

---

# Traversability Estimator for Legged robot.

Master's Thesis submitted to the  
Faculty of Informatics of the *Università della Svizzera Italiana*  
in partial fulfillment of the requirements for the degree of  
Master of Science in Informatics  
Artificial Intelligence

presented by  
Francesco Saverio Zuppichini

under the supervision of  
Prof. Student's Alessandro Giusti  
co-supervised by  
Prof. Student's Co-Advisor

June 2019



---

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

---

Francesco Saverio Zuppichini  
Lugano, 28 June 2019



# Abstract

Effective identification of traversable terrain is essential to operate mobile robots in different environments. Historically, to estimate traversability, texture and geometric features were extracted before train a traversability estimator in a supervised way. However, with the recent deep learning breakthroughs in computer vision, terrain features may be learned directly from raw data, images or heightmaps, with higher accuracy.

We implement a full pipeline to estimate traversability with high accuracy using only data generated through simulation using a deep convolutional neural network. The method is based on the framework proposed by Chavez-Garcia et al. [6] and was originally tested on two wheels small robot. Collecting data in a simulation environment yields several advantages over real world. Ground can be easily generated to include any features and can be loaded on the fly into the simulator. Multiple simulations can be run in parallel reducing the cost for real world hardware.

The dataset was generated using thirty synthetic surfaces with different features such as slopes, holes, bumps, and walls. Then, we let the robot walk each map on for a certain amount of time while storing its interactions. Later, precisely crop a patch from each simulation trajectory's point that includes the whole robot's footprint and the amount of ground it should reach without obstacles in a decided time window. Then, we label each patch as traversable or not traversable based on a minimum advancement depending on the robot used. We open source the framework and the same methodology can be used with any type of robots. We test the methodology on a legged crocodile-like robot that presents challenging locomotion to be learned in order to correctly predict its traceability.

We select and test different networks architecture and select the best performing one. We quantity shows its performance with different numeric metrics on different real-world terrains. In addition, we qualitatively evaluate the network by showing the predicted traversability probability on different grounds to better visualize the acquired knowledge.

Later, we utilize model interpretability techniques to understand which patches confuse the network. We visualize the ground regions that the network fails to classify and find which part of the inputs was responsible for the wrong predictions. Finally, we test the model strength and robustness by comparing its prediction on custom patches composed by crafted features, such as a patch with a wall ahead, to the ground truth obtained by running again Krock on that ground in the simulator. The results suggest that the model was able to match the ground truth in different situations.



# Contents

<b>Contents</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Related Work</b>	<b>3</b>
<b>3 Methodology</b>	<b>5</b>
<b>4 Implementation</b>	<b>11</b>
4.1 Robot . . . . .	11
4.2 Data Gathering . . . . .	12
4.2.1 Heightmap generation . . . . .	13
4.2.2 Simulator . . . . .	16
4.2.3 Real world maps . . . . .	16
4.3 Simulation . . . . .	17
4.4 Postprocessing . . . . .	20
4.4.1 Parse stored data . . . . .	20
4.4.2 Extract patches . . . . .	20
4.4.3 Label patches . . . . .	24
4.5 Estimator . . . . .	26
4.5.1 Vanilla Model . . . . .	26
4.5.2 ResNet . . . . .	26
4.5.3 Preactivation . . . . .	27
4.5.4 Squeeze and Excitation . . . . .	27
4.5.5 MicroResNet . . . . .	28
4.6 Normalization . . . . .	29
4.7 Data Augmentation . . . . .	29
4.7.1 Dropout . . . . .	30
4.7.2 Coarse Dropout . . . . .	30
4.7.3 Simplex Noise . . . . .	30
4.8 Tools . . . . .	32
4.8.1 ROS Melodic . . . . .	33
4.8.2 Numpy . . . . .	33
4.8.3 Matplotlib . . . . .	33
4.8.4 Pandas . . . . .	33
4.8.5 OpenCV . . . . .	34

4.8.6	PyTorch . . . . .	34
4.8.7	FastAI . . . . .	34
4.8.8	imgaug . . . . .	34
4.8.9	Mayavi . . . . .	34
<b>5</b>	<b>Results</b>	<b>35</b>
5.1	Experiment setup . . . . .	35
5.2	Dataset . . . . .	35
5.2.1	Experimental validation . . . . .	35
5.2.2	Metrics . . . . .	36
5.3	Quantitative results . . . . .	36
5.3.1	Model selection . . . . .	36
5.4	Final results . . . . .	37
5.5	Qualitative results . . . . .	37
5.5.1	Quarry . . . . .	37
5.5.2	Bars . . . . .	38
5.5.3	Small village . . . . .	40
<b>6</b>	<b>Interpretability</b>	<b>43</b>
6.0.1	Features separability . . . . .	43
6.0.2	Train set . . . . .	44
6.0.3	Test set . . . . .	47
6.1	Grad-CAM . . . . .	50
6.2	Robustness . . . . .	51
6.2.1	Untraversable wall ahead at increasing distance . . . . .	51
6.2.2	Increasing height walls ahead . . . . .	54
6.2.3	Increasing height and distance walls ahead. . . . .	55
6.2.4	Tunnel . . . . .	57
6.2.5	Ramps . . . . .	57
6.2.6	Holes . . . . .	59
<b>7</b>	<b>Conclusions</b>	<b>61</b>

# Chapter 1

## Introduction

All living beings, including humans, need to traverse ground to eat, sleep and breed. Animals through millenia of evolution have developed different techniques to classify ground regions. Usually, local sensors are used to map new terrain to effectively navigate the environment. Humans, for instance, are equipped with a powerful traversability estimator, called vision, that is able to extract features from a local region, a patch, such as elevation, size, and steepness and combine them with the brain to produce a local planner able to find a suboptimal path to cross.

Similarly, ground robots need to walk in order to fulfill their tasks. Taking inspiration from nature, robots must be able to correctly aggregate terrain features to predict their traversability. Two main approaches have emerged over the years to collect the data, geometric and appearance. The first one utilize purely geometric features to label a patch, while the second relies on label categories, glass, rocks, etc, with fixed traversability probabilities extract mostly on camera's images. In both cases, those samples are used in supervised learning to train a traversability estimator. While appearance data must be collected directly from a real or simulated environment, geometric features do not directly depend on the robot's interactions with the ground.

In most indoor scenarios, data is collected trough specific hardware, such as infrared sensors, cameras or LIDAR, during a robot exploration of the enviroment. Due to their artificial design, indoor enviroments shared similar features across the world, flatness of the floor, stairs and ceil. So, even if an agent is trained with samples collected in one location, due to physical limitations, the learned mapping can be effective in different locations. Moreover, in indoor enviroment, traversability estimation is mostly solved with obstacle avoidance since, by design, the floor alone do not include bumps, ramps or holes. In addition, while the robot operates there may be other agents interacting with the enviroment, such as humans, therefore a correct function to avoid collision must be properly learn to effective move safely the robot.

On the other hand, outdoor grounds may have less artificial obstacle but they have a not homogeneous ground making challenging to estimate where the robot can properly travel. Also, a given patch with a specific shape may not be traversable by all direction due to the not uniform features of the terrain. On top of that, using real world data may be unfeasible due to the time required to move the robot on different grounds and the possibility to introduce bias if the data samples are not varied enough. Fortunately, ground

can be generated artificially and used in a simulated environment to let the robot walk in it while storing its interactions. Moreover, ground maps can be easily obtained nowadays by using third-party services such as google maps or flying drone equipped with mapping technologies.

We proposed a full pipeline to estimate traversability tested on legged crocodile-like robot on uneven terrain based entirely on data gathered through simulation. Simulation offers several advantages than real world data collection. First, we can include a rich array of different terrains without the need to physically move them. Data is collected faster and multiple simulations can be launched in parallel keeping the cost of the real world hardware low. We first generated thirty synthetic maps encoded as heightmaps with different features: bumps, walls, slopes, steps and holes. Then, we run the robot on each one of them for a certain amount of time while storing its interactions with the environment, position and orientation. Later, the procedure to create the dataset by cropping a patch for each stored data point in a such way that includes the robot's footprint and the maximum possible amount of ground that can traverse in a selected time window. After, we label each patch by traversable or not traversable if the robot's advancement computed in that time using the same time window is less or greater than a threshold. The threshold depends on the robot's locomotion and is calculated by spawning it in front of different obstacle and observing its advancement, we set the value for the tested robot to twenty centimeters. Those patches are used to fit a deep convolutional neural network to predict the traversability leaving to the network the task to extract important ground's features.

The report is organized as follows, the next chapter 2 introduces the related work, chapter 3 gives a high level overview of our approach, chapter 4 talk in depth about the implementation details, chapter ?? shows the results and chapter ?? discuss conclusions and future work

## Chapter 2

# Related Work

The learning and perception of traversability is a fundamental competence for both organisms and autonomous mobile robots since most of their actions depend on their mobility [24]. Usually, visual perception is used in most all animals to correctly estimate if an environment can be traversed or not. Similar, a wide array of autonomous robots adopt local sensors to mimic the visual properties of beings to extract meaningful information of the surrounding and plan a safe path through it. This chapter gives an overview of some of the existing methods in the literature to solve this task.

Most of the methodologies to estimate traversability rely on supervised learning, where first the data is gathered and then a machine learning algorithm is trained to correctly predict the traversability of those samples. Most of the approaches can be clustered into two categories, based on the inputs data: geometric and appearance based methods.

Geometric methods aim to detect traversability using geometric properties of surfaces such as distances in space and shapes. Those properties are usually slopes, bumps, and ramps. Since nearly the entire world has been surveyed at 1 m accuracy [22], outdoor robot navigation can benefit from the availability of overhead imagery, for this reason, recently, elevation data has been widely used to generate datasets. Chavez-Garcia et al. [6], proposed a framework to learn traversability using elevation data as input in the form of height maps. They trained a convolutional neural network to predict from a dataset, entirely generated through simulation, the traversability probability of a small ground region around the robot.

Elevation data can also be estimated by flying drones. Delmerico et al. [12] proposed a collaborative search and rescue system in which a flying robot that explores the map and creates an elevation map to guide the ground robot to the goal. They train on the fly a CNN to segment the terrain in different traversable classes. This map is later used to plan a path for the mobile robot.

Whereas appearance methods, to a greater extent related to camera images processing and cognitive analyses, have the objective of recognizing colors and patterns not related to the common appearance of terrains, such as grass, rocks or vegetation. Historically, the collected data is first preprocessed to extract texture features that are used to fit a classic machine learning classified such us an SVM [24] or Gaussian models [22]. Those techniques rely on texture descriptors, for example, Local Binary Pattern [23], to extract features from the raw data images obtained from local sensors such as cameras. With the rise of deep learning methods in computer vision, deep convolution neural network has been trained

---

directly on the raw RGB images bypassing the need to define characteristic features.

One recent example is the work of Giusti et al. [3] where a deep neural network was training on real-world hiking data collected using head-mounted cameras to teach a flying drone to follow a trail in the forest. Geometric and appearance methods can be used together, one example is the work of Delmerico et al.[7] extended the previous work [12] by proposing the first on-the-spot training method that uses a flying drone to gather the data and train an estimator in less than 60 seconds.

Data can also extract in simulations, where an agent interacts in an artificial environment. Usually, no-wheel legged robot able to traverse harder ground, can benefits from data gathering in simulations due to the high availability. For example, Klamt et al. [13] introduced a locomotion planner that is learned in a simulation environment.

On other major distinction, is between different types of robots: wheel and no-wheel. We will focus on the later since we adopt a legged crocodile-like robot to extend the existing framework proposed by Chavez-Garcia et al. [6].

Legged robots show their full potential in rough and unstructured terrain, where they can use a superior move set compared to wheel robots. Different frameworks have been proposed to compute safe and efficient paths for legged robots. Wermelinger et al. [25] use typical map characteristics such as slopes and roughness gather using onboard sensors to train a planner. The planner uses a RRT\* algorithm to compute the correct path for the robot on the fly. Moreover, the algorithm is able to first find an easy local solution and then update its path to take into account more difficult scenarios as new environment data is collected.

Due to uneven shape rough terrain, legged robots must be able to correctly sense the ground to properly find a correct path to the goal. Wagner et al. [14] developed a method to estimate the contact surface normal for each foot of a legged robot relying solely on measurements from the joint torques and from a force sensor located at the foot. This sensor at the end of a leg optically determines its deformation to compute the force applied to the sensor. They combine those sensors measurement in an Extended Kalman Filter (EKF). They showed that the resulting method is capable of accurately estimating the foot contact force only using local sensing.

While the previous methods rely on handcrafted map's features extraction methods to estimate the cost of a given patch using a specific function, new frameworks that automatize the features extraction process has been proposed recently. Lorenz et al. [16] use local sensing to train a deep convolutional neural network to predict the terrain's properties. They collect data from robot ground interaction to label each image in front of the robot in order to predict the future interactions with the terrain showing that the network is perfectly able to learn the correct features for different terrains. Furthermore, they also perform weakly supervised semantic segmentation using the same approach to divide the input images into different ground classes, such as glass and sand, showing respectable results.

## Chapter 3

### Methodology

We developed a framework to learn traversability directly on ground patches for any robot using purely simulated data. The core idea, originally proposed by Chavez-Garcia et al. [6], is simple yet elegant, a robot is let travel into a simulator on different synthetic terrains, while storing its interactions to later crop a region of ground, a patch, around each traveled position and directly train a neural network on them to predict traversability. Figure 3.1 shows the framework's steps.

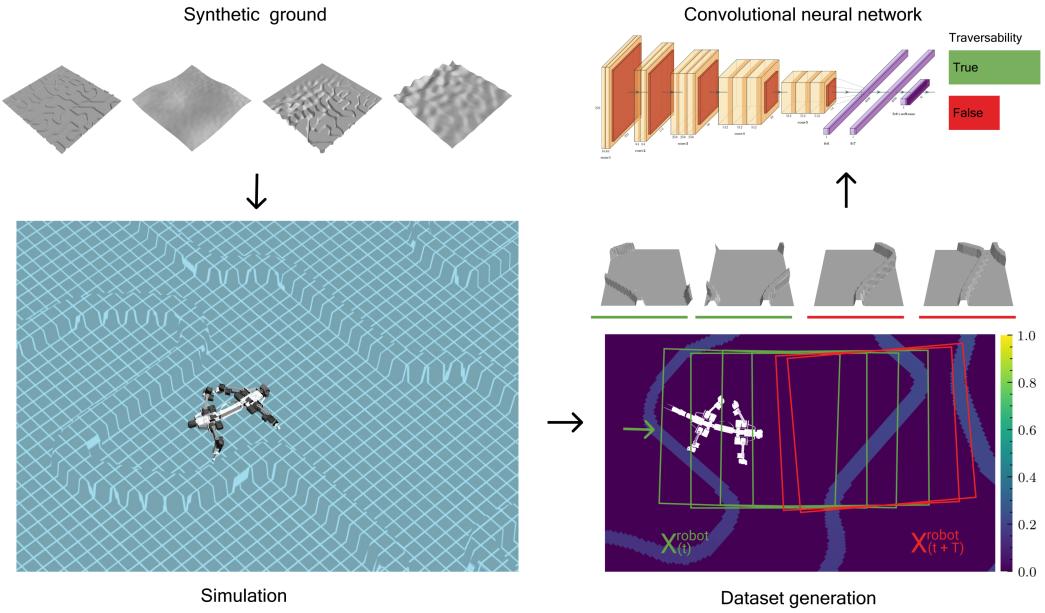


Figure 3.1. The framework’s main building block in counter-clockwise order. First, we generated meaningful synthetic grounds, then we let the robot spawn and walk on them in a simulated environment while storing its interactions. Later, we crop a region of ground, a patch, for each simulation trajectory around the robot according to its locomotion. We labeled those images using a defined threshold and fit a deep convolutional neural network to predict traversability.

Collecting data through simulation has several main advantages such as variety, cost, and speed. With simulations, we can easily increase the number of data samples at any time by simply incorporate more maps. Contrarily, a real robot requires to first identify the suitable ground and then physically transport the machine there and finally let it walk on it. Furthermore, we are not limited by real constraint and we can design the ground to maximize the variety of robot interactions. We can artificial design maps to include any kind of situations and challenges. Clearly, this also reduces the time required to collect the robot’s interactions with the terrain. The time could be further minimized by running different simulations in parallel bypassing the demand for more physical hardware. To create a quality dataset, we must generate a series of various surfaces and ensure their diversity. Intuitively, those maps should be small enough to allow fast exploration but big enough to include several features. With modern ground generation techniques, it is fairly straightforward to generate a rich array of terrains with different characteristics such us bumps, ramps, slopes in different levels and sizes. There are several methods to represent terrain, we utilize heightmap where the height component of the ground is stored as pixels in a gray image. Practically, the brightness describes the height. The following figure shows a small collection of synthetic terrains with different features.

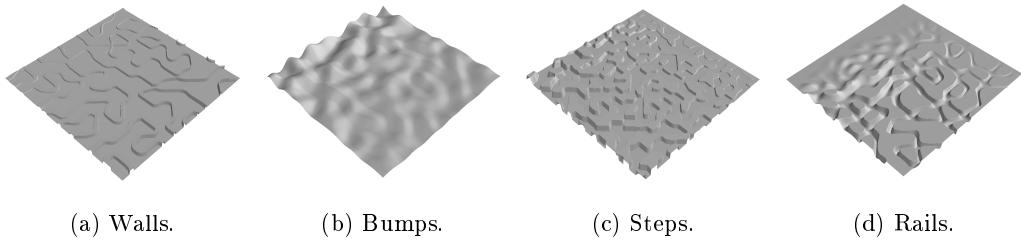


Figure 3.2. Some artificially generated terrains,  $10 \times 10\text{m}$ , with one specific feature each.

To let the robot interacts with them, each map can be simply loaded into a simulator. To ensure a correct exploration of each terrain, we randomly spawn the robot multiple times on the same map and let it walk for a fixed amount of time. Randomly spawn the robot on the surface reduce the possibility to repeatedly visiting the same spot and it maximizes the robot's exploration.

The robot position is tracked and stored during simulation in order to extract a small portion of ground, patches, around each of its trajectory's position. In practice, the patches are cropped from the original image using a specific size based on the robot's footprint and its velocity. Each resulting image is labeled using a minimum advancement based on the robot's characteristic. We tested our framework on a legged crocodile-like robot called *Krock*. Due to its four legs, the robot has very unique locomotion allowing it to overcome different obstacles making estimate traversability more challenging. Figure ?? helps to visualize the procedure. The dataset generation process is explained in detail in section 4.2. while

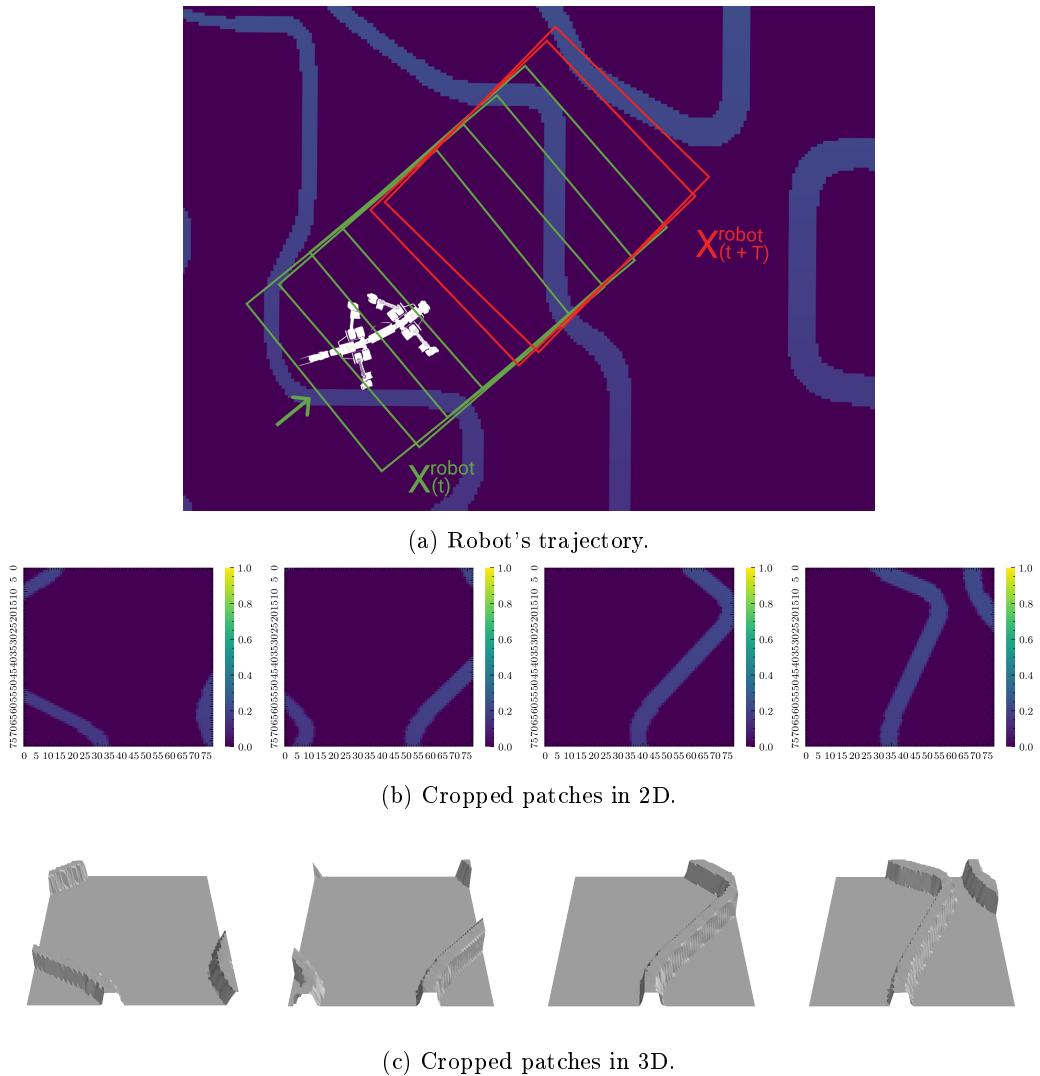


Figure 3.3. Example of a robot trajectory (a) extracted during training on a map with different walls. Robot's initial position is shown by its white silhouette. Patches borders are labeled with greed if traversable and red if not and showed as 2D (b) and 3D(c) rendered images. The robot traverses the patches from left to right.

This dataset is fed to a deep convolutional neural network. The network directly learns to extract features from raw data, in our case images, without needing to preprocess the inputs. After tested different model's, we selected a convolutional neural network three times lighter than the original one with comparable performance. To properly evaluate the architectures, we used real-world terrains, mostly obtained with flying droned and ground mapping software. The results are quantively and qualitatively shown in chapter 5. Later, in chapter 6, we investigated the model's learning by first visualizing the inputs that confuse the model, section ???. We adopted a technique that allows highlighting the regions in the input images that contribute the most to the prediction, in this way we can understand

which part of the terrain was responsible for the wrong prediction. Lastly, in section 6.2, we tested the model’s robustness by creating different patches with different characteristics and compare the prediction to the ground truth from the simulator. We shown that the model was able to match the simulator’s outputs in most scenarios but most important, we understand the network’s limitations.

To summarize our contributions to literature are the implementation of a framework to learn traversability entirely through simulation that can be employed with any mobile robot, a new small neural network architecture able to reach high accuracy, different real-world evaluations, and a model interpretability case study to understand the strengths and limitations of the trained model.



# Chapter 4

## Implementation

This section talks in detail about the implementation of the framework. We will first talk about the utilized robot, then describe each step in the dataset generation, illustrate the utilized estimators and finish by listing the employed software.

### 4.1 Robot

We tested our framework on a legged crocodile robot called *Krock* developed at EPFL. The next figure shows the robot in real life with a coat and in a simulated environment.



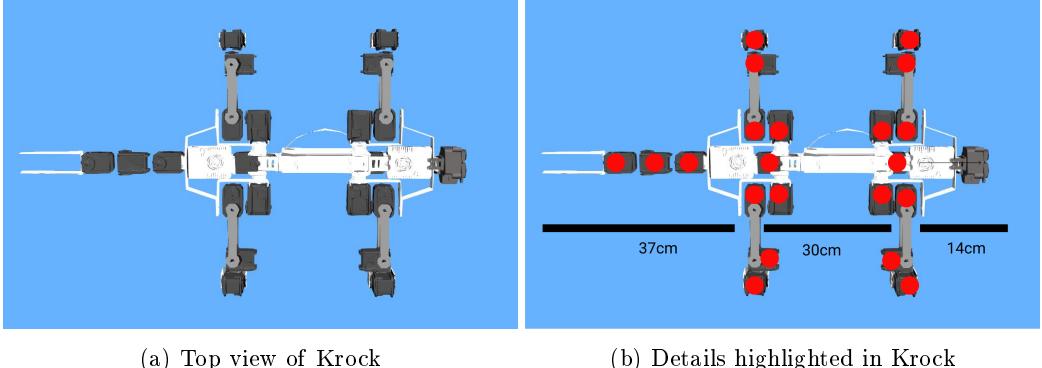
(a) Krock in real life.

(b) Krock in the simulator.

Figure 4.1. Krock

Krock has four legs, each one of them is equipped with three motors in order to rotate in each axis. The robot can raise itself in three different configurations, gaits, using the four motors on the body connected to the legs. In addition, there are another set of two motors in the inner body part to increase Krock's move set. The tail is composed by another set of three motors and can be used to perform a wide array of tasks. The robot is 85cm long and weighs around 1.4kg. The next figure ?? shows *Krock* from the top helping the reader

understanding its composition and the correct ratio between its parts. Also, each motor is highlighted with a red marker.



(a) Top view of Krock

(b) Details highlighted in Krock

*Krock's* moves by lifting and moving forward one leg after the other. The following figure shows the robots going forward.

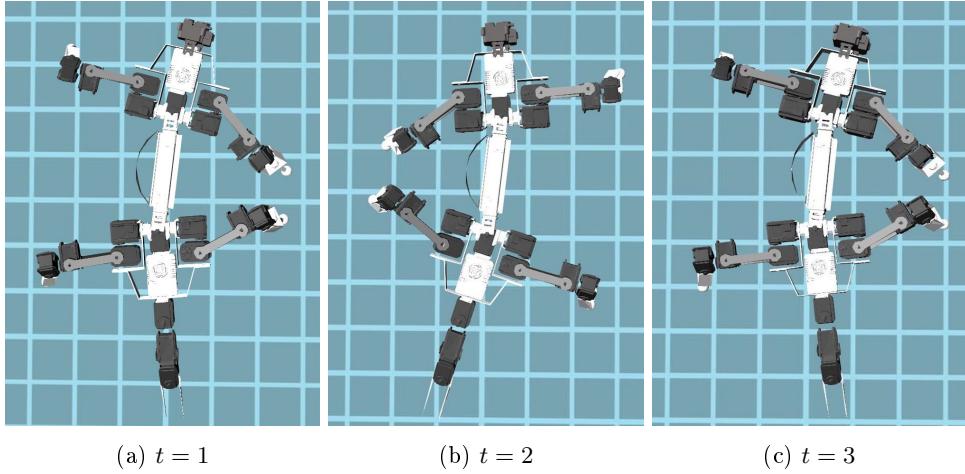


Figure 4.3. Krock moving forward.

In our framework we fixed the gait configuration to normal, showed in figure ??, where the body is at the same legs' motors height.

Add Krock gait picture

## 4.2 Data Gathering

To train a traversability estimator we need a dataset. This section describe in detail how to create, collect and process the data gather from a simulated enviroment to generate a dataset we can use to perform supervised learning.

### 4.2.1 Heightmap generation

To collect the data through simulation we first need to generate meaningful terrain to be explored by the robot. Each map is stored as heightmap, a 2D array, an image, where each pixel's value represents the terrain height. For example, the following image shows an heightmap representing a real world quarry and the relative terrain.

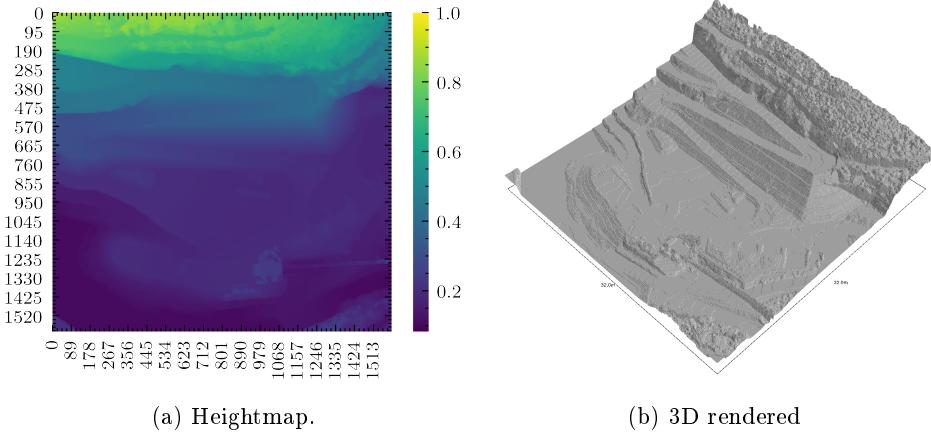


Figure 4.4. This image shows an heightmap and the 3D render of a quarry.

We created thirty maps of  $513 \times 513$  pixel with a resolution of  $0.02\text{cm}/\text{pixel}$  in order to represent a  $10 \times 10\text{m}$  terrain using 2D simplex noise [17]. Simplex noise is a variant of Perlin noise [1], a widely used technique in the terrain generation literature. We divide the maps into five main categories of terrains to cluster different ground features: *bumps*, *rails*, *steps*, *slopes/ramps* and *holes*. For some map we add three different rocky texture to create more complicated situations, all the maps configuration are shown in detail in table ??.

**Bumps:** We generated four different maps with increasing bumps' height using simplex noise with features size  $\in [200, 100, 50, 25]$ .

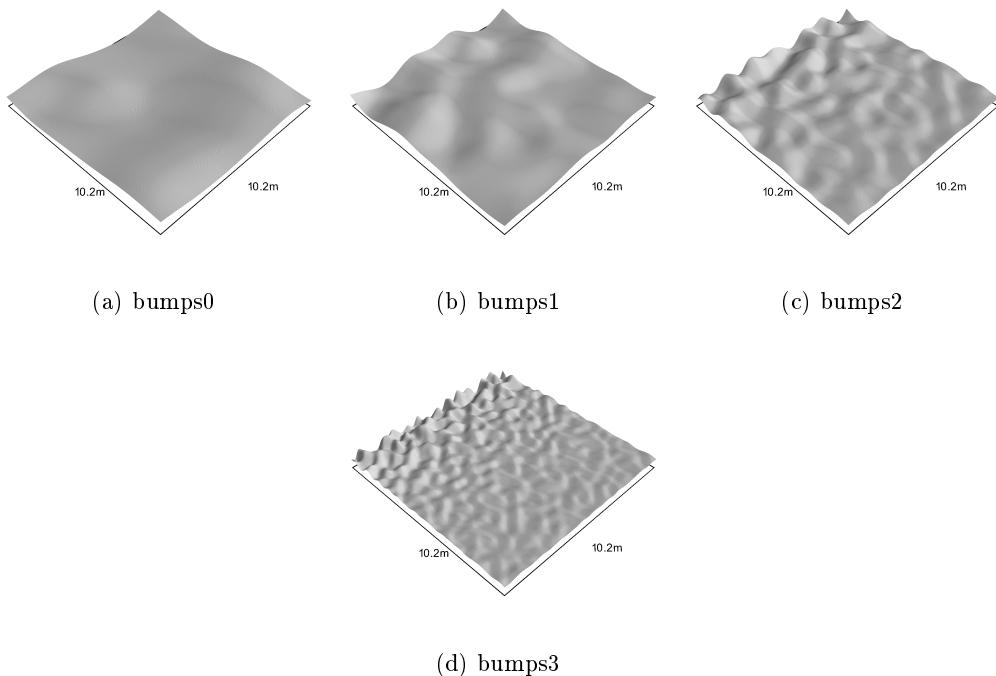


Figure 4.5. Bumps maps ( $10 \times 10$ m).

**Bars:** In these maps there are wall with different shapes and heights. In

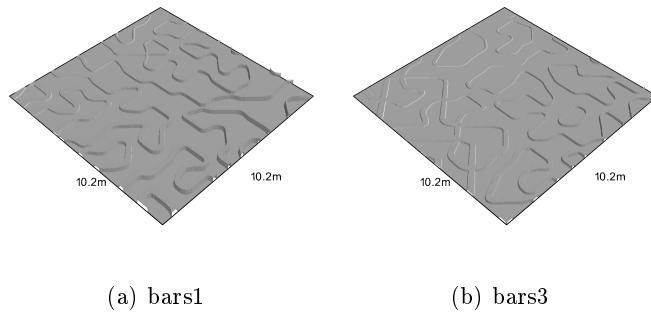


Figure 4.6. Bars maps ( $10 \times 10$ m).

Rails: Flat grounds with slots.

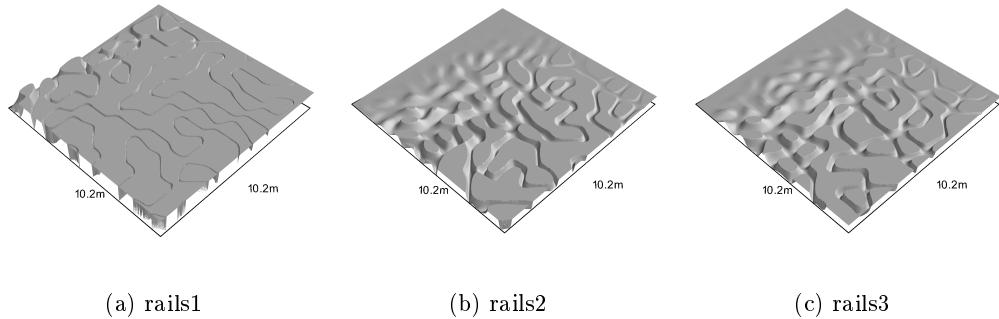


Figure 4.7. Rails maps ( $10 \times 10$ m).

Steps: These are maps with various steps at increasing distance and frequency.

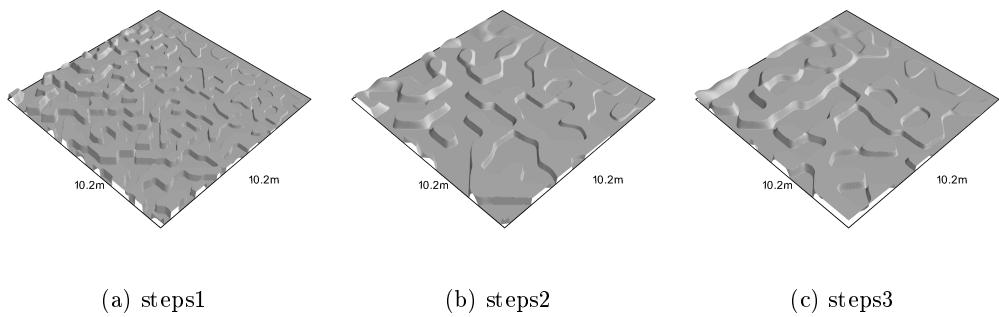


Figure 4.8. Steps maps ( $10 \times 10$ m).

**Slopes/Ramps:** Maps composed by uneven terrain scaled by different height factors from 3 to 5 used to include samples where *Krock* has to climb.

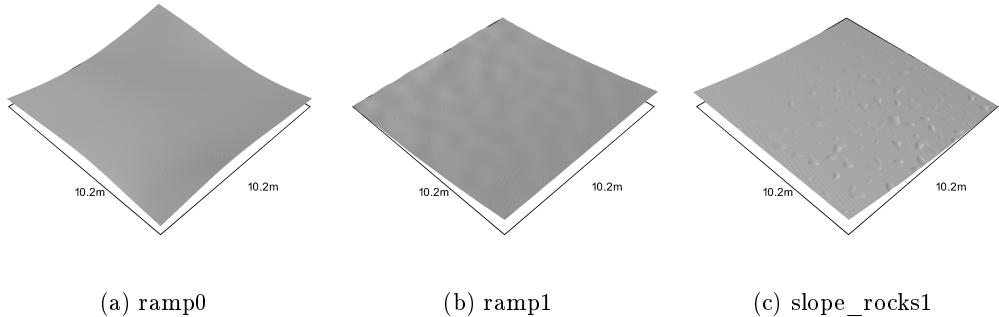
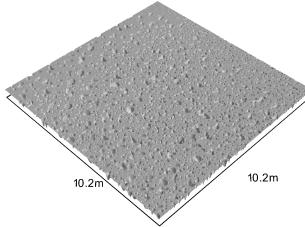


Figure 4.9. Slopes maps ( $10 \times 10$ m).

Holes We also included a map with holes



(a) holes1

Figure 4.10. Holes map (10 × 10m).

#### 4.2.2 Simulator

We used Webots to move *Krock* on the generated terrain. The robot controlled was implemented by EPFL and handed to IDSIA . The controller implements a ROS' node to publish *Krock* status including its pose at a rate of 250hz. We decide to reduce it to 50hz by using ROS build it `throttle` command. To load the map into the simulator, we first had to convert it to Webots's `.wbt` file. Unfortunately, the simulator lacks support for heightmaps so we had to use a script to read the image and perform the conversion.

cite them?  
cite?

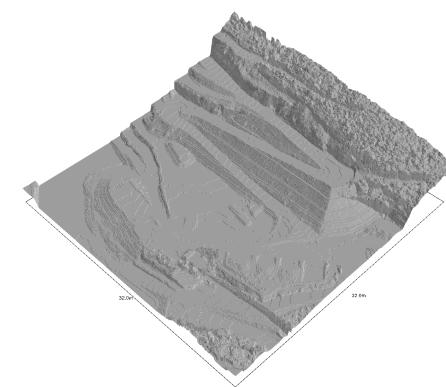
To communicate with the simulator, Webots exposes a wide number of ROS services, similar to HTTP endpoints, with which we can communicate. The client can use the services to get the value of a field of a Webots' Node, for example, if we want to get the terrain height, we have to ask for the field value `height` from `TERRAIN` node. In addition, to call one service, we first have to get the correct type of message we wish to send and then we can call it. We decided to implement a little library called `webots2ros` to hide all the complexity needed to fetch a field value from a node.

We also implement one additional library called `Agent` to create reusable robot's interfaces independent from the simulator. The package supports callbacks that can be attached to each agent adding additional features such as storing its interaction. Finally, we used `Gym` [5], a toolkit to develop and evaluate reinforcement learning algorithms, to define our environment. Due to the library's popularity, the code can easily be shared with other researches or we may directly experiment with already made RL algorithm in the future without changing the underlying infrastructure.

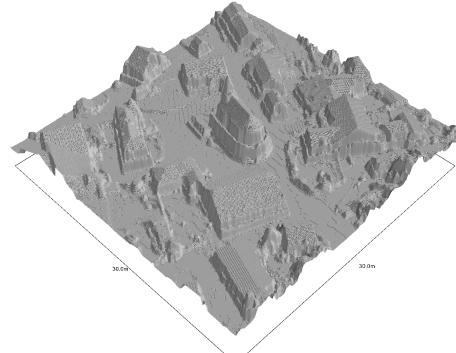
#### 4.2.3 Real world maps

To later evaluate the model's performance, we decide to use real world terrain. We gather several heightmaps produced by ground mapping flying drones from sensefly's dataset.

include sullens



(a) Quarry



(b) A small village.

Figure 4.11. Real world maps obtained from sensefly’s dataset. The left images shows the real world location while on the right a render in 3D of the coresponding heightmaps. Booth images have a maximum height of 10m.

### 4.3 Simulation

To collect *Krock*’s interaction with the environment, we spawn the robot on the ground and let it move forward for  $t$  seconds. We repeat this process  $n$  times per each map. Unfortunately, spawning the robot is not a trivial task. In certain maps, for instance, *bars1*, a map with tons of walls, we must avoid spawning *Krock* on an obstacle otherwise the run will be ruined by *Krock* getting stuck at the beginning introducing noise in the dataset. To solve the problem, we defined two spawn strategies, a random spawn and a flat ground spawn strategy. The first one is used in most of the maps without big obstacles such as *slope\_rocks*. This stragety just spawn the robot on random position and rotation. While, the flat ground strategy isfirst selects suitable spawn positions by using a sliding window on the heightmap of size equal *Krock*’s footprint and check if the mean pixel value

is lower than a small threshold. If so, we store the center coordinates of the patch as a candidate spawning point. Intuitively, if a patch is flat then its mean value will be close to zero. Since there may be more flat spawning positions than simulations needed, we have to reduce the size of the candidate points. To maintain the correct distribution on the map to avoid spawning the robot always in the same cloud of points, we used K-Means with  $k$  clusters where  $k$  is equal to the number of simulations we wish to run. By clustering, we guarantee to cover all region of the map removing any bias. The following picture shows this strategy on *bars1*.

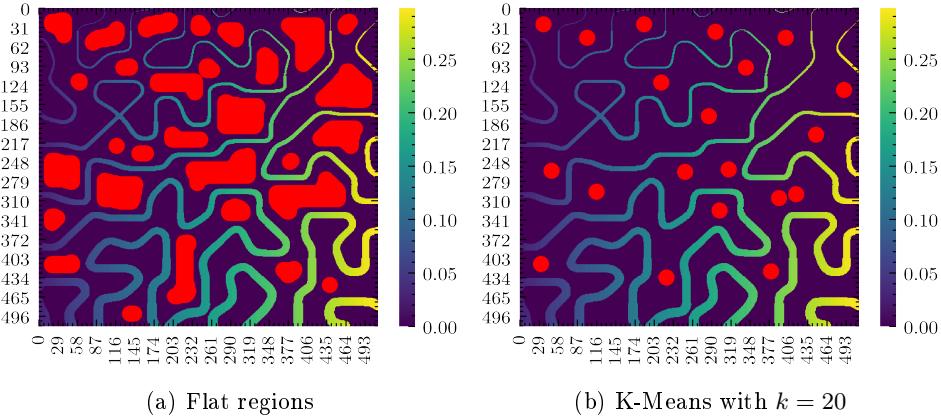


Figure 4.12. Examples of the spawning selection process (marked as red blobs) for the map *bars1*

The following table shows the maps configuration used in the simulator.

Map	Height(m)	Spawn	Texture	Simulations	max time(s)
<i>bumps0</i>	2	random	- rocks1 rocks2	50	10
<i>bumps1</i>	1	random	- rocks1 rocks2	50	10
<i>bumps2</i>	1	random	- rocks1 rocks2	50	10
	2		-		
<i>bumps3</i>	1	random	- rocks1 rocks2	50	10
<i>steps1</i>	1	random	-	50	10
<i>steps2</i>	1	flat	-	50	10
<i>steps3</i>	1	random	-	50	10
<i>rails1</i>	1	flat	-	50	20
<i>rails2</i>	1		flat	-	10
<i>rails3</i>	1		flat	-	10
<i>bars1</i>	1	flat	-	50	10
	2		-		
<i>bars3</i>	1	flat	-		
<i>ramp0</i>	1	random	- rocks1 rocks2	50 50	10
	3				
<i>ramp1</i>	4	random	-	50	10
	5				
	3				
<i>slope_rocks1</i>	4	random	-	50	10
	5				
<i>holes1</i>	1	random	-	50	10
<i>quarry</i>	10	random	-	50	10
Total: 1600					

Table 4.1. Maps configuration used in the simulator.

## 4.4 Postprocessing

After the run Krock on each map, we need to extract the patches for each stored pose  $p_i$  and compute the advancement for a given time window,  $\Delta t$ .

### 4.4.1 Parse stored data

First, we turn each `.bag` file into a pandas dataframe and cache them into `.csv` files. We used `rosbag_pandas`, an open source library we ported to python3, to perform the conversion. Then, we load the dataframes with the respective heightmaps and start the data cleaning process. We remove the rows corresponding to the first second of the simulation time to account for the robot spawning time. Then we eliminate all the entries where the *Krock* pose was near the edges of a map, we used a threshold of 22 pixels since we notice *Krock* getting stuck in the borders of the terrain during a simulation. After cleaning the data, we convert *Krock* quaternion rotation to Euler notation using the `tf` package from ROS. Then, we extract the sin and cos from the Euler orientation last component and store them in a column. Before caching again the resulting dataframes into `.csv` files, we convert the robot's position into heightmap's coordinates that are used later to crop the correct region of the map.

To compute the robot's advancement in a time window  $\Delta t$  we look for each stored pose,  $p_t$ , in the future,  $p_{t+\Delta t}$ , and see how far it went. This is described by the following equation:

ask omar

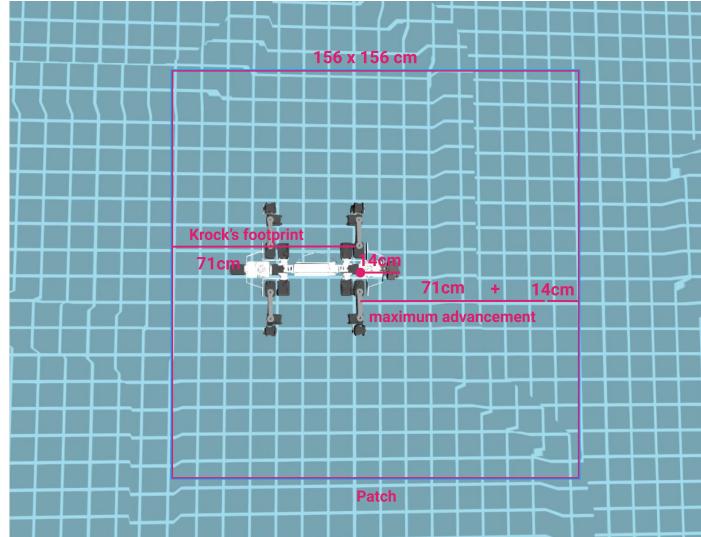
The correct value of  $\Delta t$  is crucial. We want a time window small enough to avoid smoothing too much the advancement and making obstacle traversing, and big enough to include the full legs motion. We empirically set  $\Delta t = 2$  since it allows Krock to move both its legs and does not flatten the advancement too much.

### 4.4.2 Extract patches

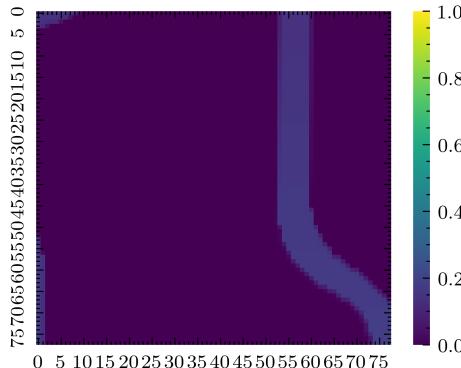
Each patch must contain both Krock's footprint, to include the situations where the obstacle is under the robot, and certain amount of ground region in front of it. Intuitively, we want to include in each patch the correct amount of future informations according to the selected time window. Thus, we should add the maximum possible ground that Krock could traverse.

To find out the correct value, we must compute the maximum advancement on a flat ground for the  $\Delta t$  and use it to calculate the final size of the patch. We compute it by running some simulations of *Krock* on flat ground and averaging the advancement getting a value of 71cm in our  $\Delta t = 2$ s.

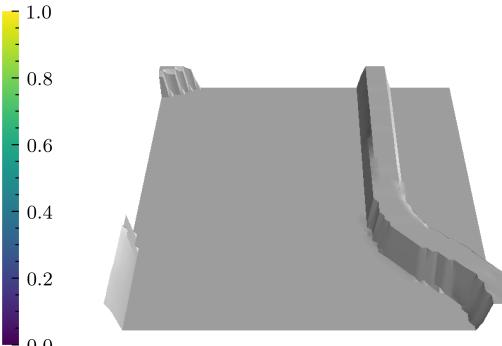
Each patch must include Krock's footprint and the maximum possible distance it can travel is a  $\Delta t$ . So, since Krock's pose was stored from IMU located in the juncture between the head and the legs, we have to crop from behind its length, 85cm minus the offset between the IMU and the head, 14cm. Then, we have to take 71cm, the maximum advancement with a  $\Delta t = 2$ s plus the removed offset. The following figure visualizes the patch extraction process.



(a) Robot in the simulator.



(b) Cropped patch in 2d.



(c) Cropped patch in 3d.

Figure 4.13. Patch extraction process for  $\Delta t = 2\text{s}$ .

Lastly, we create a final dataframe containing the map coordinates, the advancement, and the patches paths for each simulation and store them to disk as *.csv* files.

The whole pipeline takes less than one hour to run the first time with 16 threads, and, once it is cached, less than fifteen minutes to extract all the patches. In total, we created almost half a million images.

Once we extract the patches, we can always re-compute the advancement without re-running the whole pipeline. The next figure show the proposed pipeline.

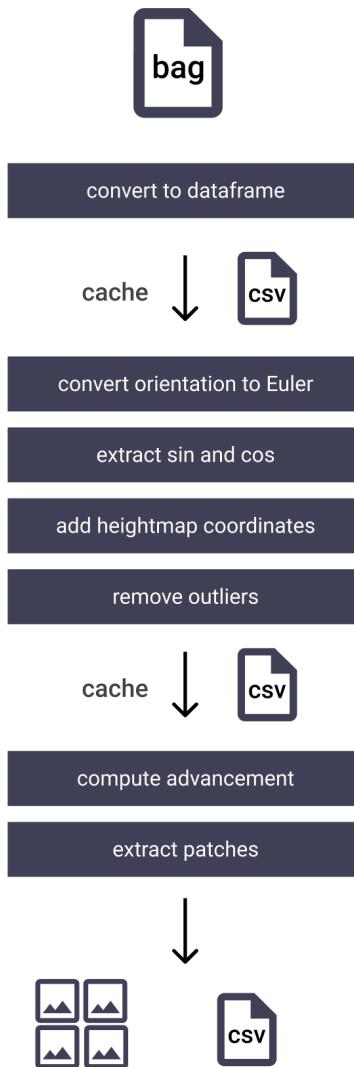


Figure 4.14. Postprocessing Pipeline flow graph, starting from the top.

The following figure shows the mean advancement across all the maps used to train the model in a range of  $\pm 0.70\text{cm}$ , the maximum advancement the used time window,  $\Delta t = 2\text{s}$ .

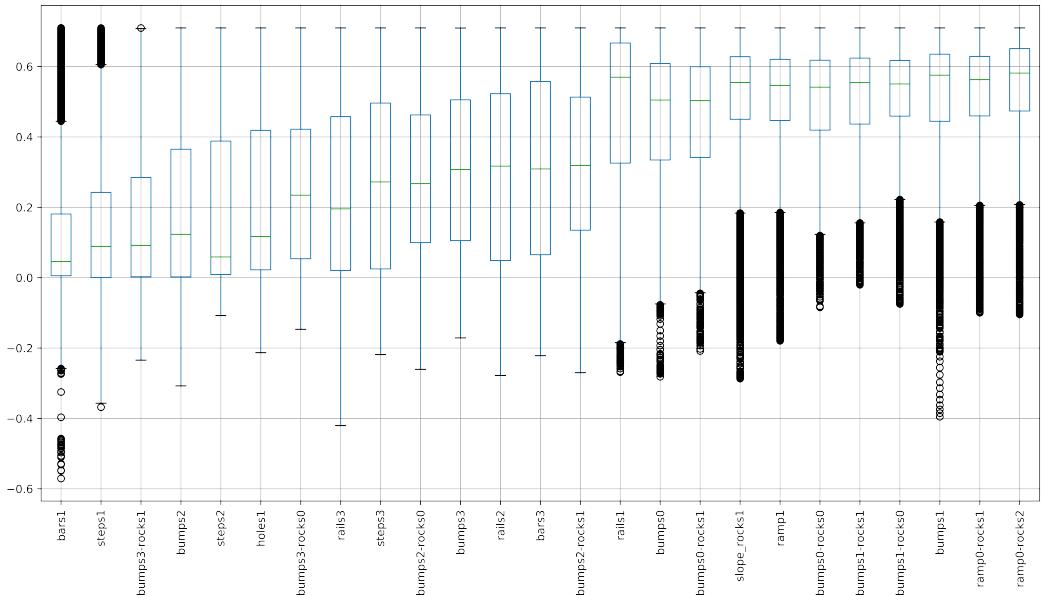


Figure 4.15. Advancement on each map with a  $\Delta t = 2\text{s}$  in ascendent order.

As sanity check, we visualize some of the patches with best and worst advancement. Patches with high advancement should not include big obstacle and viceversa.

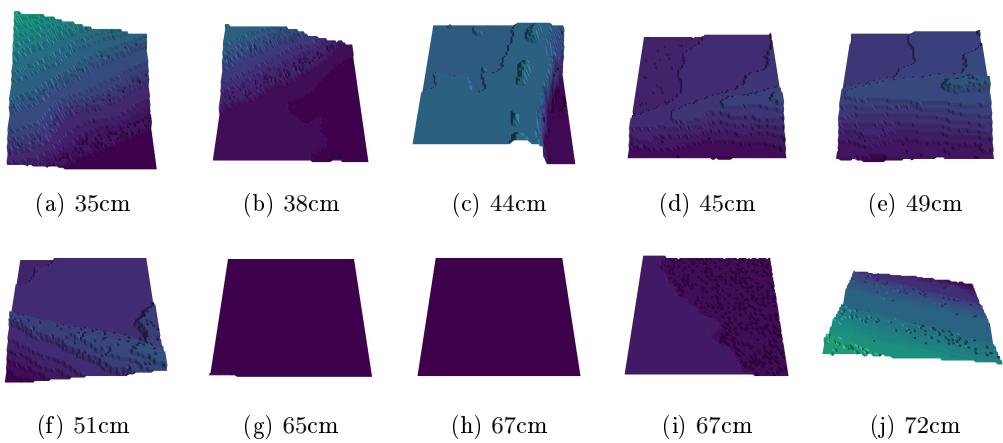


Figure 4.16. Patches with high advancement Quarry using a  $\Delta t = 2\text{s}$ .

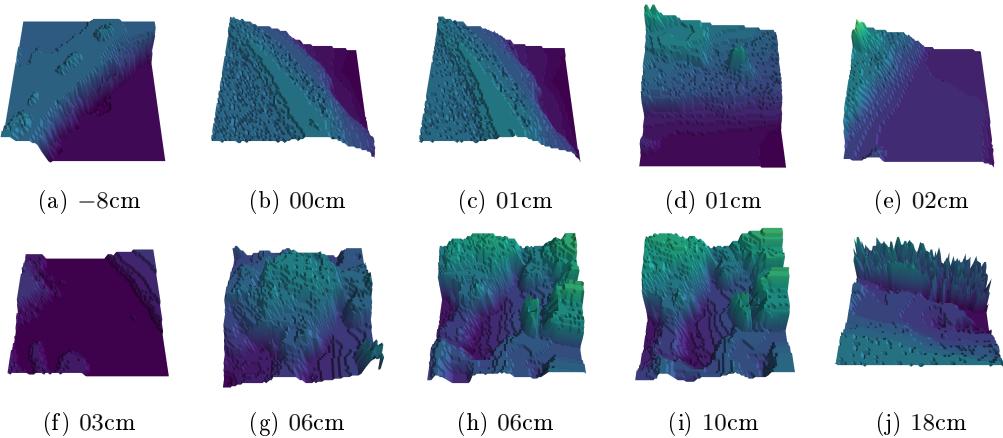


Figure 4.17. Patches with low advancement Quarry a  $\Delta t = 2\text{s}$ .

**make them gray**

Correctly, the patches with a high advancement have descent, small obstacle or they are flat. Contrarily, samples with low advancement have untraversable features such as bumps or too steep slopes.

All the handles used to postprocess the data are available as a python package. Also, we create an easy to use API called `pipeline` to define a cascade stream of function that is applied one after the other using a multi-thread queue to speed-up the process.

#### 4.4.3 Label patches

To decide whether a patch is traversable or not traversable we need to decide a *threshold*,  $tr$ , such as if a patches has an advancement major than  $tr$  it is label as traversable and viceversa. Formally, a patch  $p_i$  is label as *not traversable* if the advancement in a given time window  $\Delta t = 2$  is less than the threshold  $tr_{\Delta t=2}$

Ideally, the threshold should small enough to include as less as possible false positive and big enough to cover all the cases where Krock gets stuck. We empirically compute the threshold's value by spawning Krock in front of a bumps and a ramp and let it walk.

**Bumps** We spawned the robot on the *bumps3* map close to the end and let it go for 20 seconds. The following figure shows Krock in the simulated environment.

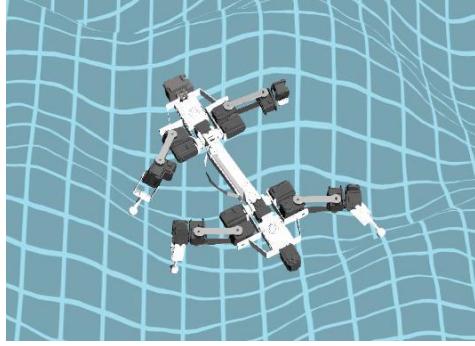
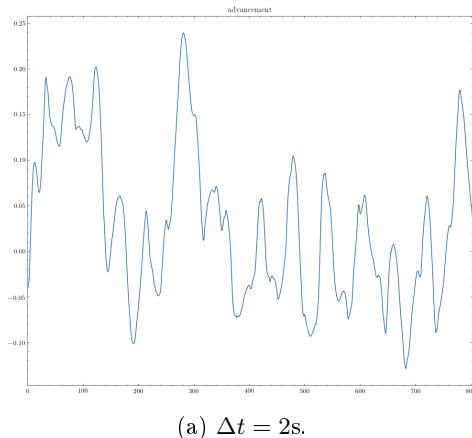


Figure 4.18. Krock tries to overcome an obstacle in the bumps3 map.

Due to its characteristic locomotion, Krock tries to overcome the obstacle using the legs to move itself to the top but it fell backs producing a spiky advancement where first it is positive and then negative. The following picture shows the advancement on this map with different time windows,  $\Delta t$ , the greater the smoother.



(a)  $\Delta t = 2\text{s}$ .

Figure 4.19. Advancement over time with  $\Delta t = 2$  on bumps3.

A wheel robot, on the other hand, will not produce such graph since it cannot free itself easily from an obstacle. Image a wheel robot moving forward, once it hits an obstacle it stops moving, if we plot its advancement in this situation it will look more like a step than a series of spikes.

**Ramps** The threshold should be small enough to not create any false positive, patches that are traversable but were classified as not. So, we let the robot walk up hill on the *slope-rocks1* map with a height scaling factor set to 5. We knew from empirical experiment that the robot is able to climb this steep ramp. The following figure shows the advancement.

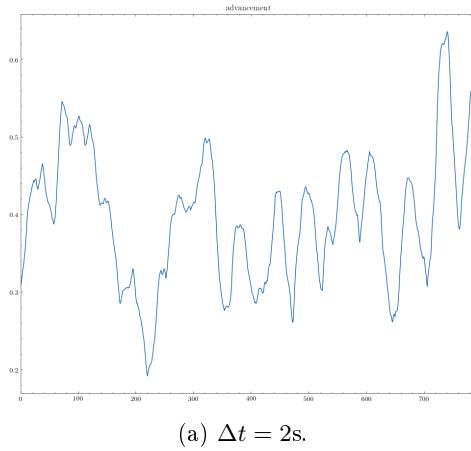


Figure 4.20. Advancement over time with  $\Delta t = 2\text{s}$  on slope\_rocks3.

The mean value is around 0.4cm over a  $\Delta t = 2\text{s}$  that is impressive considering the steepest of the surface. We can use this information combined with the previous experiment to choose a threshold that is between the upper bound of *bumps* and between the lower bound of *slope\_rocks1*. This will ensure to minimize the false positive. A good value is  $tr_{\Delta t=2\text{s}} = 20\text{cm}$ .

## 4.5 Estimator

In this section we described the choices behind the evaluated network architecture.

### 4.5.1 Vanilla Model

The original model proposed by Chavez-Garcia et all [6] is a CNN composed by a two  $3 \times 3$  convolution layer with 5 filters;  $2 \times 2$  Max-Pooling layer;  $3 \times 3$  convolution layer with 5 filters; a fully connected layer with 128 outputs neurons and a fully connected layers with two neurons.

### 4.5.2 ResNet

We adopt a Residual network, ResNet [8], variant. Residual networks are deep convolutional networks consisting of many stacked Residual Units : Intuitively, the residual unit allows the input of a layer to contribute to the next layer's input by being added to the current layer's output. Due to possible different features dimension, the input must go through and identify map to make the addition possible. This allows a stronger gradient flows and mitigates the degradation problem. A Residual Units is composed by a two  $3 \times 3$  Convolution, Batchnorm [11] and a Relu blocks. Formally defined as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + h(\mathbf{x}) \quad (4.1)$$

Where,  $x$  and  $y$  are the input and output vector of the layers considered. The function  $\mathcal{F}(\mathbf{x}, \{W_i\})$  is the residual mapping to be learn and  $h$  is the identity mapping. The next figure visualises the equation.

add resnet image or table

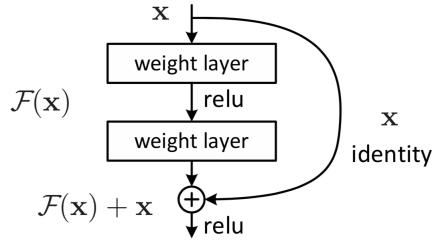


Figure 4.21. Resnet block [8]

When the input and output shapes mismatch, the *identity map* is applied to the input as a  $3 \times 3$  Convolution with a stride of 2 to mimic the polling operator. A single block is composed by a  $3 \times 3$  Convolution, Batchnorm and a ReLU activation function.

#### 4.5.3 Preactivation

Following the recent work of He et al. [9] we adopt *pre-activation* in each block. *Preactivation* works by just reversing the order of the operations in a block.

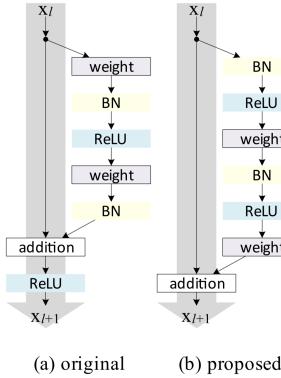


Figure 4.22. Preactivation [9]

#### 4.5.4 Squeeze and Excitation

Finally, we also used the *Squeeze and Excitation* (SE) module [10]. It is a form of attention that weights the channel of each convolutional operation by learnable scaling factors. Formally, for a given transformation, e.g. Convolution, defined as  $\mathbf{F}_{tr} : \mathbf{X} \mapsto \mathbf{U}$ ,  $\mathbf{X} \in \mathbb{R}^{H' \times W' \times C'}$ ,  $\mathbf{U} \in \mathbb{R}^{H \times W \times C}$ , the SE module first squeeze the information by using average pooling,  $\mathbf{F}_{sq}$ , then it excites them using learnable weights,  $\mathbf{F}_{ex}$  and finally, adaptive recalibration is performed,  $\mathbf{F}_{scale}$ . The next figure visualises the SE module.

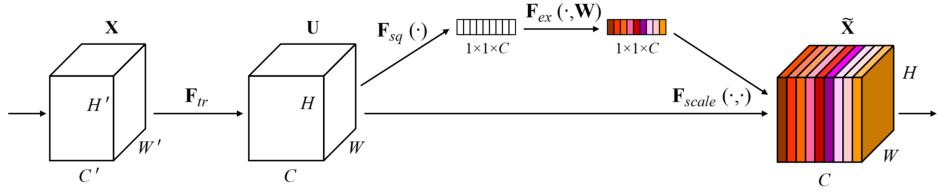


Figure 4.23. Squeeze and Excitation [10]

#### 4.5.5 MicroResNet

Our network is composed by  $n$  ResNet blocks, a depth of  $d$  and a channel incrementing factor of 2. Since ResNet assumed an input size of  $224 \times 224$  and perform an aggressive features extraction in the first layer, we called it *head*, as showed in [we decided to adopt](#) a less aggressive convolution. We tested two kernel sized of  $7 \times 7$  and  $3 \times 3$  with stride of 2 and 1 respectively. Lastly, we used LeakyReLU [4] with a negative slope of 0.1 instead of ReLU to allow a better gradient flow during backpropagation. LeakyRelu is defined as follows

$$\text{LeakyRelu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1x & \text{otherwise} \end{cases} \quad (4.2)$$

We called this model architecture *MicroResNet*. We evaluated  $n = [1, 2], d = 3$  with and without squeeze and excitation and with the two different *head*'s convolution. All the networks have a starting channel size of 16. The following table shows the architecture from top to bottom.

Input	(1, 78, 78)			
Layers		3 × 3, 16 stride 1	7 × 7 16 stride 2	
		2 × 2 max-pool		
		3 × 3, 16	x 1	
		3 × 3, 32		
	SE	-	SE	-
		3 × 3, 32	x 1	
		3 × 3, 64		
	SE	-	SE	-
		3 × 3, 64	x 1	
		3 × 3, 128		
Parameters	SE	-	SE	-
	average pool, 1-d fc, softmax			
Parameters	313,642	302,610	<b>314,282</b>	303,250
Size (MB)	5.93	5.71	<b>2.41</b>	2.32

Table 4.2. MicroResNet architecture. First layer is on the top. Some architecture's blocks are equal across models, this is shows by sharing columns in the table.

ref to Resnet Table

this table sucks

Our models have approximately 35 times less parameters than the smalles ResNet model, ResNet18, that has 11M parameters. To simplicity we will use the following notation to describe each architecture variant: **MicroResNet-3x3/7x7-/SE**.

ask omar help to add margin in the rows

add model picture

## 4.6 Normalization

Before feeding the data to the models, we need to make the patches height invariant to correctly normalize different patches taken from different maps with different height scaling factor. To do so, we subtract the height of the map corresponding *Krock*'s position from the patch to correctly center it. The following figure shows the normalization process on the patch with the square in the middle.

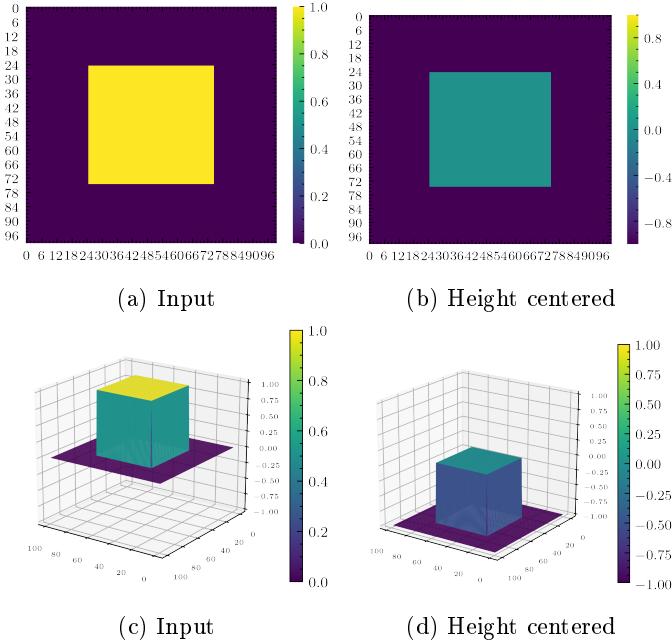


Figure 4.24. Normalization process. Each patch is normalized by subtracting the height value corresponding to the robot position to center its height.

## 4.7 Data Augmentation

Data augmentation is used to change the input of a model in order to produce more training examples. Since our inputs are heightmaps we cannot utilize the classic image manipulations such as shifts, flips, and zooms. Imagine that we have a patch with a wall in front of it, if we random rotate the image the wall may go in a position where the wall is not facing

the robot anymore, making the image now traversable with a wrong target. We decided to apply dropout, coarse dropout, and random simplex noise since they are traversability invariant. To illustrate those techniques we are going to use the same square patch showed before 4.24.

### 4.7.1 Dropout

Dropout is a technique to randomly set some pixels to zero, in our case we flat some random pixel in the patch.

### 4.7.2 Coarse Dropout

Similar to dropout, it sets to zero random regions of pixels with defined boundaries. Figure

### 4.7.3 Simplex Noise

Simplex Noise is a form of Perlin noise that is mostly used in ground generation. Our idea is to add some noise to make the network generalize better since lots of training maps have only obstacles in flat ground. Since it is computationally expensive, we randomly fist apply the noise to five hundred images with only zeros. Then, we randomly scaled them and add to the input image.

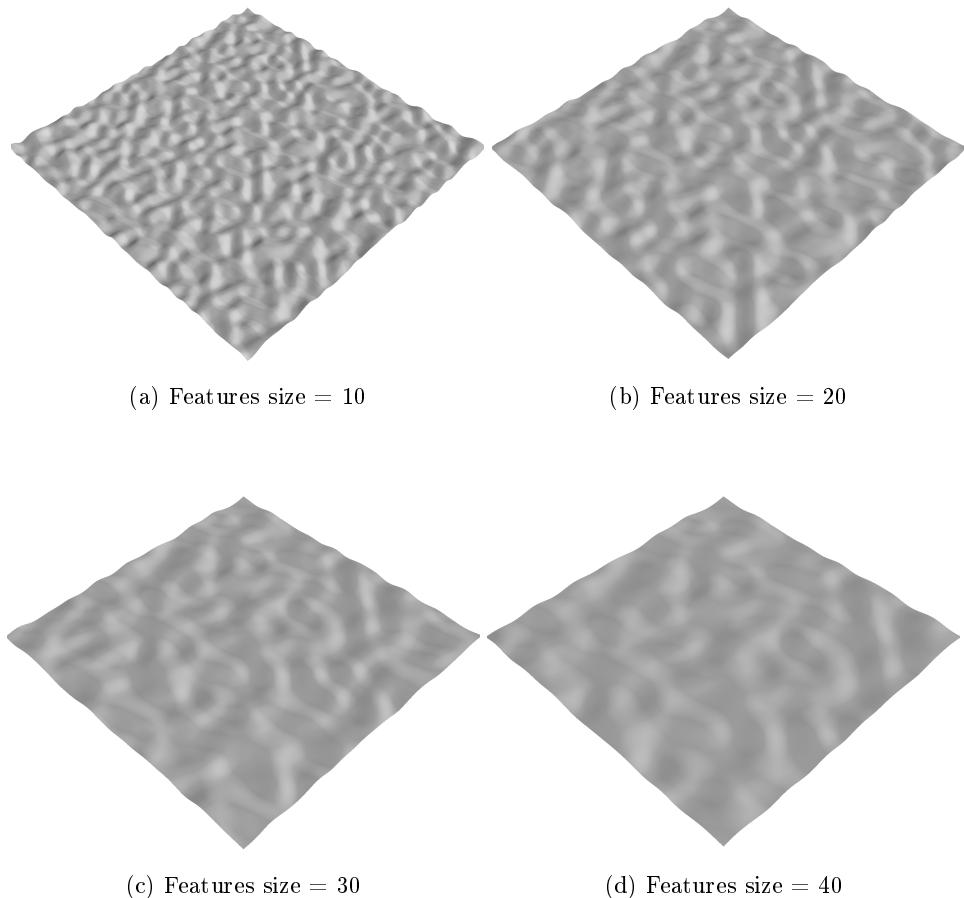


Figure 4.25. Simplex Noise on flat ground. The feature size corespond to the amount of noise we introduce in the surface. A lower value will produce rocky grounds, while a bigger one bumps.

The following images show the tree data augmentation techniques used applied the input image.

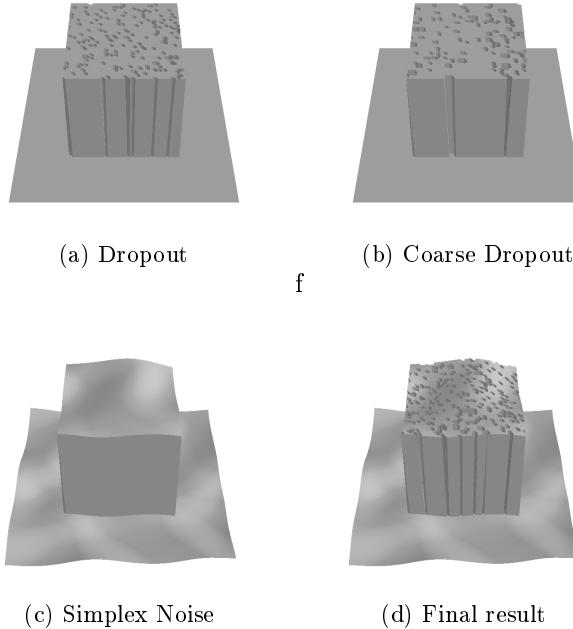


Figure 4.26. Data augmentation applied on a patch. First we randomly set to flat some small regions of the ground. Then, we slightly mutate the surface using simplex noise.

To ensure we do not create untraversable patches from traversable patch we limited the coarse dropout region to only few centimeters and we apply a smoother simplex noise on traversable patches. In detail,

in all the training epochs, we augmented 80% of the input images with dropout and coarse dropout. Dropout has a probability between 0.05 and 0.1 and coarse dropout has a probability of 0.02 and 0.1 with a size of the lower resolution image from which to sample the dropout between 0.6 and 0.8. Simplex noise was applied on the 70% of the training data samples with a feature size between 1 and 50 with a random scaling factor between 15 – 25 and 5 – 15 from traversable and not traversable patches respectively.

## 4.8 Tools

We quickly list the most important tools and libraries adopted in this project:

- ROS Melodic
- Numpy
- Matplotlib
- Pandas
- OpenCV

- PyTorch
- FastAI
- imgaug
- Blender

The framework was entirely developed on Ubuntu 18.10 with Python 3.6.

#### 4.8.1 ROS Melodic

The Robot Operating System (ROS) [18] is a flexible framework for writing robot software. It is *de facto* the industry and research standard framework for robotics due to its simple yet effective interface that facilitates the task of creating a robust and complex robot behavior regardless of the platforms. ROS works by establishing a peer-to-peer connection where each *node* is to communicate between the others by exposing sockets endpoints, called *topics*, to stream data or send *messages*.

Each *node* can subscribe to a *topic* to receive or publish new messages. In our case, *Krock* exposes different topics on which we can subscribe in order to get real-time information about the state of the robot. Unfortunately, ROS does not natively support Python3, so we had to compile it by hand. Because it was a difficult and time-consuming operation, we decided to share the ready-to-go binaries as a docker image.

#### 4.8.2 Numpy

Numpy is a fundamental package for any scientific use. It allows to express efficiently any matrix operation using its broadcasting functions. Numpy is used across the whole pipeline to manipulate matrices.

#### 4.8.3 Matplotlib

Almost all the plots in this report were created by Matplotlib, is a Python 2D plotting library. It provides a similar functional interface to MATLAB and a deep ability to customize every region of the figure. All It is worth citing *seaborn* a data visualization library that we inglobate in our work-flow to create the heatmaps. It is based on Matplotlib and it provides an high-level interface.

#### 4.8.4 Pandas

To process the data from the simulations we rely on Pandas, a Python library providing fast, flexible, and expressive data structures in a tabular form. Pandas is well suited for many different kinds of data such as handle tabular data with heterogeneously-typed columns, similar to SQL table or Excel spreadsheet, time series and matrices. We take advantages of the relational data structure to perform custom manipulation on the rows by removing the outliers and computing the advancement.

Generally, pandas does not scale well and it is mostly used to handle small dataset while relegating big data to other frameworks such as Spark or Hadoop. We used Pandas to store the results from the simulator and inside a Thread Queue to parse each .csv file efficiently.

#### 4.8.5 OpenCV

Open Source Computer Vision Library, OpenCV, is an open source computer vision library with a rich collection of highly optimized algorithms. It includes classic and state-of-the-art computer vision and machine learning methods applied in a wide array of tasks, such as object detection and face recognition. We adopt this library to handle image data, mostly to pre and post-process the heatmaps and the patches.

#### 4.8.6 PyTorch

PyTorch is a Python open source deep learning framework. It allows Tensor computation (like NumPy) with strong GPU acceleration and Deep neural networks built on a tape-based auto grad system. Due to its Python-first philosophy, it is easy to use, expressive and predictable it is widely used among researches and enthusiast. Moreover, its main advantages over other mainstream frameworks such as TensorFlow [2] are a cleaner API structure, better debugging, code shareability and an enormous number of high-quality third-party packages.

#### 4.8.7 FastAI

FastAI is a library based on PyTorch that simplifies fast and accurate neural nets training using modern best practices. It provides a high-level API to train, evaluate and test deep learning models on any type of dataset. We used it to train, test, and evaluate our models.

#### 4.8.8 imgaug

Image augmentation (imgaug) is a python library to perform image augmenting operations on images. It provides a variety of methodologies, such as affine transformations, perspective transformations, contrast changes and Gaussian noise, to build sophisticated pipelines. It supports images, heatmaps, segmentation maps, masks, key points/landmarks, bounding boxes, polygons, and line strings. We used it to augment the heatmap, details are in section 4.7

#### 4.8.9 Mayavi

Mayavi is a scientific data visualization and plotting in python library. We adopt it to render in 3D all the surfaces used in our framework.

# Chapter 5

## Results

In the section we show and evaluate the model’s results. We will start by presenting to the reader the networks score on each metric, then we will use the best model to predict the traversability in real world terrains.

### 5.1 Experiment setup

We run all the experiment on a Ubuntu 18.10 work station equipped with a Ryzen 2700x, a powerful CPU with 8 cores and 16 threads, and a NVIDIA 1080 GPU with 8GB of dedicated RAM.

### 5.2 Dataset

To perform classification, we select a threshold of 0.2m on a time window,  $\Delta t$ , of two seconds to label the patches, meaning that a patch with an advancement less than 20 centimeters is labeled as *no traversable* and viceversa. This processed is explained in detail in the previous chapter. While for the regression, we did not label the patch and directly regress on the advancement.

Initially, to train the models we first use Standard Gradient Descent with momentum set to 0.95, weight decay to  $1e - 4$  and an initial learning rate of  $1e - 3$  as was originally proposed in He et al. [8]. However, we later utilize Leslie Smith’s 1cycle policy [21] that allows us to trian the network faster and with an higher accuracy. We minimise the binary Cross Entropy for the classifier and the Mean Square Error (MSE) for the regressor.

#### 5.2.1 Experimental validation

We select as *validation* set ten percent of the training data. Since we store each run of Krock as a *.csv* file, validation and train set do not overlap. The test set is composed entirely by the Quarry map, a real world scenario. Table ?? tells in detail the configuration used in each map of all sets.

We also evaluate the model on the following additonal maps

add arck rocks

### 5.2.2 Metrics

**Classification:** To evaluate the model's classification performance we used two metrics: *accuracy* and *AUC-ROC Curve*. Accuracy scores the number of correct predictions made by the network while AUC-ROC Curve represents degree or measure of separability, informally it tells how much model is capable of distinguishing between classes. For each experiment, we select the model with the higher AUC-ROC Curve during training to be evaluated on the test set.

**Regression:** We used the Mean Square Error to evaluate the model's performance.

## 5.3 Quantitative results

### 5.3.1 Model selection

We compared two different *micro-resnet* and the *vanilla* cnn from the previous Chapter. We evaluate those models using a time window of two second, a threshold of 20cm and the data augmentation techniques described before. We run five experiments per architecture and we select the best performing network, the results are showed in the following table.

		Vanilla	MicroResnetSE	
			$3 \times 3$ stride 1	$7 \times 7$ stride 2
AUC	Top	0.892	0.888	<b>0.896</b>
	Mean	<b>0.890</b>	0.883	0.888
Params		974,351	313,642	314,282

Table 5.1. Model comparison on the test set.

Luca told me is better to split the models like Model1 and Model2 etc

Based on this data We select *micro-resnet* with squeeze and excitation and a starting convolution's kernel size of  $7 \times 7$  with stride of 2. This model has one third of the parameter of the origal model proposed by Chavez-Garcia et all [6].

As proof of work, we also train the best network architecture, MicroResnetSE with a first convolution's kernel size of  $7 \times 7$  and stride= 2, with and without the Squeeze and Excitation operator.

	MicroResnet $7 \times 7$	MicroResnet $7 \times 7$ -SE	Improvement
Top	0.875	<b>0.896</b>	+0.021
Mean	0.867	<b>0.888</b>	+0.021

Table 5.2. AUC top value and mean value for MicroResnet with a fist convolution of  $7 \times 7$  and stride = 2 with and without the SE module. The improvement is the same.

## 5.4 Final results

The following table shows in deep the score of the best network for each dataset.

Dataset			micro-resnet	Size	Resolution(cm/px)
Type	Name	Samples	ACC	AUC	
Synthetic	Training	429312	-	-	2
	Validation	44032	95.2 %	0.961	2
	Arc Rocks	37273	85.5 %	0.888	2
Real evaluation	Quarry	36224	88.2 %	0.896	2
	foo	TODO			
	baaa	TODO			

Table 5.3. Final results on different datasets.

## 5.5 Qualitative results

We qualitative evaluate the model predictions by plotting the traversability probability on different maps in 3D. We used a sliding window to extract the patches from the tested maps to collect the model's predictions. Then, we created a texture based on the traversable probability, we have colored it using a colormap and finally applied on the 3D model of each map. We evaluated four rotations, from bottom to top, top to bottom, left to right and right to left since those are the most human understandable. By using those angles, we can optimize the cropping process since there is not need to rotate each patch based on the Krock's head position. Instead, we can just rotate the entire map beforehand.

### 5.5.1 Quarry

The first map we evaluate is Quarry, this map is  $32 \times 32$ m long and has a maximum height of 10m. We can use some of the terrain interesting features, such as are three bis slopes and the rocky ground on top, to evaluate the model. For instance, we expect the trail on the slopes to be traversable at almost any rotation, especially from left to right and viceversa. While, the top part should be hard to traverse in almost any case. The following figure shows the traversability probability directly on the map.

some where place the colormap bar

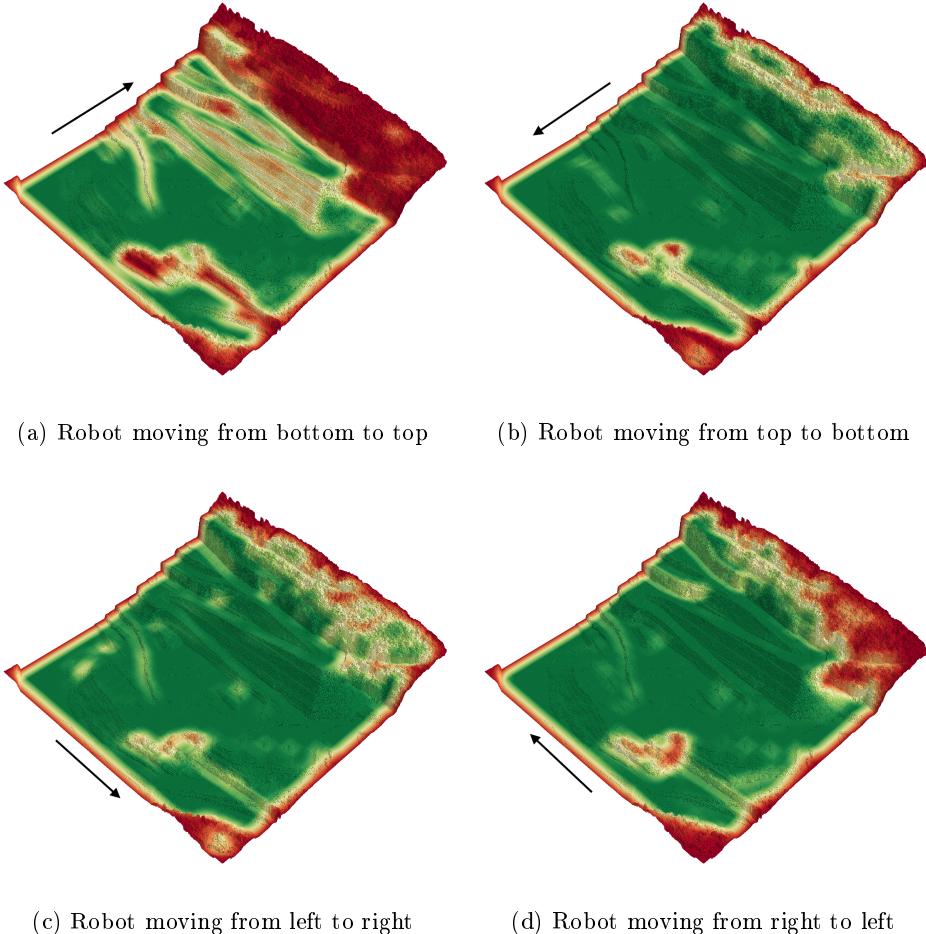


Figure 5.1. Traversability probability on the Quarry map,  $32 \times 32$ m, for different Krock's rotation. The values are obtained by sliding a window on the map to create the patches and then predict the traversability for each one of them.

Correctly, the lower part of the map, composed by flat regions, was labeled with high confidence as traversable in all rotations. On the other hand, the traversability of the slopes and the bumps on the top region depends on the robot orientation.

### 5.5.2 Bars

Bars is a map composed by different heights wall, thus we expect similar probabilities with different orientation.

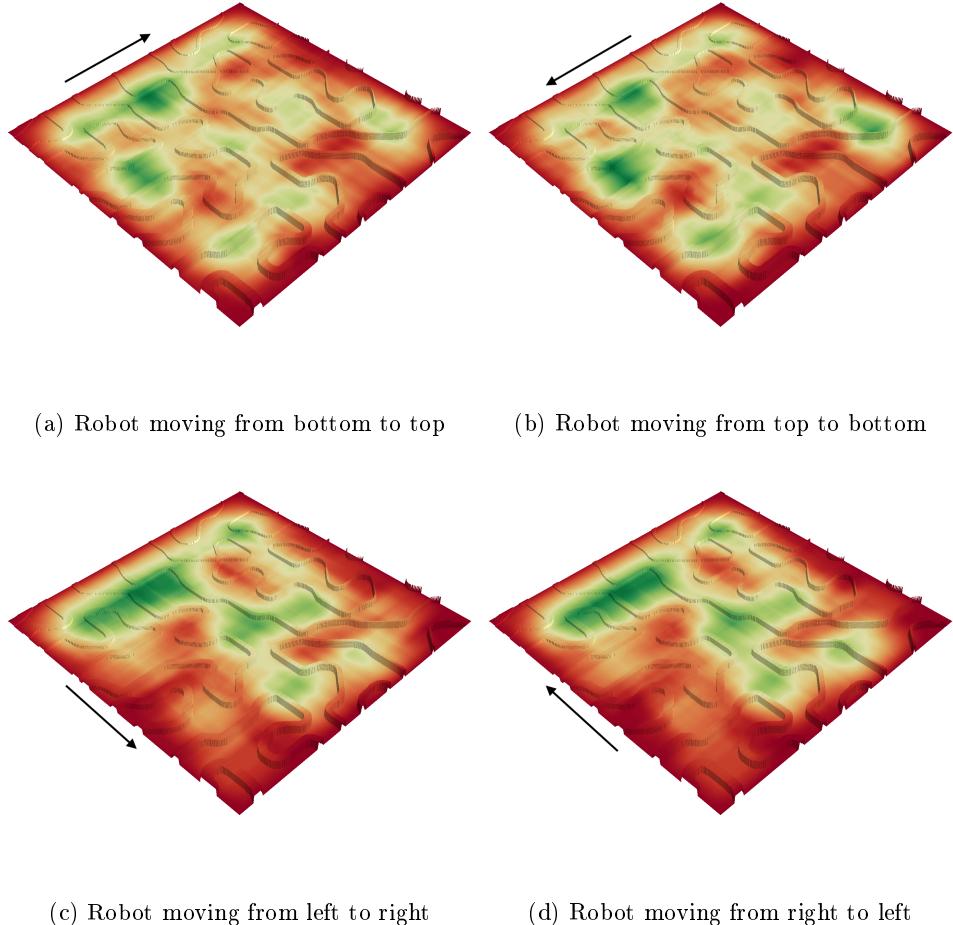


Figure 5.2. Traversability probability on the bars map,  $10 \times 10\text{m}$ , for different Krock's rotation. The values are obtained by sliding a window on the map to create the patches and then predict the traversability for each one of them.

This is a hard map for the robot due to the high number of not traversable walls. Interesting, we can identify a tunnel near the bottom center of the maps that shows how the model correctly label those patches depending on the orientation. The following figure highlight this detail.

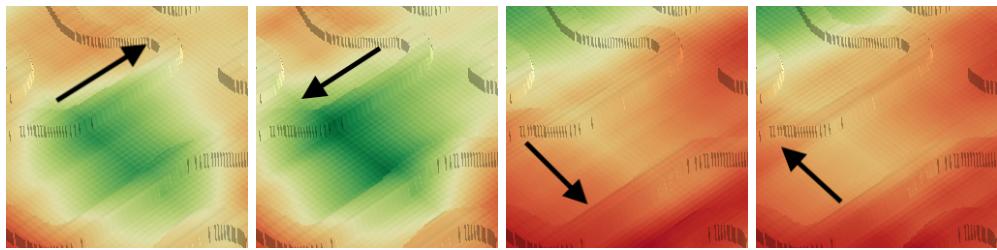


Figure 5.3. Detail of a region in the bars map where there are two walls forming a tunnel. Correctly, when the robot is travel following the trail the region is label as traversable.

### 5.5.3 Small village

The following image shows the traversability probability on the small image map.

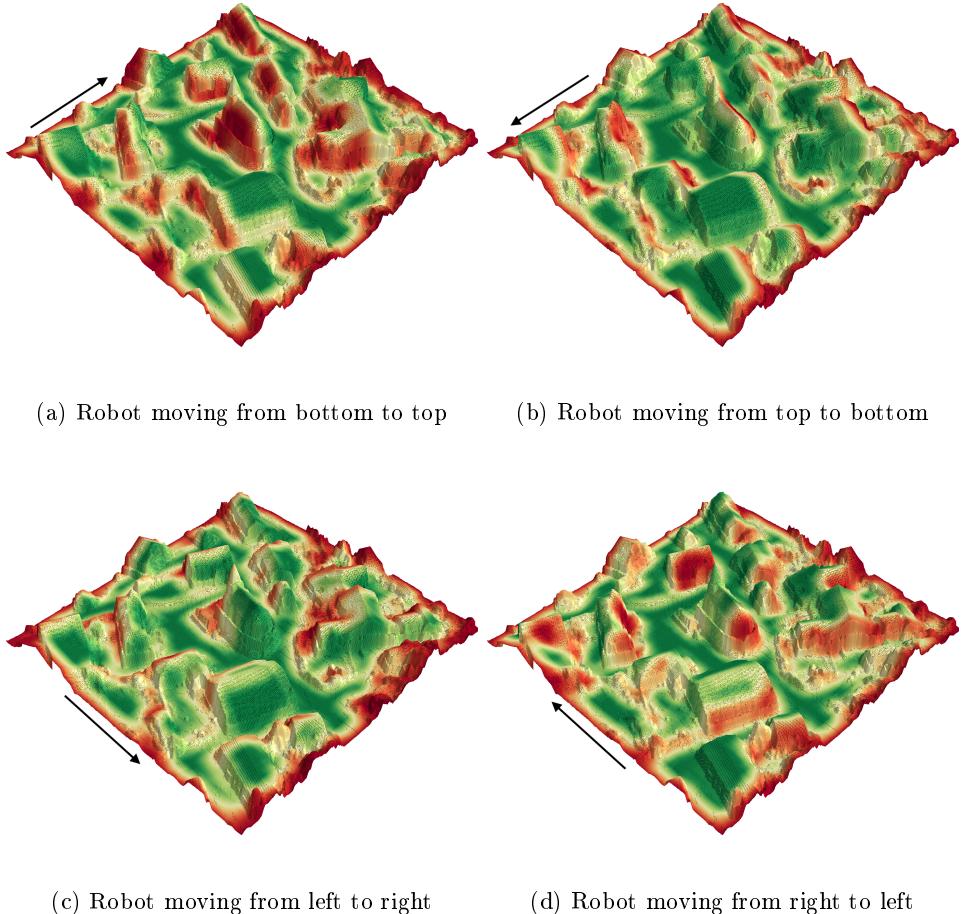
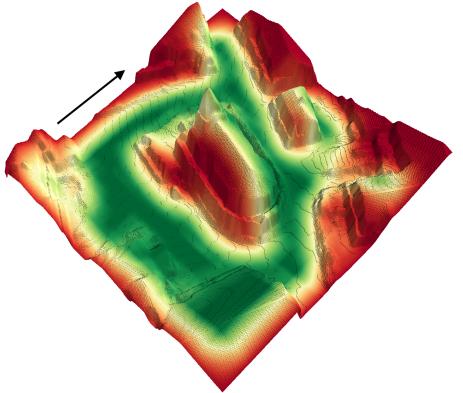
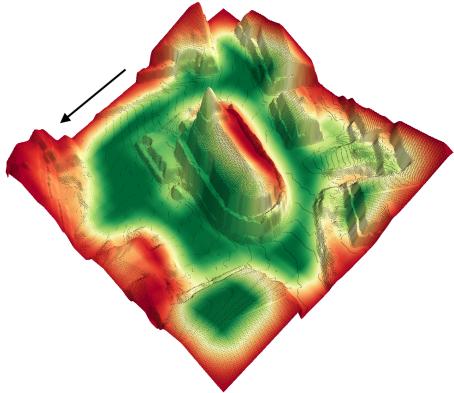


Figure 5.4. Traversability probability on the map of a small village for different Krock's rotation. The surface cover  $30 \times 30$ m and has a maximum height of 10m. The values are obtained by sliding a window on the map to create the patches and then predict the traversability for each one of them.

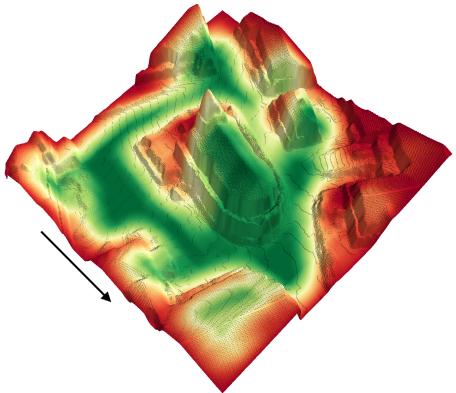
All the streets were labeled as traversable, while the roofs' traversability depend on the orientation. For example, most steep roofs are not traversable when walking up hill. On the other hand, if krock walk side by side they can be traverse. The following figure shows this behaviour on the church's roof.



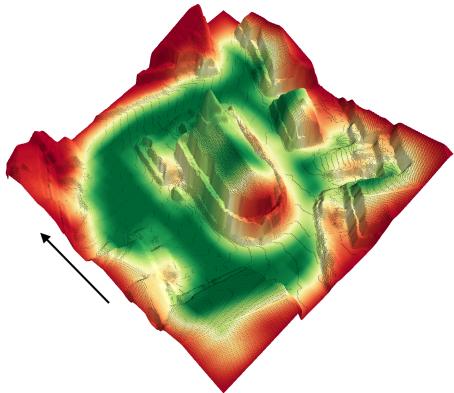
(a) Robot moving from bottom to top



(b) Robot moving from top to bottom



(c) Robot moving from left to right



(d) Robot moving from right to left

Figure 5.5. Detail of the church in the small village map for different Krock's rotation. We can notice for the first two image one part of the roof is traversable and viceversa. While for the last two the robot can correctly travel til the end.

This chapter is under development!

# Chapter 6

## Interpretability

In this section, we will evaluate the model's prediction to better understand it. We will find if there are any features in the patches that can confuse it and if the model's output is robust. First, we will show how the model learn to correctly separate the features from the two classes, then we will introduce one technique used to highlight the region of the input image that contribute the most to the model predictions. Then, we will use it on the data from the *Quarry* test set to find out the patches where the model fails and analyze them.

Later, we will work with custom created patches with different features, walls, bumps, etc, to test the robustness of the model by comparing its predictions to the real data gathered from the simulator.

### 6.0.1 Features separability

In general, convolutional neural network learns to encode images by applying filters of increasing sizes at each layer. Usually, the first layers learn basic features, such as edges, while the final one encodes complex shapes. Those final informations are combined and mapped to the correct classes by one or more fully connected. For example, the following image was generated by plotting the learned features for different categories at different layers by Lee et al. [15]

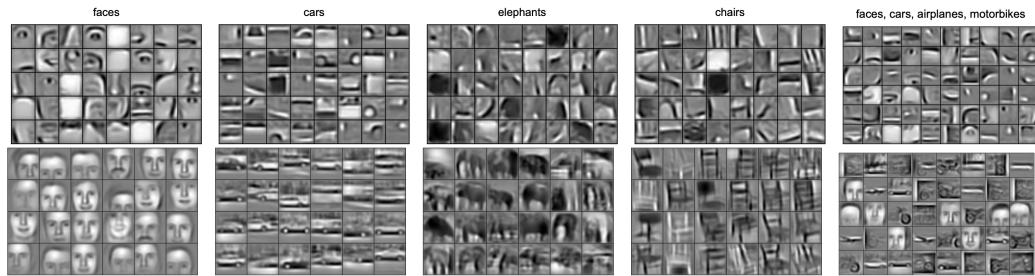


Figure 6.1. Figure from Lee et al. [15] paper where they shown for different classes the low-level features (up) and the high-level features (down) learned by a convolution neural network.

So, a correctly trained network should be able to separate those features based on

predicted classes. Intuitively, given two classes  $\mathcal{A}$  and  $\mathcal{B}$ , for example *cat* and *dog*, the high-level features for each class should not be the same, otherwise the model may output a wrong prediction since the same feature is mapped to different classes. Thus, visualizing the inputs features vectors against each class can give a good quality estimation of the model since highly separable features yield a correct learning procedure. Our model, MicroResNet, has a last convolution layer's features size of [128, 3, 3]. Since visualize a 128 dimension vector is impossible, we reduce its dimension to 2 by applying Principle Component Analysis (PCA) [20].

### 6.0.2 Train set

The following figures shows the features of 11K images sampled from the train set label with their classes, *traversable* and *not traversable*.

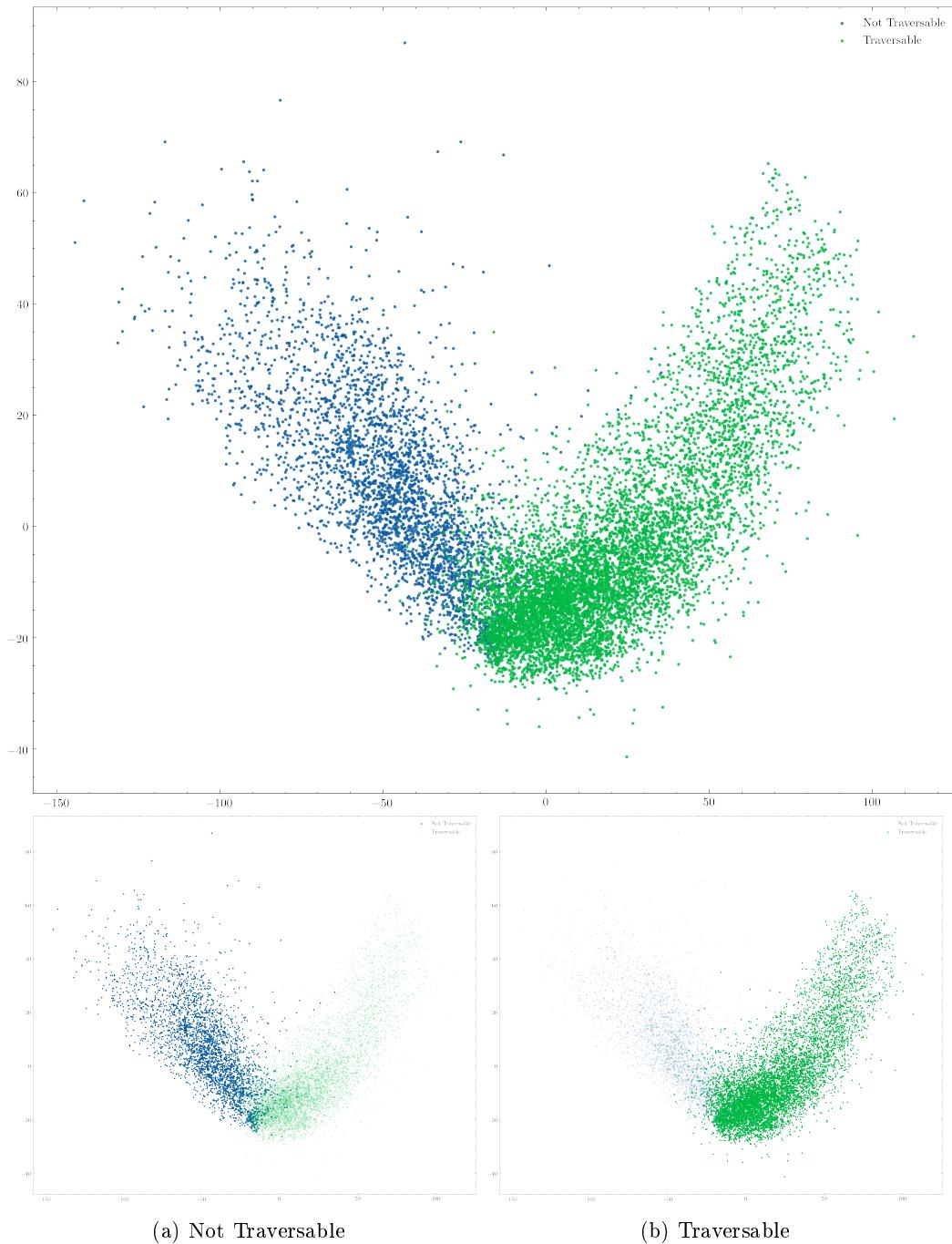


Figure 6.2. Principal Component Analysis on the features space computed using the features from the last convolutional layers on the train dataset.

We can clearly distinguish two main clusters based on the labels, one on the left and one on the right. Those points are easily separable, even by human eyes, meaning that

the model was able to learn meaning features from the dataset. Moreover, if we plot the density center, we can see that most of the samples per class are not very close.

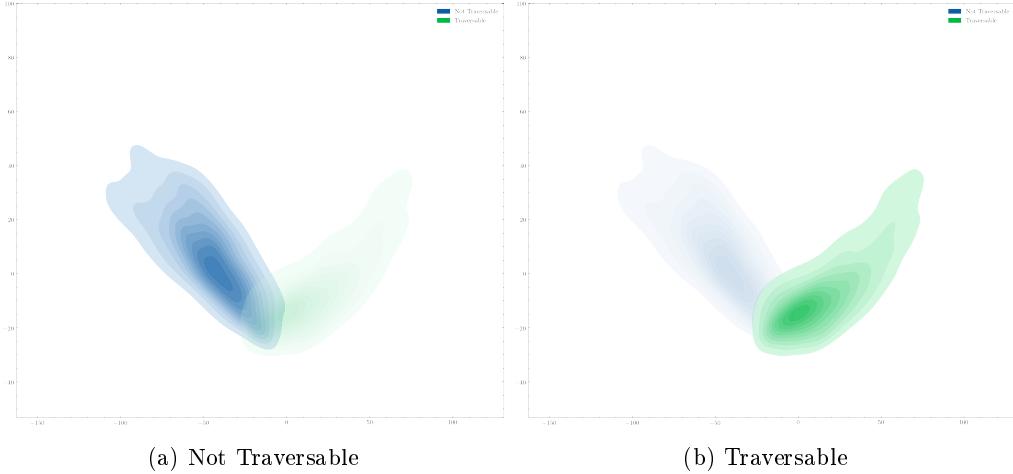


Figure 6.3. Density plot for the points sampled from the train dataset in the features space. The centers of the cluster are not overlapping yielding a good separability and a correct learning.

We can also identify patches based on their position in the features space, for example, the points on the top left cloud, are patches with walls or big bumps. On the other hand, the hardest patch to classify, or in separate in the features space, are the ones in the middle. Those are the patches with a relatively flat ground and small obstacles. One advantage of this procedure is that intrinsically, similar patches will be close to each other helping to identify clusters of inputs. We decided to not show all images on the same plot to avoid overcrowding the image. Instead, we clustered the points using K-Means with  $k = 200$  clusters and then we took the patch that corresponded to the center point in each cluster. In this way, even if we are showing only a few inputs, we included all the meaningful features. The following image shows the result.

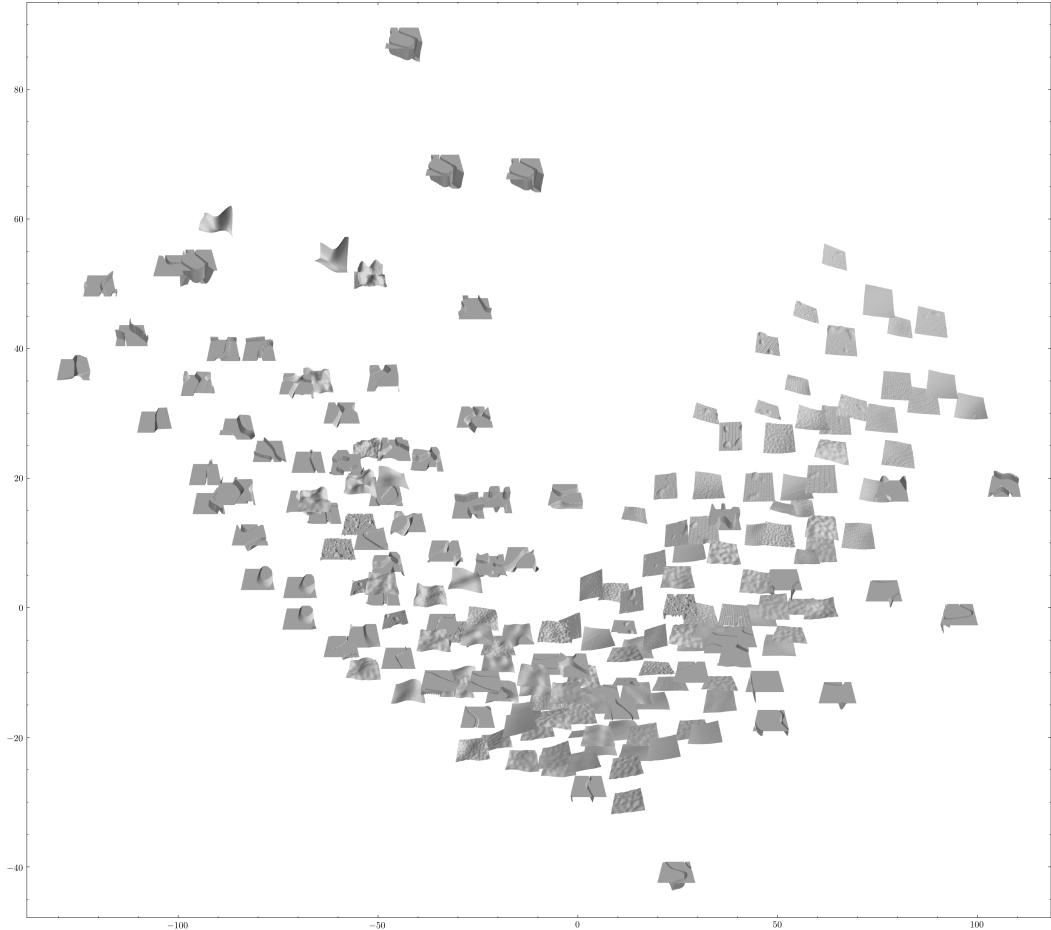


Figure 6.4. Patches that cooresponds to coordinates in the features space of the last convolutional layers on the train dataset. Similar grounds are close to each other.

Patches with similar features are close to eachother. On the left-top side, we can distinguish highly non traversable patches with walls/bumps in front of the robot. Going down, we start to encoter patches with ligher obstacles. On the platoue, there are traversable patches with small obstacles such as light bumps. It is important to notice that those patches are the closest ones to the non traversable ones, meaning that those are the most harder inputs for the model to classify. Going up on the right side, we see some small steps. Finally, on the top, we find all the downhill patches, the easiest ones to traverse.

### 6.0.3 Test set

We can apply the same procedure to evaluate the test set. This dataset is harder and present challanging situation for the robot, so we expected some points to be mixedup.

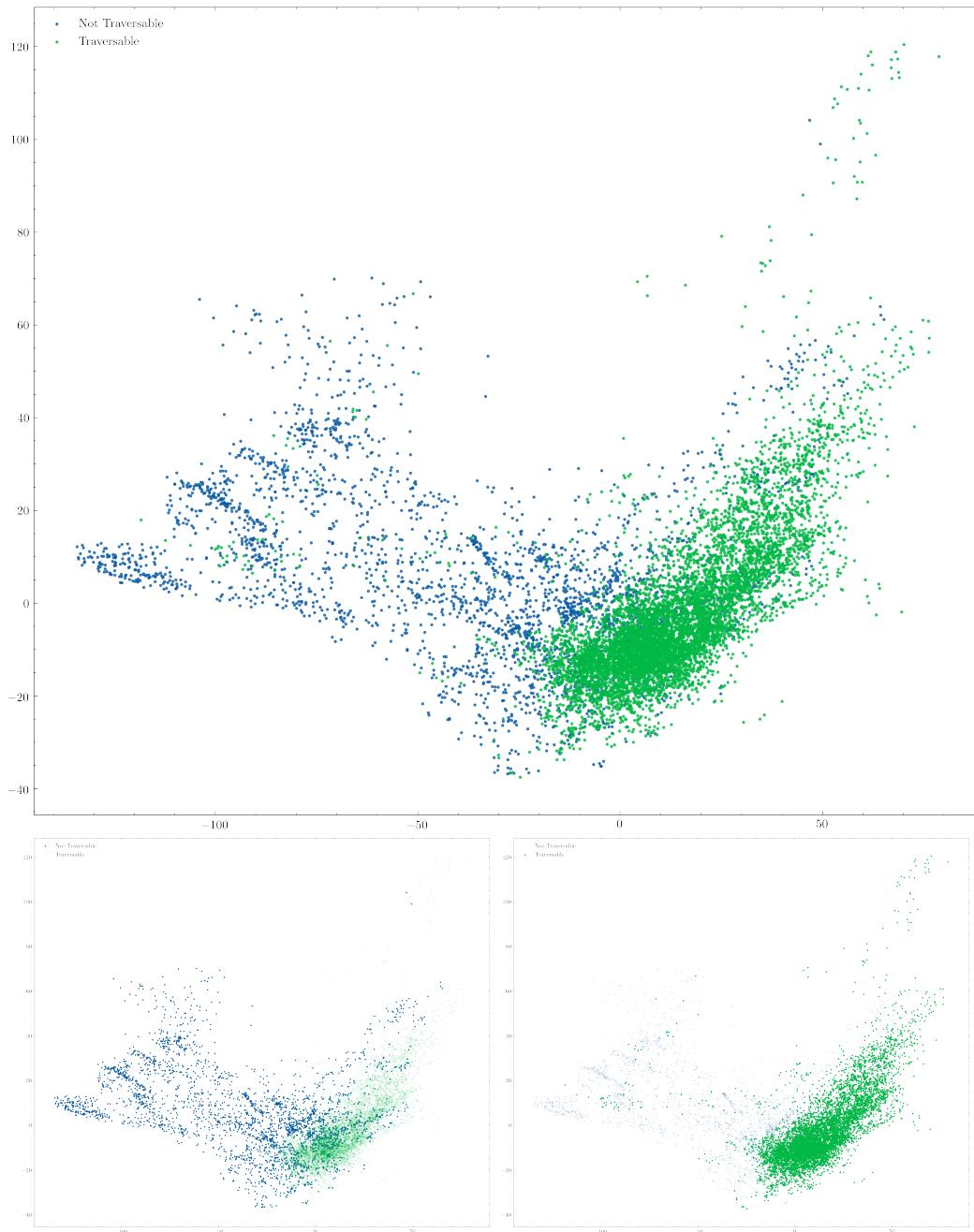


Figure 6.5. TODO

Interesting, the traversable patches are really close to each other, while the others spans a very big surface. This means that there are lots of different terrains with different features that are not traversable. The traversable points are clustered near the center. Moreover, some points are not perfect separable. We plotted the density for each class to better understand where the most points are clusted,

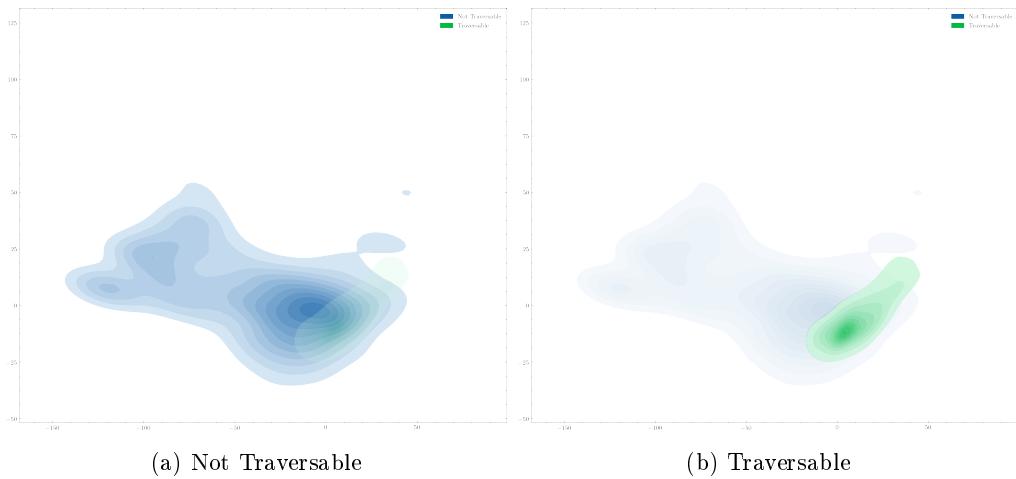


Figure 6.6. Density plot for the points sampled from the test dataset in the features space. The centers of the cluster are not overlapping yielding a good separability and a correct learning.

The two centers are really close to each other, this explain the lower AUC score than the validation set. We can also visualize the patches by plotting them using their features coordinates



Figure 6.7. Patches that corresponds to coordinates in the features space of the last convolutional layers on the test dataset. Similar grounds are close to each other.

On the top left, from the not traversable cloud, we can see patches with a high level of bumps. Going down we find surfaces with huge walls in front of the robot, while going close to the center we start to see all the traversable patches. Those samples have not too steep slopes. If we move to the density center, green bouble shown in figure ??, we encounter lots of flat patches with little obstacles. Going up on the right branch we find downhill and on the top there are falls.

In the following section we will take a deep look at the test set to find which patches confuse the model. Most probably, those samples will be located between the two clusters center where the difference between classes' features is minimum.

## 6.1 Grad-CAM

Gradient-weighted Class Activation Mapping (Grad-CAM) [19] is a technique to produce visual explanations for convolutional neural networks. It highlights the regions of the input image that contribute the most to the prediction.

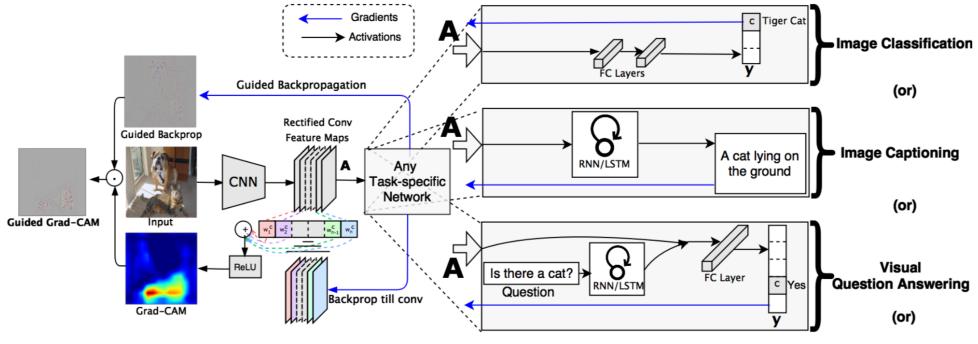


Figure 6.8. Grad-CAM procedure on an input image. Image from the original paper [19].

In detail, the output with respect to a target class is backpropagate while storing the gradient and the output in the last convolution. Then, global average is applied to the saved gradient keeping the channel dimension in order to get a 1-d tensor, this will represent the importance of each channel in the target convolutional layer. We each element of the convolutional layer outputs is multiplied with the averaged gradients to create the grad cam. This whole procedure is fast and it is architecture independent.

In detail, the output with respect to a target class is backpropagate while storing the gradient and the output in the last convolution. Then global average is applied the saved gradient keeping the channel dimension in order to get a 1-d tensor, this will represent the importance of each channel in the target convolutional layer. We each element of the convolutional layer outputs is multiplied with the averaged gradients to create the grad cam. This whole procedure is fast and it is architecture independent.

## 6.2 Robustness

To test the model's robustness we created custom patches with different features, walls,bumps,ramps, and test the model's predictions against the real robot advancement obtained from the simulator. Hopefully, model's outputs should match the ground truth or show a degree of uncertainty in the edge cases. According to the previous experiments, we used a threshold of 20cm and a time window of two seconds.

### 6.2.1 Untraversable wall ahead at increasing distance

The most trivial test is to place a not traversable wall in front of *Krock* at an increasing distance from the its head. At a certain distance, we expect the model's predictions to be traversable even if the wall itself is too tall. Why? Because the robot will be able to travel more than the threshold before being stopped by the obstacle.

We created fifty-five different patches by first placing wall 16cm long exactly in front of the robot and then move it by 1cm at the time towards the end. It follows some of the input patches ordered by distance from the robot. We remind to the reader that *Krock* traverse the patch from left to right.

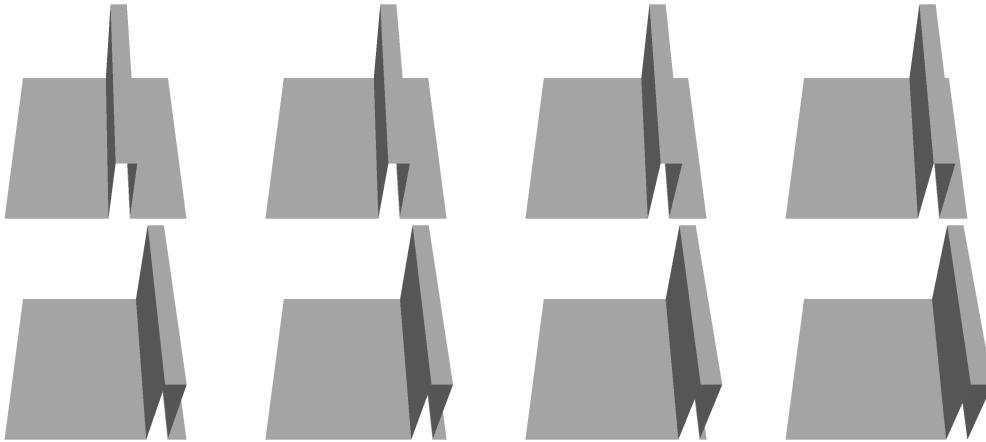


Figure 6.9. Some of the tested patches with a non traversable wall at increasing distance from the robot.

The model's predictions are shown in the following plot. We can see how the two classes invert their values around 20cm. Moreover, the predictions are uniform and do not change multiple times. Intuitively, if a wall is traversable from a certain distance, it will still be if we place even further.

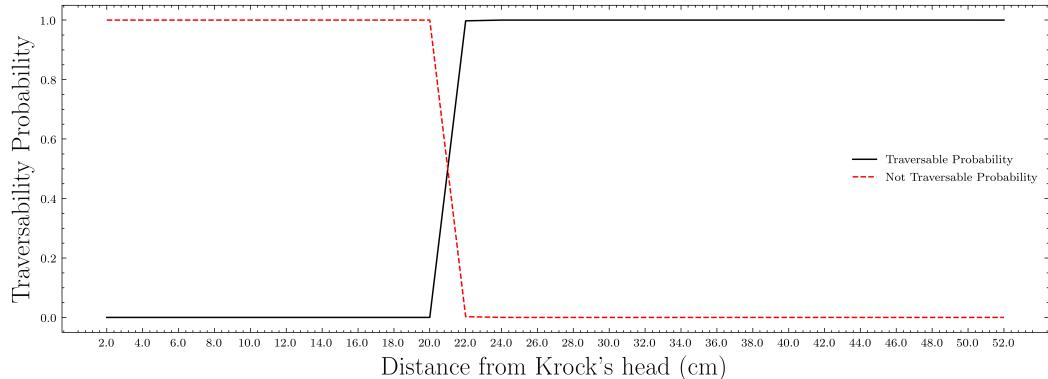


Figure 6.10. Traversability probabilities against wall distance from Krock's head.

graph too tall, adjust the figure size

Summarized by the following table:

Distance(cm)	Prediction
0 - 20	Not traversable
22 - end	Traversable

Table 6.1. Model prediction from the wall patches.

To be sure the results are correct, we run the last not traversable and the first traversable patch on the simulator to get real advancement. In the simulator, Krock advances 18.3cm

on the first not traversable patch ?? (a) where the wall is at 20cm from the robot's head. While, on the first traversable patch, with a wall at 22cm, the robot was able to travel for 20.2cm. Correctly, the network's predictions are supported by the ground truth obtained from the simulation. Moreover, the model correctly understand that the distance from the obstacle is more relevant than its height.

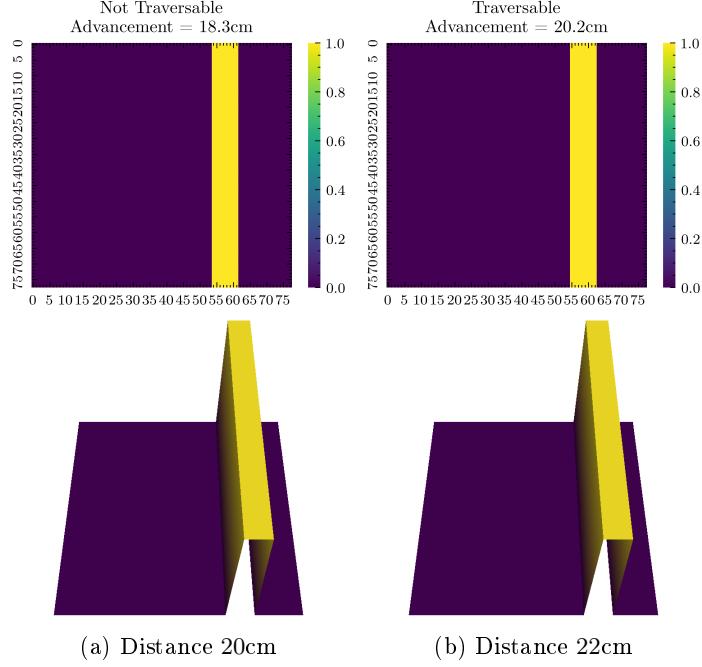


Figure 6.11. Correctly, when the distance between the robot and the wall is greater than the select treshold, in our case 20cm, the patch is label as traversable.

Furthermore, we increased the wall size of the first traversable patch, figure ??(b), to 10 and to 50m to see if the model will be confused. Correctly, the predictions did no change and those patches were still labeled as traversable

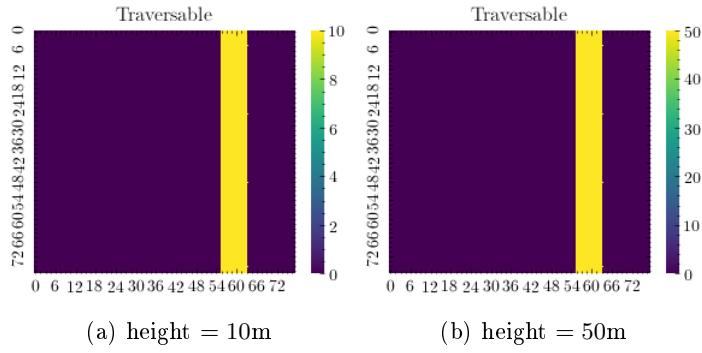


Figure 6.12. Two patches with a very tall wall at a distance  $> tr$ .

Correctly, the model classifies the patches as traversable and was not confused by the enormous height of the two walls.

### 6.2.2 Increasing height walls ahead

An other test we performed was to place walls in front of the robot with increasing heights to check whether the prediction matches the real data. We run forty patches in the simulator from a wall's height of 1cm to 20cm. The following figure shows some of the inputs.

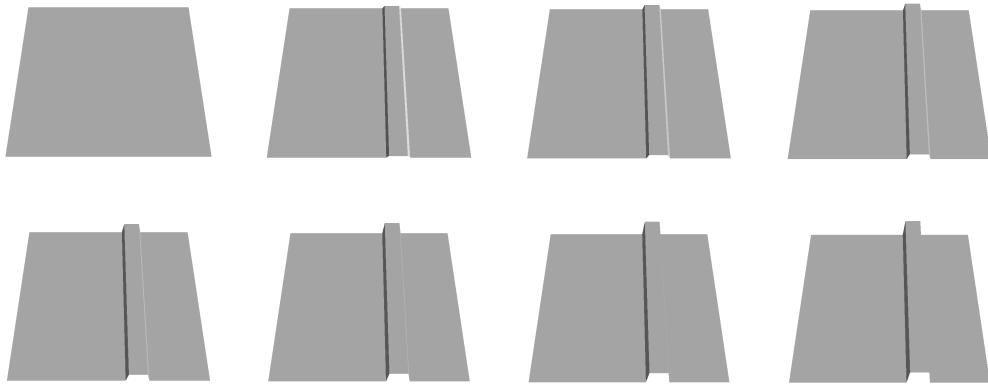


Figure 6.13. Some of the tested patches with a wall at increasing height ahead of Krock.

The models predicted that the walls under 10cm are traversable. We can see that in the edge case,  $\approx 10\text{cm}$ , the model's prediction change smoothly revealing a degree of uncertainty.

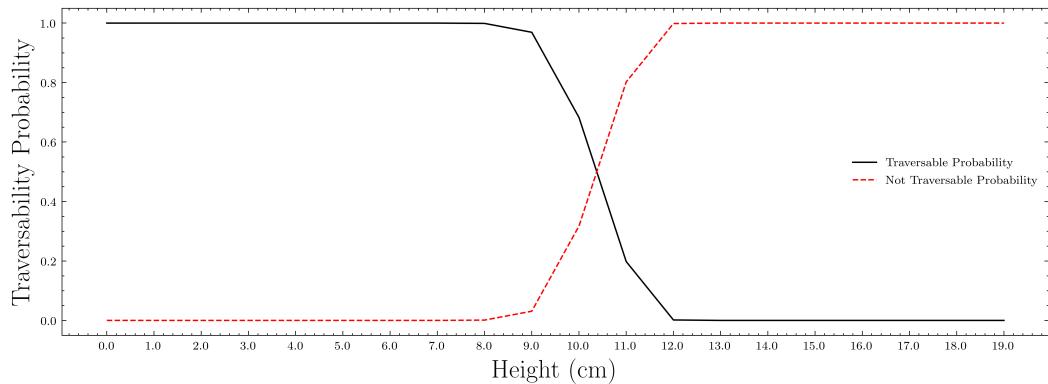


Figure 6.14. Traversability probabilities against walls height in front of Krock.

Height (cm)	Prediction
0 - 9	Traversable
10 - end	Not Traversable

Table 6.2. Model prediction for the wall patches

We can compare the model's prediction with the advancement computed in the simulator using the same approach from the last section. The following figure shows the results from the last traversable patch and the first non traversable.

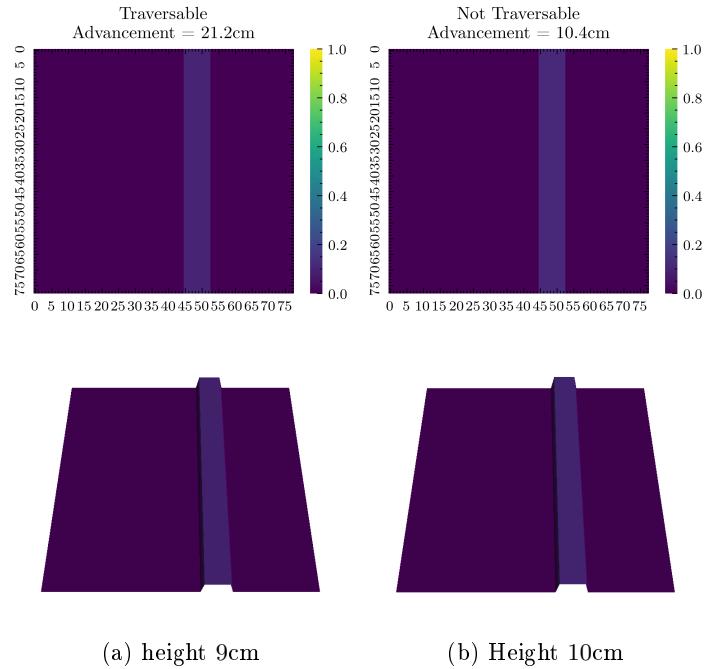


Figure 6.15. The last traversable and the first non traversable patches with a increasing height wall ahead of Krock. Correctly the model's prediction matches the advancement from the simulator.

In the first case, the simulator outputted an advancement of 21.2cm meaning that Krock was able to overcome the obstacle, while it failed in the second case. Correctly, the predictions matched the real data.

### 6.2.3 Increasing height and distance walls ahead.

We combined the previous experiments and tested the model predictions against the ground truth for each height/distance combination. To reduce the number of samples and improve readability, we limited ourselves to consider only patches with a wall tall between 5cm and 20cm, we know from previous sections patches with a value smaller and bigger obstacle are traversable and not traversable respectively. Similar, we set the wall's distance from Krock's head between 1cm to 30cm for the same reasons. To evaluate the model's prediction, we

run all the patches several time on the simulator and average the results. We highlighted the false positive and the false negative by red and blue respectively. Since we spawned the robot directly on the patch inside the simulator, the outputs may change across different runs. Sometimes, two runs on the same patch can produce different advancement due to some really small changes in the initial position of the robot caused the spawning procedure or some lag. So, to completeness, we also shown the variance between all simulation's runs to highlighted cases where the advancement changes the most across different runs.

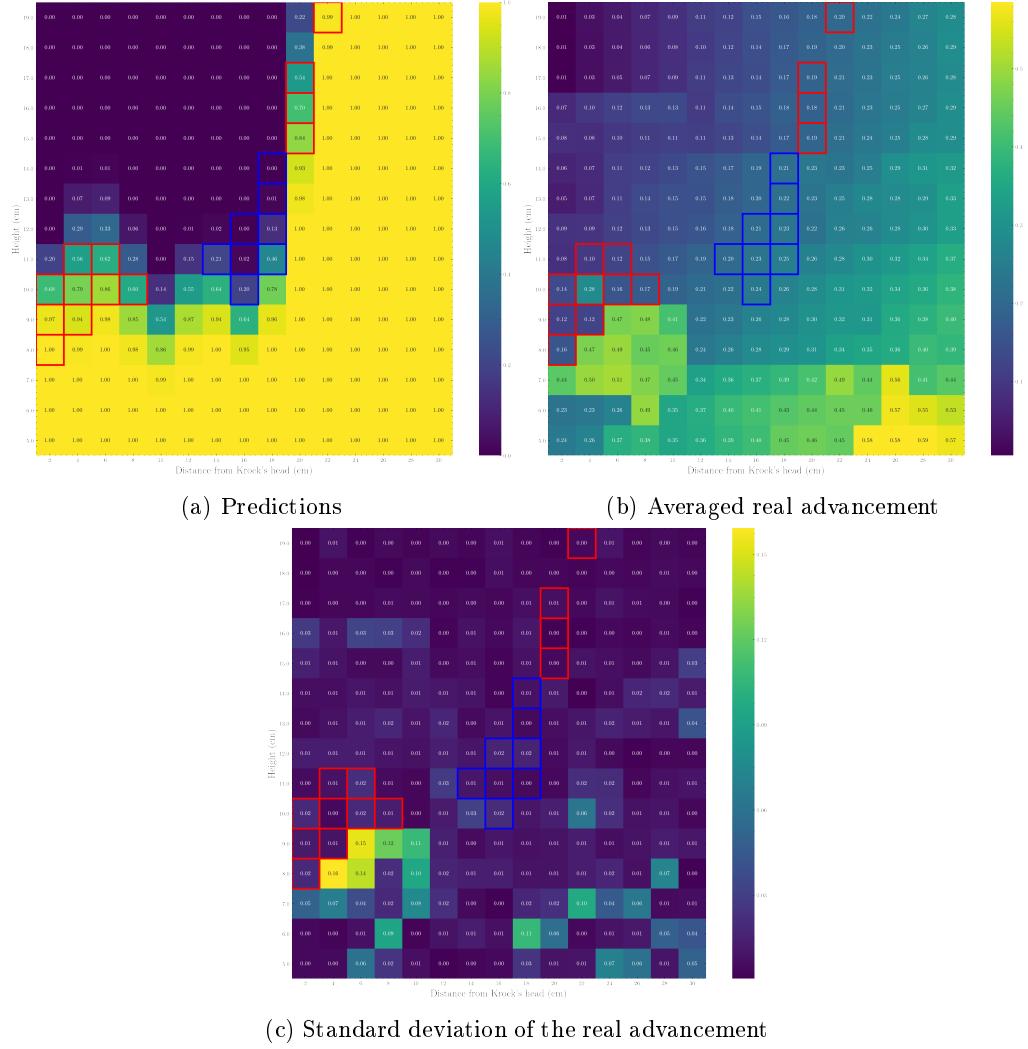


Figure 6.16. TODO

The model's fails to predicts some of the edge cases. The false positive are located in two regions: on the bottom left and on the top center. The first are the patches with a wall just ahead Krock of heights between  $\approx 8 - 11$  combination. The second cluster of false positive appears when the wall is at 20cm, the treshold. Even if the model failed to classify those inputs, it shows a correct degree of uncertainty.

The false negative, in the middle, are the inputs at distance close to the threshold and with a wall near the edge between traversable and not traversable. The overall accuracy is 91%. Even if the model wrongly classified some of the inputs, all those errors are in the edge cases where the predicted classes' probability is not maximum. Moreover, in most cases the model's shown uncertainty, especially on the false negative. Also, the prediction changes smoothly without any spikes accordendly to the features of the terrain. This shows a correct degree of understanding of the surface inputs. For instance, If the model outputs not traversable at height of 10cm and at distance of 16cm, then all the taller wall are correctly labeled as not traversable showing consistency and predibility.

**FINISH**

#### 6.2.4 Tunnel

**do it**

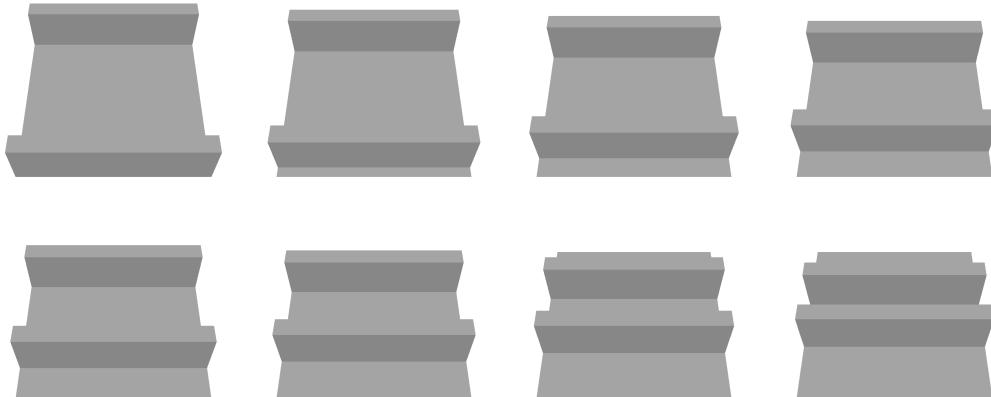


Figure 6.17. Some of the tested patches with tunnel at different distances.

#### 6.2.5 Ramps

**explain we had to square the linear ramps to create a small flat region**

We generate twenty ramps with a maximum height from 0.25m to 4m. Below we plot the traversability probabilities against the maximum height of each ramp.

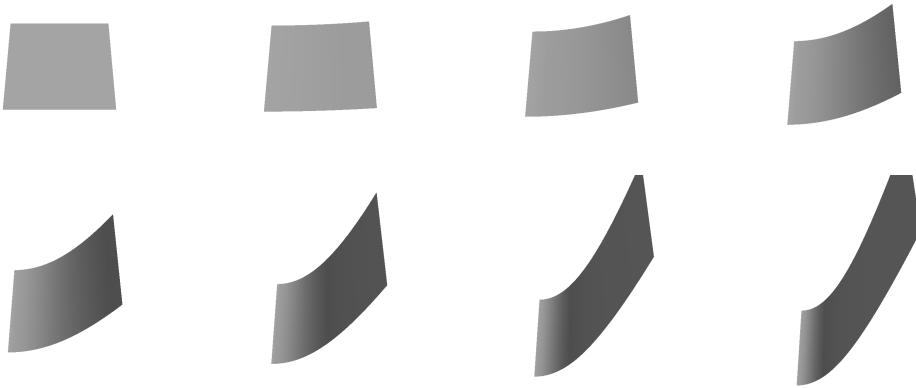


Figure 6.18. Some of the tested patches with steep ramps.

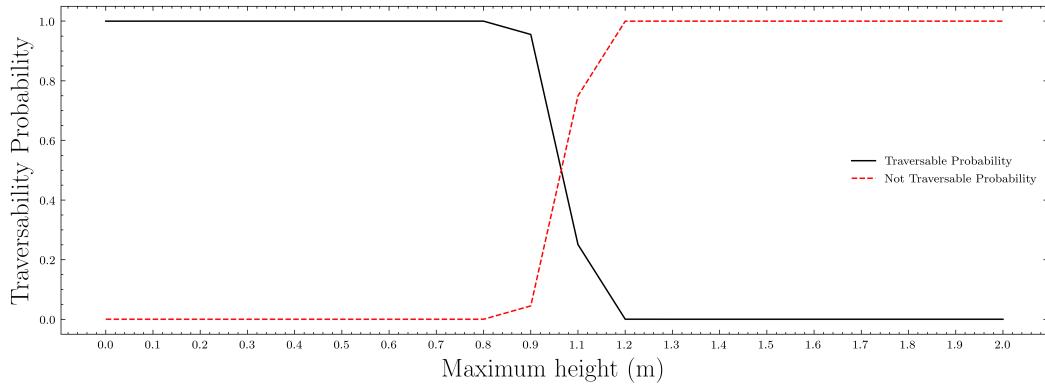


Figure 6.19. Traversability probabilities against maximum height of each ramp.

x labels are wrong, why?

The following table summarizes the results.

Height(m)	Prediction
0.5 - 1	Traversable
1 - end	Not traversable

Table 6.3. Model prediction for the ramps patches

We test the last traversable patch and the first not traversable with the real advancement gather from the simulator.

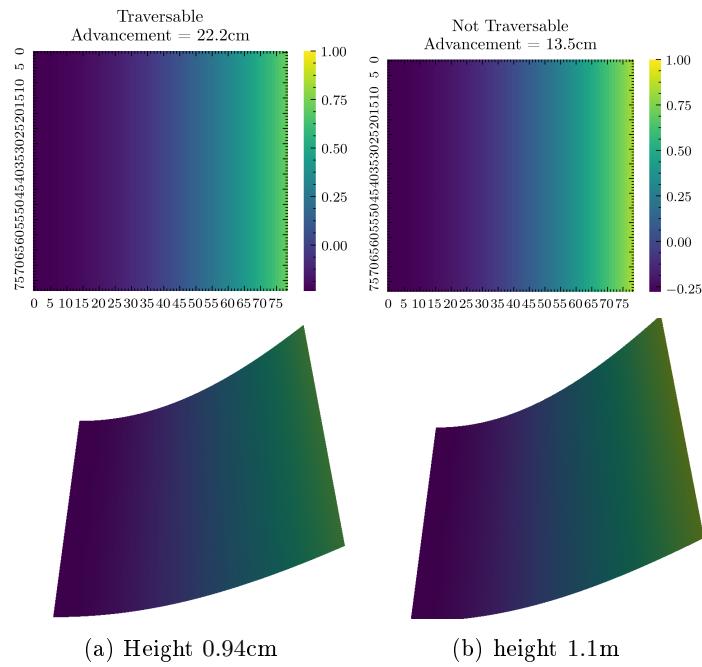


Figure 6.20. The last traversable and the first non traversable patches with a steep ramp ahead of Krock.

scale is wrong

Krock is able to traverse up to 1m height ramps, this is confirmed using the simulation.

We can add rocks to those patches to give Krock the ability to climb them better.

add rocks

### 6.2.6 Holes

do it



## Chapter 7

### Conclusions



# Bibliography

- [1] et al. A. Lagae. “"A survey of procedural noise functions"". In: *Comp. Graphics Forum* 29.8 (2010), pp. 2579–2600.
- [2] Martin Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. eprint: [arXiv:1605.08695](https://arxiv.org/abs/1605.08695).
- [3] Jérôme Guzzi Alessandro Giusti et al. “A Machine Learning Approach to Visual Perception of Forest Trails”. In: (2016).
- [4] Andrew Y. Ng Andrew L. Maas Awni Y. Hannun. “Rectifier Nonlinearities Improve Neural Network Acoustic Models.” In: () .
- [5] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: [arXiv:1606.01540](https://arxiv.org/abs/1606.01540).
- [6] R. Omar Chavez-Garcia et al. “Learning Ground Traversability from Simulations”. In: (2017). doi: [10.1109/LRA.2018.2801794](https://doi.org/10.1109/LRA.2018.2801794). eprint: [arXiv:1709.05368](https://arxiv.org/abs/1709.05368).
- [7] Jeffrey Delmerico et al. “On-the-spot Training” for Terrain Classification in Autonomous Air-Ground Collaborative Teams”. In: (2017).
- [8] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. eprint: [arXiv:1512.03385](https://arxiv.org/abs/1512.03385).
- [9] Kaiming He et al. *Identity Mappings in Deep Residual Networks*. 2016. eprint: [arXiv:1603.05027](https://arxiv.org/abs/1603.05027).
- [10] Jie Hu et al. *Squeeze-and-Excitation Networks*. 2017. eprint: [arXiv:1709.01507](https://arxiv.org/abs/1709.01507).
- [11] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: [arXiv:1502.03167](https://arxiv.org/abs/1502.03167).
- [12] Julia Nitsch Jeffrey Delmerico Elias Mueggler and Davide Scaramuzza. “Active Autonomous Aerial Exploration for Ground Robot Path Planning”. In: (2016).
- [13] Tobias Klamt and Sven Behnke. “Anytime Hybrid Driving-Stepping Locomotion Planning”. In: (2017).
- [14] M. Bloesch L. Wagner P. Fankhauser and M. Hutter. “Foot Contact Estimation for Legged Robots in Rough Terrain.” In: (2016).
- [15] Honglak Lee et al. *Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations*.
- [16] Wellhausen Lorenz et al. “Where Should I Walk? Predicting Terrain Properties from Images via Self-Supervised Learning.” In: (2019).
- [17] K. Perlin. “Improving noise”. In: *ACM Transactions on Graphics* ().

- [18] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: 2009.
- [19] Ramprasaath R. Selvaraju et al. *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization*. 2016. eprint: [arXiv:1610.02391](https://arxiv.org/abs/1610.02391).
- [20] Jonathon Shlens. *A Tutorial on Principal Component Analysis*. 2014. eprint: [arXiv:1404.1100](https://arxiv.org/abs/1404.1100).
- [21] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. 2018. eprint: [arXiv:1803.09820](https://arxiv.org/abs/1803.09820).
- [22] Boris Sofman et al. “Improving Robot Navigation Through Self-Supervised Online Learning”. In: (2006).
- [23] Matti Pietikainen Timo Ojala and Topi Maenpaa. “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns.” In: (2002).
- [24] E. Ugur and E. Sahin. “Traversability: A case study for learning and perceiving affordances in robots”. In: (2010).
- [25] Martin Wermelinger et al. “Navigation Planning for Legged Robots in Challenging Terrain.” In: (2016).