method for deriving "unevenness" features from surface normals in RGB-D images is described in [13]. This allows them to distinguish smooth flooring from walls and uneven terrain. Additional handcrafted features derived from coloured point clouds are described in [5].

In contrast to this, [8] proposes the use of an inertial measurement unit which can measure acceleration and angular velocity components in three dimensions to classify terrain. This lets them distinguish between flat ground and slopes as well as some different surface types like grass and stone plates. A similar concept is presented in [3] where the authors evaluated a terrain classification system for planetary rovers based on vibration measurements on multiple points of the rover structure. Unlike camera based sensors, accelerometers and gyroscopes are invariant to lighting conditions which change rapidly between sunny and shady areas and also vary throughout the day.

Similarly, [6] shows a method for extrapolating information from proprioception for the purpose of classifying terrain in the near and far-field of stereo image data. The authors describe that data obtained from wheel slip can be combined with stereo cameras to distinguish different types of terrain. Further more they describe methods to bootstrap RGB based classification in the mid and far-field, where no or very coarse depth information is available from the near-field RGB-D data where information is much denser. In [2] this is taken a step further. Here, overhead satellite imagery is combined with on-board colour and infrared camera data and laser scanner point clouds to enable better path planning over longer ranges.

A method for unsupervised online learning of terrain traversability is proposed in [9]. The

robot uses stereo cameras to distinguish different types of terrain and a bumper to subsequently categorize it as traversable or not. The authors use a mixture of geometric and visual features to train their classifier. We adapt parts of this strategy of drivability classification for our air-based obstacle classification system which also assigns traversability scores to cluster IDs but does so contactlessly.

The successful application of neural networks for terrain classification is demonstrated in [11] where a convolutional neural network is trained to detect landing zones for an autonomous helicopter in LiDAR point clouds. The shown approach outperforms the traditional classifiers based on different handcrafted features it is compared to quite significantly.

State of the art results in pixel-wise segmentation and labeling of road scenes are shown in [1]. Their deep convolutional encoder-decoder network performs pixel wise classification at high accuracies in part due to their custom upsampling layer. The design of their encoder-decoder stages serve as a starting point for our neural network. Comparable results in this field were achieved by [18] albeit through a different network architecture. Here, a recurrent neural network is used together with a conditional random field to produce the per pixel labels.

Chapter 3

Methodology

In this chapter we describe in detail the operating principles behind our terrain classifier. First we present the hardware platform and sensor systems on which we implement and test our system. Following this, we describe the software components necessary for the actual classification tasks and their interconnectivity. Finally, we provide an overview over our testing environment and the experimental methodology through which we evaluate our system.

3.1 Hardware Platform

3.1.1 Mobile Robot Platform

The testbed for our experiments is the Volksbot "Achim", a modular outdoor robot platform. It is powered by the two motorised, individually articulable front wheels and has a fixed rear axle. This makes an Ackermann type motion controller necessary to prevent wheels from skipping. Unlike most other Ackermann vehicles, there is no physical linkage between the wheels and the interdependent wheel angles are regulated in software. The powerful motorisation makes this vehicle capable of navigating most reasonably flat surfaces with ease, making it ideal for our test environment which consists of mostly grassy ground.

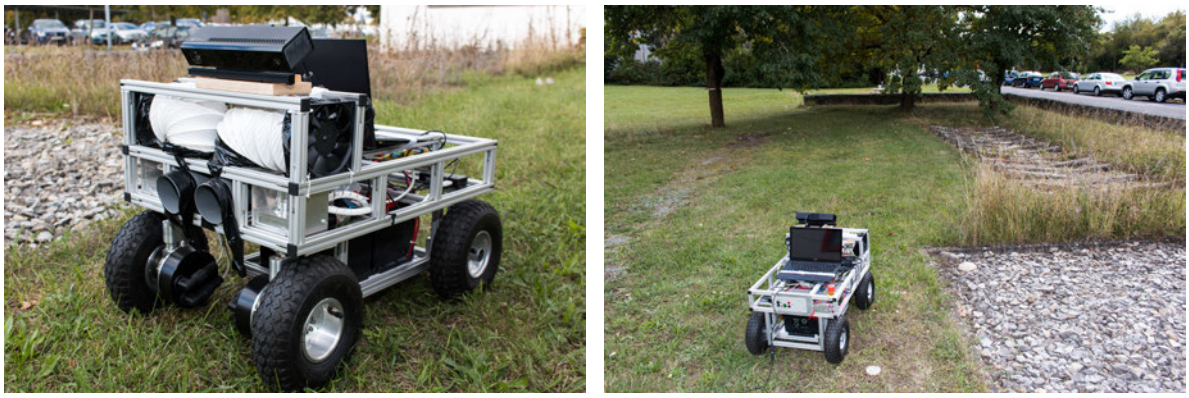


Figure 3.1: The robot vehicle "Achim" in our test environment.

Figure 3.2: Custom mounting bracket for the Kinect v2 sensor.

The vehicle is constructed from aluminum profile bars that allow easy reconfiguration and expansion. This allows us to attach our required sensors and actuators to the platform as required with little effort.

Power for the robot is provided by two 12 V car batteries that supply the motors and motor controllers with 24 V. To power our fans, we use a DC-DC converter to produce a 12 V supply from the batteries.

All computation is handled by a laptop attached to the robot. The notebook has an Intel Core i7-4500U CPU with 8 GB of ram and is running Ubuntu linux 16.04 LTS. It uses ROS, the Robot Operating System, to control the robot. The motor controllers are communicating on a CAN bus which is attached to the computer through a CAN to USB adaptor. To manually steer the vehicle, we use a joystick also connected to the laptop. Figure 3.1 depicts the robot in our outdoor testing field.

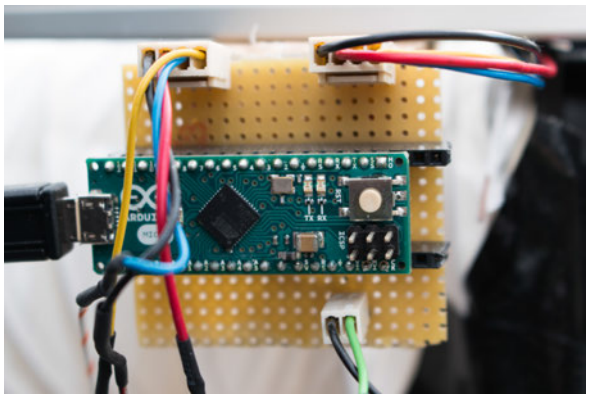
3.1.2 Kinect v2 Depth Camera

We use a Microsoft Kinect v2 depth camera as the main sensor of the vehicle. The Kinect v2 is an affordable RGB-Depth sensor which was originally designed for and sold with the Microsoft Xbox One game console, but is now also available separately. It can also be attached to an USB 3.0 port on a PC and powered from the wall with the help of an adapter kit which is available separately from the sensor.

Unlike its predecessor version which was based on the recognition of a projected dot pattern in space, the Kinect v3 uses time-of-flight measurements for its depth sensing. This makes it more precise and less noisy than the original Kinect sensor and its clones. The new version is also capable of limited operation in outdoor environments [4] which the previous iteration was not. Performance varies with the amount of sunlight present in the scene since the sunlight can overpower the built in infrared emitter leading to more noise or outright measurement failure. Nevertheless, the sensor works quite reliably at close range even in the sun, which is sufficient for our use case.

The sensor provides [12] a 512×424 pixel depth image at 30 Hz with a field of view (FOV) of $70° \times 60°$. The operational range is listed as 0.5 m to 4.5 m which in our tests decreases to

(a) Blower assembly with fans on the left and right and ducting to the middle.



(b) Fan speed controller board with 12 V input at the bottom and fans connected at the top.

Figure 3.3: The actuator for the movement detection system.

roughly 2 m in sunny conditions. Colour images are captured at up to 1920 × 1080 pixels at 30 Hz which results in a lower vertical FOV than the depth image. Further more we have access to the raw infrared image captured for the depth measurement which shows the illumination by the infrared light source of the sensor.

A multi-array microphone which can record directional audio is also present in the unit but it is not used in our experiments.

The Kinect v2 is fastened on top of the robot with a custom mounting bracket, shown in Figure 3.2. By tilting the sensor downward the field of view of the camera contains the ground immediately in front of the robot, up to a certain distance mostly limited by the sensor range. The Kinect Adapter for Windows which connects the sensor to the notebook requires a 12 V supply, normally provided by a AC power brick which plugs into the wall. To bypass this, we cut the proprietary coaxial power connector from said AC supply and directly connected it to the 12 V source on the vehicle that also powers the fans.

3.1.3 Air-based Obstacle Detector

In order to distinguish actual obstacles, such as trees or walls, from apparent obstacles, such as high grass, we utilise a custom air-pressure based actuator in tandem with the depth camera. The actuator is constructed from high-power fans intended for the PC server market and various ducting to redirect the air stream. The concept of the system is to apply air pressure to the obstacle and use the depth sensor to determine whether any objects in front of the robot are moved by it or if they remain stationary.

To this end, we have attached two speed-controllable 120 mm fans to the sides of the robot. The fans are Delta TFC1212DE-PWM server fans with brushless DC motors that can operate at up to 5500 RPM and deliver a very high air flow and air pressure. We use ducting constructed from drainage pipes and extractor ducting to redirect the air stream of both fans to the front of the vehicle so that it can blow against potential obstacles. The complete assembly is shown in Figure 3.3a.

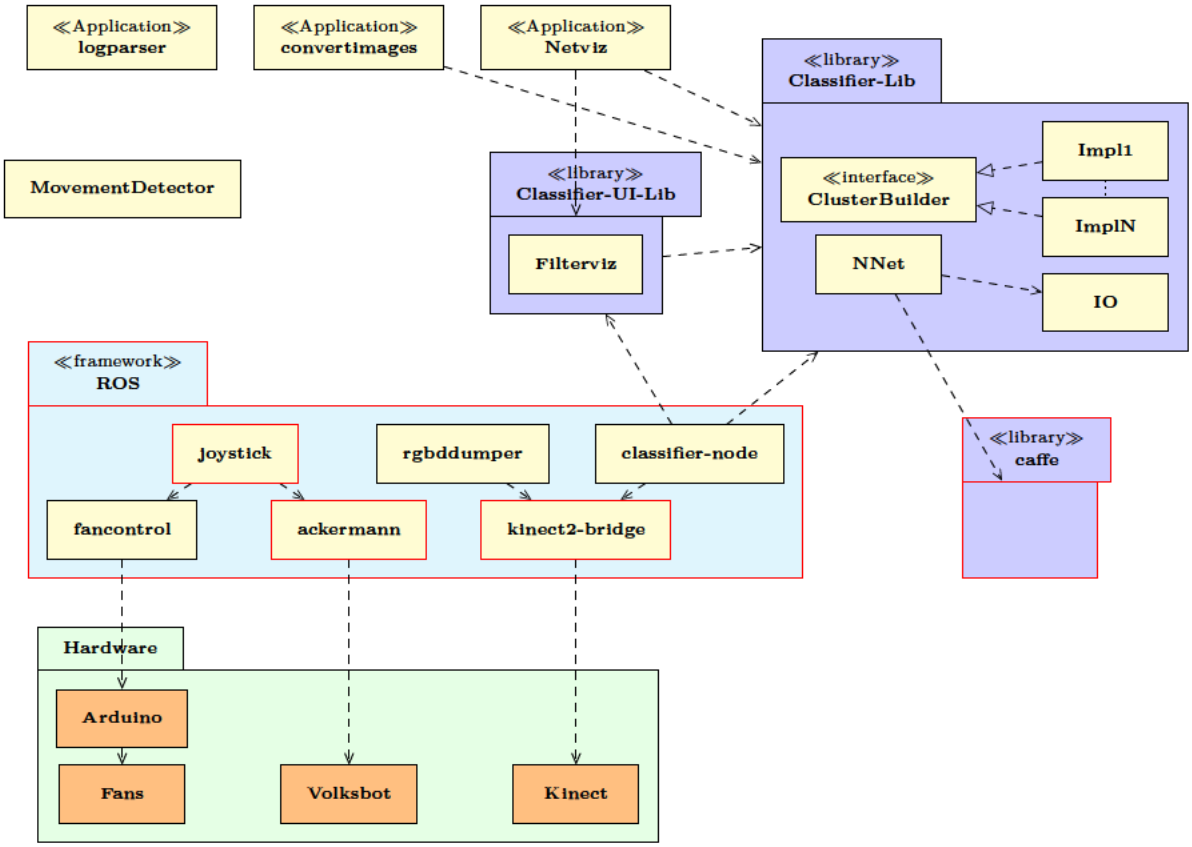


Figure 3.4: Overview over software components. Red borders indicate third-party modules.

Since the fans are intended for the PC server market, they use regular four-pin connectors with a standardised [7] PWM signal to control their speed. Hereby, the speed is set by the duty cycle of a 25 kHz PWM signal where 100% duty cycle corresponds to full speed and everything below some lower bound duty cycle to the minimum RPM of the fan. This particular model can additionally be turned off completely by setting the duty cycle to 0% which is convenient for our purposes.

We use an Arduino Micro microcontroller to implement this type of speed control which allows us to set the fan speeds from software. Figure 3.3b shows the simple circuit assembly for this task.

3.2 Software Architecture

3.2.1 Overview

Much of our work is realised in the form of software components. Figure 3.4 gives a high level overview over the various modules. Entities with red borders in the diagram represent third party code and orange boxes indicate hardware components.

Our main work resides in the `CLASSIFIER-LIB` C++ library. It contains the `NNET` class

which acts as a wrapper around the caffe library and provides functionality for loading and using neural networks. Image preprocessing and loading is handled by the IO class. The abstract ClusterBuilder class defines the common interface for our various segmentation implementations, here symbolised as Impl1 ... ImplN and further described in Section 3.2.3. Further more the library contains code that can generate 3D point clouds from the RGB-D images which can then be transformed to different coordinate systems.

To supplement the functionality of the classifier-lib, we also provided visualisation functionality in a second library called classifier-ui-lib. This library uses the Qt framework and provides UI elements that can be integrated into applications to show filter and parameter visualisations. It is used by our Netviz helper application which we use for neural network evaluation as well as our classifier-node ROS node to provide live images.

Our training and test images are converted into a format that caffe can load natively using our convertimages console application. It preprocesses the image files using the IO functionality exposed by classifier-lib and converts each triplet of colour, depth and infrared images to a caffe blob that is then saved as hdf5 database file.

The logparser helper application can be used during network training to generate a live plot of the error values using the gnuplot plotting software. Further more, it can parse caffe training log files and write the output into formatted text files for offline processing. We use these files to generate the training graphs in this work.

Our obstacle classification as described in Section 3.2.4 is implemented in the class called MovementDetector which is wrapped in a console application for testing.

The vehicle uses ROS, the robot operating system as its underlying software framework. It is designed to run on the Ubuntu linux distribution and provides infrastructure allowing individual system components running in nodes to communicate with each other. It contains various reusable modules that can be connected together to perform various tasks such as path planning, image capturing or mapping. We use the framework to drive the robot using the ackermann and joystick nodes that were developed for the Volksbot hardware. Our fancontrol node interfaces with the Arduino microcontroller's firmware to control the fans, while the third-party kinect2_bridge node provides access to the Kinect data feeds that we use in our rgbddumper and classifier-node nodes. Our use of ROS is described in more detail in Section 3.2.6.

In order to control the fan speeds through ROS, we also wrote firmware for the Arduino microcontroller to which they are attached. The firmware accepts commands on the serial interface of the Arduino and translates them into PWM signals according to the specifications described in Section 3.1.3.

3.2.2 Neural Network

Recent work has shown that neural networks can often outperform solutions relying on handcrafted features in computer vision tasks [15]. We are therefore using a neural network to extract features from the camera images we capture with the Kinect v2 sensor.

The selected network architecture is that of a deep convolutional autoencoder, similar to the first stages in SegNet [1]. The goal is to train the encoder to produce meaningful features that can be used to segment the image. Ideally, the feature representations found by the network should outperform handcrafted features such as surface normals, spin features or SIFT features

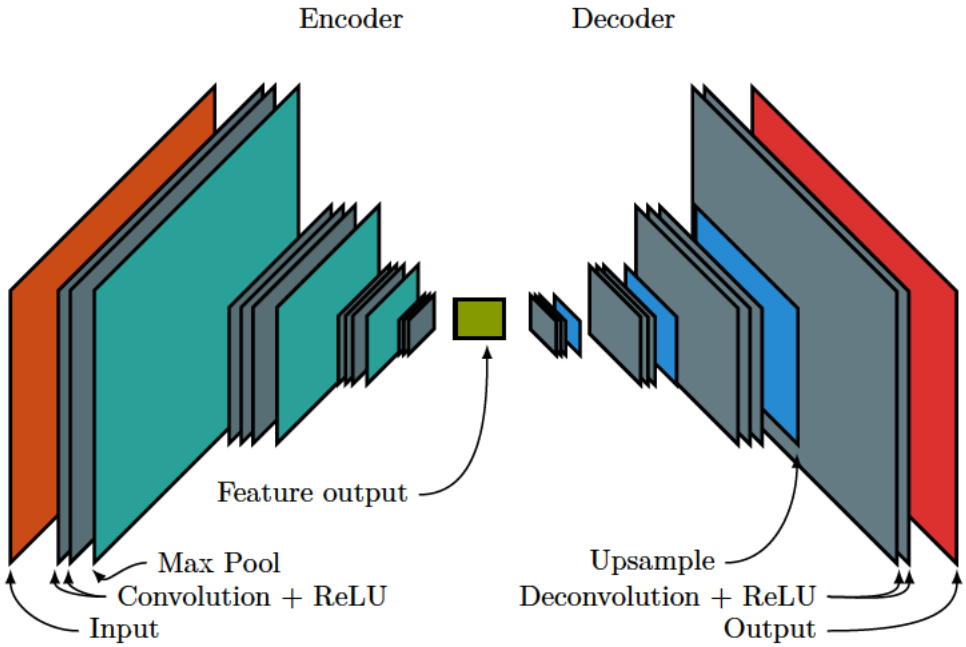


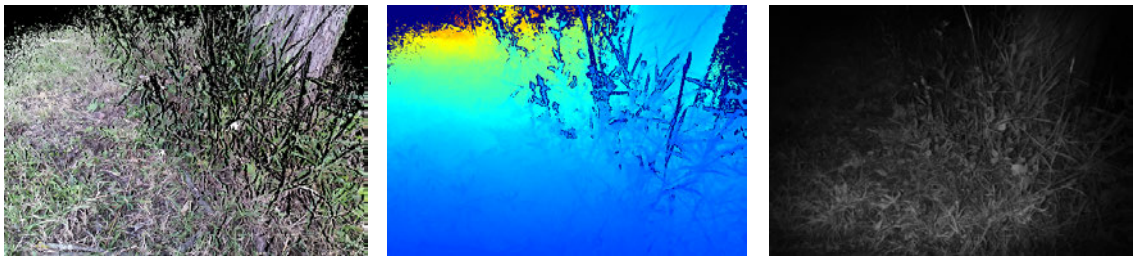
Figure 3.5: Graphical representation of our network architecture.

which are used in traditional approaches.

The network input consists of the RGB colour image, the depth image and the infrared image captured by the Kinect sensor. The raw images are first cropped to the lowest common FOV and then converted from integer pixel representations to floating point data blobs spanning all images. This is necessary due to the different bit depths of the raw data which is 8 bit per pixel for each channel in the colour image and 16 bit per pixel for the greyscale infrared and depth images respectively. During this preprocessing, the data is rescaled to a common baseline of $[0, 1]$. For the depth image specifically, we only expect raw values of 0 mm to 4000 mm due to the camera angle and the resulting FOV together with the sensor range limitations. Any pixels that fall outside this range are ambiguities created by objects that are too close to the sensor or other misreadings. Therefore we set these pixels to 0 and use a scaling factor of $1/4000$ for the remaining areas instead of $1/65\,536$ for the full 16 bit input range.

The input data is then fed through a series of convolution nodes, each with a 3×3 input window, a stride 1 and padding of 1. Each convolution is followed by a rectified linear unit (ReLU) with activation function $f(x) = \max(0, x)$. The convolution layers are stacked in sets of threes followed by a 2×2 max pooling layer. The first stack has only two convolution operations due to hardware limitations. In total, there are four such segments with increasing number of filter banks in the encoder part of the net. The input and second stage convolution layers have 64 filters each, the third stack has 256 and the innermost stage has 512 filters per convolution layer.

This effectively encodes our input in a $8\times$ downscaled image with 512 channels. We can



(a) Colour channels

(b) Depth channel (coloured by distance for visualisation only)

(c) Infrared channel

Figure 3.6: Example of training image consisting of colour, depth and infrared channels. Black regions in the colour image indicate areas with no depth information either due to shading or sensor range limitations.

interpret this as a 512 dimensional feature vector for each of the output pixels that represent a 8×8 input area.

To be able to train the network, we also require a decoder that performs the inverse operations of the encoder. In the decoder, the convolutions are replaced by deconvolutions and the pooling layers are replaced by upsampling layers. The corresponding convolution and deconvolution layers share their filter banks and parameters and are therefore performing exactly inverse operations. This means that in an optimal network, the output of the network equals the input. Since information is lost during the convolution and pooling operations, this forces the network to find ways to represent the input in a different dimensionality. Figure 3.5 shows a graphical representation of our network architecture.

Prior to the first convolution operation, dropout noise is added. For each pixel in the input, there is a 50% chance that it will be set to 0 before it is fed into the network. This can improve the robustness of the features discovered during the training process as it forces the network to reconstruct the missing data itself.

The network is trained in an unsupervised manner using images captured while manually steering the robot through the expected terrain. The raw data is prerocessed in the same way as the network input described above and then stored in binary files that caffe can read. The images are split up into two separate sets: A *training* set and a *test* set. The network optimisation is run on the *training* set and periodically evaluated against the *test* set to detect when the solver starts to overfit the training data. We use 104 test images and 366 training images to perform our network optimisation. Since the training is unsupervised, no label data is required for this process. The order of the training data is randomized each epoch to prevent potential biasing through a static sequence. Figure 3.6 shows an example of a training image.

We use an AdaDelta [17] solver for the optimisation process. This is a type of gradient descent solver that is very robust and automatically updates its learning rate. We have found this solver to perform better than a simple stochastic gradient descent which often did not converge at all. The error function is chosen in such a way that the network is forced to reproduce its input at the output. This is accomplished by using the Euclidean distance between the input and output

as the error function. This is calculated as

$$\frac{1}{2N}\sum_{i=1}^{N}\left\|x_i^{in}-x_i^{out}\right\|_2^2, \tag{3.1}$$

where N is the number of pixels, x_i^{in} is the ith pixel in the input image and x_i^{out} the ith pixel in the output image. A pixel in this case is just a one-dimensional number as each input channel (red, green, blue, depth, infrared) is considered separately.

To implement our neural network architecture, we use the caffe framework citejia2014caffe which provides implementations for many different neural network layers and solvers. Since the upscaling layer we require for our architecture is not part of the stock caffe library, we use the SegNet fork of caffe with backported support for Nvidia's cuDNN 4 and 5 from `https://github.com/m00nkeh/caffe-segnet`. The cuDNN support provides implementations of some layers optimised for Nvidia GPUs based on their CUDA compute architecture. To accelerate the training process, we use a desktop computer with two Nvidia GTX970 GPUs with 4 GB of VRAM each, an Intel Core i7-4600K and 16 GB of RAM. Training is performed on both cards in parallel, with each card processing half of the training images.

Our network definitions, training and test data as well as a pre-trained model are available at `http://fzeltner.de/thesis`.

3.2.3 Segmentation

Since the goal of our work is to distinguish between different terrain types, we have to implement a strategy that can segment images and recognize previously encountered structures in new camera data and assign the right labels to them. A label in this case is merely a numeric index as the system has no need for semantic names like "tree" or "grass". A lookup table that is updated by the classifier described in Section 3.2.4 maps each index to one of the three possible states of *unknown*, *drivable* or *obstacle*. In the following, we use the term *cluster* for these segments.

Strategy

Algorithm 1 describes the process with which the cluster indices are assigned to the regions of the captured image. The input to the function is the matrix of feature vectors M that is obtained from the neural network. This matrix has the dimensions $w \times h \times d$ where w and h are determined by the number of max pooling operations in the network. d is the number of filters of the last convolution layer. In our case M therefore has dimensions $64 \times 44 \times 512$ which is a reduction of the input image dimensions by a factor of 8. *Labels* has the same width and height as M but only a single channel. It will contain a cluster ID for each input pixel by the end of the algorithm.

We now iterate over M pixel by pixel and try to find the corresponding cluster through the function `getCluster`. If a matching cluster is found, we write its ID to the corresponding pixel of the *Labels* output matrix. Should no matching cluster be found, a new one is created. Its center vector is initialised with the feature vector of the current pixel and its state set to *unknown*. The cluster's ID is set to the next highest free number and its radius is initialised

Algorithm 1 Obtaining per pixel cluster labels

```
1: function getLabels(Net output blob M)
2:     Labels ← −1        // Matrix with same width and height as M
3:     for each pixel in M do
4:         c ← getCluster(pixel)
5:         if c not found then
6:             c ←newCluster(pixel)
7:         end if
8:         Labels[pixel] ← c.ID
9:     end for
10:    updateClusters(Labels)
11:    return Labels
12: end function
```

Algorithm 2 Cluster update function

```
1: procedure updateClusters(Labels)
2:     for each c in clusters do
3:         assignedpixels ←filter(Labels, c.id)
4:         c.hits ← c.hits + assignedpixels.count
5:         c.radius ←updateRadius(c, assignedpixels)
6:         c.stddev ←updateStddev(c, assignedpixels)
7:     end for
8:     update global means if necessary
9: end procedure
```

from some initial value determined by the specific `ClusterBuilder` implementation. Finally, it is appended to the array of known clusters and therefore accessible to subsequent calls to `getCluster`.

Before returning the label matrix *Labels*, the attributes of the known clusters are updated by calling function `updateClusters` with the labels. The attributes that are actually used for the segmentation process differ between implementation, but an example is given in Algorithm 2. In this case the hit counter is increased by the number of matches found in the input image. The cluster radius is also updated through averaging all encountered radii. Some implementations also update the standard deviation of the radii as they may be used in the `getCluster` function to determine if a feature vector matches a class or not.

In some implementations we generate a set of initial clusters using the k-means algorithm. This process is described in Algorithm 3. The k-means algorithm tries to group similar vectors into k clusters such that the sum over all square distances of the points to their cluster centers is minimal. We use the implementation provided by the openCV library for this task. It provides us with the k cluster centers as well as a matrix mapping input pixels to cluster indices.

For each cluster center we now create new clusters as described for `getLabels` above. These are then updated through the `updateClusters` function, again as described above.

Algorithm 3 ClusterBuilder initialisation

```
1: function INIT(Net output blob M, number of clusters k)
2:     Labels, centers ← KMEANS(M, k)
3:     for each center c in centers do
4:         NEWCLUSTER(c)
5:     end for
6:     UPDATECLUSTERS(Labels)
7:     return Labels
8: end function
```

The function `getCluster` that is used by both Algorithm 1 and 3 searches the existing clusters for a match given a feature vector. There are currently four different search strategies implemented:

Distance First Match returns the first cluster that fulfills the condition

$$\|c.center - v\|_2 < c.radiusLimit, \tag{3.2}$$

where c is the current cluster and v is the feature vector to be matched. $\|\|_2$ is the Euclidean distance. In other words, the algorithm checks whether the feature vector is inside a hyper-sphere around the cluster center or not and ends the search as soon as this condition is met.

Distance Best Match operates identical to the above but checks against *all* existing clusters and returns the one where the distance between the two vectors is the shortest. The feature vector does however still have to fulfill the condition

$$\|c_{closest}.center - v\|_2 < c_{closest}.radiusLimit \tag{3.3}$$

for it to be considered a match. The radius limit in both cases may be a global constant, fixed per cluster or continuously updated by the `updateClusters` function, depending on the specific ClusterBuilder implementation.

Best Cosine Similarity works like the search strategy above but uses the cosine similarity between the feature vector and the cluster center instead of the Euclidean norm. The cosine similarity is defined as

$$\frac{a \cdot b}{\|a\|\|b\|}, \tag{3.4}$$

i.e. the cosine between the vectors a and b. Since our feature space is positive or zero due to the ReLU non-linearity, the result of this calculation is always in the range of $[0, 1]$ where values closer to 1 indicate higher similarity. The radius limit is therefore used as a lower bound rather than as upper bound as in the previous strategies.

Best Angle Similarity is identical to the previous search strategy but uses the angle similarity in place of the cosine similarity. The angle similarity is defined as

$$1 - \frac{\arccos\left(\frac{a \cdot b}{\|a\|\|b\|}\right)}{\pi}. \tag{3.5}$$

Type	Init	Limit	New clusters	Search strategies
Fixed	k-means	$0.5r_{\max}$	$\mu_{r_{\max}}$	FD, BD
Averaging	k-means	$\omega_1 r_{\max} + \omega_2(\mu_r + \sigma_r)$	Global mean	FD, BD
k-means	k-means	$0.5\mu_r$ for distance and fixed for similarity	μ_r of new cluster	FD, BD, BCS, BAS
Simple	N/A	fixed	N/A	FD, BD, BCS, BAS

Table 3.1: Overview over the different ClusterBuilder implementations. Symbols: $r_{\max}$: Maximum radius to cluster center, μ_r: mean radius to cluster center, σ_r: Standard deviation of radii to center, $\omega_1 + \omega_2 = 1$: Weight factors, $\mu_{r_{\max}}$: mean over maximum radii of all clusters. Search strategies: FD: Distance First Match, BD: Distance Best Match, BCS: Best Cosine Similarity, BAS: Best Angle Similarity

This function takes the cosine similarity as an input and transforms it into a function that is linear over the cosine of the angle between the two vectors. The result is again in the range of $[0, 1]$ and higher values indicate higher similarity.

Implementations

We provide different implementations of the clustering algorithm based on the steps above. The implementations mostly differ in the way the radius or similarity limit is calculated and updated. Table 3.1 lists the available implementations and gives a comparison over the parameters. They all provide the same interface to the rest of the software and can therefore be swapped in and out for each other as desired.

The *Fixed* ClusterBuilder bases its radius limit on the largest distance between the cluster center and its assigned points. The factor of 0.5 was determined through experimentation and is dependent on the neural network. An initial set of clusters is generated through the k-means algorithm, as described above. For each of these clusters we find the maximum radius so we can search matching clusters on subsequent getLabels calls. If a point does not match any of the existing clusters, a new one is created. It will initially be assigned the mean value of the radius limits of the other clusters as its limit, so it can be matched against in the current processing iteration. During the following call to updateClusters, the maximum radius is updated for all clusters. After this is completed, the mean over the maximum radii is also updated.

Enhancing on the idea of this strategy is the *Averaging* ClusterBuilder. It works similarly to the *Fixed* ClusterBuilder, but instead of basing its limit just on the largest distance, it uses a weighted mean of the maximum radius and the mean radius plus one standard deviation. The weights ω_1 and ω_2 are fixed and biased $5 : 1$ in favour of the mean radius. During the update step, we now naturally also have to update the radius mean and standard deviation for each cluster. Since both of these strategies base their limits on radii, they only work for the distance based search strategies.

A different approach is implemented in the *k-means* ClusterBuilder. Instead of iterating over the pixels of a new image, we apply the k-means algorithm to it, like we do in the initialisation step. We then try to match the resulting clusters to the existing ones using one of the

available search strategies. For the distance based approaches we use half of the mean radius as the limit and for the similarity comparison we use predefined constants. Unlike the previous methods, this implementation does not require any global averages, as new clusters are generated by the k-means algorithm and we never need to compare any pixels from the current image to clusters that were newly created in the same time step.

Finally, we provide a *Simple* implementation that does not perform any initialisation and just uses predefined constants as the limits when matching pixels to clusters. This means we do not have to perform any averaging or update calculations. We just count hits per cluster for statistical purposes and eliminate clusters with a hit count below a certain threshold. This helps reduce the number of clusters that contain very few pixels which are commonly found in regions of the image where there is no actual data in the source captures.

3.2.4 Classification

The output of the segmentation process alone is insufficient for our drivability analysis as it only differentiates between different *types* of terrain but does not yet provide information about *traversability*. Therefore we employ an additional processing step to classify the detected segments as either drivable or obstacle. Since we have depth information, we can map the image containing our labels back into 3D space and assign them to grid cells on a global map. When the vehicle encounters a grid cell that is marked as unknown, the higher order planning system should stop the vehicle and begin the classification sequence.

Classification is performed via the fan system attached to the robot for this purpose described in Section 3.1.3 and the depth camera. The goal is to detect whether the encountered structure is a solid obstacle or a pseudo-obstacle, high grass for example, that the robot could easily traverse. The underlying assumption is that pseudo-objects will move when the fans blow against them while actual obstacles, such as trees, will not.

To detect this movement, we first capture a baseline depth image with the Kinect camera with the fans turned off. Next, the fans are engaged and a second depth image is captured. The two data sets can then be compared to find areas that show movement.

This is done by calculating the difference in the depth images. Since we are not interested in the direction of the movement, we take the absolute value of the difference to simplify the subsequent operations. Prior to calculating the difference in the images, we set any pixels larger than a certain threshold to zero, similar to the preprocessing we perform on the neural network input. This removes outliers and sensor misreadings. We repeat this thresholding operation on the resulting image as well with a lower maximum value to filter out regions where the depth change is very high. This occurs especially in the edges, as the sensor data there is most noisy and often only available in one of the source images.

The final two operations on the differential image are the application of a binary threshold and a subsequent erosion pass with a rectangular kernel to filter out most of the single pixel regions. The remaining large chunks of pixels are the areas in which movement occurred.

An obvious limitation of this system is that it does not work for the classification of the ground. Concrete for example will not move when exposed to an airstream and would therefore be classified as non-traversable. Therefore this system would have to be combined with some form of existing ground plane extraction methods to limit the output area of our classifier to

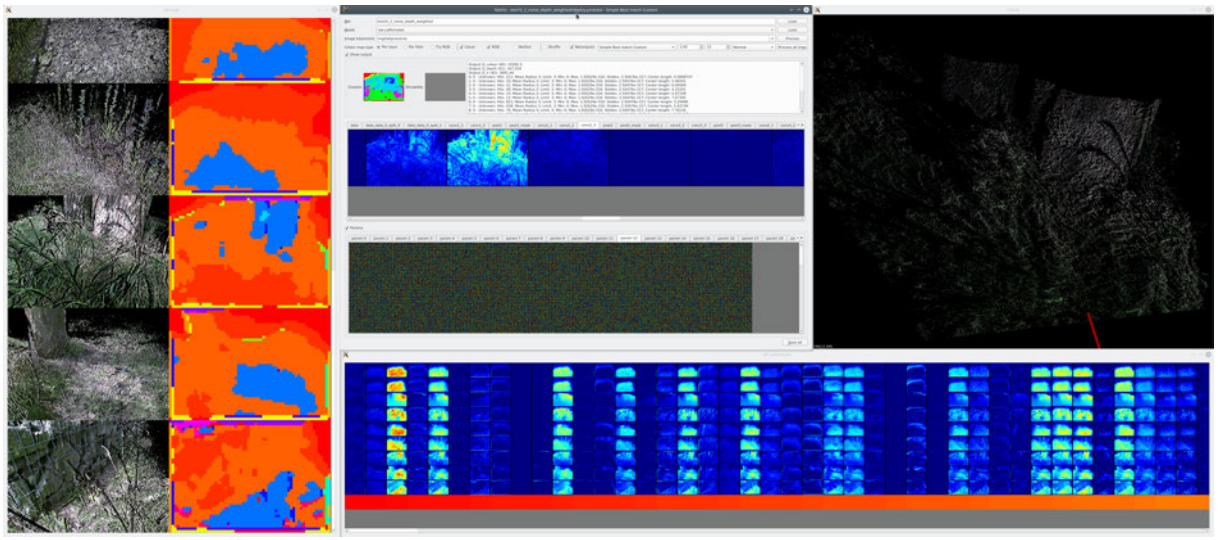


Figure 3.7: Screenshot of the Netviz application showing the point cloud view to the right, a filter output comparison at the bottom and clustering test to the left. The controlling UI is visible in the middle with the outputs of the third convolution layer of the second stage.

objects protruding from the floor.

3.2.5 Netviz

To help us gain insight into the operation of the neural network and to test our various clustering methods, we developed a GUI application that can create various visualisations and comparisons for us. It uses our abstraction layer for the caffe framework to load network definitions and weights, process images and visualise all intermediate outputs. This allows us to inspect the results of the various convolution and deconvolution layers throughout the network. It can also generate 3D point clouds from the Kinect depth images and either colour them by cluster ID or with their original RGB data.

By processing images in bulk, we can compare the network performance on different images and also visually inspect the difference in filter outputs they create. We can also feed the network outputs into one or multiple instances of the clustering algorithm to inspect and compare its performance.

Most of the visualisation elements are implemented in our `CLASSIFIER-UI-LIB` as library functions and can therefore be reused and integrated into other applications easily. Further more, all visualisations can be exported to png image files.

A screenshot of the application is given in Figure 3.7 which shows some of the functionality it provides.

3.2.6 ROS

As we already outlined above, we use the ROS framework for our hardware control and data capturing. Figure 3.8 gives an overview over the data flow between the ROS nodes we use in

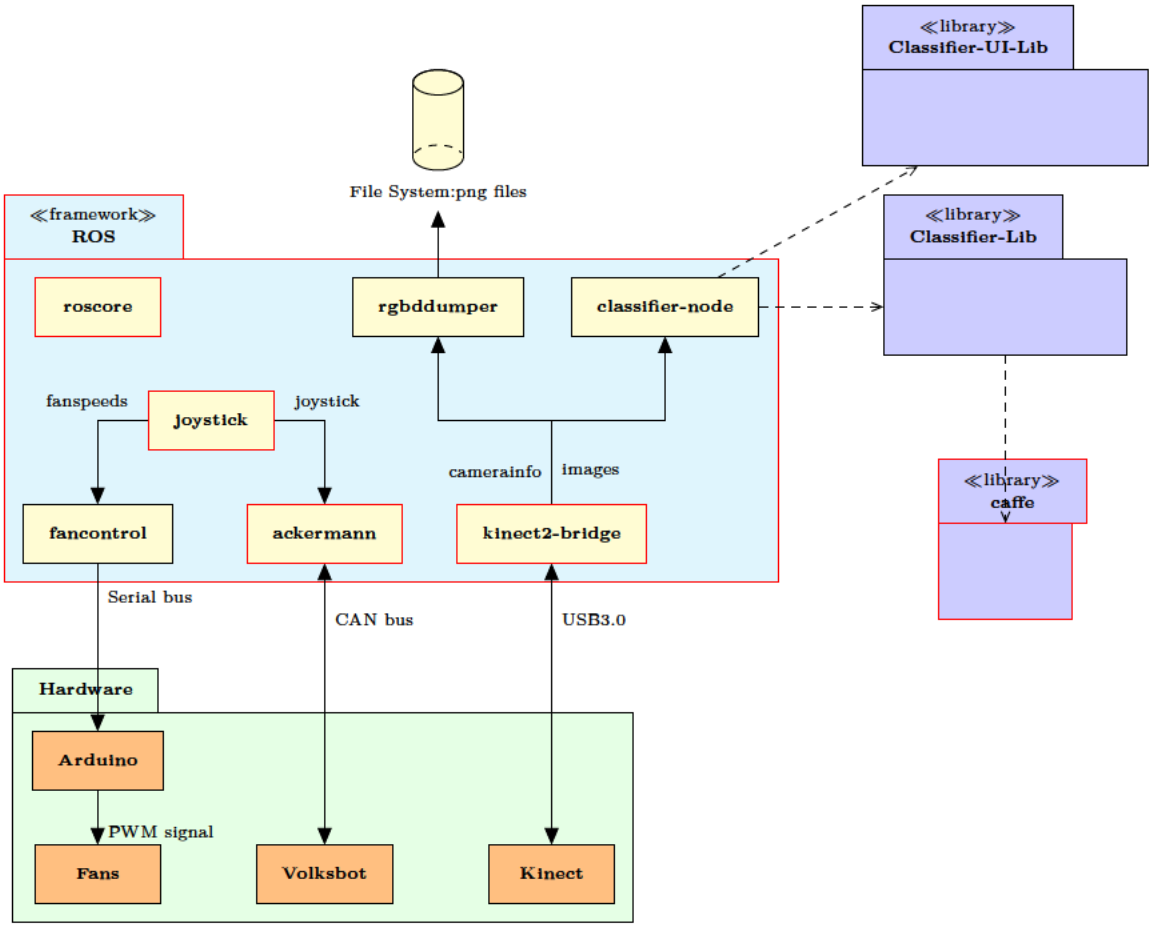


Figure 3.8: ROS data flow and hardware communication overview. Solid arrows show communication channels with the given names. In case of hardware communication, the interface type is given instead. Dashed arrows represent library dependencies

our system. Again, third party components are outlined in red and hardware components are shown in orange. Communication in ROS is handled by a message based system in which nodes advertise their provided *topics* via the `ROSCORE` service. Other nodes can then subscribe to these topics, from which point on they can then receive messages from the sending node.

The motor controls for steering and driving are both handled by the `ACKERMANN` node which implements and controls the Ackermann constraints for the vehicle in software. Both the steering and driving motors are accessed via hardware controllers on a CAN bus. We attach to this CAN bus by means of an USB to CAN adapter. The commands for speed and direction are generated by the `JOYSTICK` node and published on the *joystick* topic. The `JOYSTICK` node supports an array of gamepads and joysticks as hardware input devices to steer the robot. We modified this node to also send fan speed commands on the *fanspeeds* topic to our `FANCONTROL` node which then translates the commands into a format suitable for the Arduino microcontroller and forwards them over the serial connection which connects the device to the computer.

To get access to the Kinect data, we use the KINECT2_BRIDGE node which is part of the IAI Kinect2 [16] ROS package. It uses the LIBFREENECT2 library to connect to the camera over its USB 3.0 interface. The node provides a number of topics for other components to subscribe to. Namely it provides topics for the colour, depth and infrared images in various resolutions, point clouds at various resolutions, and the corresponding *camerainfo* topics that transmit metadata for the recorded frames. We usually limit the frame rate to 1 image per second for our operations to limit the performance drain on the computer.

We provide two nodes that can consume this image data, the RGBDDUMPER node and the CAFFE-CLASSIFIER node. The RGBDDUMPER node is a helper node that can be used to save images captured by the Kinect to disk. Prior to saving them, we optionally crop the frames to a common field of view to compensate for the discrepancies in camera aspect ratios as described in earlier sections. The main purpose of this node is to record images for our network test and training which can then be converted into the proper format by our convertimages application. Additionally, it provides a realtime view of the captured data so that it can be inspected during operation.

The second node operating on the live data provided by the bridge is the CLASSIFIER-NODE. This node serves as a testbed to evaluate our algorithms in realtime on the robot. It links against our CLASSIFIER-LIB which lets us load our neural network definition and weights. We can then feed the data we receive from the Kinect directly into the network and then inspect the output with our visualisation helpers. This way we can inspect layer activations as the robot is capturing data. Through the CLASSIFIER-LIB we also have access to our segmentation code which we can therefore also evaluate as we drive the robot in the environment.

3.3 Experimental Methodology

The environment our robot operates in is our outdoor test course. The ground of the course is mostly low cut grass with earth patches in some spots. We sectioned off a region where the grass could grow higher to be able to test our classification algorithm there. High grass also naturally occurs in some areas such as between the rocky test field and the area with wooden logs. Further more there are multiple trees and a low stone wall that serve as solid obstacles for our experiments. Depending on the weather, there are both sunny and shady regions present, which can be a challenge for our vision system. Figure 3.9 shows two views onto our test course.

We acquire our neural network training and test data by manually driving the robot through the environment while it continuously records images with the Kinect camera. We also capture data from a neighbouring tarmac path as well as the path into and out of our Robotics hall to produce a more diverse data set. The samples are then split into a test set and a training set after manually removing some duplicate images to reduce any biases in the data. As mentioned above, our training data is available at `http://fzeltner.de/thesis`.

With our training data, we use the caffe framework to optimize our network parameters. The results of the optimisation are quantified by the training and test error scores produced during the learning procedure. After the optimisation is complete, we evaluate the encoder-decoder performance over the test set to identify which types of images score the lowest and highest errors. Finally, we inspect the output data the network produces after the encoding stage as

(a) Section with high grass

(b) Various obstacles and ground types

Figure 3.9: The test environment

(a) Testing classification on tall grass

(b) Testing classification on tree

Figure 3.10: Examples of experimental setup for classification

this serves as the input for the testing of our segmentation algorithms. The visualisations for the encoder-decoder performance and the network output are generated by our `NETVIZ` application.

To evaluate our segmentation algorithms, we run each of them over the same series of test images and vary their parameters and search strategies to observe the changes in the result. We then visually inspect the cluster maps produced by the segmentation methods with regards to correct segmentation of individual images and correct grouping of terrain types across multiple images. To perform these evaluations we again utilise our `NETVIZ` application which can produce the required visualisation outputs and can additionally also show direct comparisons of multiple algorithms or parameter sets.

We test our classification method by recording Kinect sensor data from objects while the fan is off and again after setting the fan to its maximum power. In Figure 3.10 we depict the configurations in which such image pairs are captured. The picture pairs are then fed into our classification algorithm which then produces a map of the areas with movement. We inspect this map to verify that our algorithm works correctly.

Chapter 4

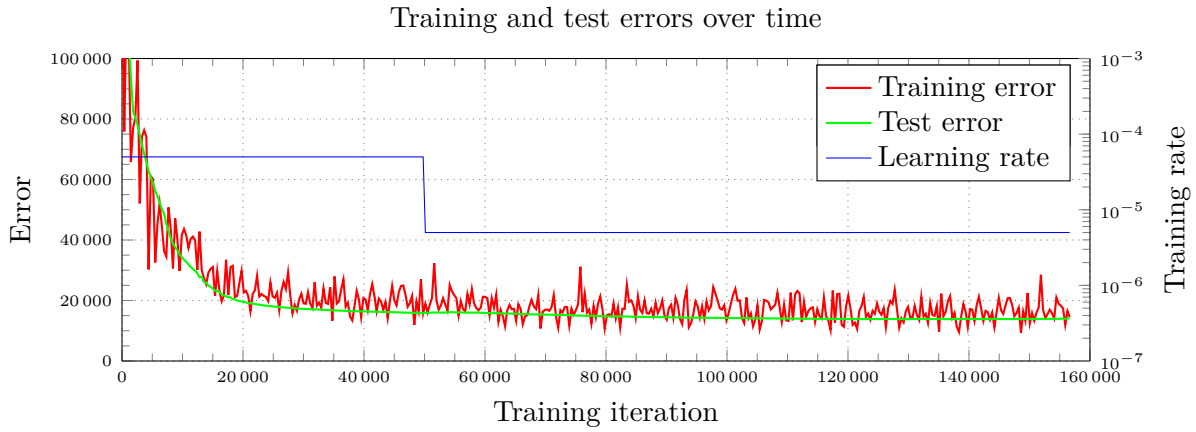
Results

4.1 Neural Network

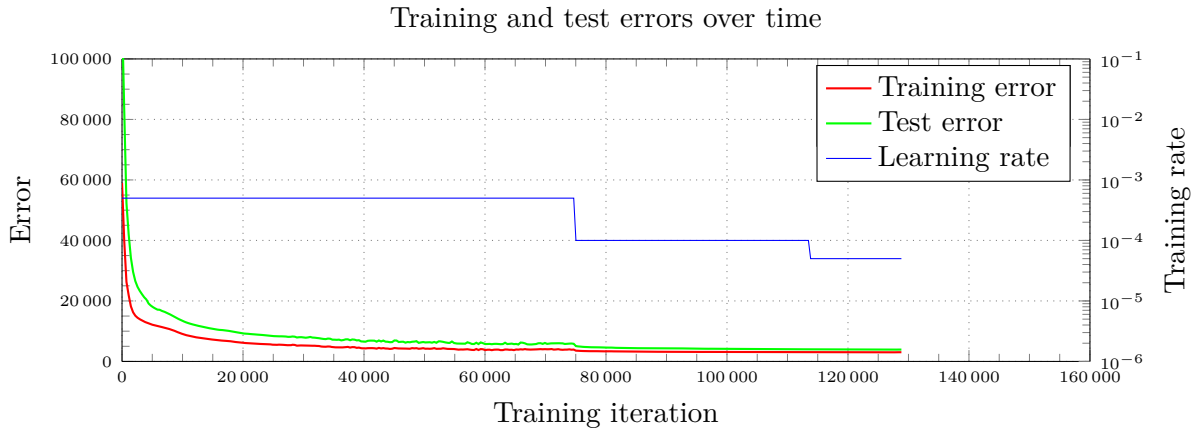
4.1.1 Training

To evaluate the performance of the learning process we first graph the training and test errors over the training iterations. Figure 4.1 shows the changes in these quantities over time. During

Figure 4.1: Plot of the training process with training error in red, test error in green and learning rate in blue



(a) Plot of the training process with 10× weighting factor on the depth error.



(b) Plot of the training process with 10× weighting factor on the depth error and no noise and shuffling on to the input.

Figure 4.2: Second training run with 10× weighting factor for the depth channel error component for networks with and without noise applied to the input.

the training procedure, the network performance is evaluated every 366 iterations by calculating the mean error over the entire test set. At the beginning of the optimisation both the test and training errors decline quickly to below 20 000. From this point on, the test error decreases more slowly with the training error oscillating between 10 000 and 20 000. After roughly 30 000 training iterations the test error starts to suddenly increase above 20 000. This is an indicator that the solver is starting to overfit the training set. From this point on the test error keeps growing steadily until we aborted the training process after 45 000 iterations.

The lowest test error with a value of 8810.19 is encountered at iteration 26 352. However, this network performed badly at reproducing the depth channel at its output. We therefore repeated the training process with a 10× weight factor applied to the depth error. Additionally, we trained a version of the network with the weighted depth error that does not add noise to the input. The graphs for the respective learning processes are given in Figure 4.2.

Figure 4.3: Filters of first convolution layer. Top row RGB, center row depth, bottom row IR. Each filter is 3x3x5 in size, 1 pixel black border added for easier separation of filters.

In Figure 4.2a we see both the training and test errors over time for the case with noise applied to the input. We started the optimisation process with a lower training rate than before and dropped it again after 50 000 iterations. This resulted in a smoother test error than we originally obtained. The error decreases quickly for the first 20 000 iterations, albeit much slower than in our first training attempt, due to the lower training rate. After this point the error decreases much more slowly and remains basically constant with minor fluctuations after 120 000 iterations. The lowest error of 13 777.6 occurs at iteration 125 904. This seems much higher than before, but after taking into consideration that the depth error is weighted by a factor of 10, we get a real error of $7237.3 + 533.8 + 1202.5 = 8973.6$ which is in line with our previous result.

Without adding the noise we obtained different results. The most obvious difference we can see in Figure 4.2b is that the training error is always lower than the test error, which is not the case in the two previous graphs. We dropped the learning rate the first time after 75 000 iterations and again after 113 500 because the error either rose or no longer fell shortly after these points. Yet again we observe an initially fast decline of the error that eventually slows down. After 128 832 iterations we stopped the process as the error no longer decreased. The error at the lowest point is 3860.18 which corresponds to a value of 2029.60 when the depth error is weighted normally.

Figure 4.3 shows a visualisation of the filters of the first convolution layer for the first network at the iteration with the lowest error. Filters of later layers are hard to visualise as they operate on the already high dimensional output of the previous layers. We can see that the depth and infrared channels produced filters that detect mostly transitions from light to dark areas as half of the pixels in most filters appear black. In the colour channels we see more non-zero pixels and mostly strong colour tones with green tones dominating.

4.1.2 Encoder-Decoder performance

To look at the encoder-decoder performance more closely we have graphed the distribution of the errors over the test set in Figure 4.4. The results were produced with the weights which produced the lowest test scores during the training of the respective networks. This gives us an overview of how well the network performs for different images. We want the error to be more or less constant, indicating that all inputs can be encoded equally well. A high tail in the distribution would mean that there are images that do not encode well.

From the graph we can observe that the variance around the mean error is not very high. The minimum error is 5327.47. The error stays below 14 000 except for one image that has an error of over 32 000. If we look at the individual contributions to the total error we can see that the depth and infrared errors are very consistent, except for the last image, while the colour channels seem to have the largest effect on the error sum.

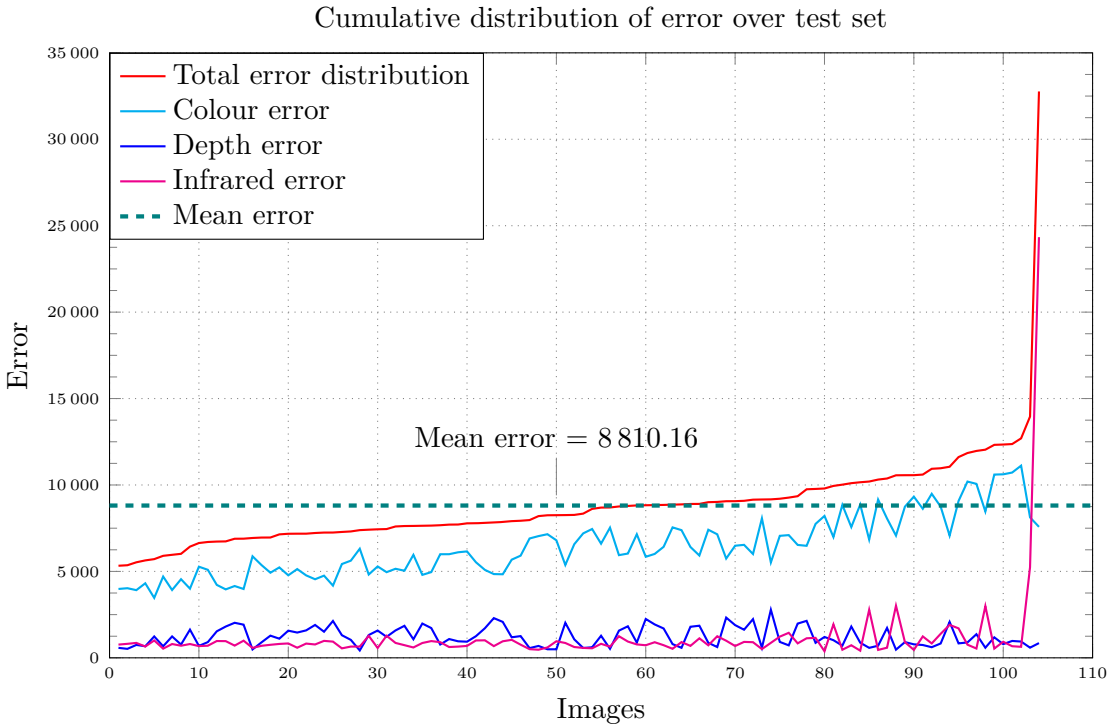


Figure 4.4: Distribution of the input to output errors over the test set. The cyan, blue and magenta curves show the separate colour, depth and infrared errors while the red curve shows the sum of the individual errors.

For comparison, Figure 4.5 shows the same distribution for the network trained without added noise and with a 10× weighting of the depth error. The shape of the distribution is very similar to the one above but the error values are lower. Contributing most to the total is still the colour error which remains well above the other channels except for the extreme spike at the end. The depth error is further more considerably smoother and also lower relative to the other channels than in the previous example which is an expected result of the extra weighting factor applied to it.

To gain more insight into what kind of images can be encoded well by the network and which ones are not processed well we generated the outputs for the inputs with the three lowest and highest errors in figures 4.6 and 4.7 respectively. The images were produced by the network trained with input noise and no extra weight factors.

In the top three images we can see that the network manages to reproduce the overall structure of the input. Especially brightness information in the RGB image is reconstructed fairly accurately. Finer details like differently coloured single leaves or blossoms are lost. Interestingly, the black regions where the image is missing data are reproduced really well. The depth image seems to contain brightness variations leaked from the colour channels that are not present in the input data. Similarly, the infrared channel is missing most information.

This becomes even more obvious when we look at the three worst examples. The top row

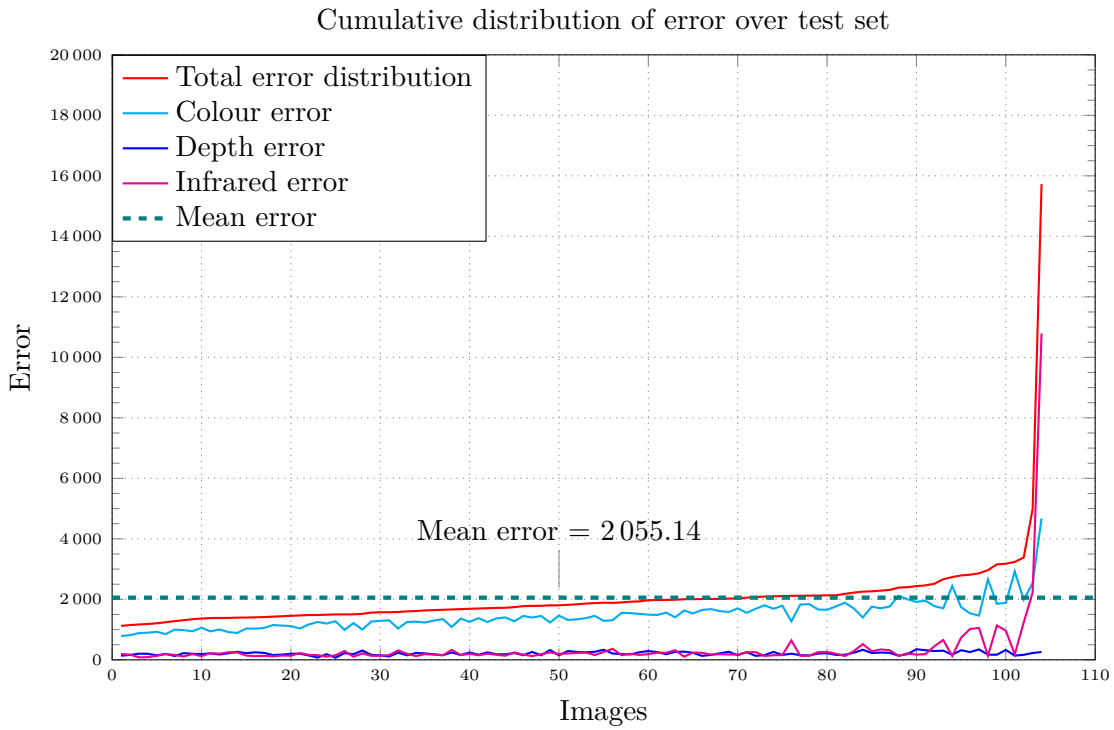


Figure 4.5: Distribution of the input to output errors over the test set for the network trained without noise and 10× weighting of the depth error. The cyan, blue and magenta curves show the separate colour, depth and infrared errors while the red curve shows the sum of the individual errors.

shows our outlier with the extremely high error. The colour and depth channel errors are in line with the other images, but the infrared channel alone has an error of over 24 000. When we look at the image we can see that the hedge in the foreground of the image is very close to the camera and the infrared image is therefore largely overexposed. These white regions are missing in the decoder output which results in the large error score. This can also be seen in the second image which shows a similar pattern in the infrared input and output.

4.1.3 Network Output

The data we are interested in for further processing is the activation of the last convolution layer `conv4_3` after applying the ReLU non-linearity. This convolution layer has 512 filters and, due to the prior downsampling operations, an output resolution of 64 by 44 pixels. This gives us a 512 dimensional feature vector for each non-overlapping 8×8 patch in the input image.

Figure 4.8 shows an example of what some of these filter responses look like. The filters are shaded to indicate activation of the pixels relative to the global minimum and maximum of the layer. We can see that only very few filters contain areas with high activation values. The highest level encountered is 6.37 which is over six times higher than the maximum network input value of 1. The least active filters contain only few pixels that are non-zero, mostly in the outer regions of the frame.

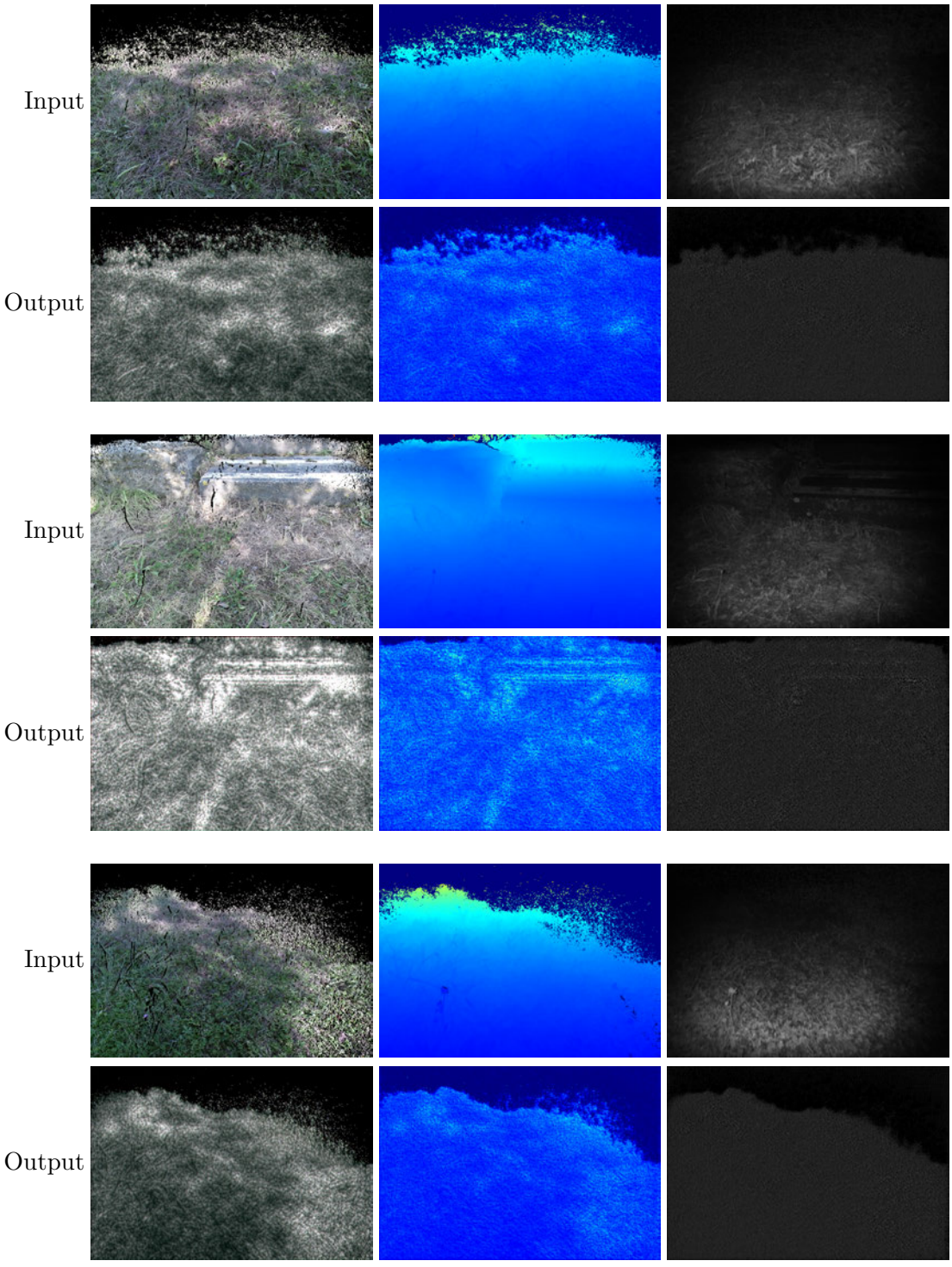


Figure 4.6: Top 3 lowest error test inputs and corresponding outputs

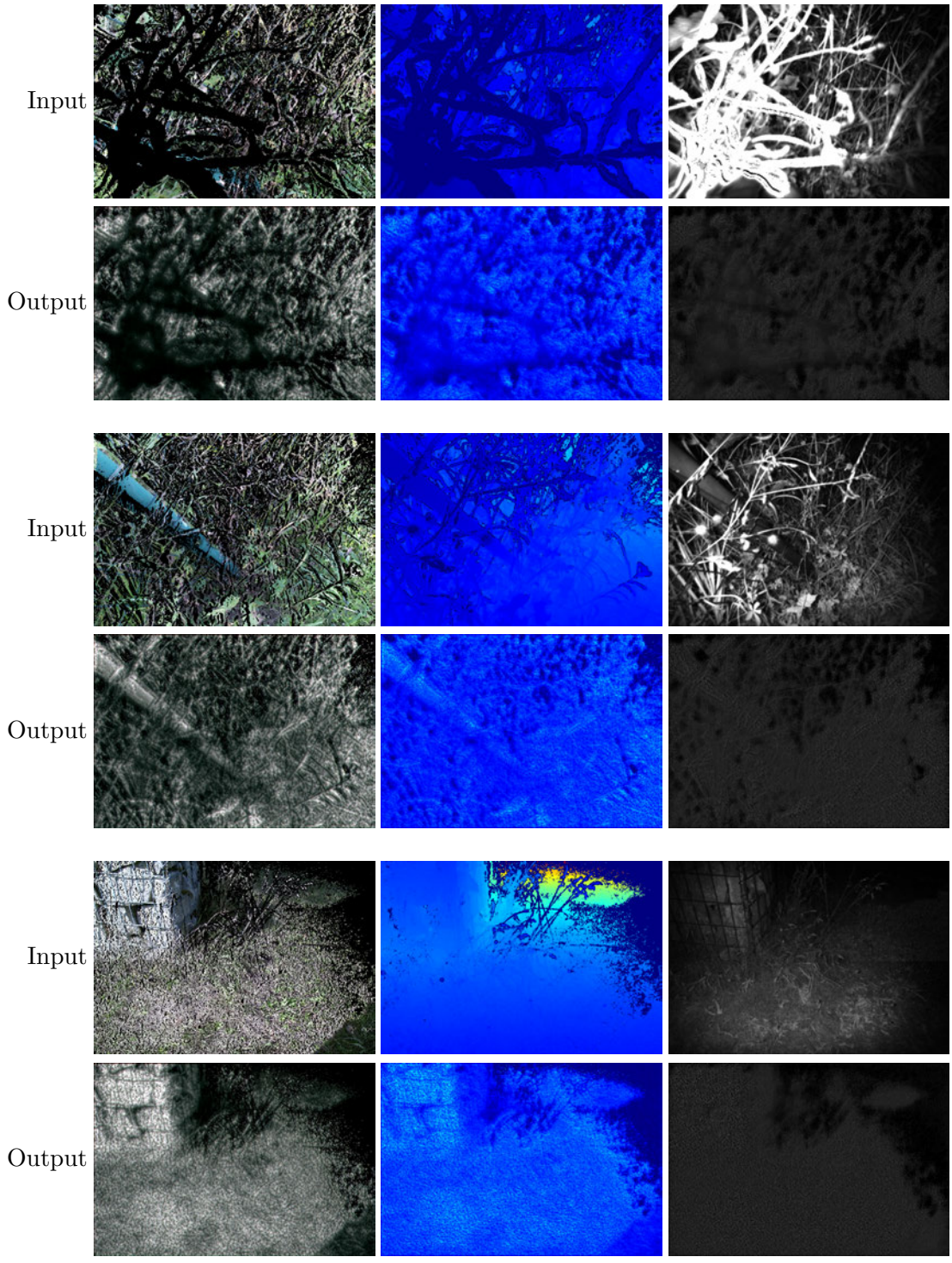


Figure 4.7: Top 3 highest error test inputs and corresponding outputs

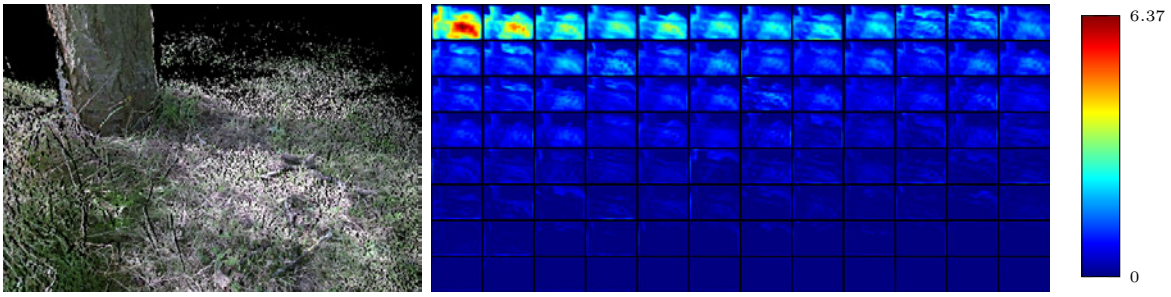


Figure 4.8: Partial neural network output for test image on the left. The channels are ordered by mean activation level and show examples throughout the range of activation levels.

Most of the very active areas seem to correspond to the very bright sections in the input. Some of the filters with lower activation around the edges seem to map to areas with high noise in the picture. There does not appear to be any direct connection between certain filters and terrain types.

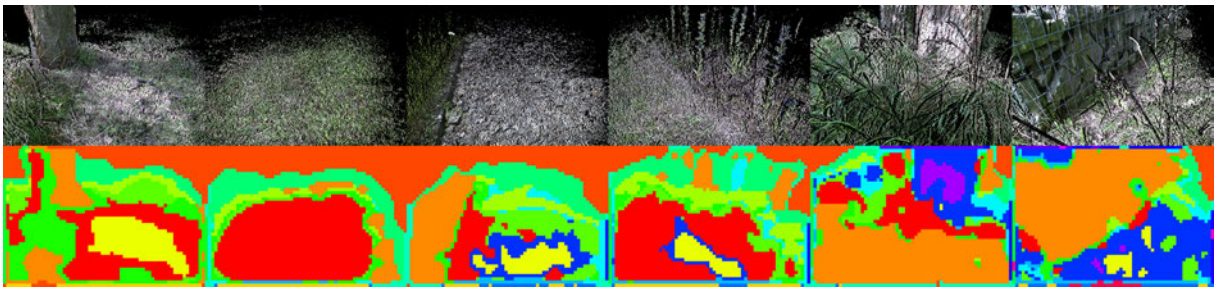
In terms of performance, the network takes 89.65 ms per image to perform a full encoder-decoder pass on our Nvidia GTX 970 GPU. Since we only need the encoder stages to obtain our network output, we can reduce this time to 27.99 ms by removing the decoder stages from the definition. These numbers are low enough to process images at over 30 Hz, which is the maximum rate our camera can produce. Our laptop does unfortunately not have a dedicated GPU and has to run the network in CPU mode on its Intel Core i7-4500U processor which results in a forward pass time of 2165.73 ms per image for the encoder stage. This is too slow for any real-time applications and we therefore recommend using hardware with a dedicated graphics card for such tasks. For comparison, the Intel Core i7-2600K in our desktop achieves a forward pass time of 1797.19 ms per image which is slightly faster than the laptop but still almost two orders of magnitude slower than the GPU accelerated runs.

4.2 Segmentation

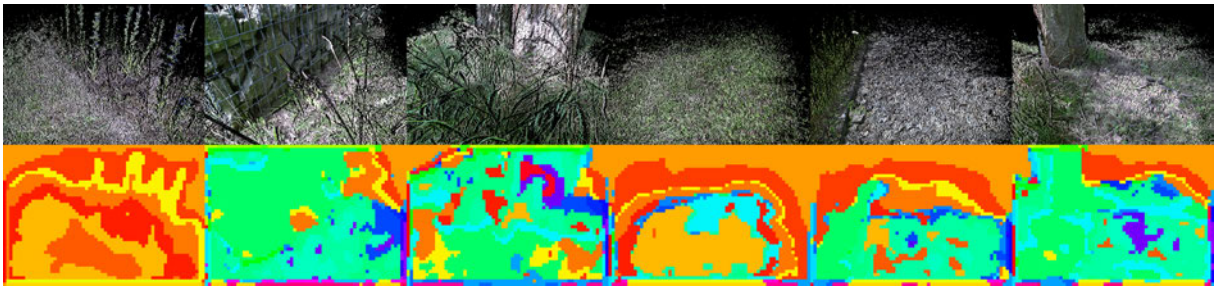
We tested our clustering strategies by applying each of them to the same set of images and observing the resulting output. We especially search for instances where segments of different terrain category are incorrectly grouped to the same segment id and instances of similar terrain being split up into multiple segment ids both in the same image and across multiple images. Both of these cases indicate poor performance in the algorithm. We use the network weights trained with input noise and without any extra weighting of the depth channel in these tests as they performed the best for segmentation.

4.2.1 Fixed `ClusterBuilder`

The first implementation we tested is the Fixed `ClusterBuilder`. Figure 4.9b shows the output of this segmentation strategy for the Distance First Match search strategy. All in all, there were 23 different cluster IDs created. The leftmost image shows the result of the k-means initialisation step of the algorithm. The dark right side of the tree is correctly separated from the surrounding



(a) Output of Fixed ClusterBuilder on example images with Distance First Match search strategy



(b) As above, but with different order of images. Colour coding of the segmentation output of the two images is unrelated.

Figure 4.9: Segmentation results obtained from the Fixed ClusterBuilder implementation with Distance First Match search strategy for a set of test images in two permutations of ordering. Upper image half shows the input picture, lower half the produced segments. Same colour labels across images indicate same segment id.

grass, but its left side gets grouped together with the taller grass on the left edge of the frame. There is also a region in the foreground that is incorrectly paired with the tree trunk. The low grass is split into three clusters: One corresponding to the bright, sunny area, one mostly matching the regions in the shadow, and one surrounding the others while bleeding into the tall grass to the left. Towards the top, where the input data gets more noisy, new segments are visible. Notably, one segment seems to closely fit the black regions at the top of all input images. On the lower border, as well as on the left and right edges of the frame we can see linear structures of segments mostly not belonging to any of the other clusters.

In the subsequent images the segments are found by either finding a match in the already existing clusters or, if no match is found, by creating them when a new region is encountered. The second input image shows flat ground with grass on it. This resulted in one dominating cluster in the output that corresponds to one of the grass clusters in the first input image. Towards the borders, a ring-like structure is visible along the areas of increasing noisiness, again similar to the first image. The cluster IDs in these bands are also found in many of the subsequent input images.

The third image shows a stony ground lined by a curbstone and grass to the left. The stones in the foreground are assigned a new label shown in blue, however further in the distance and in the middle of the blue region they share clusters occupied by grass in the previous images. The curbstone and grass on the left share a cluster ID with the tree from the first image while the

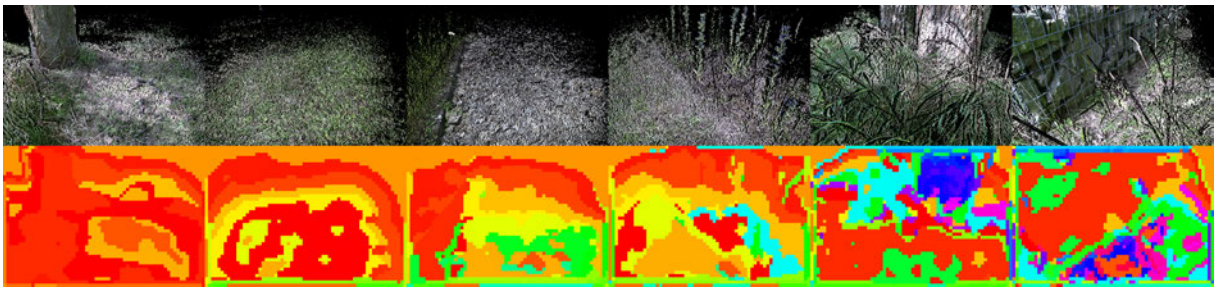


Figure 4.10: Output of the Fixed ClusterBuilder implementation with the Distance Best Match search strategy

stones next to them are grouped with the large patch of grass from image two. The next image is dominated by the red grass segment, but one of the low grass areas gets grouped together with the stones from the previous image. The taller plants to the right are partially grouped with the tree from image one and partly with the noise transition regions of the previous images.

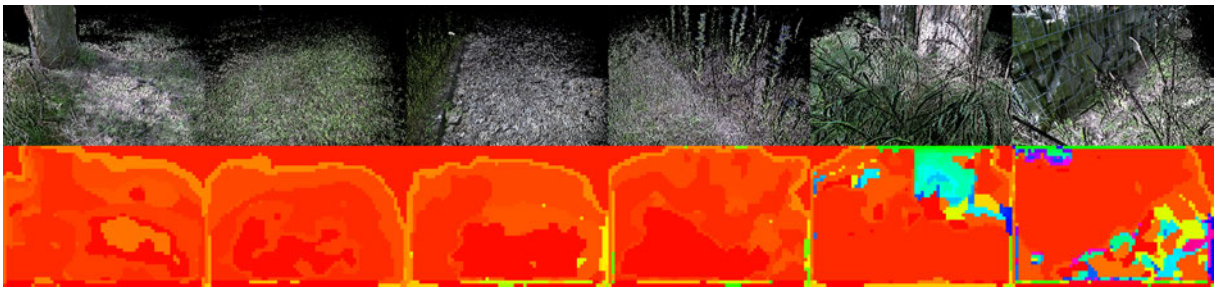
In the second image from the right we have two trees standing in the background, behind tall grass. The bright region on the right tree is represented by a new cluster id shown in purple, while the dark regions of it and some spots to the left are grouped together with the stony ground. The left tree and the region between the trees and the foreground are grouped with grass. Grouped together with the tree from the first picture is the tall grass in the foreground and the very right edge of the right tree.

Finally, we can see the algorithm segmenting the stone wall from the ground quite well in the last image. The wall is however incorrectly grouped together with the tree from the first image and the tall grass from the previous image. The rest of the image is mostly merged into the group corresponding to stones in image three.

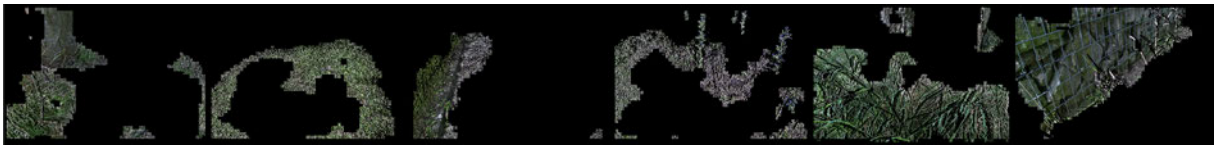
Processing the images in a different order produces mostly similar results which are shown in Figure 4.9b. Notably the colours between the two runs are entirely unrelated as we merely colour clusters in order of creation to visually separate the areas. We can immediately see that the stone wall, tall grass in front of the two trees and the single tree are again grouped into a common cluster. Also present is the ringlike structure toward the noisy outer regions. The differences lie mostly in the smaller regions such as the bright grass area in the rightmost image, which is this time assigned its own cluster. This time there were 56 clusters created, although many are only encountered in very small regions or along the edges.

When we change the cluster search strategy to Distance Best Match, we obtain different results, which are shown in Figure 4.10. The most obvious difference is an increase in cluster count at a number of 72. The leftmost image again shows the result of the k-means initialisation and has similar, but not equal structure to the output in Figure 4.9a. The tree is again merged with the tall grass on the left and there is some differentiation between brighter and darker grass regions. The stones are again partially grouped with the grass while the curbstone is paired with the tree from the first image. In the second to last image we can again observe the high grass in the foreground being grouped with the aforementioned tree cluster, together with large parts of the stone wall in the last image.

The background of the image is however split into many more small new groups than before



(a) Output of the Averaging ClusterBuilder on example images



(b) Input images masked to show only regions matching one of the classes from above

Figure 4.11: Averaging ClusterBuilder example output and example of one class

and the same can be observed in the foreground of the rightmost image. The right tree gets assigned its own class this time, which can also be found in some small areas on the floor in front of the wall.

4.2.2 Averaging ClusterBuilder

Next, we test our Averaging ClusterBuilder implementation which derives the comparison limits from mean distances to the cluster center. Figure 4.11 shows how this algorithm performs on our test images with the Distance First Match search strategy. The order of the images is the same as in Figure 4.9 to better show the differences in the outputs. The algorithm produced 106 different clusters. The most common ones are the clusters created by the k-means initialisation which cover basically all of the first four images and most of the latter two as well. The initialisation looks similar to the previous cases which is to be expected, as it is performed identically in both instances.

Again we can see from the algorithm output and the visualisation in Figure 4.11b that the stone wall, the single tree and the tall grass from the second to rightmost image are grouped together. Also in this group we find parts of the short grass and the curbstone, as well as parts of the bigger plants in the fourth image from the left. The visualisation emphasizes a similarity in colour of most of those regions which all share relatively dark green and brown tones. The stones in the third picture are grouped together with the low grass in the fourth picture as well as some of the grass in the first and second pictures. Also visible is the ring-like structure towards the noisy regions that we could already see in the previous algorithm.

The last two images are split into many small clusters in the regions not belonging to the big group described above. The right tree consists of multiple smaller groups that are either not shared with structures in the other images or only share small regions with the ground in front of the stone wall in the next picture. Similar observations can be made for this region which is also split into many smaller chunks with unique cluster IDs.

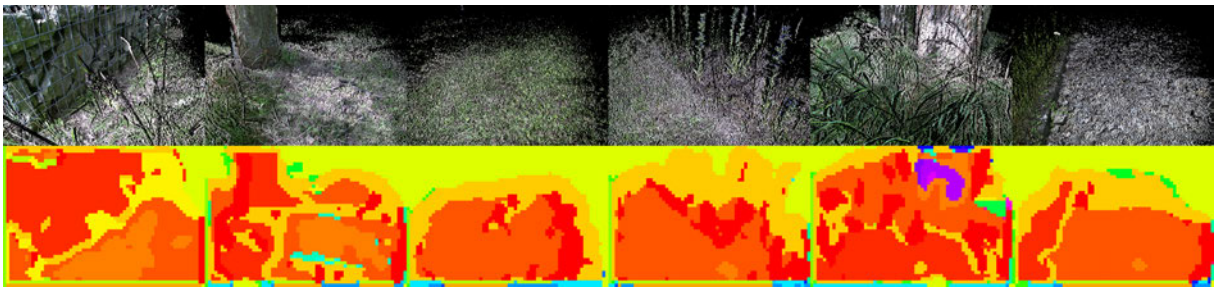


Figure 4.12: Averaging ClusterBuilder output for different order of input images

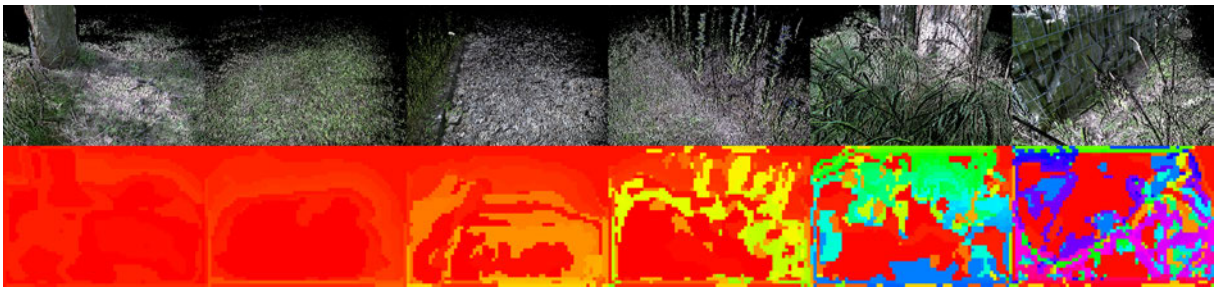


Figure 4.13: Averaging ClusterBuilder output for the Distance Best Match search strategy

Changing the order of the images yields quite different results. Figure 4.12 shows the output when the algorithm is applied after shuffling the test data. This time, only 37 different clusters were created.

We again have an overlap between the stone wall, the single tree and the tall grass in front of the two trees, but this time only the curbstone and none of the grass regions are part of this group. A second large group consists of most of the flat grass areas and the stone ground. Some of the brighter grass regions in the sun in the first two images share a group with parts of one of the trees. Another group contains the tall plants in the fourth image as well as parts of the noisy regions from the other images, but also transitions between vertical and flat structures. There is also a small cluster only containing parts of one of the trees in the second rightmost image.

The ringing effect is less pronounced this time with a single group covering the black areas and most of the very sparse regions at the top. The second ring is the aforementioned group containing the tall plants in image number four.

If we switch our search strategy to Distance Best Match, we obtain the results shown in Figure 4.13. Right away we can see a substantial number of clusters being created, 302 in total. Yet we can still see the grouping of tree, stone wall, and tall grass from the previous examples. Most of the clusters are again present in the last two images. The last two images especially account for the majority of clusters and show very fine segmentation. They share very little with the previous images apart from the aforementioned stone wall and high grass structures.

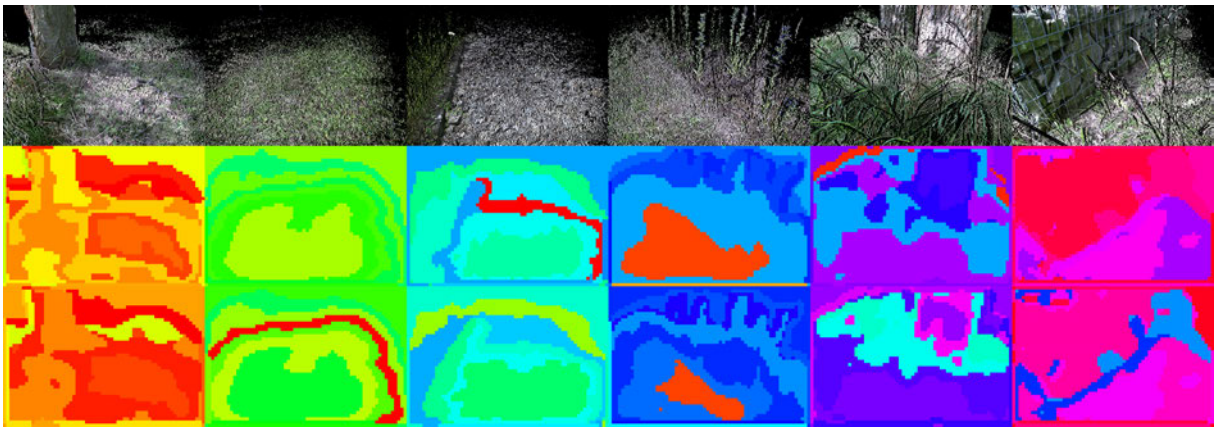


Figure 4.14: k-means `ClusterBuilder` output for Distance First and Best Match strategies respectively

4.2.3 k-means `ClusterBuilder`

The third implementation we tested is the k-means `ClusterBuilder`. It runs the k-means algorithm for every frame and tries to match the new clusters to the already existing ones. In Figure 4.14 we can see the output this produces for the Distance First Match and Distance Best Match search strategies respectively. The approaches generate 45 and 47 unique clusters respectively. Since every image is divided into ten clusters by the k-means algorithm, the maximum number of unique IDs possible in our test case is 60.

We can see that the individual images are generally segmented relatively well. For example, the stone wall is separated from the ground and the gravel pit is separated from its surroundings. We can also see the ringlike structures again, especially in the second through third images. There is however not much grouping happening between frames. Only some grass in the third image is assigned to the same cluster as grass regions in the first image in both cases. We also don't observe a common class for the black regions at the top with missing data which was always present in the previous examples.

When we apply the Angle Similarity and Cosine Similarity search strategies, we obtain different results. This is shown in Figure 4.15 where we can see the output for both search strategies. The immediately visible difference to the distance based approach is the reduced number of clusters that were produced. We now only produce 20 and 24 groups respectively. We can still see decent segmentation of the individual frames, but this time we get more matching groups between images.

With the Angle Similarity search strategy we get two groups of grass that are shared in the first two images and the stone wall is grouped with the tree of the first image and the grass foreground in the fourth image. The third image showing the gravel pit did not segment well this time and we can not see a distinction between the rocks and the grass and curbstone on the left.

Common clusters are also found with the Cosine Similarity search strategy. The largest group consists of some grass from the second image which is grouped with the rocks in the third image, more grass in the fourth image, the right tree in the fifth image and the foreground of

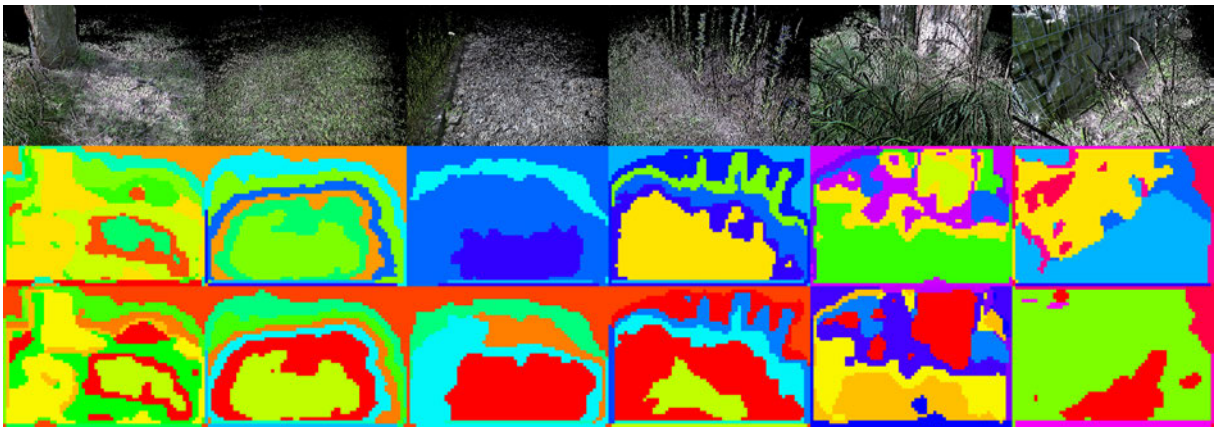


Figure 4.15: k-means ClusterBuilder output for Angle Similarity and Cosine Similarity search strategies

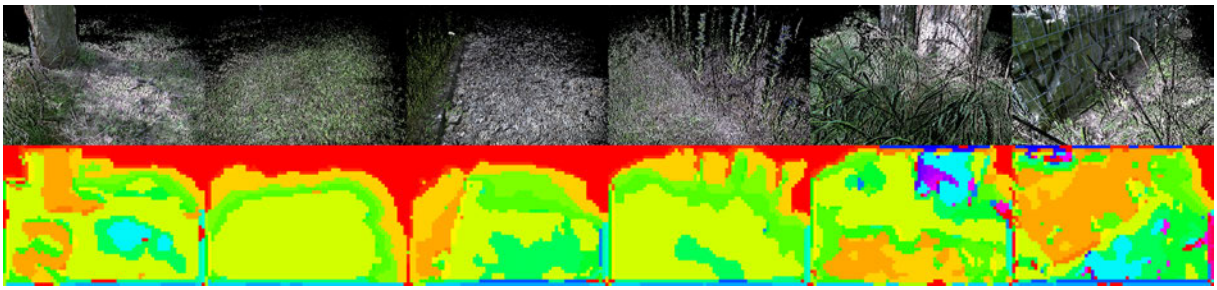


Figure 4.16: Simple ClusterBuilder output for Distance First Match search strategy with a radius limit of 6.0 and rejection limit of 10

the last image. Also grouped together are some grassy regions from the first, second, and fourth images. We can also again see a common group for the black regions with missing data at the top of the images, with the last two being the exception. Segmentation performed the worst in the last image this time, where only three clusters remain if we disregard the border regions. The wall cluster also covers part of the foreground this time.

4.2.4 Simple `ClusterBuilder`

The final implementation to consider is the Simple ClusterBuilder. Figure 4.16 shows this implementation's output with the Distance First Match search strategy and a radius limit of 6.0. It produced 36 unique clusters for our input images. The images show a dominant group that contains the grass from the second image along with parts of the grass in the first and fourth images as well as parts of the rocks in the third image and the middle area of the fifth picture. Some of the borders of the stone wall are also part of this group. We also encounter a group that contains most of the stone wall, parts of the tree in the first image, the curbstone of the third image and parts of the foreground tall grass in the fifth image. The right tree in the fifth image largely shares a cluster with sunny grass spots in the first image and some of the

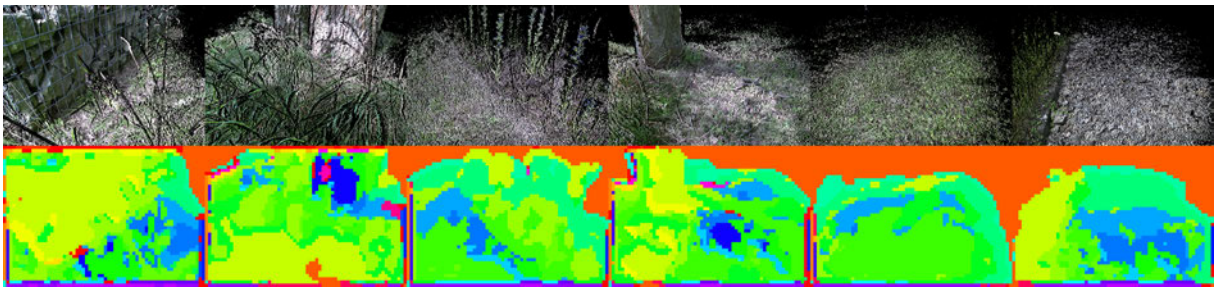


Figure 4.17: Simple ClusterBuilder output for Distance First Match search strategy with a radius limit of 6.0 and rejection limit of 10 with different order of input images

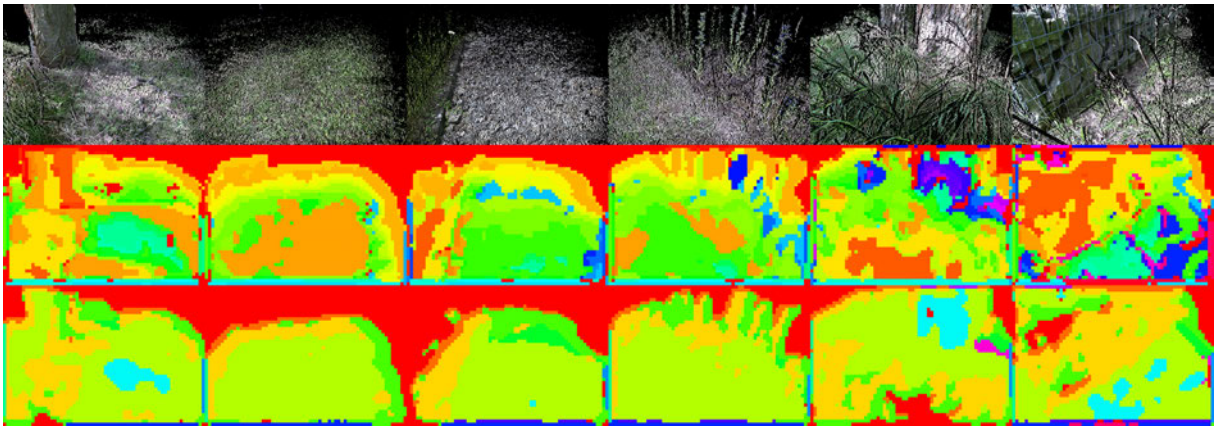


Figure 4.18: Simple ClusterBuilder output for Distance First Match search strategy with radius limits of 4.5 and 8.5 and rejection limit of 10

ground in the rightmost image.

The black and noisy regions share a common group in all of the images and the ring-like structures are less pronounced than in some of the previous examples. We can also see decent segmentation of objects such as the stone wall or the curbstone in the images.

When we change the order of the input images, we obtain the results shown in Figure 4.17. The number of clusters is now slightly lower at 34 and the structure of the output stays mostly similar. We can see differences mostly in the grass areas which are now split up slightly more than before. The decent separation of the wall and other large structures remains present.

Changing the radius limit to lower or higher values has a noticeable impact on the generated clusters. As can be seen in Figure 4.18, lowering the limit to 4.5 produces significantly more individual clusters, 67 in this case. This is most noticeable in the last three images where some areas like the tree or the ground in front of the stone wall show very high numbers of small groups. We can however still see many of the common groupings from before.

Increasing the limit to 8.5 has the opposite effect and less unique clusters are created. This time we only see 14 of them which naturally leads to a coarser segmentation in the individual images. The dominant group of grass and stones from the first example now covers almost all flat areas and the middle regions of the fifth image. We can still see some decent segmentation,

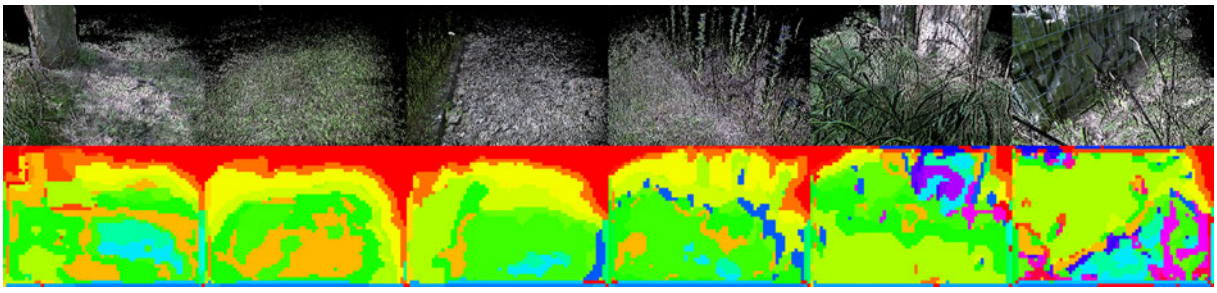


Figure 4.19: Simple ClusterBuilder output for Distance Best Match search strategy with a radius limit of 7.0 and rejection limit of 10

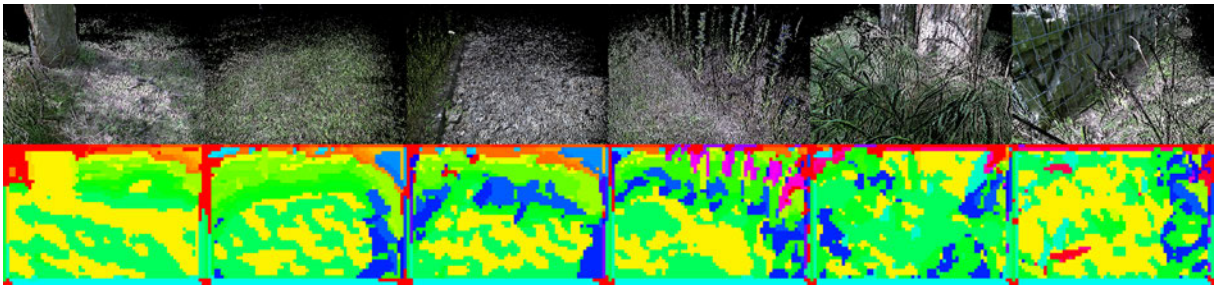


Figure 4.20: Simple ClusterBuilder output for Best Angle Similarity search strategy with a similarity limit of 0.90 and rejection limit of 10

the stone wall for example is still separated from the ground and the right tree in the fifth image is still separated from its surroundings, but the groups spanning images contain many mixed terrain types.

When we switch the search strategy to Distance Best Match we obtain the output shown in Figure 4.19. The number of created clusters is 41, slightly more than with the Distance First Match search strategy. The grass in the first image is more fragmented this time, with shadowy and sunny regions being split up. The grass in the second image is grouped with the darker grass regions from the first image, but parts are also grouped with the rocks from the third image, which share a large group with the grass in the fourth image. The stone wall is again well separated from the foreground and shares a group with the tall grass in the fifth image, the curbstone and parts of the tree in the first image. The other tree is also again separated from its surroundings, but shares groups with some of the bright grass areas in the other images. Most individual classes are seen on the tree in the fifth image and the ground in the rightmost image. We can also clearly see the group covering the black areas in all of the frames.

With the Best Angle Similarity search strategy, the results we obtain are quite different. Figure 4.20 shows the output we produced with a similarity limit of 0.90 and a rejection limit of 10. This produces 56 clusters in total. The most visible feature of the output in contrast to the previous ones is the complex interleaving of groups which is very prominent in the second and third image, as well as the last image.

The two most common groups are the ones coloured in yellow and green. The yellow cluster contains the tree from the first image as well as roughly half of the grass in the following images,

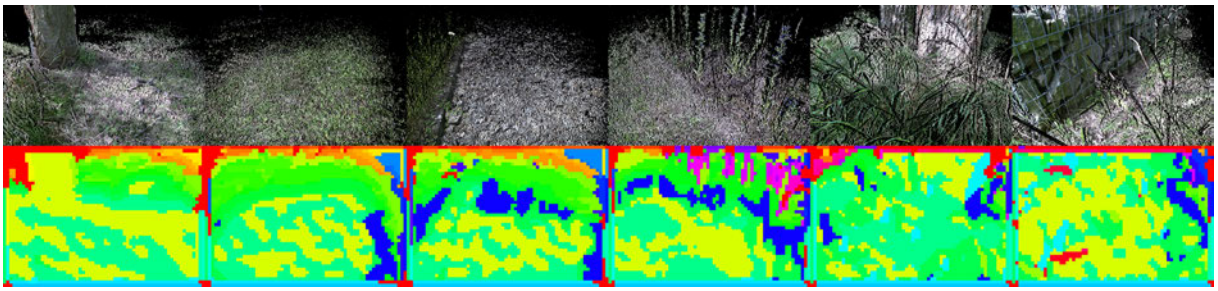


Figure 4.21: Simple ClusterBuilder output for Best Cosine Similarity search strategy with a similarity limit of 0.95 and rejection limit of 10

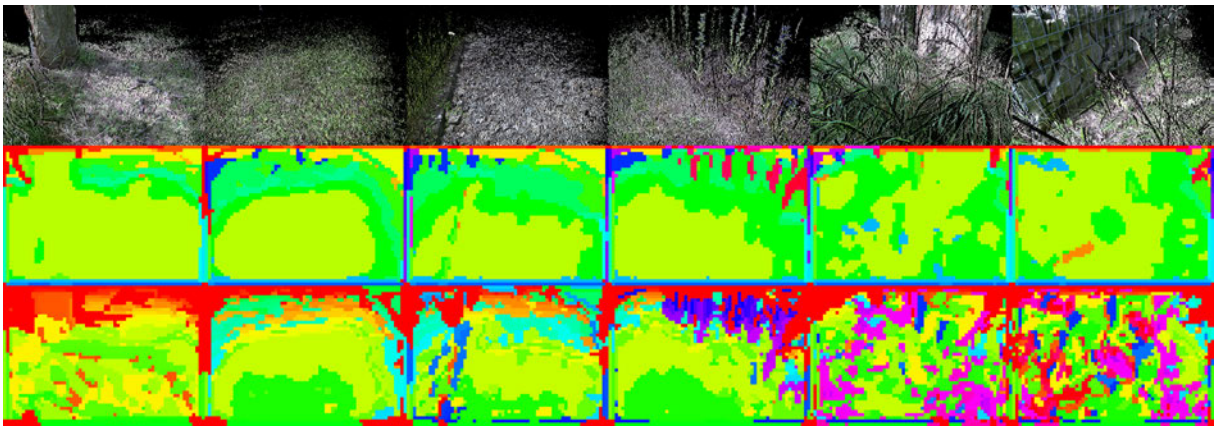


Figure 4.22: Simple ClusterBuilder output for Best Cosine Similarity search strategy with similarity limits of 0.91 and 0.98 and rejection limit of 10

a part of the rocks in the third image, the tree in the fifth image along with some other regions in this image and finally most of the wall and foreground in the last frame. The green cluster contains most of the remaining grass and rock regions as well as most of the tall grass in the fifth image and the rest of the stone wall.

Unlike before, there is also no clear group covering all of the black regions in the upper area of the images. Instead, multiple smaller clusters cover different parts of the noisy outer regions. Segmentation of the individual images is also worse than for the other search strategies. None of the big features like the wall or trees are clearly separated from their surroundings.

Similar results are obtained with the Best Cosine Similarity search strategy, shown in Figure 4.21. We can see essentially the same structures as with the Best Angle Similarity search, i.e. two dominating, interleaved clusters that cover most areas of all of the images. The clusters cover the same image regions as before and this method also fails to properly segment the big structures in the input images.

Increasing or decreasing the limit further worsens the result. With a lower limit, most regions of all images are covered by the same large cluster, with a second one covering almost everything else. When we increase the limit, we end up generating over 100 individual clusters, mostly situated in the noisy regions of the images. There are however still larger clusters present

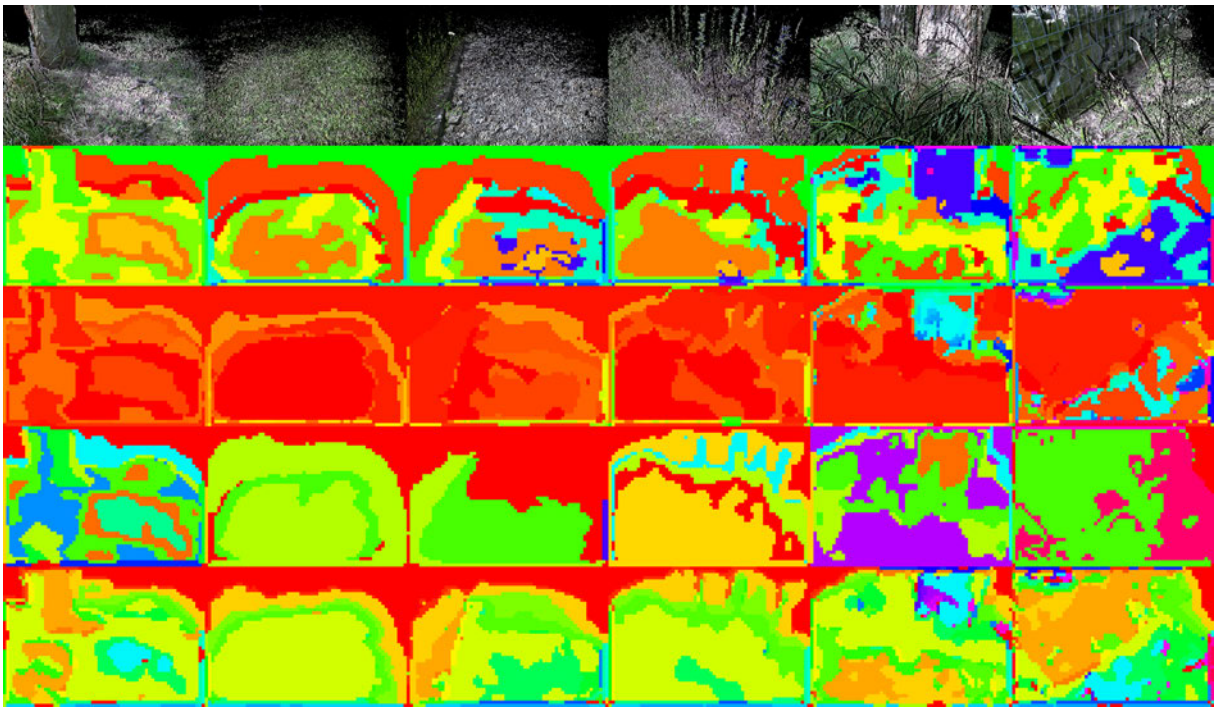


Figure 4.23: Comparison of all ClusterBuilder implementation types. Top to bottom: Fixed ClusterBuilder Distance First Match, Averaging ClusterBuilder Distance First Match, k-means ClusterBuilder Best Angle Similarity, Simple ClusterBuilder Distance First Match with 6.0 distance limit and 10 rejection limit

that cover similar, but smaller regions as above. Similarly to before, the largest group, coloured in dark green, covers almost all terrain types. In both cases, no reasonable segmentation can be made out.

4.2.5 Comparison

Figure 4.23 shows a direct comparison of the different ClusterBuilder implementations with their respectively best performing search strategy. We can clearly see the similarity in the K-means initialisations of the first three implementations. The purely discovery based Simple ClusterBuilder implementation does however not differ that much from them. We can also make out similarities in the other images such as a triangular region in the fourth image and the tree in the fifth image. Apart from the k-means implementation, we can also always clearly distinguish the stone wall from the surroundings.

Some of the grouping across images can also be found in all but the k-means implementation. One example of this is that the cluster containing the curbstone in the third image also always contains at least parts of the tall grass in the fifth image. We can also always see common groups for parts of the stone wall and the tree in the first image. Further more there is also always overlap between the grass and the stones in the first four images. Another common feature is the presence of a cluster covering the noisy and black image parts. They are found in all of the

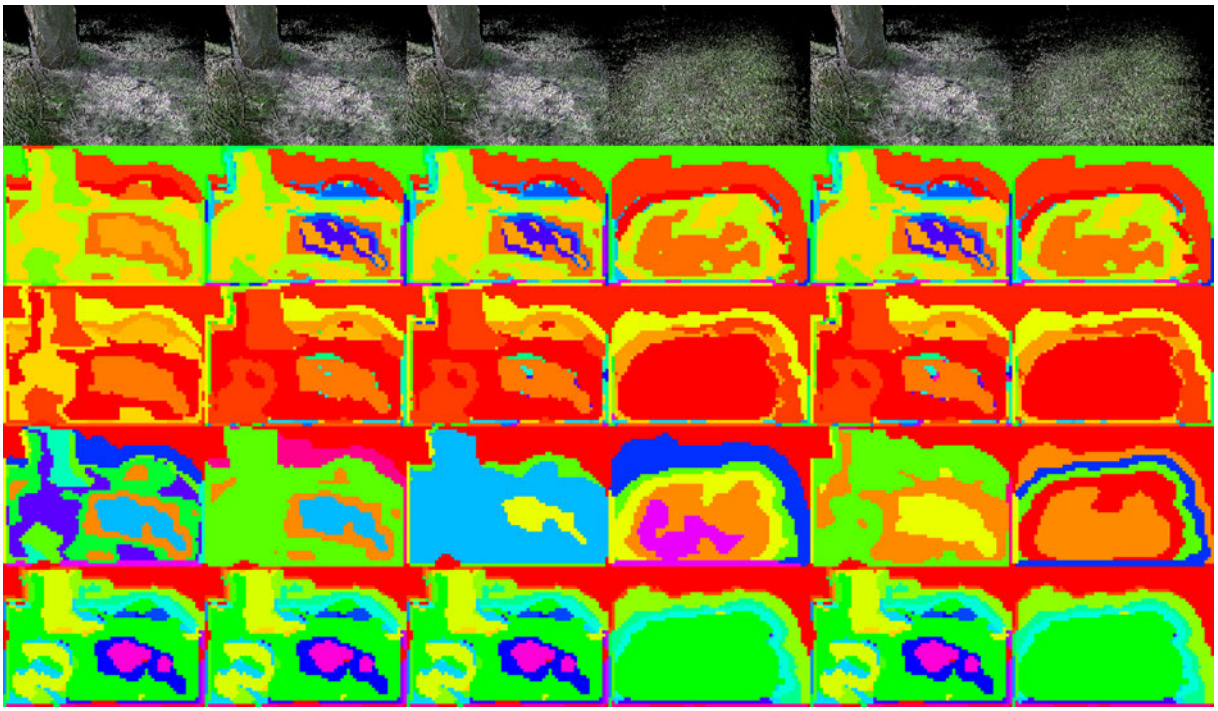


Figure 4.24: Comparison of all ClusterBuilder implementation stability. Top to bottom: Fixed ClusterBuilder Distance First Match, Averaging ClusterBuilder Distance First Match, k-means ClusterBuilder Best Angle Similarity, Simple ClusterBuilder Distance First Match with 6.0 distance limit and 10 rejection limit

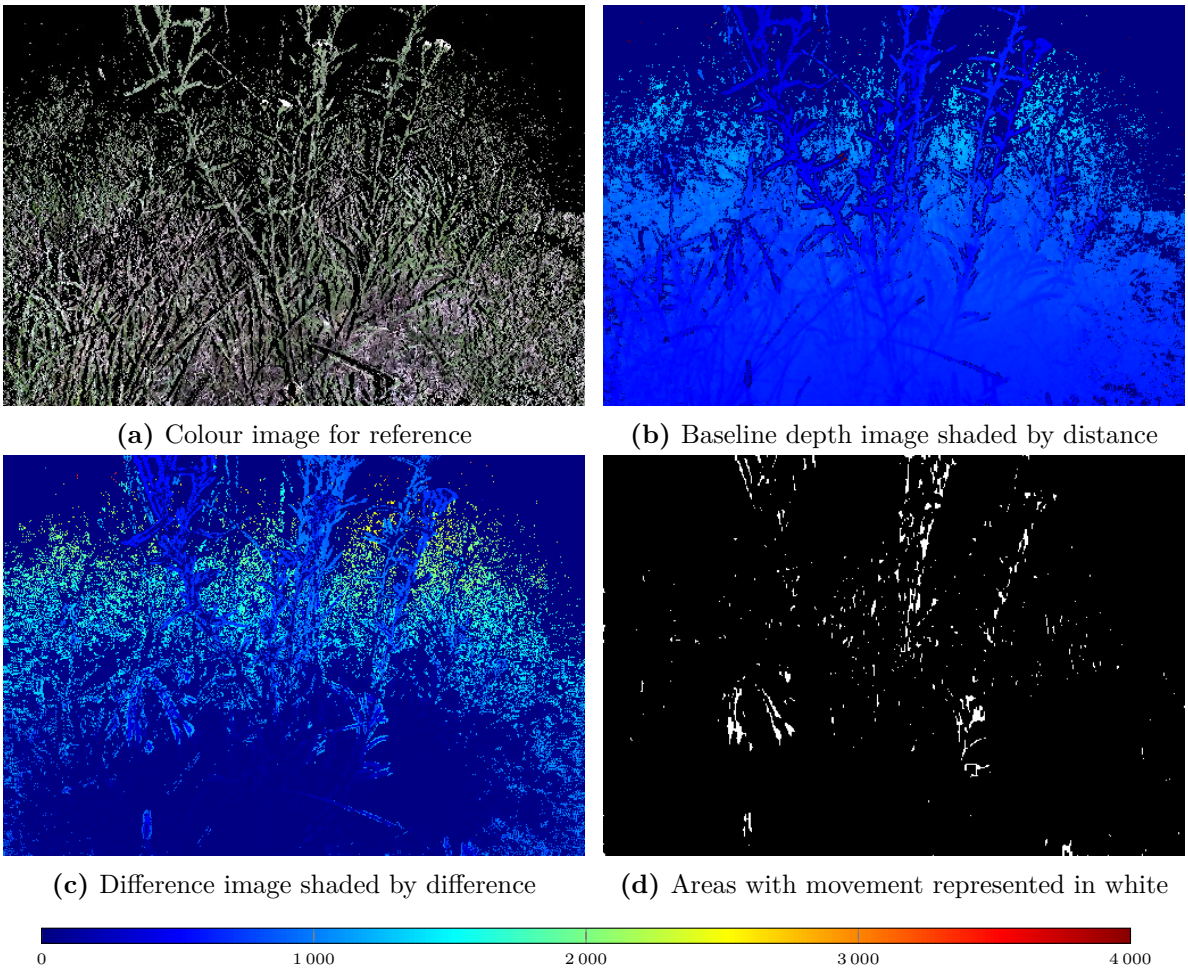
implementations and have very closely matching shapes that fit well with the input images. The exception is again the k-means implementation, in which this region often extends further into the image.

The k-means implementation stands out the most among the four as especially the last image is not segmented well. Other differences can be found in the way the stone wall is segmented. The Fixed and Simple implementations show two interleaved clusters here, while the other two implementations cover almost the entire wall in a large group. The same can be observed in the foreground of image five, where the Fixed ClusterBuilder shows the most segments.

4.2.6 Stability

To test and compare the stability of the algorithms, we applied the four implementations from the previous sections to a series of input images that consist of a repetition of the same frame three times, then a second image followed by the first image and finally the second image again. We expect the output to stay the same across repetitions of an image, even if we introduce a different frame between encounters.

Figure 4.24 shows the results of this test. We can immediately see that the k-means implementation does not produce stable output, even prior to introducing the second frame to the sequence. For the other implementations, the second encounter of the test image always



(a) Colour image for reference

(b) Baseline depth image shaded by distance

(c) Difference image shaded by difference

(d) Areas with movement represented in white

Figure 4.25: Results of movement detection applied to grass and weeds

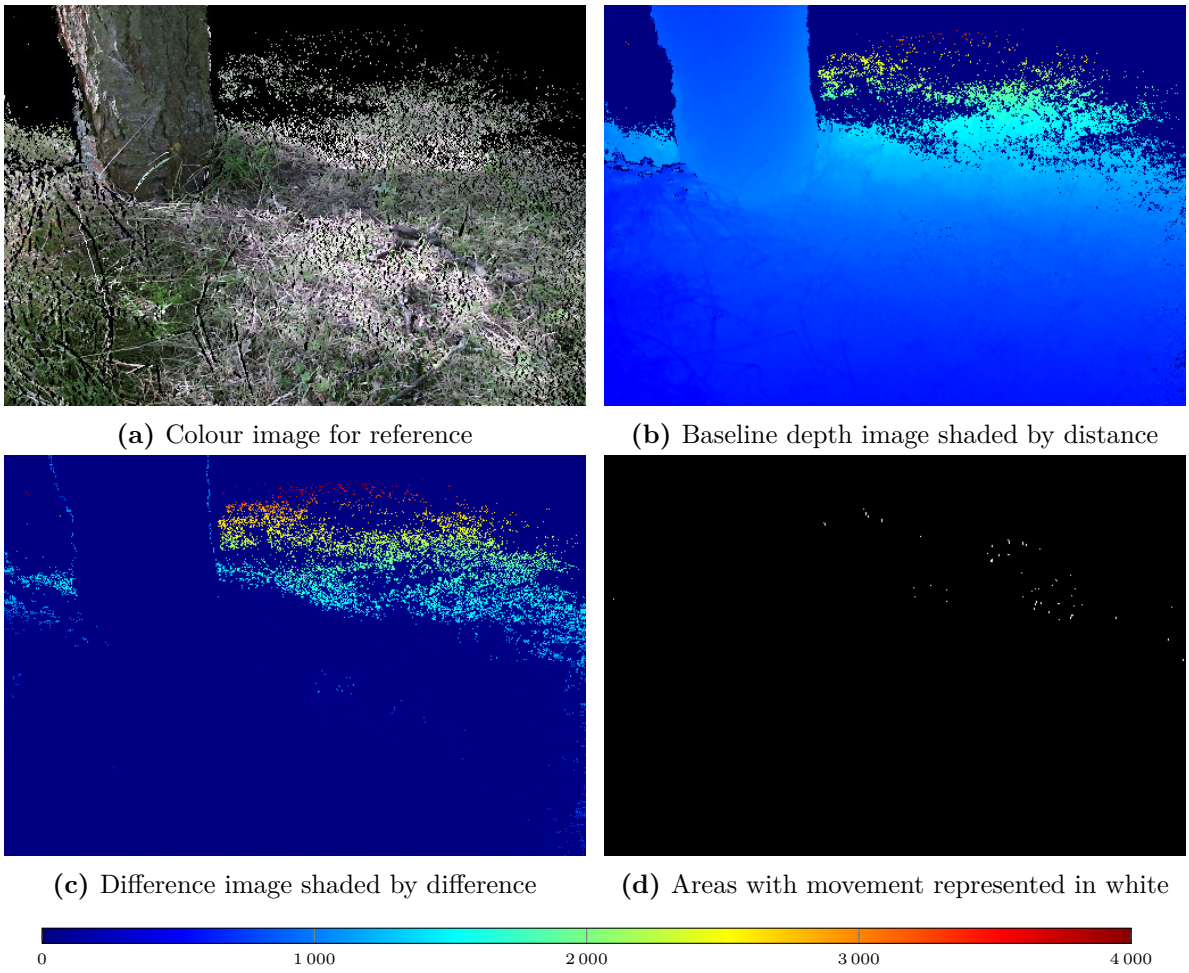
produces different results to the initialisation frame. Subsequent frames produce identical output for the Simple and Fixed `ClusterBuilder` implementations for both the original and the introduced image.

The Averaging `ClusterBuilder` implementation shows slight variations between runs both prior and post insertion of the second image. The differences are small but they can clearly be seen in the image center where a small cyan area is surrounded by a larger orange cluster.

4.3 Classification

To evaluate the performance of the movement detector, we performed experiments on different pairs of depth images captured as described in Section 3.2.4.

Figure 4.25 shows our algorithm applied to a scene containing medium height weeds and



(a) Colour image for reference

(b) Baseline depth image shaded by distance

(c) Difference image shaded by difference

(d) Areas with movement represented in white

Figure 4.26: Results of movement detection applied to tree

grass. The top row shows the baseline images in colour and as a depth image. The depth image has already been clipped here. The second depth image is omitted as it looks very similar to the baseline image. The Figure 4.25c shows the difference between the depth images. The weeds in the foreground can be made out quite clearly here. Also of note is that there is an area with a lot of high difference pixels in the upper area of the image as a result of the noisy and sparse input data there. Figure 4.25d visualises the output of the movement detection algorithm where areas in white contain movement and areas in black contain no movement.

Again, the weeds in the foreground can be clearly made out in the image. Further more, most of the noise in the background has been removed by our filtering leaving us with a usable map of areas that contain movement. This data can now easily be cross-referenced with the clusters we obtained through the clustering process to update the corresponding cluster labels to traversable or non-traversable where applicable.

To confirm that the algorithm works correctly, we also tested it on pairs of images of rigid obstacles. Figure 4.26 shows this on the example of a tree trunk. The images show the same stages in the processing flow as for the experiment above. We can see that the tree is not visible in the difference image except for a thin outline that is the result of sensor noise. Again we observe areas with large difference values in the upper area of the image where there the input data is sparse. These areas are again filtered out as expected in the final output representation.

Chapter 5

Discussion

5.1 Discussion of Results

5.1.1 Neural Network

Our network training proved to be fairly consistent. When we compare our first run with the second process that weights the depth error stronger, we see similar results. We do however arrive at the end result in different ways. The minimum was reached much quicker when we started with a higher training rate, but the error declined less smoothly. A lower training rate smoothed the curve, but convergence took much longer. Further increasing the training rate would destabilise the system and the error would not converge at all.

When we compare the effect of the noise layer on the training we can see that without it, the training error is smooth, while with noise it fluctuates significantly. Further more, removing the noise lowers the training error to below the test error. Since there is never any noise added during the network verification, it makes sense that it is lower than the training error, as is the case in the Figure 4.1 and 4.2a. Since the network directly optimises for the training set it also makes sense for the training error to be lower than the test error in the case where no noise is added.

The better test scores for the network without noise is also reasonable, since the network does not have to learn how to compensate for the missing data during training and can therefore better optimise for the input.

Examining the encoder-decoder performance over the test set gives us some insight over the processes in the network. We see issues with the replication of especially the infrared channel, especially when it differs in brightness from remaining channels. In the extreme case where the infrared input is white from overexposure but the remaining channels are black from range clipping, the network fails to reproduce the infrared input.

A possible explanation for this is the fact that in most cases, the infrared channel is not very bright and is further more just one of five channels of input. Since the colour channels do not generally contain very bright regions either, the network optimises for this type of input. The colour channels also seem to dominate the optimisation, as we can clearly make out structures from the RGB image in depth and infrared outputs.

Most of the time the colour channels are the main contributor to the total error between

input and output while the depth and infrared errors remain fairly constant. This is reasonable, as the depth images are usually fairly similar in structure and further more tend to only contain values in the lower range of the possible input range. Similarly, the infrared error stays fairly constant, with a few exceptions that are the result of the effect explained above. The colour error is naturally higher than the other two, as it is the sum of three channels, but most of the time it is more than the expected three times higher than the remaining contributors. Unlike the others it also varies over the test set.

Large colour errors appear when the input image contains large amounts of saturated colours, as those seem to be lost in the output. Only the variances in brightness are preserved relatively well. The network is able to reproduce the black regions in the input very well. This is a direct result of the optimisation, as black represents a value of 0 and is one of the extremes of the input range and usually the single most common input value.

After decoding, we therefore end up with a desaturated version of the input colour image with some noise, but largely intact structure. The patterns in the colour channels also contaminate the depth and infrared channels likely as a direct result of the optimisation for brightness differences.

The encoder-decoder performance is however only of limited interest for our application as we intend to use the result of the final convolution step, i.e. the encoder output, in our further processing. Ideally, we would have seen some direct correspondence between terrain classes and activated filters here, but this is not the case. Instead, the most active filters seem to respond to brightness variations in the input. There are also filters present that show activation in the black and dark areas of the input image and thereby encoding the position of the black and noisy regions. The weaker filters mostly show strong activation toward the edges, which we can see having an effect on the segmentation process later on. This is likely an effect of the padding used during the convolution layers.

Since we only see strong activation in a minority of the filters and the fact that those filters are usually the same ones for our test set, these have the most influence on our subsequent segmentation process. When we look at the output of the angle and cosine similarity based segmentation algorithms, we can see that these vectors usually vary very little across large parts of the image. Where we see good results are the black areas at the borders. This is consistent with our previous observation that some of the filters show strong activation in the black and dark regions of the images.

5.1.2 Segmentation

When we look at the results of our segmentation strategies from the previous chapter it is quite obvious that none of them perform well enough to be used in real-world applications. Especially the k-means `ClusterBuilder` is unsuitable as it is not stable. The instability is a result of the randomness of the k-means algorithm, which will produce slightly different results every time it is run on the same input image. We therefore obtain slightly different cluster centers every time which means that clusters in similar regions can end up being assigned to different pre-existing groups. What makes this worse is the fact that the assignment is performed on a per cluster basis as opposed to a per pixel basis, which means that if a cluster is assigned wrongly, we automatically assign a large chunk of the image wrongly. Another problem we encounter for the distance based search strategies for the k-means `ClusterBuilder` is that we seem to get

matches very rarely which means we produce new clusters frequently. In our case, only 15 and 13 matches occurred with the two strategies.

There is however also an advantage to this approach. Since we generate clusters in the new images with the k-means algorithm and try to assign those to the existing groups, we only have to perform as many searches as we have clusters in the image. In our case that is 10, which is significantly less than the 2816 searches required for pixel wise assignment. Unfortunately we gain nothing from this performance plus since the algorithm does not produce usable output.

Comparing the angle and cosine similarity search strategies with the vector distance based ones shows the latter performing better. The former are very sensitive to the limit parameter which is well shown in Figure 4.22. Lowering the limit just slightly produces output that is unusable because everything gets grouped into one or two large clusters, which makes it impossible to distinguish different terrain classes. Increasing the limit on the other hand creates so many unique clusters that one terrain type is represented by too many IDs to be useful for classification of obstacle types.

The cosine similarity is dependant on the angle between two vectors. Therefore we can conclude that not enough information can be derived from the angle difference of the neural network output of two pixels to properly distinguish different types of terrain. The same applies for the angle similarity, as it just maps the cosine similarity into a linear scale over the angle.

This leaves us with the vector distance based search strategies. We have two variants to consider here, namely first match and best match. First match runs faster, since it allows for early stopping during the searches. In the worst case scenario it still has the same performance as the best match search. This implementation also tended to produce less small clusters. The reason for this is that when we search for the best matching cluster we do this by finding the closest distance and then checking if we are below the given radius limit for the cluster. There is now a case where the closest cluster has a radius limit such that the pixel is still too far away from its center and therefore prompts the creation of a new one when it might have matched for a different cluster that is slightly further away but has a larger radius limit.

For these reasons we see the Distance First Match strategy as the best performing solution. We now have to compare the results of the three remaining segmentation algorithms. The most complex algorithm of the tree is the Averaging `ClusterBuilder`. Unlike the other implementations, the radius limit for the clusters is updated over time here. Despite this, it still stays stable as our analysis has shown. It does however share a weakness with the Fixed `ClusterBuilder` implementation and that is its initialisation. The output of both algorithms changes quite drastically, if the first encountered image is changed. This is caused by the fact that the radius limit is in both cases derived from the average radii of the initial groups which naturally depends on the initial clusters. These differ slightly over different runs of the k-means algorithm on the same input and drastically for runs over different images. This makes the algorithms very fragile and unpredictable as the first image the system encounters is usually unknown. A potential solution for this problem is to select the starting image by hand.

Another effect of this initialisation becomes evident in the stability comparison in Figure 4.24 where we can see that on the second iteration additional clusters are created. This is caused by pixels that are at the very extremes of the groups produced by the k-means algorithm which fall outside the limit radius for their respective clusters.

This problem is mitigated in the Simple `ClusterBuilder` as it uses a constant predefined

radius limit for all clusters. Since this parameter has to be tuned by hand this is comparable to initialising the aforementioned algorithms with a preselected image. However since there is no k-means initialisation involved, this approach is much more tolerant to changes in the image sequence, which makes it the least fragile algorithm over all. The purging of very small new clusters also improves its stability compared to the other implementations as it dampens the rate at which new clusters are created.

The algorithm is however still incapable of distinguishing between different terrain and obstacle types, just like the other three implementations. The groups are instead seemingly based on colour and brightness similarity. Since the filters in our neural network, which we use as our input feature vectors, seem to largely encode brightness information and not so much the structure in the depth image, this seems to be at least in part an issue with the input to our algorithms and not necessarily the algorithms itself. The fact that the object classes we are trying to distinguish are often very similar in colour only makes this worse.

5.1.3 Classification

Our classification algorithm is able to successfully detect regions where objects move when they are exposed to an air stream. This allows us to detect whether an obstacle is solid or not. Based on this distinction, we can update our lookup table for cluster IDs and update the information that would be available to a path planning system. This allows us to drive through regions containing tall grass or other protruding but traversable objects, which would be considered solid by traditional, purely height based obstacle detection algorithms.

The fact that the method is contactless can be an advantage for use-cases in which it is undesirable to drive into potential obstacles with a bumper for any reason.

An obvious limitation of the system is that the base assumption that objects which move can be safely traversed. A counterexample would be a rope net spanned between two poles. Our fans might be powerful enough to move the net and classify it as a pseudo-obstacle but it would certainly not be traversable. However due to the flexibility, it would be easy to incorporate more sensor types such as a bumper or wheel slip measurements which could overrule the prior air-based measurements and update the drivability classification of the respective cluster type. The result would be a tiered system in which prior miss-classification could be corrected through secondary measurements with different sensors.

5.2 What We Learned

The analysis above shows that our current approach is insufficient and is unable to distinguish reliably between terrain types. The main problem seems to be the feature space that was learned by the neural network. While our simple denoising autoencoder architecture can reproduce most of the structure of the input image, the common handling of colour information and depth or infrared information through the same filters does not work well. Structure from the RGB channels contaminate the output of the other channels, especially when they differ significantly.

Further more, the purely unsupervised learning strategy has not produced features that are useful for distinguishing terrain types. Instead, the bias for black areas has optimised the network to reproduce these areas fairly accurately.

We believe that the following factors could improve the performance of our approach and should be investigated in further research:

Split up the neural network By creating parallel paths for the colour, depth and infrared channels and optimising them separately instead of training the network on a single five channel input, the kind of crosstalk we encountered between them could be eliminated. Later on, the three network outputs can be considered as three separate feature vectors for the purpose of the segmentation process.

Avoid gaps in inputs To avoid the bias for the dark areas which we encountered during our experiments, an option would be to use full colour images instead of our versions, where areas with no depth information are set to 0. Another option would be to mask out these areas for the error calculation, which has the advantage of also being applicable to the depth channel.

Increase size of training set Increasing the size of the training and test sets might also benefit the performance of the neural network. This can be achieved by capturing more images and by modifying existing images through scaling or mirroring so they can be used multiple times.

Use supervised learning A drastic change would be the switch to a supervised training method. The downside of this is that it requires large amounts of labeled data which can be costly to produce. It would however eliminate the need for our segmentation and classification algorithms, as this could simply be implemented as a fully connected layer on top of our existing network which would produce a network structure similar to SegNet. The downside of this is however that it limits the system to the types of terrain that it has been trained on, making it unsuitable for unknown environments which is the strength of the approach presented in this work.

5.3 Further Work

As the system is currently not in a state where it can be used in real world applications, the first step is to implement and test the points mentioned above. Once we are capable of reliably distinguishing and recognize different types of terrain, we can proceed with the next steps towards a fully integrated system.

Our cluster data has to be transformed into a global reference frame so we can build a navigation map from it. For this we can use the point cloud creation code from our library to combine the depth image and our cluster ID map into 3D space. This data can then be used to update a global grid map, where each cell contains information about the types of terrain present in it through the cluster IDs in that cell. Since we maintain a lookup table from cluster ID to drivability status, we can obtain an occupancy grid from this, where cells are either free, occupied or unknown. A path planner can then use this map to find the best route to a given goal location.

When a cell marked as unknown is encountered by the robot along the path, we can use our `FANCONTROLLER` node together with our classification algorithm to determine whether the

terrain at the cell is free or not. With the result of this test, we update our lookup table and regenerate our occupancy map, which updates all other grid cells that also contain the cluster ID we just checked. Finally, the path planner reruns its path search to update the route to the goal position given this new information and the robot can continue driving. Cells that do not contain points above or below a certain height can immediately be marked free as they only contain flat ground and our system does not contain sensors for flat undrivable terrain.

To implement this, the following additional components are required:

- An algorithm to merge the cluster ID point cloud into a global grid map
- Ground plane detection to immediately mark flat cluster IDs as free
- ROS planning module that triggers the obstacle test
- ROS node that performs the full obstacle check

Most of these components already exist in some form and would merely have to be adopted and extended for our special use-case.

Further more, the system can be extended with more sensors. In its current state, we can only check obstacles protruding from the ground for solidity. We cannot test the ground itself with regard to its stability. Our system could however easily be combined with more types of sensors such as wheel slippage to also mark certain floor categories as undrivable.

Chapter 6

Conclusion

In this work we have evaluated the performance and viability of our terrain classification system.

We have described a viable low-cost hardware setup that is capable of capturing colour and depth information and implemented our air-based obstacle detection sensor on our robot platform. With this setup we were able to capture data in our test environment on which we trained our neural network. We were able to implement all described algorithms in software and run our neural network on GPU accelerated desktop class hardware as well as on a laptop without GPU acceleration and produce results on both platforms. With GPU acceleration available, our system is fast enough to process data at a rate capable of real-time navigation at slow driving speeds.

Our autoencoder network architecture was able to reasonably reproduce the colour channel and some aspects of the depth and infrared channels after being trained on our data set. Shortcomings were encountered with the reproduction of very bright input in either of depth and infrared channels while black regions were reproduced best.

Consequently, the feature space represented by the encoder output proved to resemble mostly brightness information from the colour channels and not correspond to depth and infrared strongly. This has made the evaluation of our segmentation algorithms difficult. We were however able to show the difference in their behaviours, based on which we could rule out the viability of the k-means algorithm and the cosine and angle similarity cluster search variants of the Simple `ClusterBuilder`. While none of the algorithms produced results good enough to be used in any real-world application, we found the Simple `ClusterBuilder` to be the most reasonable variety. With a well tuned radius limit and its rejection threshold it proved to be the most consistent implementation of the ones presented here.

Future research is required to determine whether this algorithm can produce useful results with a better feature space produced by a different or better trained neural network. In the discussion we have provided some ideas for changes that could yield the desired improvements.

Our experiments on the classification algorithm based on movement detection was demonstrated to work within expectations. We were able to clearly detect the regions where the movement had occurred in the captured depth images after we enabled our blower actuator. By combining this with the available 3D environment map, this system would be able to produce drivability labels for our classifier and is further more easily expandable to incorporate more

sensor types.

In conclusion, we have shown that our system is not yet ready for real-world use but we expect to see continuing research into similar, convolutional neural network based approaches.

Bibliography

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[2] J. Andrew (Drew) Bagnell, David Bradley, David Silver, Boris Sofman, and Anthony (Tony) Stentz. Learning for autonomous navigation: Advances in machine learning for rough terrain mobility. 17(2):74–84, June 2010.

[3] C. A. Brooks and K. Iagnemma. Vibration-based terrain classification for planetary exploration rovers. *IEEE Transactions on Robotics*, 21(6):1185–1191, Dec 2005.

[4] P. Fankhauser, M. Bloesch, D. Rodriguez, R. Kaestner, M. Hutter, and R. Siegwart. Kinect v2 for mobile robot navigation: Evaluation and modeling. In *Advanced Robotics (ICAR), 2015 International Conference on*, pages 388–394, July 2015.

[5] D. Gibbins and L. Swierkowski. A comparison of terrain classification using local feature measurements of 3-dimensional colour point-cloud data. In *2009 24th International Conference Image and Vision Computing New Zealand*, pages 293–298, Nov 2009.

[6] Andrew Howard, Michael Turmon, Larry Matthies, Benyang Tang, Anelia Angelova, and Eric Mjolsness. Towards learned traversability for robot navigation: From underfoot to the far field. *Journal of Field Robotics*, 23(11-12):1005–1017, 2006.

[7] Intel Corporation. *4-Wire Pulse Width Modulation (PWM) Controlled Fans*, July 2004. Revision 1.2.

[8] R. Jitpakdee and T. Maneewarn. Neural networks terrain classification using inertial measurement unit for an autonomous vehicle. In *SICE Annual Conference, 2008*, pages 554–558, Aug 2008.

[9] Dongshin Kim, Jie Sun, Sang Min Oh, J. M. Rehg, and A. F. Bobick. Traversability classification using unsupervised on-line visual learning for outdoor robot navigation. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 518–525, May 2006.

[10] Artur Maligo and Simon Lacroix. Classification of Outdoor 3D Lidar Data Based on Unsupervised Gaussian Mixture Models. In *IEEE International Symposium on Safety, Security, and Rescue Robotics*, West Lafayette, United States, October 2015.

[11] Daniel Maturana and Sebastian Scherer. 3d convolutional neural networks for landing zone detection from lidar. In *International Conference on Robotics and Automation*, March 2015.

[12] Microsoft. Kinect hardware. `https://developer.microsoft.com/en-us/windows/kinect/hardware`, 2016. Accessed: 2016-08-16.

[13] Giulio Reina, Mauro Bellone, Luigi Spedicato, and Nicola Ivan Giannoccaro. 3d traversability awareness for rough terrain mobile robots. *Sensor Review*, 34(2):220–232, 2014.

[14] Christophe Reymann and Simon Lacroix. Improving LiDAR Point Cloud Classification using Intensities and Multiple Echoes. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015)*, page 7p., Hamburg, Germany, September 2015.

[15] Richard Socher and Brody Huval and Bharath Bhat and Christopher D. Manning and Andrew Y. Ng. Convolutional-Recursive Deep Learning for 3D Object Classification. In *Advances in Neural Information Processing Systems 25*. 2012.

[16] Thiemo Wiedemeyer. IAI Kinect2. `https://github.com/code-iai/iai_kinect2`, 2014 – 2015. Accessed 2016-08-16.

[17] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[18] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip Torr. Conditional random fields as recurrent neural networks. In *International Conference on Computer Vision (ICCV)*, 2015.

Proclamation

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Würzburg, November 2016