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. **?** and was originally tested on two wheels small robot. Collecting data in a simulation enviroment yields several advantanges 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 wasgenerated using thirty synthetic 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 or 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 qualitative 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 groun truth in different situations.

Contents

Chapter 1

Introduction

All living beings, including humans, need to traverse ground to eat, sleep and breed. Anymals 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 possibily to introduce bias if the data samples are not varied enough. Fortunately, ground can be generated articially and used in a simulated enviroment to let the robot walk in it while storing its interactions. Moreover, grounds maps can be easily obtained nowadays by using third-party services such as google maps or flying drone equiped with mapping technologies.

We proposed a full pipeline to estimate traversability tested on legged crocodile-like robot on uneven terrain based entirely on data gather trought 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 multiplice simulations can be launch in parallel keeping the cost of the real world hardware low. We first generated thirty synthetic maps encoded as heghtmaps 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 enviroment, position and orientation. Later, the procede 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 treshold. The treshold depends on the robot's locomotion and is calculated by spawing it in front of different obstacle and observing its advancement, we set the value for the tested robot to twenty centiments. Those patches are used to fit a deep convolutional neural network to prediction the traversability leaving to the network the task to extract important ground's features.

The report is organized as follow, the next chapter **??** introduces the related work, chapter **??** gives an high level overview of our approach, chapter **??** talk in deep 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 **?**. 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 **?**, 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. **?**, 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. **?** 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 **?** or Gaussian models **?**. Those techniques rely on texture descriptors, for example, Local Binary Pattern **?**, 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. **?** 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.**?** extended the previous work **?** 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. **?** 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. **?**.

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. **?** 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. **?** 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. **?** 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. **?**, 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 **??** shows the framework's steps.

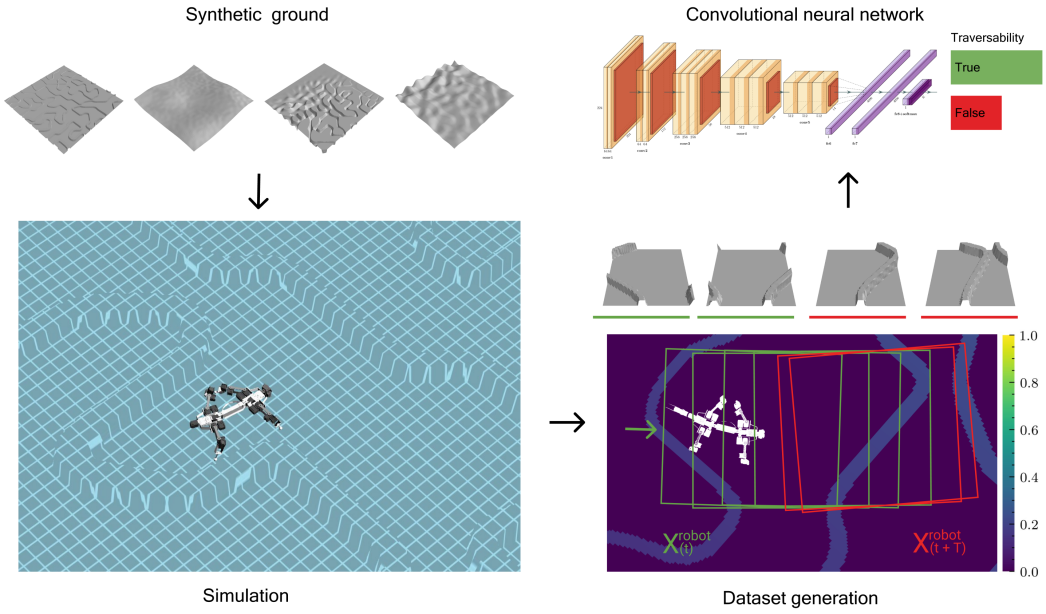
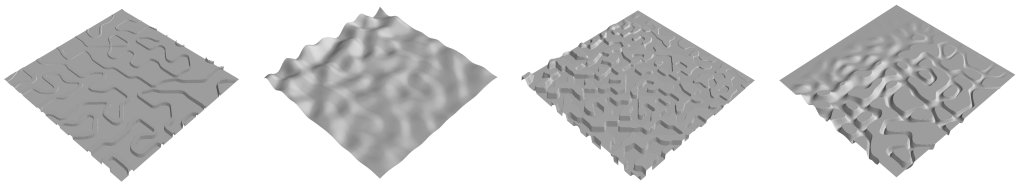


Figure 3.1. The framework's main building block in counter-clockwise order. First, we generated meaningful synthetic grounds, then we let the robot spawn and walk on them in a simulated environment while storing its interactions. Later, we crop a region of ground, a patch, for each simulation trajectory 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. To let the robot interacts with them, each map can be simply loaded into a simulator.



(a) Walls. (b) Bumps. (c) Steps. (d) Rails.

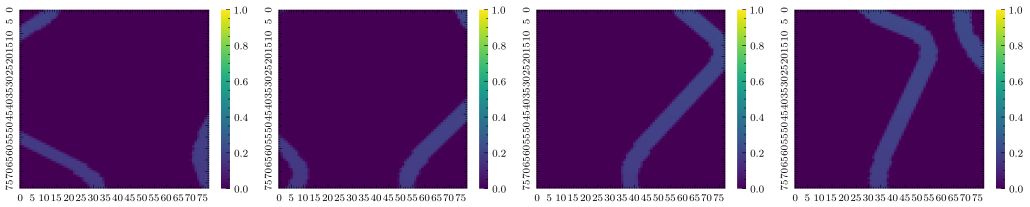
Figure 3.2. Some artificially generated terrains, 10×10m, with one specific feature each.

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.

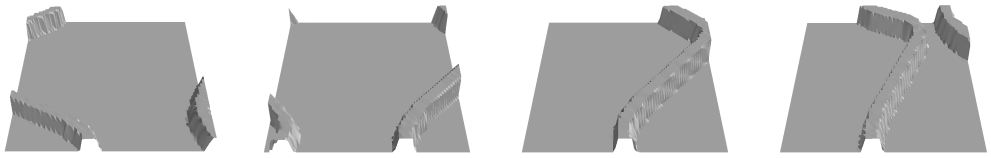
The robot position is tracked and stored during simulation in order to extract a small portion of ground, patches, around each of its trajectory's position. In practice, the patches are cropped from the original image using a specific size based on the robot's footprint and its velocity. Each resulting 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 in explain in detail in section **??**. while

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 qualitively shown in chapter **??**. Later, in chapter **??**, 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.

(a) Robot's trajectory.



(b) Cropped patches in 2D.



(c) Cropped patches in 3D.

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.

Lastly, in section **??**, 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 understandood 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 sections talks in detail about the implementation of the framework. We will first talk about from 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 framewotk on a a legged crocodile robot called *Krock* developed at EPFL. The next figure shows the robot in real life with a cloat and in a simulated enviroment.

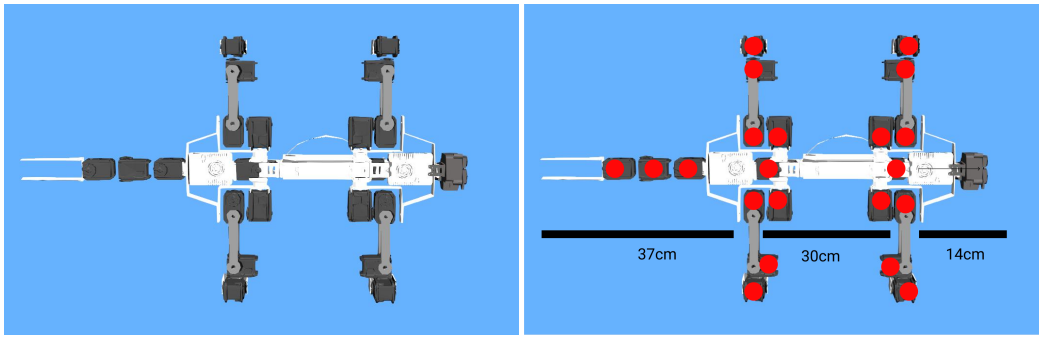
(a) Krock in real life.

(b) Krock in the simulator.

Figure 4.1. *Krock*

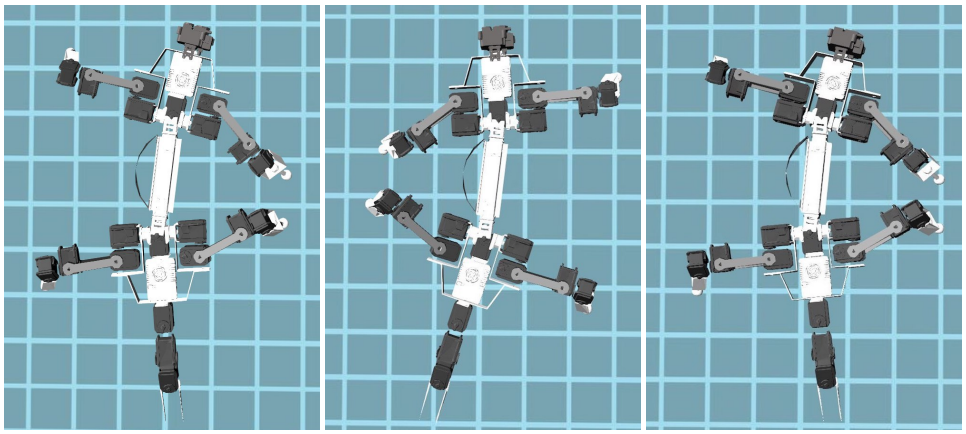
Krock has four legs, each one of them is equipped with three motors in order to rotate in each axis. The robot can raise itself in three different configurations, gaits, using the four motors on the body connected to the legs. In addition, there are another set of two motors in the inner body part to 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 weights 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. *Krock*'s moves by lifting and



(a) Top view of *Krock* (b) Details highlighted in *Krock*

moving forward one leg after the other. The following figure shows the robots going forward. In



(a) $t = 1$ (b) $t = 2$ (c) $t = 3$

Figure 4.3. *Krock* moving forward.

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

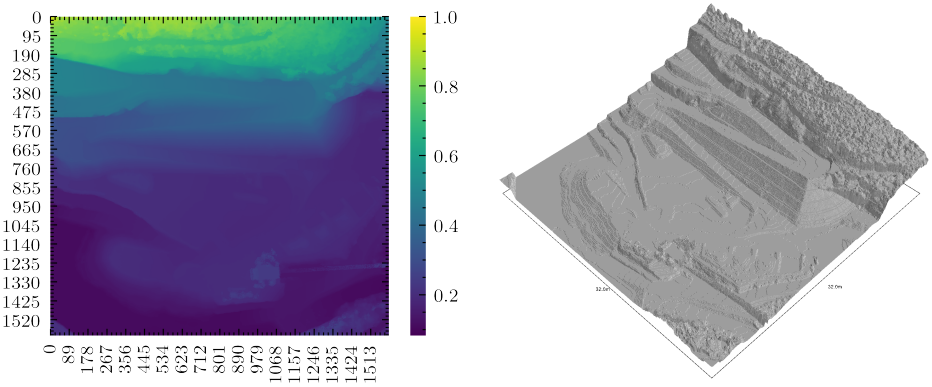
4.2 Data Gathering

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 throught 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. We created thirty maps of 513×513 pixel with a



(a) Heightmap. (b) 3D rendered

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

resolution of $0.02cm/pixel$ in order to represent a $10 \times 10m$ terrain using 2D simplex noise **?**. Simplex noise is a variant of Perlin noise **?**, a widely used technique in the terrain generation litterature. We divide the maps into five main categories of terrains to cluster differents 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 genered four different maps with increasing bumps' height using simplex noise with features size $\in [200, 100, 50, 25]$.

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

Rails: Flat grounds with slots.

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

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.

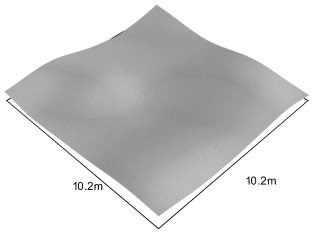
Holes We also included a map with holes

4.2.2 Simulator

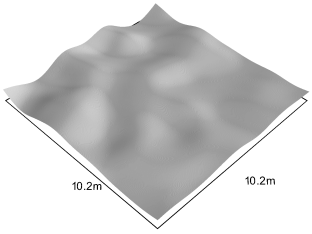
We used Webots to move *Krock* on the genereted terrain. The robot controlled was implemented by EPFL and handed to IDSIA . The controller implements a ROS' node to publish *Krock* status including its pose at a rate of $250hz$. We decide to reduce it to $50hz$ by using ROS build it

cite them?

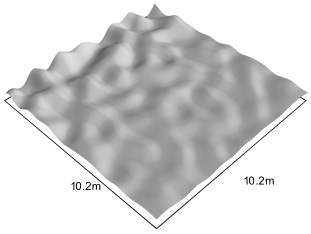
cite?



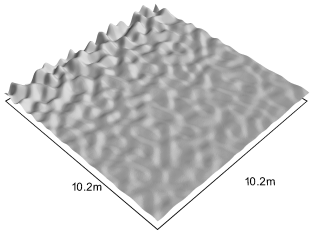
(a) *bumps0*



(b) *bumps1*

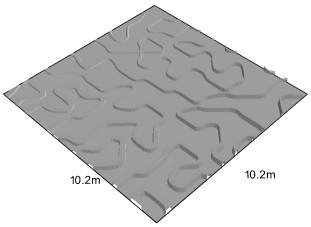


(c) *bumps2*

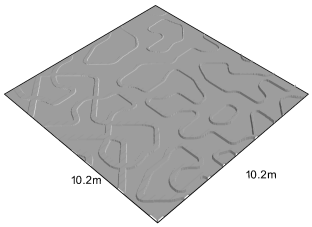


(d) *bumps3*

Figure 4.5. Bumps maps (10×10m).

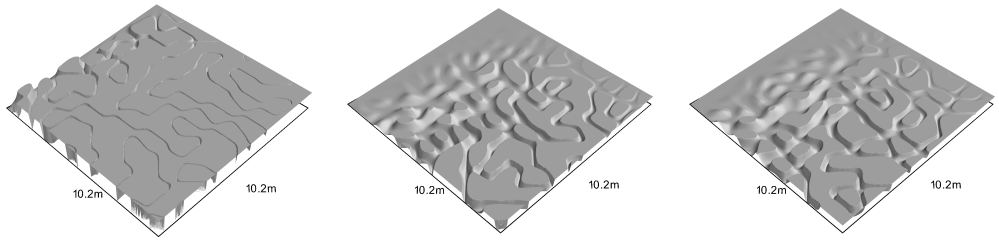


(a) *bars1*



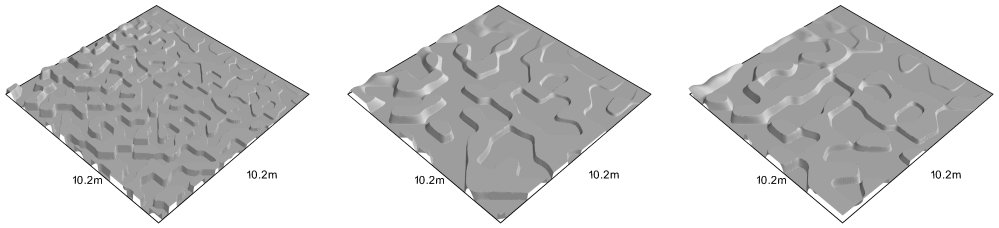
(b) *bars3*

Figure 4.6. Bars maps (10×10m).



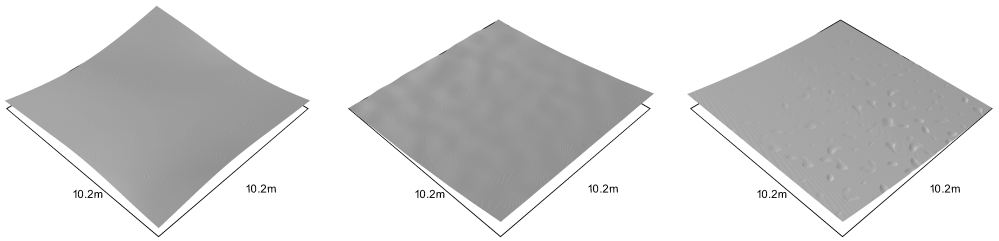
(a) *rails1* (b) *rails2* (c) *rails3*

Figure 4.7. Rails maps (10×10m).



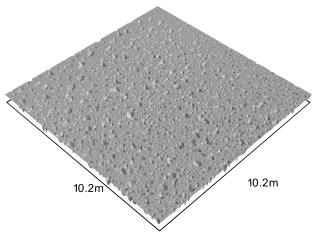
(a) *steps1* (b) *steps2* (c) *steps3*

Figure 4.8. Steps maps (10×10m).



(a) *ramp0* (b) *ramp1* (c) *slope_rocks1*

Figure 4.9. Slopes maps (10×10m).



(a) *holes1*

Figure 4.10. Holes map (10 × 10m).

`throttle` command. To load the map into the simulator, we fist 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.

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* **?**, 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

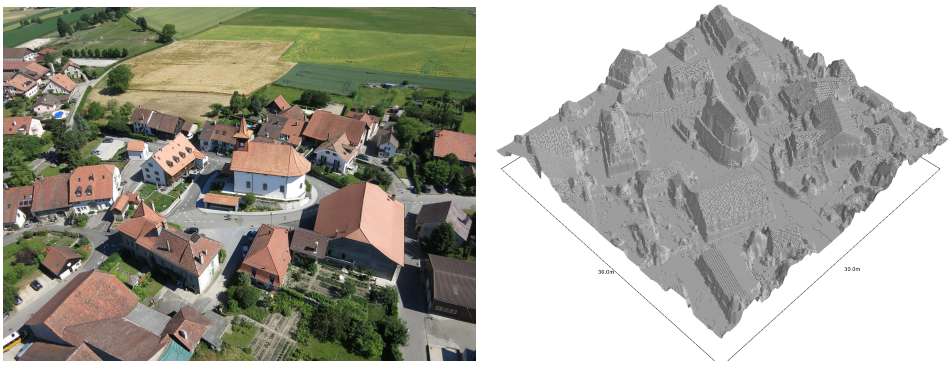
To later evaluate the model's performance, we decide to use real world terrain. We gather several heightmaps produced by ground mapping flying drones from sensefly's dataset.

include sullens

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

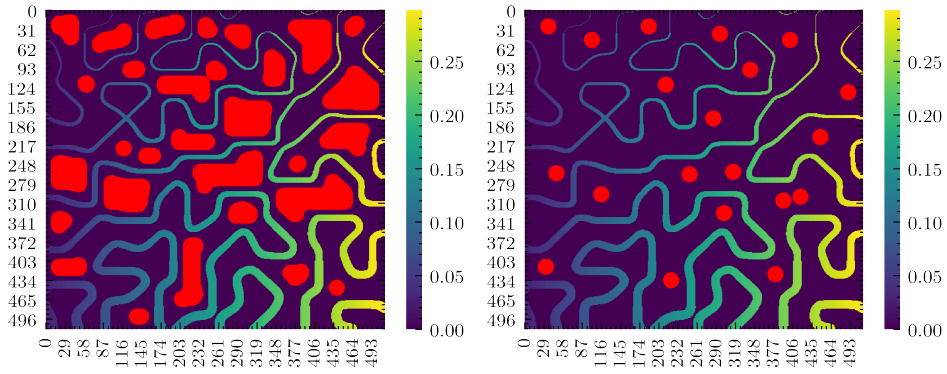
(a) Quarry



(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.

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 spawing point. Intuitively, if a patch is flat then its mean value will be close to zero. Since there may be more flat spawing positions than simulations needed, we have to reduce the size of the candidate points. To maintaing the correct distribution on the map to avoid spawing 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*. The following table shows



(a) Flat regions (b) K-Means with $k = 20$

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

the 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, Δt.

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 tp 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.

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

Table 4.1. Maps configuration used in the simulator.

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

ask omar

The correct value of Δt is crucial. We want a time window small enough to avoid smoothing to much the advancement and making obstacle traversale, 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 flat 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 Δt 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 Δt. 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. 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 milion 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. The following figure shows the mean advancement across all the maps used to train the model in a range of ± 0.70cm, the maximum advancment the used time window, $\Delta t = 2$s. As sanity check, we visualize some of the patches with best and worst advancement. Patches with high advancemnet should not include big obstacle and viceversa.

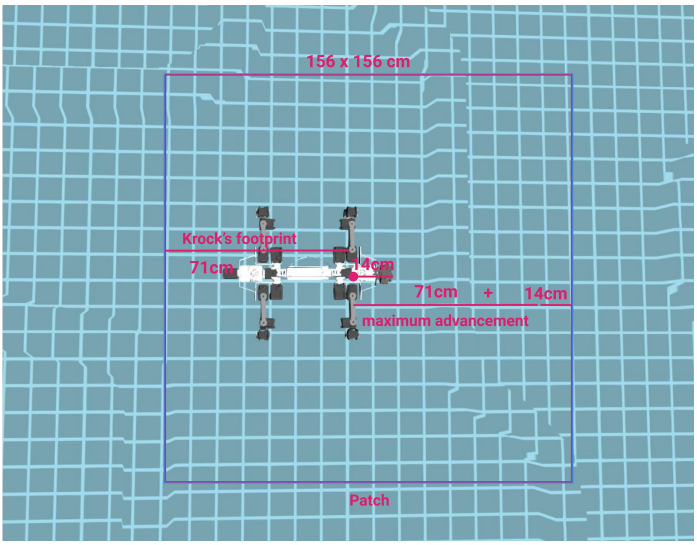
make them gray

Corretly, 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.

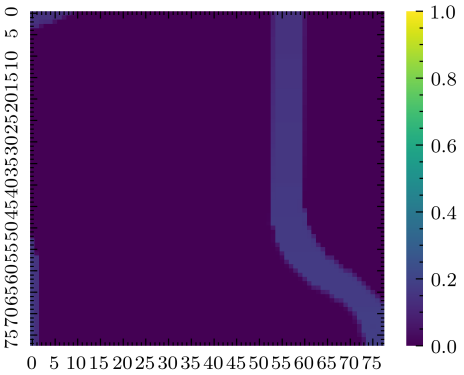
All the handles used to postprocess the data are available as a python package. Also, we create an easy to use API called `pipeline` to define a cascade stream of function that is applied one after the other using a multi-thread queue to speed-up the process.

4.4.3 Label patches

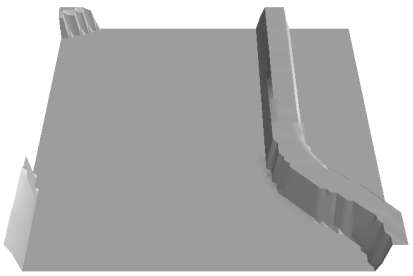
To decide whether a patch is traversable or not traversable we need to decide a *threshold*, tr, such as if a patches has an advancement major than tr it is label as traversable and viceversa. Formally,



(a) Robot in the simulator.



(b) Cropped patch in 2d.



(c) Cropped patch in 3d.

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

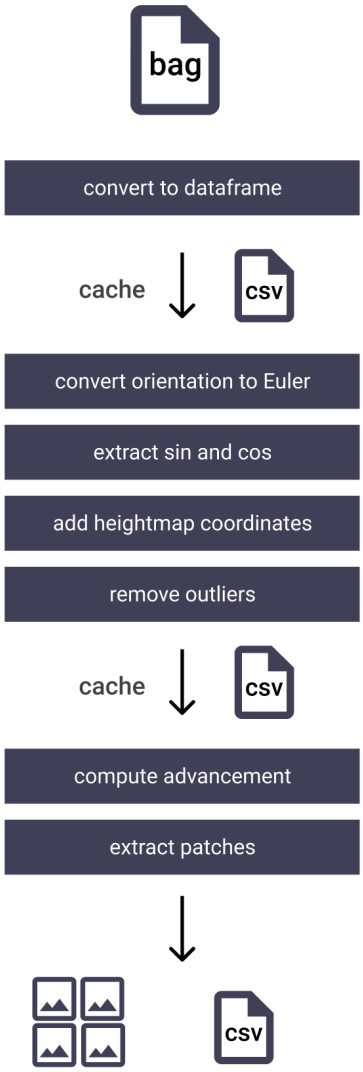


Figure 4.14. Postprocessing Pipeline flow graph, starting from the top.

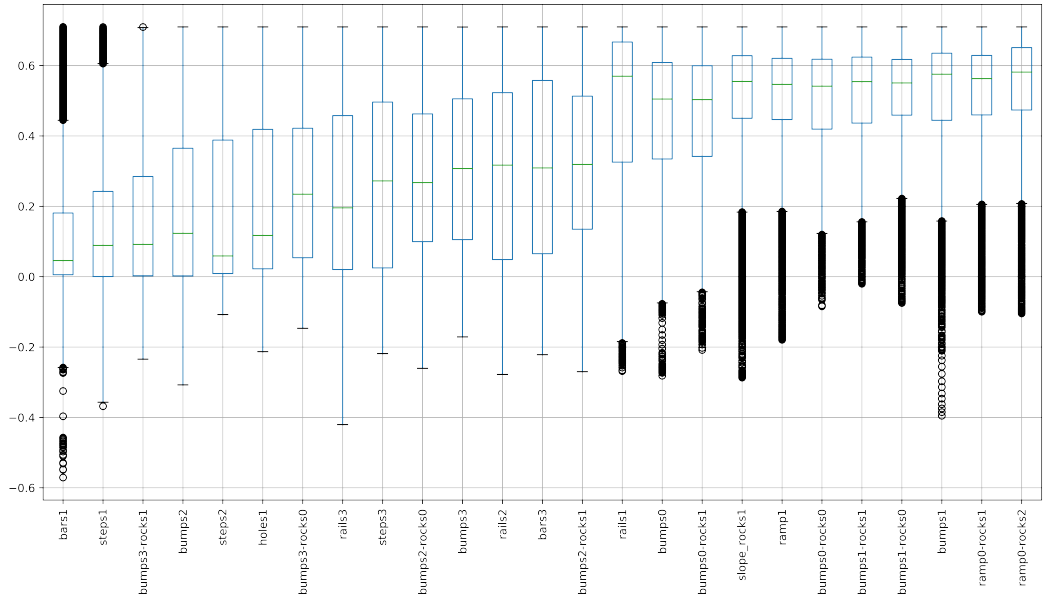
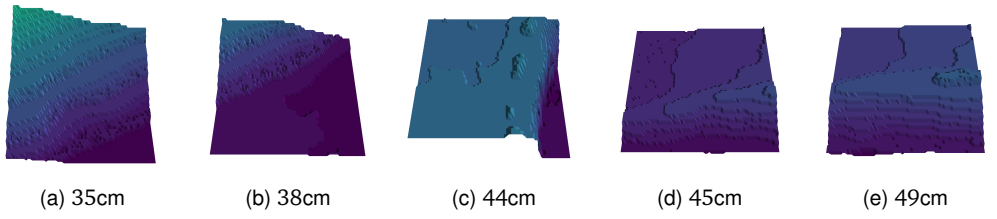


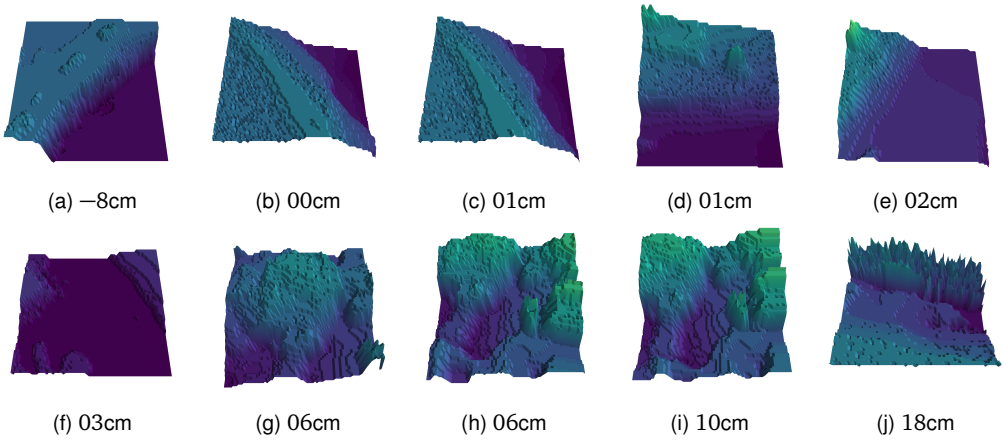
Figure 4.15. Advancement on each map with a $\Delta t = 2$s in ascendent order.



(a) 35cm (b) 38cm (c) 44cm (d) 45cm (e) 49cm

(f) 51cm (g) 65cm (h) 67cm (i) 67cm (j) 72cm

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



(a) −8cm (b) 00cm (c) 01cm (d) 01cm (e) 02cm

(f) 03cm (g) 06cm (h) 06cm (i) 10cm (j) 18cm

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

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 $tr_{\Delta t=n}$

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 spawing 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.

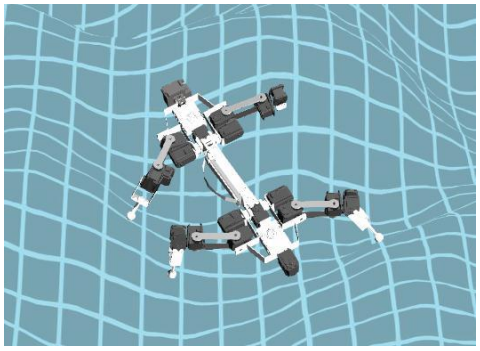
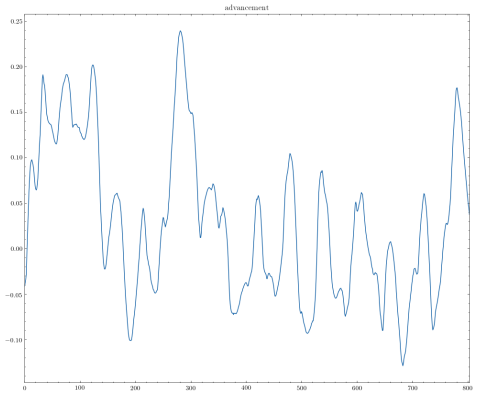


Figure 4.18. Krock tries to overcome an obstacle in the *bumps3* map.

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 spicky advancement where first it is positive and then negative. The following picture shows the advancemnet on this map with different time windows, Δt, the greater the smoother.

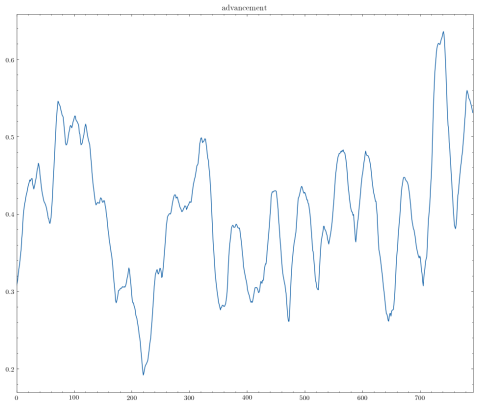
A wheel robot, on the other hand, will not produce such graph since it cannot free itself easily from an obstacle. Image a wheel robot moving forward, once it hits an obstance it stops moving, if we plot its advancement in this situation it will look more like a step than a series of spikes.



(a) $\Delta t = 2$s.

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

Ramps The trehsold 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 climp this steep ramp. The following figure shows the advancement. The mean value is around



(a) $\Delta t = 2$s.

Figure 4.20. Advancement over time with $\Delta t = 2$s on *slope_rocks3*.

0.4cm over a $\Delta t = 2$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 beetween the upper bound of *bumps* and beetween the lower bound of *slope_rocks1*. This will ensure to minimize the false positive. A good value is $tr_{\Delta t=2s} = 20cm$.

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 **?** is a CNN composed by a two 3×3 convolution layer with 5 filters; 2×2 Max-Pooling layer; 3×3 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 **?**, variant. Residual networks are deep convolutional networks consisting of many stacked R̈esidual 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 R̈esidual Units ïs composed by a two $3x3$ *Convolution*, *Batchnorm* **?** and a *Relu* blocks. Formally defined as:

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

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

add resnet image or table

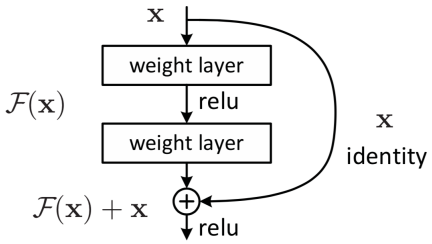


Figure 4.21. *Resnet* block **?**

When the input and output shapes mistmatch, the *identity map* is applyed to the input as a $3x3$ Convolution with a stride of 2 to mimic the polling operator. A single block is composed by a $3x3$ *Convolution*, *Batchnorm* and a *Relu* activation function.

4.5.3 Preactivation

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

4.5.4 Sqeeze and Excitation

Finally, we also used the *Squeeze and Excitation* (SE) module **?**. 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 $\boldsymbol{F}_{tr} : \boldsymbol{X} \mapsto \boldsymbol{U}, \boldsymbol{X} \in mathbbR^{H' \times W' \times C'}$, $\boldsymbol{U} \in \mathbb{R}^{H \times W \times C}$, the SE module first squeeze the information by using average pooling, $\boldsymbol{F}_{sq}$, the it excitates them

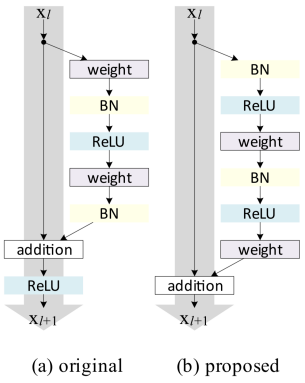


Figure 4.22. *Preactivation* **?**

using learnable weights, $\boldsymbol{F}_{ex}$ and finally, adaptive recalibration is performed, $\boldsymbol{F}_{scale}$. The next figure visualises the SE module.

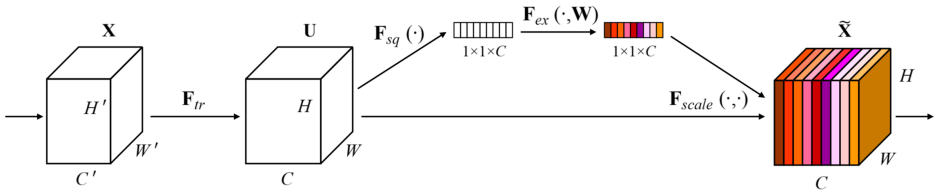


Figure 4.23. *Squeeze and Excitation* **?**

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 224×224 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 7×7 and 3×3 with stride of 2 and 1 respectively. Lastly, we used LeakyReLU **?** with a negative slope of 0.1 instead of ReLU to allow a better gradient flow during backpropagation. LeakyRelu is defined as follows

ref to Resnet Table

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

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

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`.

Input	$(1, 78, 78)$			
Layers	3×3, 16 stride 1		7×7 16 stride 2	
	$2 \times$ 2 max-pool			
	$\begin{bmatrix} 3 \times 3, & 16 \\ 3 \times 3, & 32 \end{bmatrix}$ x 1			
	SE	-	SE	-
	$\begin{bmatrix} 3 \times 3, & 32 \\ 3 \times 3, & 64 \end{bmatrix}$ x 1			
	SE	-	SE	-
	$\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 128 \end{bmatrix}$ x 1			
	SE	-	SE	-
	average pool, 1-d fc, softmax			
Parameters	313,642	302,610	**314,282**	303,250
Size (MB)	5.93	5.71	**2.41**	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.

ask omar help to add margin in the rows

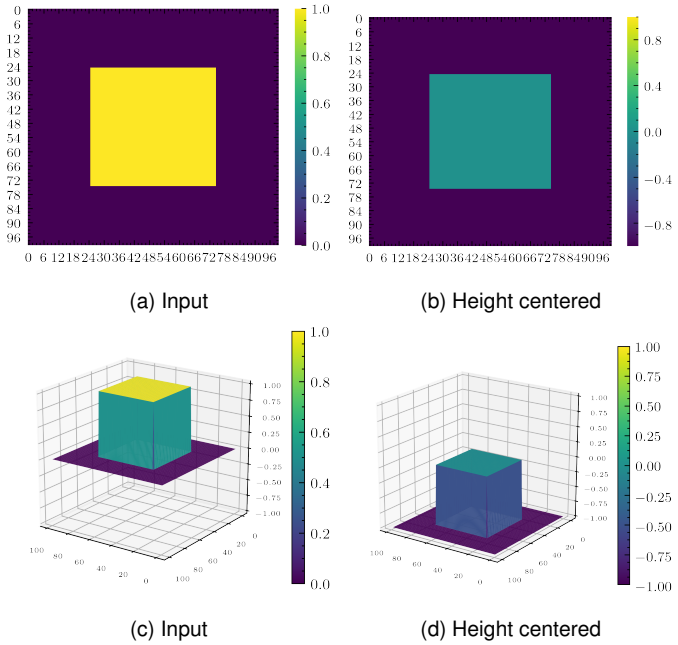
add model picture

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.

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 **??**.



(a) Input (b) Height centered

(c) Input (d) Height centered

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

4.7.1 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

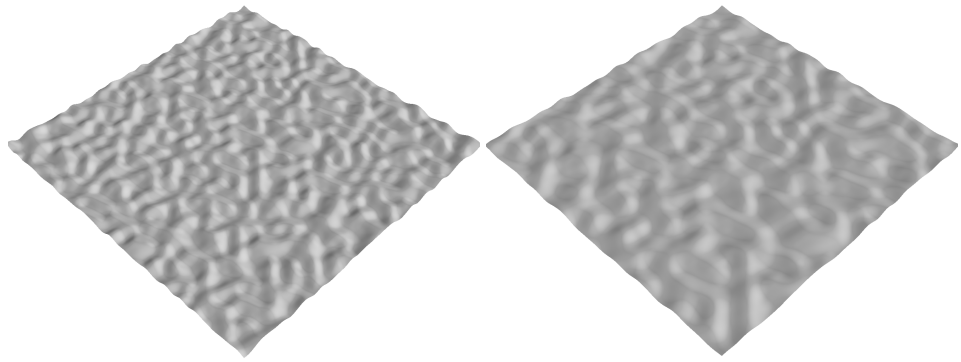
Similar to dropout, it sets to zero random regions of pixels with defined boundaries. Figure

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.

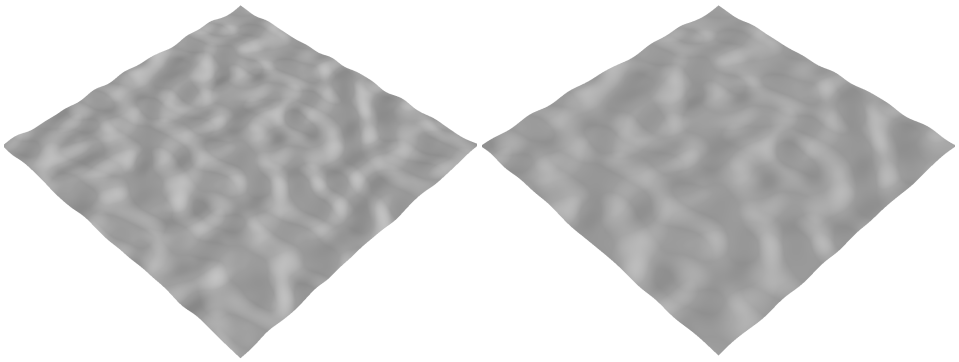
The following images show the tree data augmentation techniques used applied the input image. To ensure we do create untraversable patches from traversable patch we limited the coarse dropout region to only few centiments and we apply a smoother simplex noise on traversable patches. In detail,

un all the traning 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



(a) Features size = 10

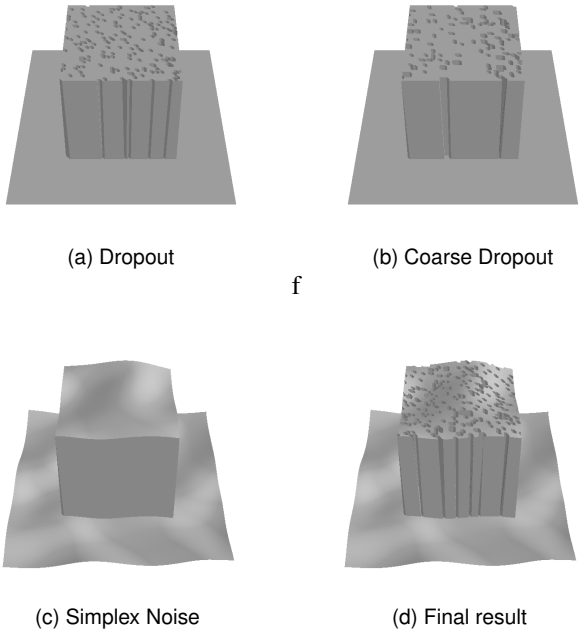
(b) Features size = 20



(c) Features size = 30

(d) Features size = 40

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.



(a) Dropout (b) Coarse Dropout

f

(c) Simplex Noise (d) Final result

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

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) **?** 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 relationa data structure to perform custom manipulation on the rows by removing the outliears and computing the advancement.

Generally, pandas does not scale well and it is mostly used to handle small dataset while relegating b̈ig 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 heightmaps 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 amoung researches and enthusiast. Moreover, its main advantages over other mainstream frameworks such as TensorFlow **?** 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 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 heightmap, details are in section **??**

4.8.9 Mayavi

Mayavi is a scientific data visualization and plotting in python library. We adopt it to rendered 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, Δt, 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. **?**. However, we later utilize Leslie Smith's 1cycle policy **?** 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	
			3×3 stride 1	7×7 stride 2
AUC	Top	0.892	0.888	**0.896**
	Mean	**0.890**	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 7×7 with stride of 2. This model has one third of the parameter of the origal model proposed by Chavez-Garcia et all **?**.

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

	MicroResnet7×7	MicroResnet7×7-SE	Improvement
Top	0.875	**0.896**	+0.021
Mean	0.867	**0.888**	+0.021

Table 5.2. AUC top value and mean value for MicroResnet with a fist convolution of 7×7 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

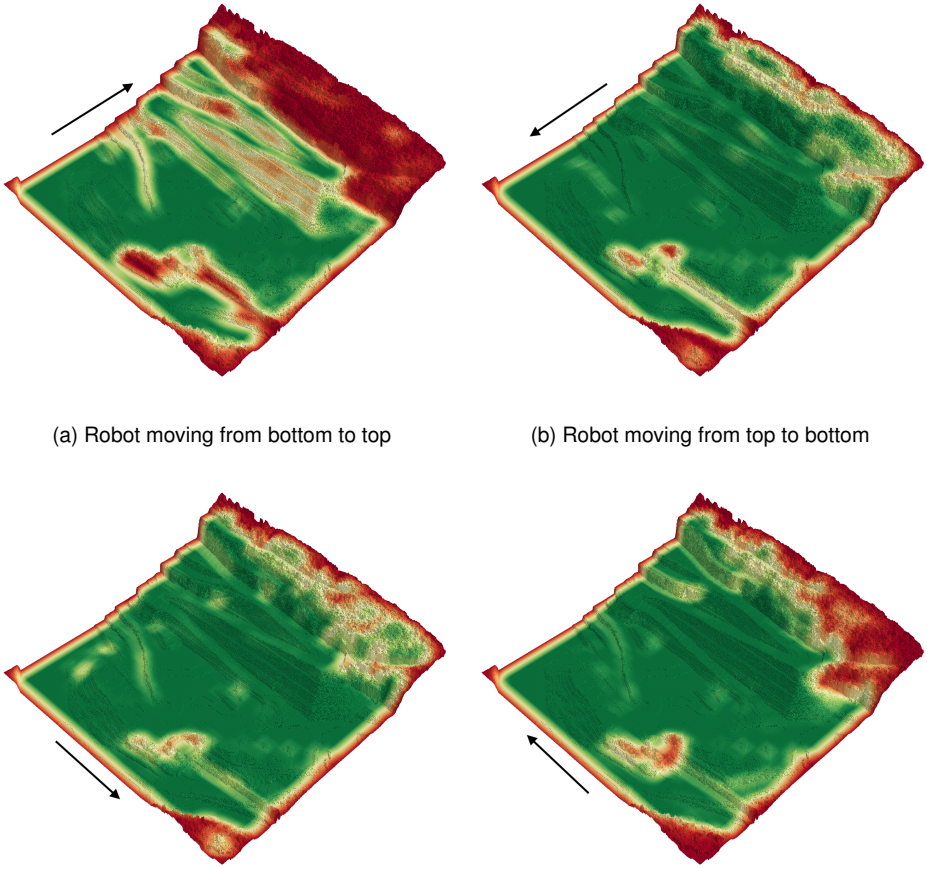
The first map we evaluate is Quarry, this map is 32×32m long and has a maximum height of 10m. We can use some of the terrain interesting features, such as are three bis slopes and the rocky ground on top, to evaluate the model. For instance, w expect the trail on the slopes to be traversable at almost any rotation, expecially 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

Correctly, the lower part of the map, composed by flat regions, was label 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.



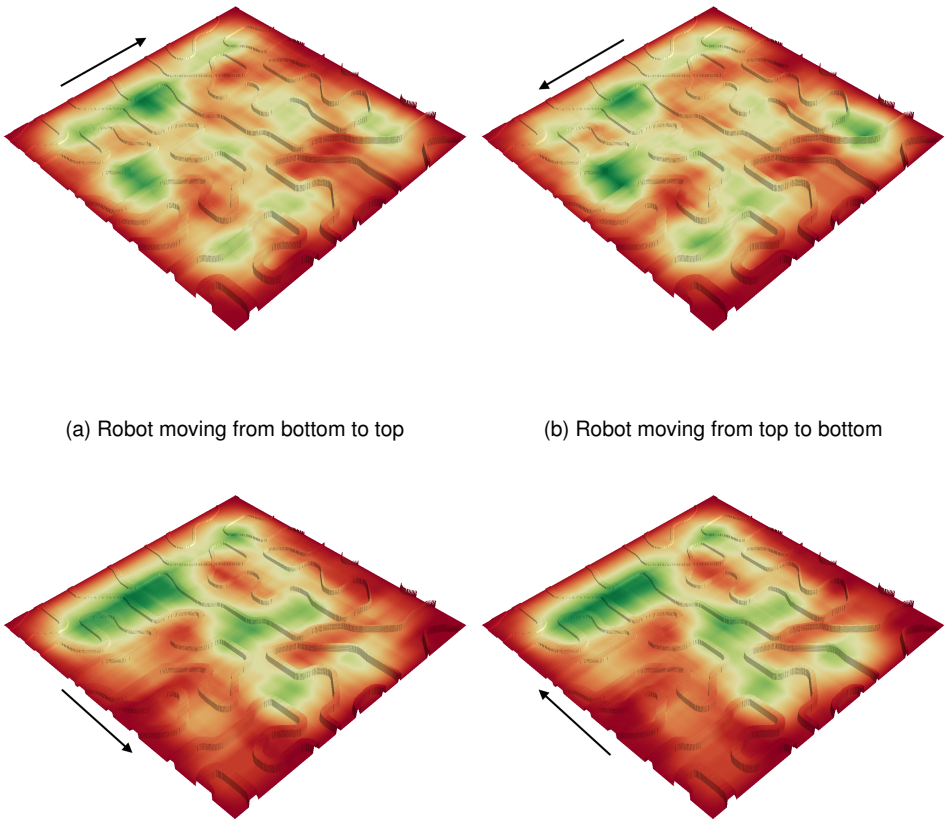
(a) Robot moving from bottom to top

(b) Robot moving from top to bottom

(c) Robot moving from left to right

(d) Robot moving from right to left

Figure 5.1. Traversability probability on the Quarry map, 32×32m, 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.



(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 5.2. Traversability probability on the bars map, 10×10m, 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.

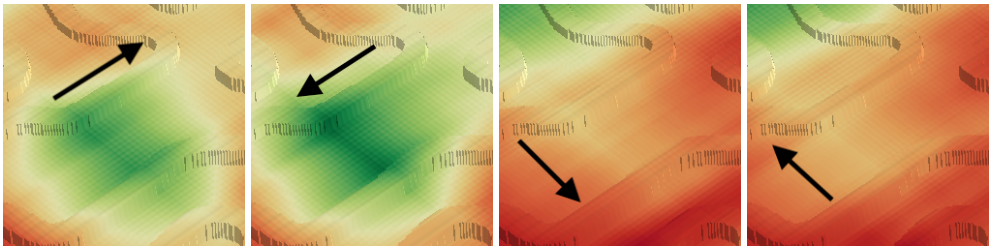


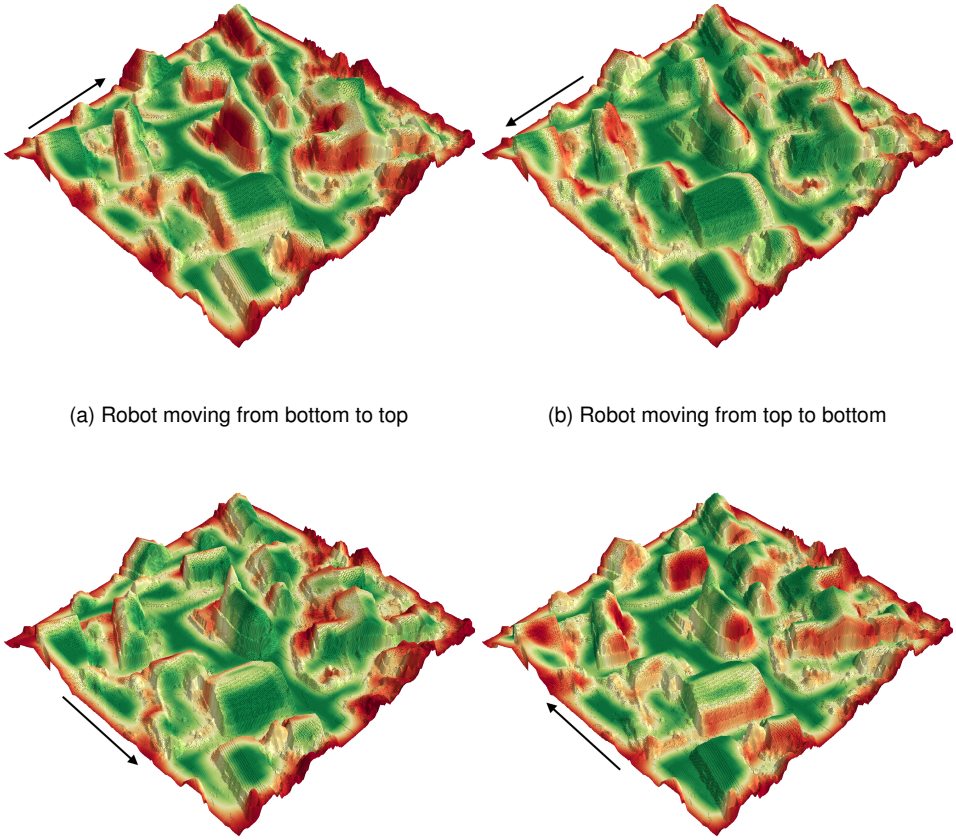
Figure 5.3. Detail of a region in the bars map where there are two walls forming a tunnel. Correctly, when the robot is travel following the trail the region is label as traversable.

5.5.3 Small village

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

All the streest were label as traversble, while the roofs' traversability depend on the orientation. For example, most steep roofs are not traversble 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.

This chapter is under development!



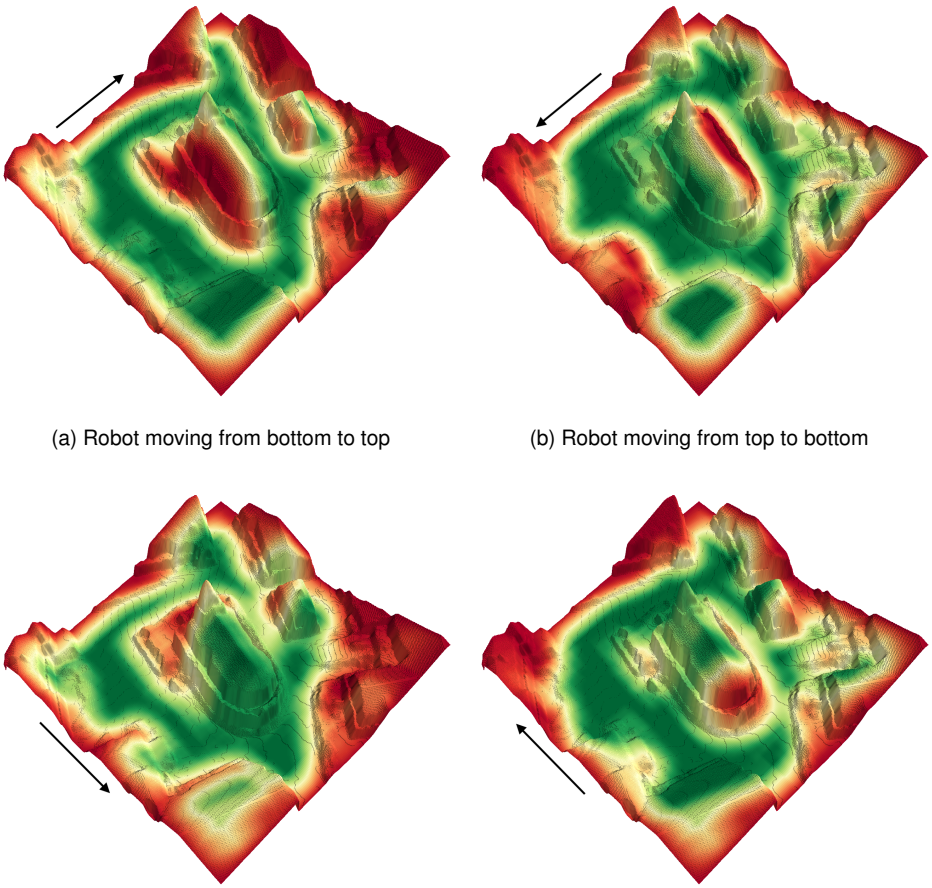
(a) Robot moving from bottom to top

(b) Robot moving from top to bottom

(c) Robot moving from left to right

(d) Robot moving from right to left

Figure 5.4. Traversability probability on the map of a small village for different Krock's rotation. The surface cover 30×30m 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.



(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 vicevera. While for the last two the robot can correctly travel til the end.

Chapter 6

Interpretability

Understanding the model's strength, robustness and, limitations is crucial to gain a better understanding of its outputs. This is even more important when working with controllers that can be deployed in real scenarios. In this section, we evaluated the quality of our traversability estimator with different methodology. First, we showed that the model has correctly learned grounds' features and was able to separable terrains based on them. Second, we utilized the challenging Quarry dataset to visualize the most traversable, the lest traversable and the misclassified patches. Utilizing a special method, we determined that the model always looked at the correct features in the ground even if when it fails. Lastly, we crafted patches with different unique features, such as walls, bumps, etc, to test the robustness of the model by comparing its outputs to the real data gathered from the simulator.

6.0.1 Features separability

In general, convolutional neural networks learn to encode images by applying filters of increasing size at each layer. Usually, the first layers learn basic features, such as edges, while the final one encodes complex shapes. The final convolutional layer's outputs are usually referred to as features space, consequently, a feature vector is just the output of the last layer for a given image. Those last features are combined and mapped to the correct classes by one or more fully connected. The following image help to visualize the different features learned at each layer. It was generated by plotting the learned features for different categories at different layers by Lee et al. **?**. So, a correctly

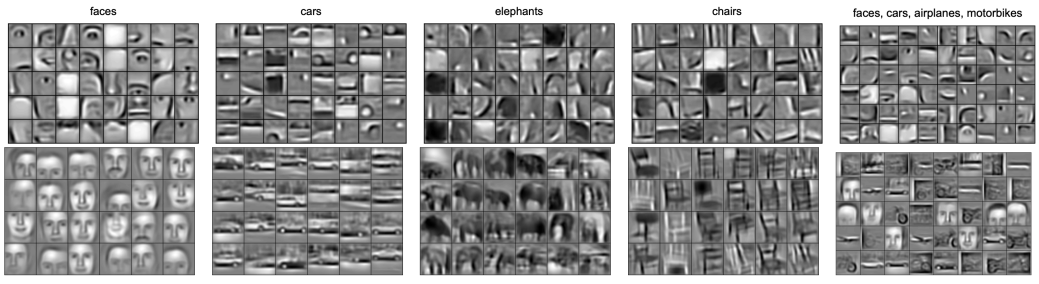


Figure 6.1. Figure from Lee et al. **?** paper where they showed for different classes the low-level features (up) and the high-level features (down) learned by a convolution neural network.

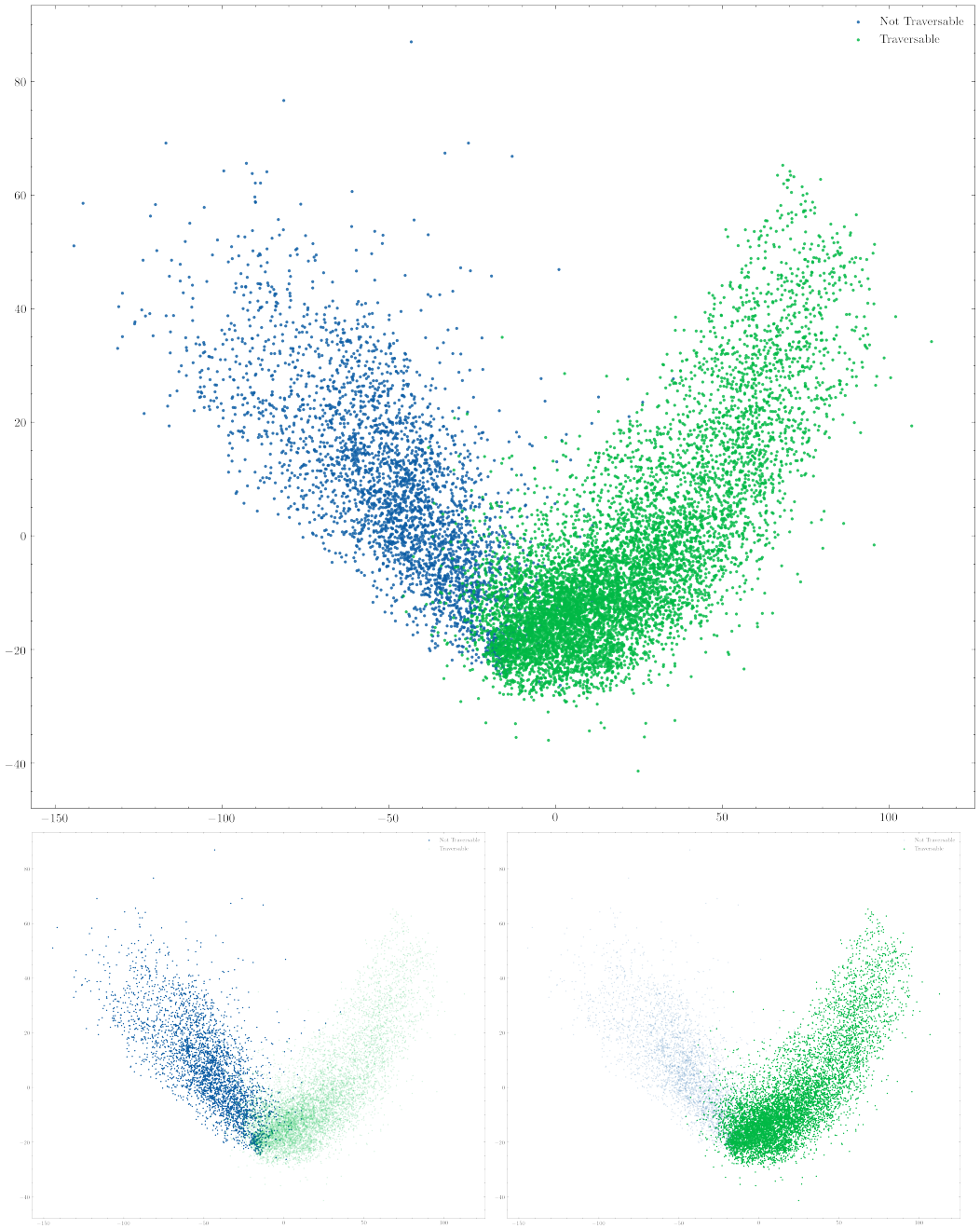
trained network should be able to separate those features based on predicted classes. Intuitively, given two classes $\mathscr{A}$ and $\mathscr{B}$, for example, *cat* and *dog*, the high-level features for each class should not be the same, otherwise, the model may misclassify the input due to the overlap of different classes' features. One technique to discover the degree of separability of our network is to directly visualize the inputs features vectors for each class. Unfortunately, like most models, our network, MicroResNet, has a high dimensional feature space. Each patch is mapped to a $[128, 3, 3]$ vector. We cannot directly visualize a 128 dimension space, we reduced each feature vector dimension to a two-dimension by applying Principle Analysis Component (PCA) **?** to visualize it. We investigate the features space of the model booth in the train and test set.

6.0.2 Train set

The following figures shows the features of 11K images sampled from the train set labeled with their classes, *traversable* and *not traversable*. We can clearly recognize two main clusters based on the labels' color, one on the left and one of the right. Those points are easily separable, even by human eyes, meaning that the model was able to learn meaning features from the dataset. To be sure the center of each class' point cloud is not overlapping we plotted the density of each cluster. Clearly, there is some distance between the centers. Furthermore, we can directly plot the patch corresponding to each feature vector to identify clusters of patches based on their position. Intuitively, if similar inputs are close to each other in the features space then the model also learned to effectively encode terrains features. 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. Definitely, patches with similar features are close to each other yielding a quality features encoding. On the left-top side, we can distinguish highly untraversable patches with walls/bumps in front of the robot. Going down, we encounter patches with smaller obstacles. On the plateau, there are traversable patches with small obstacles such as light bumps. Importantly, those patches are the closest ones to the not traversable ones, so they were the hardest to separate, thus, to classify. Going up on the right side, we see some grounds with small steps. Finally, on the top, we find all the downhill patches, the simplest ones to traverse.

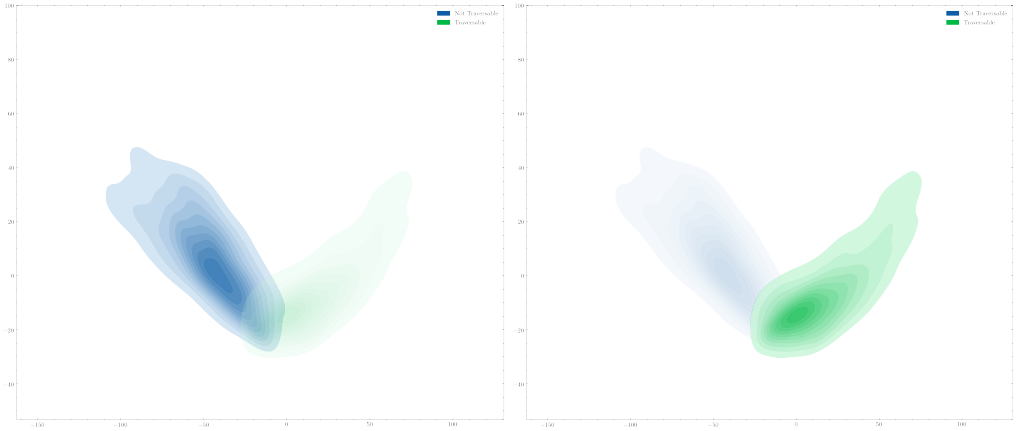
6.0.3 Test set

We can apply the same procedure on the test set. Since it is a real world quarry, this dataset is harder than the train set and present challenging situations for the robot. The following image shows the features space after reducing its dimension to two using PCA. Interesting, the traversable patches are very near to each other, while the others span a very big surface. This suggests that there are many not traversable terrains with different features. The traversable points are clustered near the center, this implies that most of them share similar features. We plotted the density for each class to better understand where the most points are mapped. The two centers are really close to each other, making those samples harder to separate and some not traversable points are mixed up with the traversable ones. This explains the elevated number of false negative that lower down the AUC score on this dataset. We can also visualize the patches by plotting them using their features coordinates 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



(a) Not Traversable (b) Traversable

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



(a) Not Traversable (b) Traversable

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

move to the density center, green double 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) **?** is a technique to produce v̈isual explanationsf̈or convolutional neural networks. It highlights the regions of the input image that contribute the most to the predictions. In detail, the output with respect to a target class is backpropagated while storing the gradient and the output in the last convolution. Then, a 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. After, 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 Quarry dataset

After showing the model's capability of correctly separate classes' features we utilized Grad-CAM to visualize some patches inside the quarry dataset. The aim of this section is to show how the model looks at meaningful features in each input to make the prediction even when it fails. For instance, imagine we feed to the model a not traversable patch with an obstacle and the network label is as traversable. Clearly, the output is wrong but the model's error may be caused by two different situations. The first one, the model could just have ignored the obstacle and looked away, meaning that it was not even able to understand there was an obstacle in the first place. Second, the network could have correctly look at the obstacle but thought that obstacle is traversable, showing

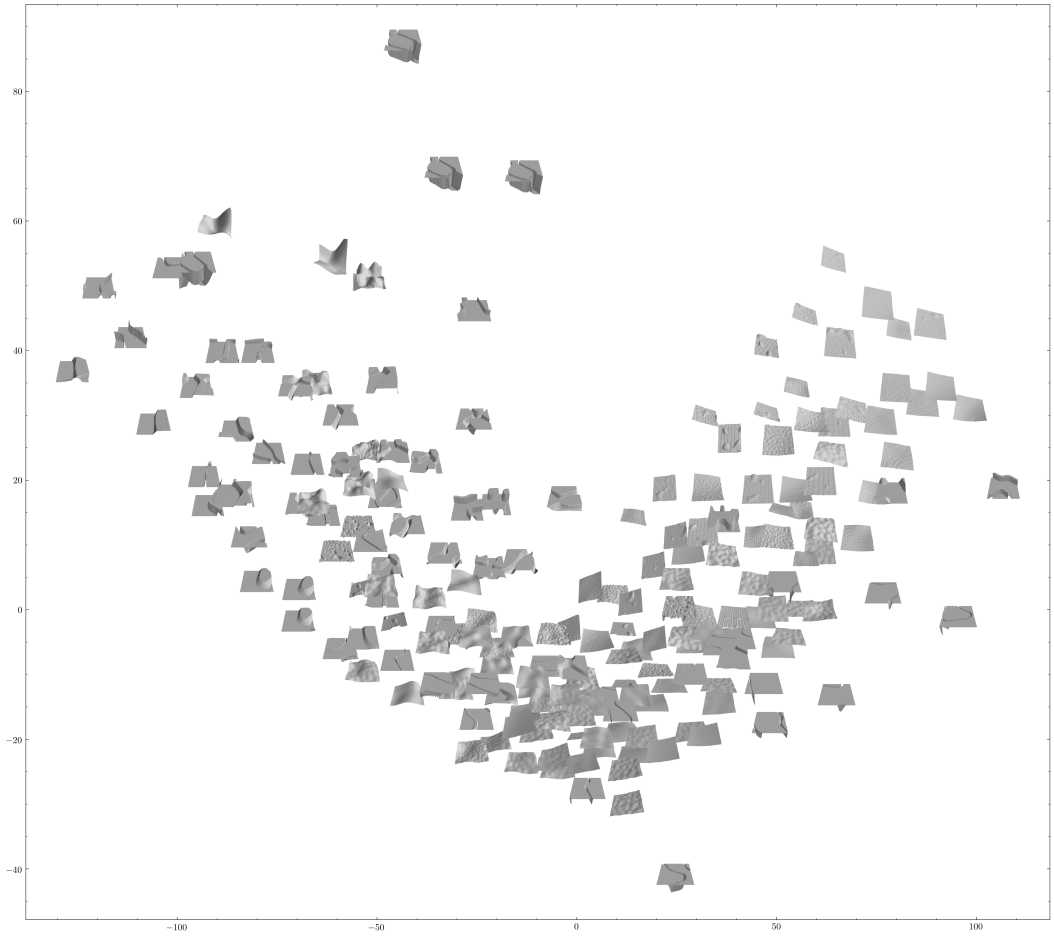


Figure 6.4. Patches plotted using the coordinate of the features vector obtained from the last convolutional layer's output and then reduced using PCA to a two-dimensional vector. Similar grounds are close to each other.

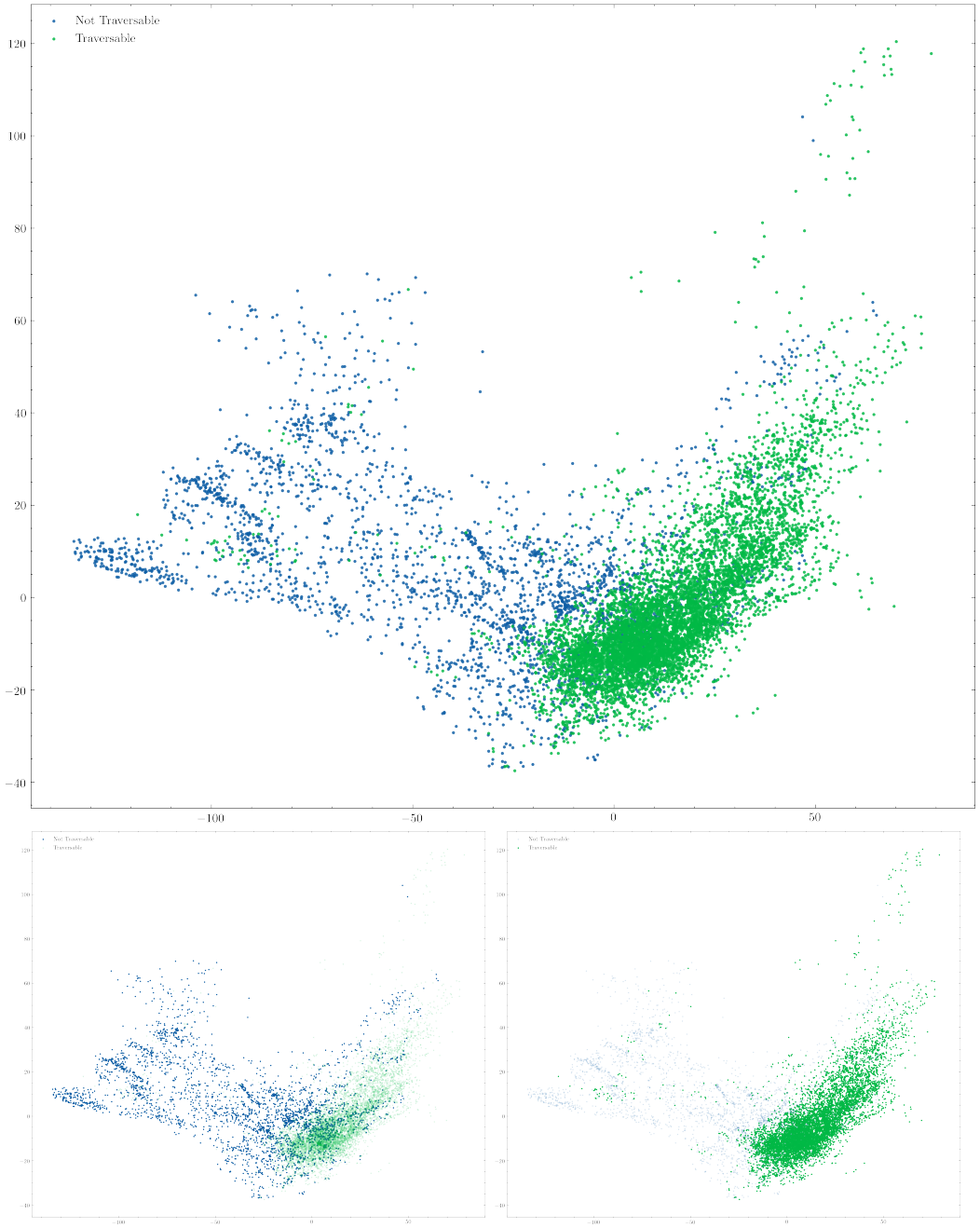
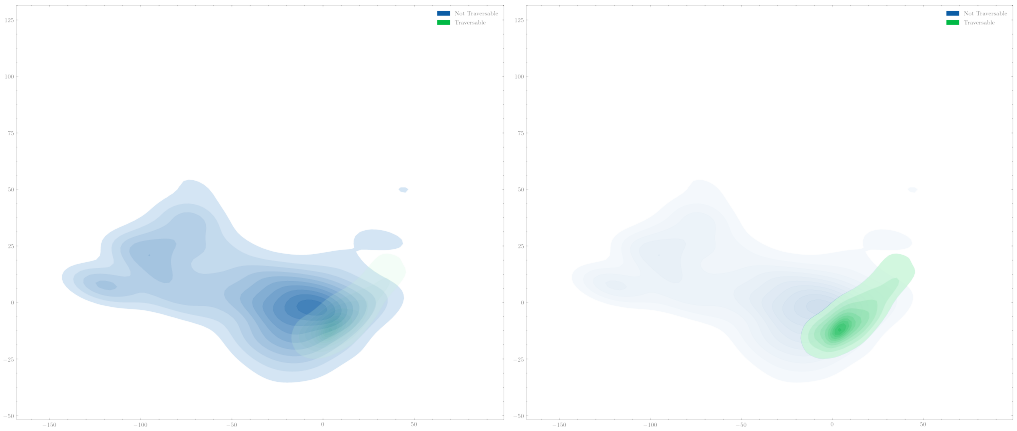


Figure 6.5. TODO



(a) Not Traversable (b) Traversable

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 correct learning.

the correct ability to find and use important features in the map. In the following sections, we showed that, even when the predictions are wrong, our model always look at the most important features of each input to determine its traversability.

We divided the dataset patches into four classes based on the model's performance: worst, best, false positive and false negative. Then, we took twenty inputs from those sets and applied Grad-CAM to generate a texture that we placed on 3D render to better visualize which region of the inputs caused the prediction.

6.2.1 Best

The best patches are the inputs labeled and predicted traversable. We plotted twenty different images sampled uniformly according to the robot advancement to include as many interesting situations as possible. We recognized two main clusters of images: flat and slopes. The models is mostly interested in the left part of the patches with uneven ground on the left region, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**. This is due to the fact that an obstacle in that reason will block the rear legs and prevent the robot to advance.

Moreover, in other patches **??**, **??**, **??**, **??**, the model's attention is ahead of the robot. In those situations, the robot is able to properly move at the beginning so the network must evaluate the possibility of obstacles ahead. There are two obvious cases, **??** and **??**. The first one is a totally flat surface, so the model will look as far as possible to check if there are obstacles. Similarly, in the second, a surface with a bump in the end, the network verifies that spot. So, rightly, the network analysis the first region of the patch that may contain an untraversable obstacle.

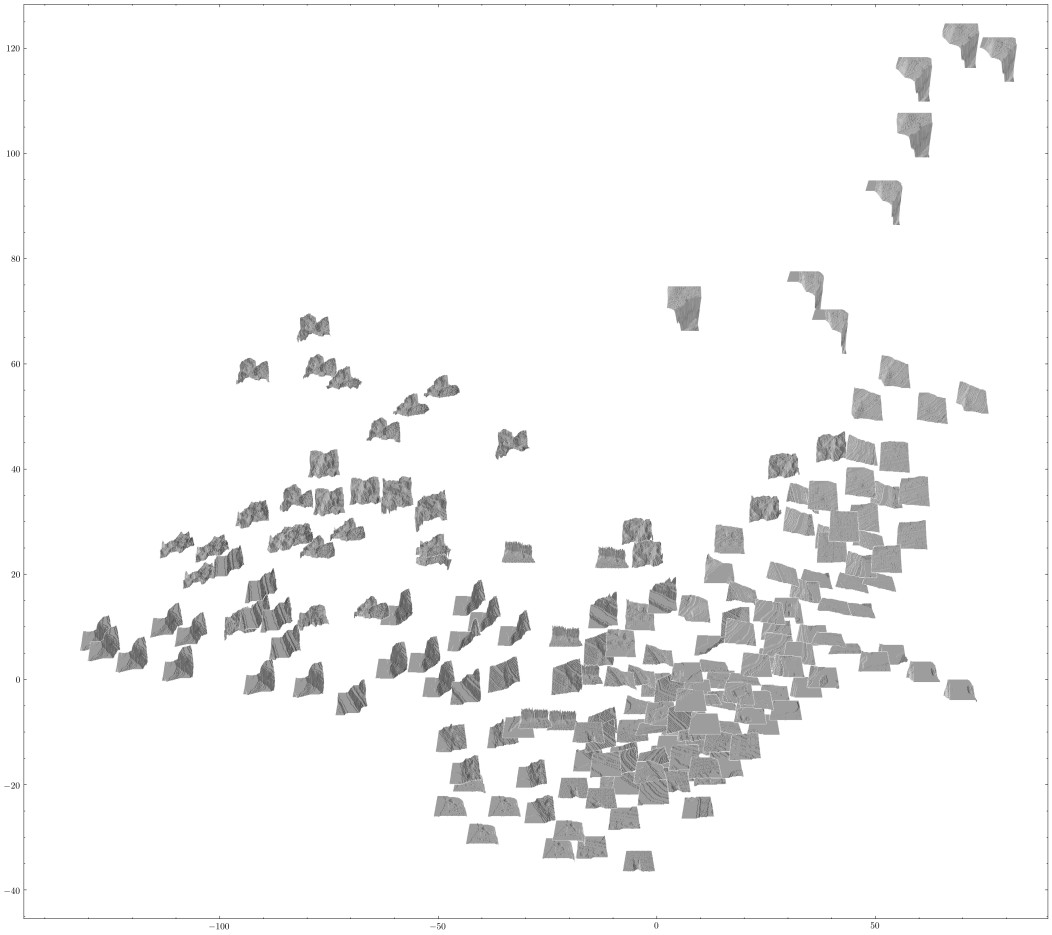


Figure 6.7. Patches that correspond to coordinates in the features space of the last convolutional layers on the test dataset. Similar grounds are close to each other.

??

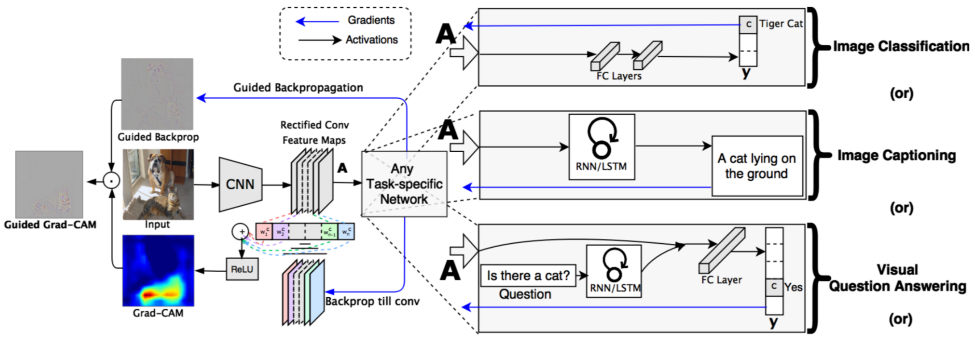
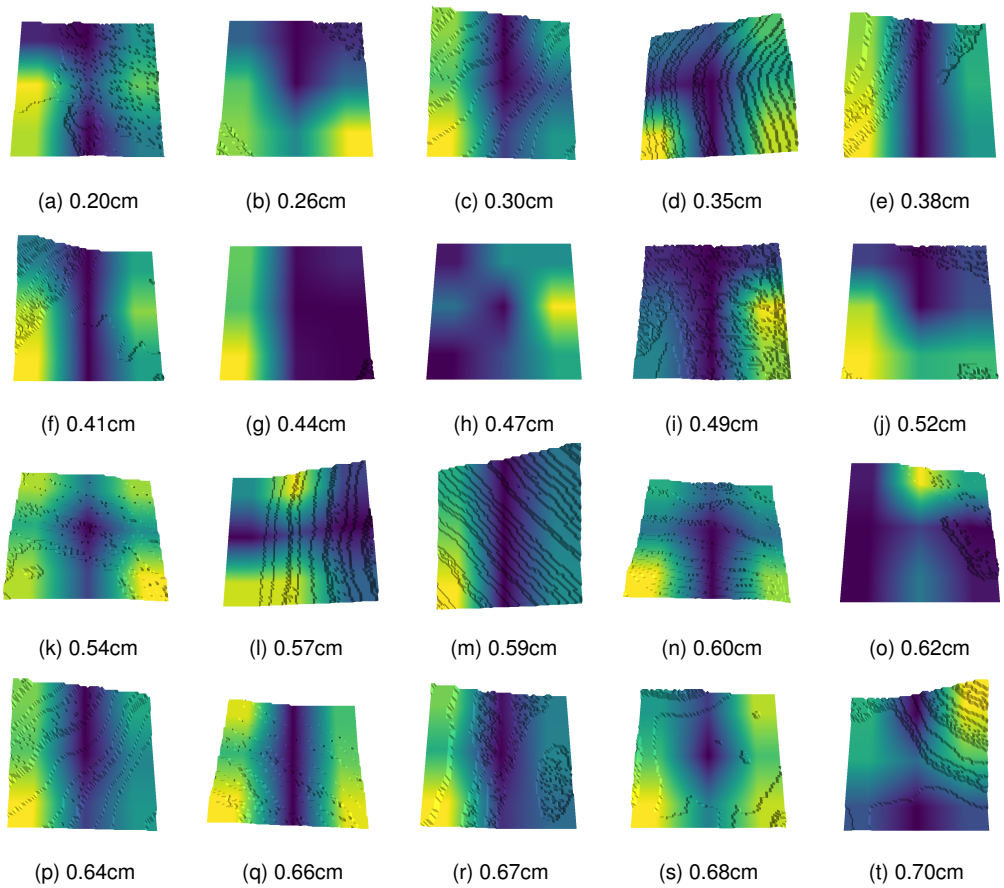


Figure 6.8. Grad-CAM procedure on an input image. Image from the original paper ?.



(a) 0.20cm (b) 0.26cm (c) 0.30cm (d) 0.35cm (e) 0.38cm

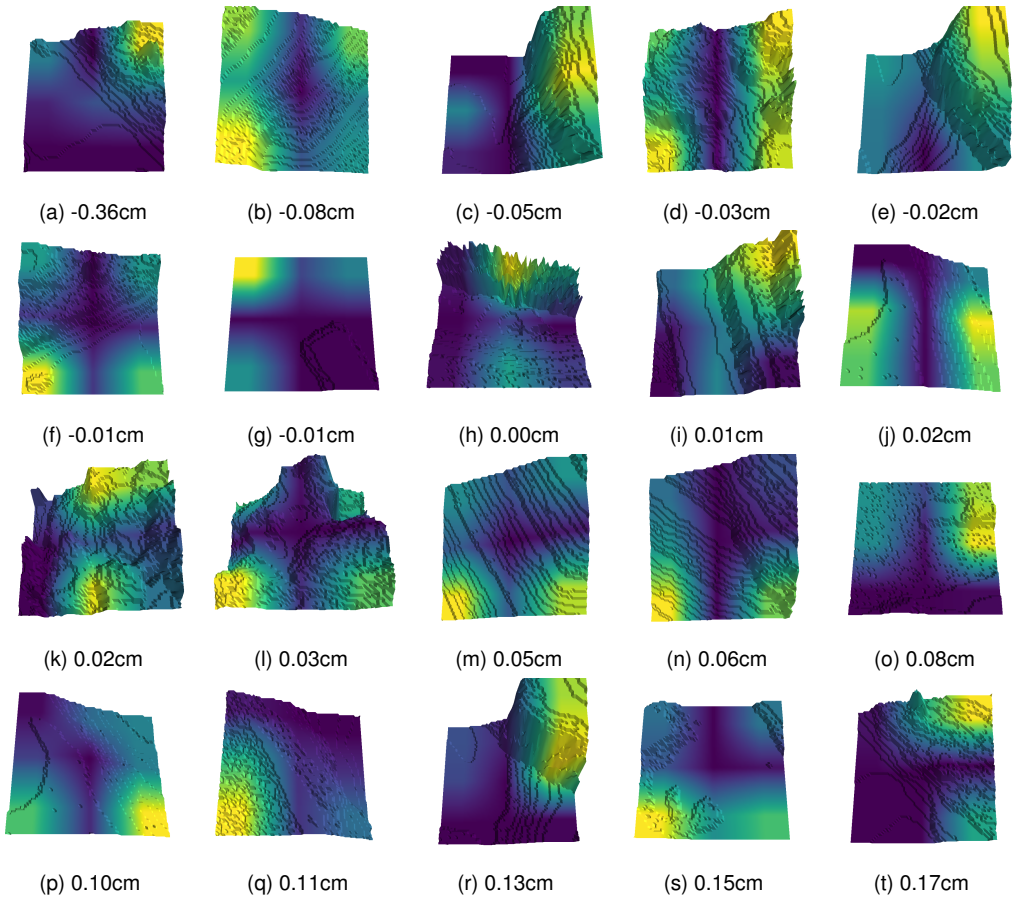
(f) 0.41cm (g) 0.44cm (h) 0.47cm (i) 0.49cm (j) 0.52cm

(k) 0.54cm (l) 0.57cm (m) 0.59cm (n) 0.60cm (o) 0.62cm

(p) 0.64cm (q) 0.66cm (r) 0.67cm (s) 0.68cm (t) 0.70cm

6.2.2 Worst

The worst patches are the inputs labeled and predicted not traversable. In this case, the region highlighted by the Grad-CAM are the features in the ground that contribute the most to make the patch not traversable. For instance, in some patches, **??**, **??**, **??**, the most not traversable features is directly under the robot legs, similar as before. While, for the others samples, **??**, **??**, **??**, **??**, ..., the attention of the network is on the obstacles in front of the robot. This is evident in **??**, **??**, **??**, where the big wall on the right is almost totally highlighted. So, without any doubts, also on this samples, the model is able to perfectly understand where to look to make the correct prediction



(a) -0.36cm (b) -0.08cm (c) -0.05cm (d) -0.03cm (e) -0.02cm

(f) -0.01cm (g) -0.01cm (h) 0.00cm (i) 0.01cm (j) 0.02cm

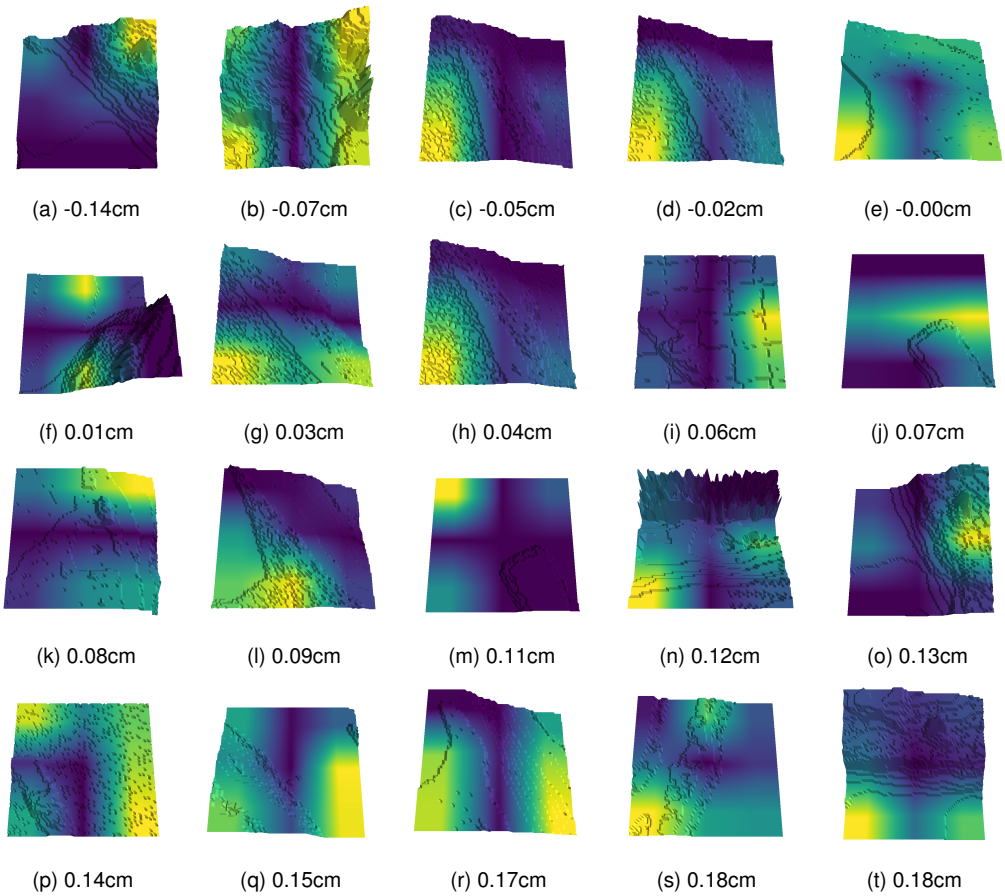
(k) 0.02cm (l) 0.03cm (m) 0.05cm (n) 0.06cm (o) 0.08cm

(p) 0.10cm (q) 0.11cm (r) 0.13cm (s) 0.15cm (t) 0.17cm

6.2.3 False Negative

Those are the inputs that were labeled as negative but predicted as positive. There are lots of interesting cases, we can cluster those patches in three groups: obstacles ahead, slopes and wall parallele to the robot. In figures **??**, **??**, **??**, the models is looking at the obstacle in front of the robot. Slopes, figures **??**, **??**, **??**, **??**, **??**, **??**, are not completely smooth, making harder to classify them. The last cluster of inputs are the ones with a wall parallel to the robot, figures **??** and **??**. In those cases the model looks at the initial position of the robot, left, and the final part of the patch. Intuitivelly, is trying to understand if the robot fits on the trail.

In general, most of the region of interested on those patches are located on the left, close to the position of the robot's leg. Correctly, even if the prediction is wrong, the model looks at the first region of the surface, located near the legs, that can cause non traversability. This shown a correct behaviour even with wrongly prediciton meaning that the network is always looking in the correct spot.



(a) -0.14cm (b) -0.07cm (c) -0.05cm (d) -0.02cm (e) -0.00cm

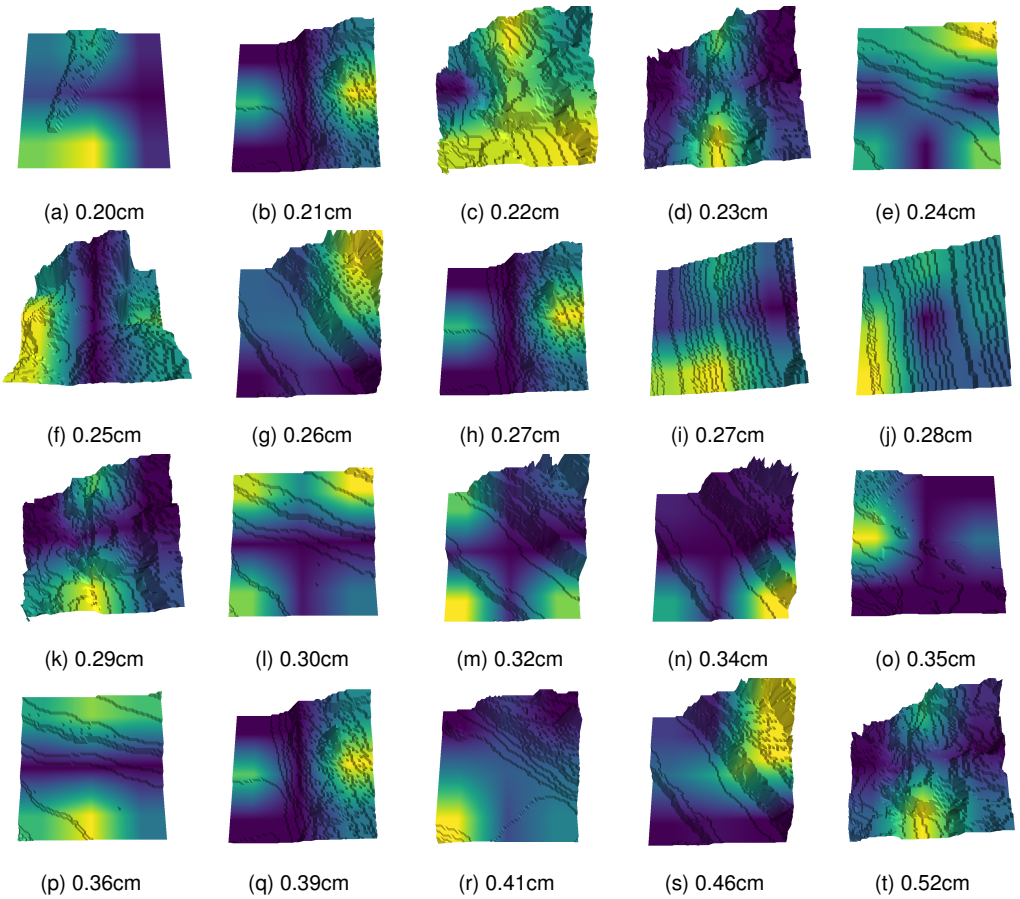
(f) 0.01cm (g) 0.03cm (h) 0.04cm (i) 0.06cm (j) 0.07cm

(k) 0.08cm (l) 0.09cm (m) 0.11cm (n) 0.12cm (o) 0.13cm

(p) 0.14cm (q) 0.15cm (r) 0.17cm (s) 0.18cm (t) 0.18cm

6.2.4 False Positive

Those are the most interesting ones, they are the samples labeled as traversable but predicted as not. They include different types of patches, some with obstacles ahead, slopes and flat regions. In each case, the model looks at meaningful features that can effect traversability. In the slopes, **??** and **??**, the first part of the surface is highlighted. Similar to before, the model is looking if the rear legs will be able to move and thought that those small steps will cause the robot to not advance enough. The same consideration can be done for figure **??**. On the other hand, some inputs. definitely confused the model. For instance, in figures **??** and **??** the model is more interested in the big obstacle on the right part. This lead to classify those patches as not traversable, even if the obstacle is more distant than the threshold, in this case, the network was totally confused. Interesting, figure **??** shows an almost identical situation in which the network correctly looks at the initial part of the ground. We can deduce that even if in almost all cases the network's prediction is caused by the correct part of the inputs, it can be confused by a big obstacle near the end.



(a) 0.20cm (b) 0.21cm (c) 0.22cm (d) 0.23cm (e) 0.24cm

(f) 0.25cm (g) 0.26cm (h) 0.27cm (i) 0.27cm (j) 0.28cm

(k) 0.29cm (l) 0.30cm (m) 0.32cm (n) 0.34cm (o) 0.35cm

(p) 0.36cm (q) 0.39cm (r) 0.41cm (s) 0.46cm (t) 0.52cm

Chapter 7

Conclusions