Abstract

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

We trained a deep convolutional neural network on data gathered entirely through simulation to predict traversability for a legged crocodile-like robot called Krock. The training data was generated by letting the robot walking for a certain amount of time on thirty synthetic maps with different grounds features such as slopes, holes, bumps, and walls in a simulator environment while storing his pose, position, and orientation. We later used that information to precisely crop a patch from each simulation trajectory's point. This small portion of ground includes the whole robot's footprint and the ground it would reach using its maximum speed on flat ground. Each patch is label as traversable if Krock's advancement in that point, computed using a time window of two seconds, is greater than a threshold of twenty centimeters, not traversable otherwise.

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. Also, we qualitative evaluate the network by showing the predicted traversability probability on those grounds.

Later, we utilize model interpretability techniques to understand which patches confuse the most the network. We visualized the ground regions that the network fails to classify and visualize 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.

0.1 Introduction

Effective identification of traversable terrain is essential to operate mobile robots in every type of environment. Today, there two main different approaches used in robotics to properly navigate a robot: online and offline. The first one uses local sensors to map the surroundings "on the go" while the second equip the mobile robot with an already labeled map of the terrain.

In most indoor scenarios, specific hardware such as infrared or lidar sensors is used to perform online mapping while the robot is exploring, this is the case of the most recent vacuum cleaner able to map all the rooms in an apartment. With the recent breakthroughs

of deep learning in computer vision, more and more cameras have been used in robotics. For example, self-driving cars utilize different cameras around the vehicle to avoid obstacle using object detection. Indoor scenarios share similar features across different places shifting the problem from which ground can be traversable to which obstacle must be avoided. For instance, the floor is always flat in almost all rooms due to is artificial design. Moreover, usually, traversability must be estimated on the fly due to the high number of possible obstacles and to the layout of the objects in each room may not be persistent in time.

On the other hand, outdoor scenarios may have less artificial obstacle but their homogeneous ground makes challenging to determine where the robot can properly travel. Moreover, a given portion of the ground may not be traversable by all direction due to the not uneven terrain. Fortunately, a height map of the ground can be obtained easily by using third-party services or special flying drones. Those maps are extremely valuable in robotics applications since they provide an efficient way to examine the features of terrains, such as bumps, holes, and walls.

These scenarios have different difficulties. In indoor environments, it is easier to move the robot on the ground since it is designed for humans, but harder to perform obstacle avoidance. While in outdoors scenario it is maybe more challenged to the first estimate where the robot can go due to the huge variety of ground features that may influence traversability. To learn where to move, an artificial controller must be trained to predict the robot interaction with the environment. Such a process requires to collect some data to train the controller. However, this may not be a straight forward process. In indoors terrain, most of the times, data is collected by driving the robot directly in the environment by a human or an artificial controller. While in the outdoors scenario data is assembled using the simulation for convenience.

Our approach aims to estimate traversability of a legged robot krocodile-like robot called Krock. We generate different uneven grounds in form of height map and let the robot walk for a certain amount of time on each one of them while recording its interaction, position and orientation. After, for each stored robot position we crop the corresponding ground portion in which the robot was during the simulator, those patches composed the training dataset. We select a minimum space in a fixed amount of time that the robot must travel to successfully traverse a ground region and use it to label the dataset. Then, we fit a deep convolutional neural network to predict the traversability probability. Later, we evaluate it using different metrics using real-world terrains.

The report is organized as follow, the next chapter introduces the related work, Chapter 2 describes our approach, Chapter 3 talks in deep about the implementation details, Chapter 4 shows the results and Chapter 5 discuss conclusion and future work.

0.2 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 [21]. Visual perception is known to be 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 animals to extract geometric information of the surrounding and plan a safe path through it.

Different methodologies have been proposed to collect the data and then learn to cor-

rectly navigate the environment. Most of the methodologies rely on supervised learning, where first the data is gathered and then a machine learning algorithm is trained sample to correctly predict the traversability of those samples. Among the huge numbers of methods proposed, there are two categories based on the input 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 [19], outdoor robot navigation can benefit from the availability of overhead imagery. For this reason, elevation data has also been used to extract geometric information. Chavez-Garcia et al. [6], proposed a framework to estimate traversability using only elevation data in the form of height maps.

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 convolutional neural network to segment the terrain in different traversable classes.

Whereas appearance methods, to a greater extent related to camera images processing and cognitive analyses, have the objective of recognising colours and patterns not related to the common appearance of terrains, such as grass, rocks or vegetation. Those images can be used to directly estimate the traversability cost.

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 [21] or Gaussian models [19]. Those techniques rely on texture descriptors, for example, Local Binary Pattern [20], 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 also be used together to train a traversability classifier. 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] proposed a locomotion planning that is learned in a simulation environment.

On other major distiction we can made 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. [22] uses 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 automatise the features extraction process has been proposed recently. Lorenz et al. [15] use local sensing to train a deep convolutional neural network to predict 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.

0.3 Implementation

In this section we reveal the details about the implementation tools and choices. We will start from the employed software, then we will describe each step in the dataset generation and finishing by illustrate the utilized estimators.

0.3.1 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.

ROS Melodic

The Robot Operating System (ROS) [**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.

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.

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.

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 dataẗo 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.

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.

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 [2] are a cleaner API structure, better debugging, code shareability and an enormous number of high-quality third-party packages.

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.

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 0.3.4

Blender

Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation. We used Blender to render some of the 3D terrain utilized to evaluate the trained model.

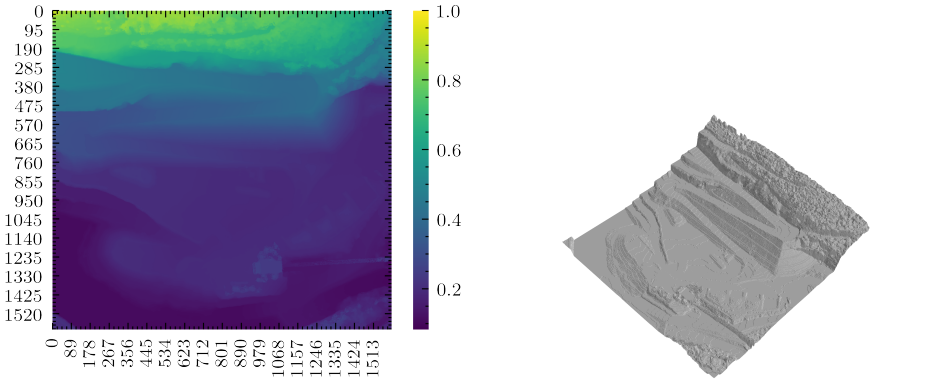
I switched to mayavi

0.3.2 Data Gathering

In this section we describe how we generate the synthetic maps, how we the robot interacts with them and all the postprocessing needed to create the final dataset.

Heightmap generation

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

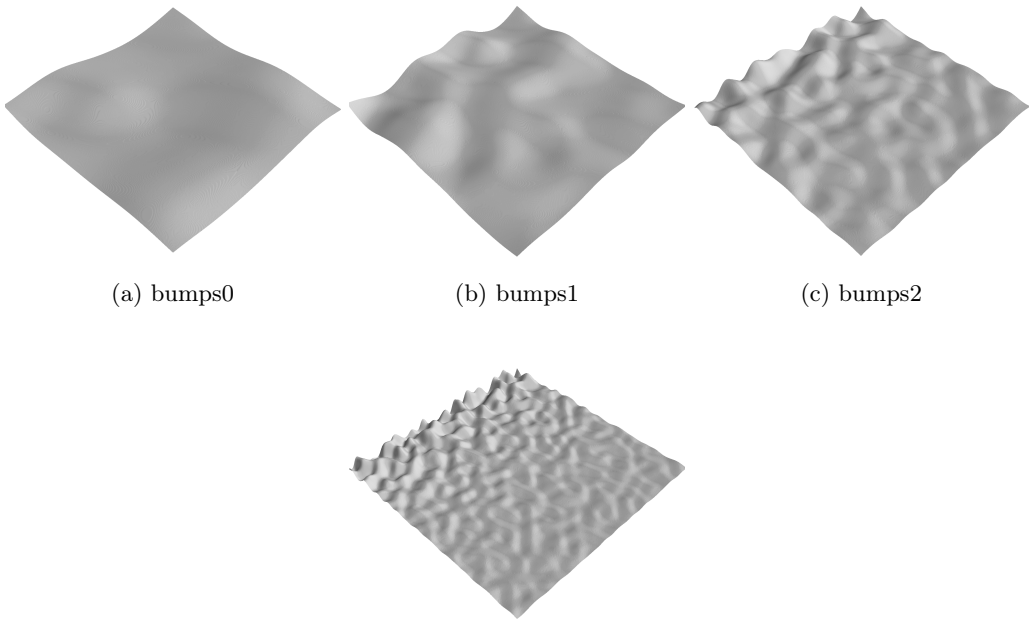


(a) An heightmap representing a Quarry. (b) 3D rendered

Figure 1. This image shows an heightmap and the 3D render of a Quarry.

We created thirty maps of 513×513 pixel with a resolution of $0.02cm/pixel$ in order to represent a $10 \times 10m$ terrain. To generate them with 2D simplex noise [16], a variant of Perlin noise [1], a widely used technique in the terrain generation litterature. We divide the maps into five main categories of terrains: *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 table **??**.

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

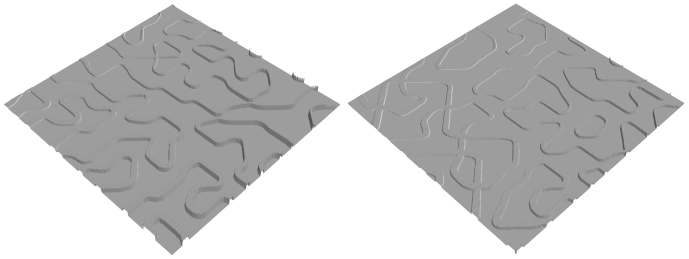


(a) bumps0 (b) bumps1 (c) bumps2

(d) bumps3

Figure 2. Bumps maps (10×10m).

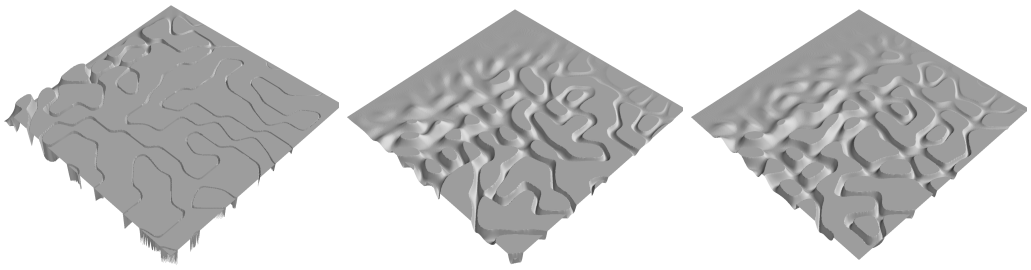
Bars: In these maps there are wall with different shapes and heights. In



(a) bars1 (b) bars3

Figure 3. Bars maps (10×10m).

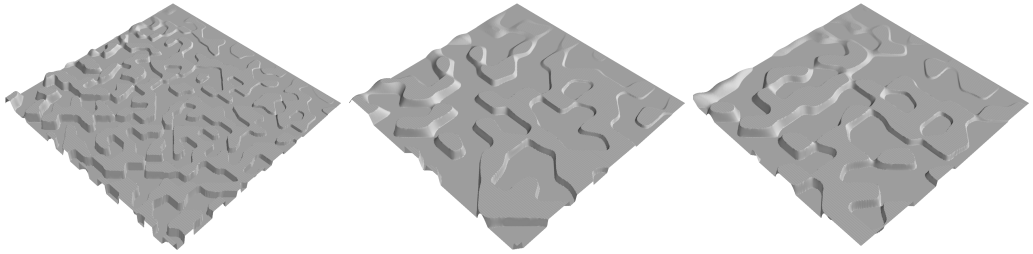
Rails: Flat grounds with slots.



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

Figure 4. Rails maps (10×10m).

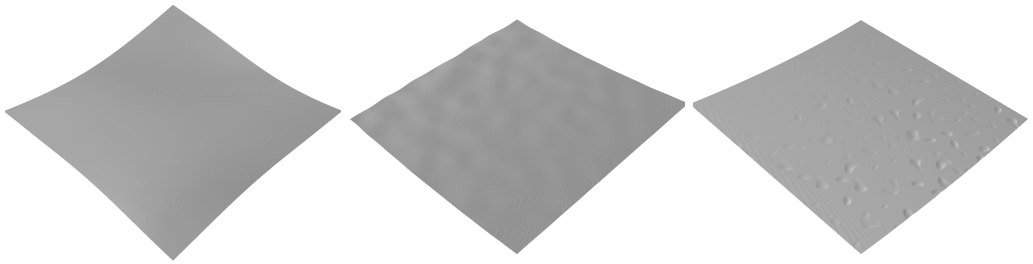
Steps: These are maps have huge wall and holes



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

Figure 5. Steps maps (10×10m).

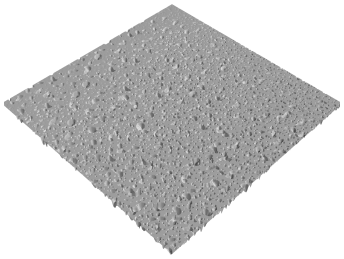
Slopes/Ramps: Maps composed by uneven terrain scaled by different height factors from 3 to 5 used to include samples where *Krock* has to climb. Those maps were scaled up with different height factors.



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

Figure 6. Slopes maps (10×10m).

Holes We also included a map with holes



(a) holes1

Figure 7. Holes map (10×10m).

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 $50hz$ by using ROS build it `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.

cite them?

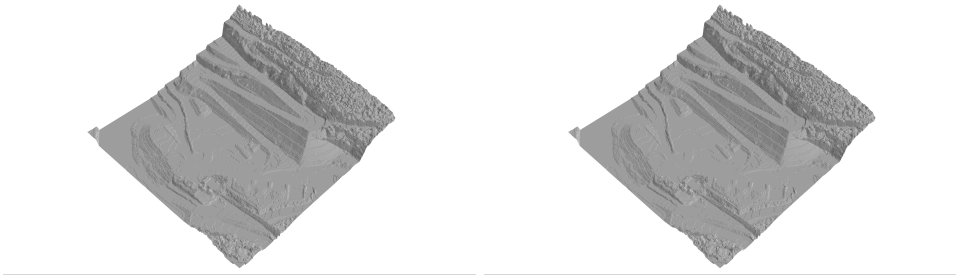
cite?

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. 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.

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. The following images shows them.



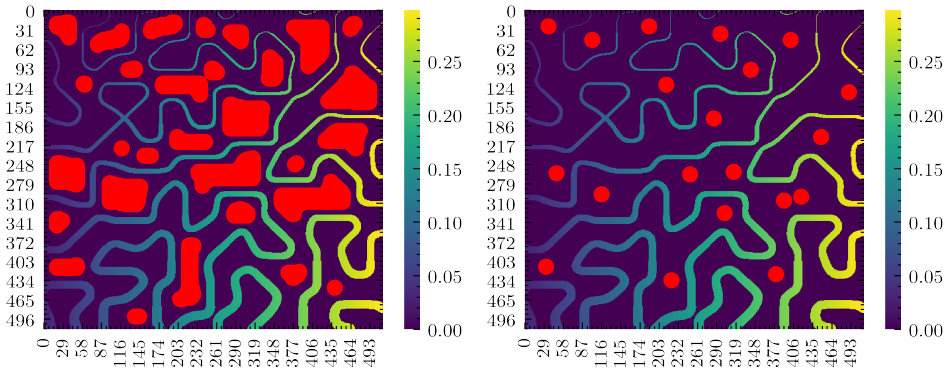
(a) Quarry (b) Quarry

Figure 8. Real world maps obtained from sensefly's dataset..

Quarry is used as test set to evaluate the model performance.

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 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*.



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

Figure 9. 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)
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 1. Maps configuration used in the simulator.

0.3.3 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.

Parse simulation 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.

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

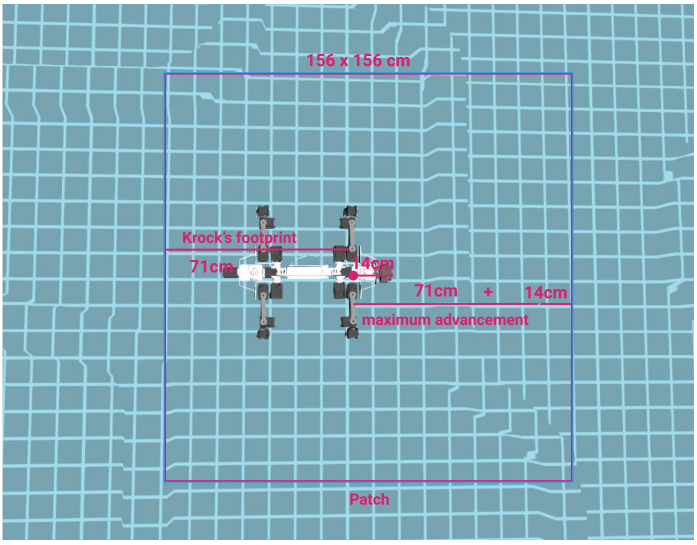
The correct value of Δt is crucial. We want a time window small enough to avoid smoothing to much the advancement and making obstacle traversale, and big enough to include the full legs motion. We empirically set $\Delta t = 2$ since it allows Krock to move both its legs and does not flat the advancement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

Extract patches

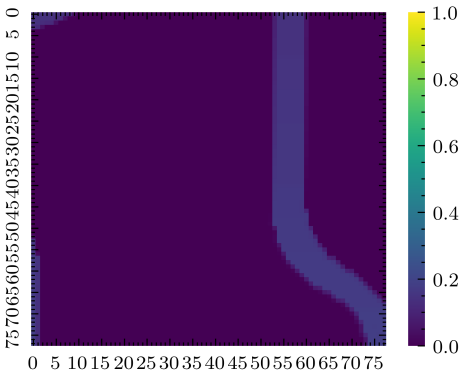
Each patch must contain both Krock's footprint, to include the situations where the obstacle is under the robot, and certain amount of ground region in front of it. Intuitively, we want to include in each patch the correct amount of future informations according to the selected time window. Thus, we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



(a) Robot in the simulator.



(b) Cropped patch in 2d.

(c) Cropped patch in 3d.

Figure 10. Patch extraction process for $\Delta t = 2$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 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.

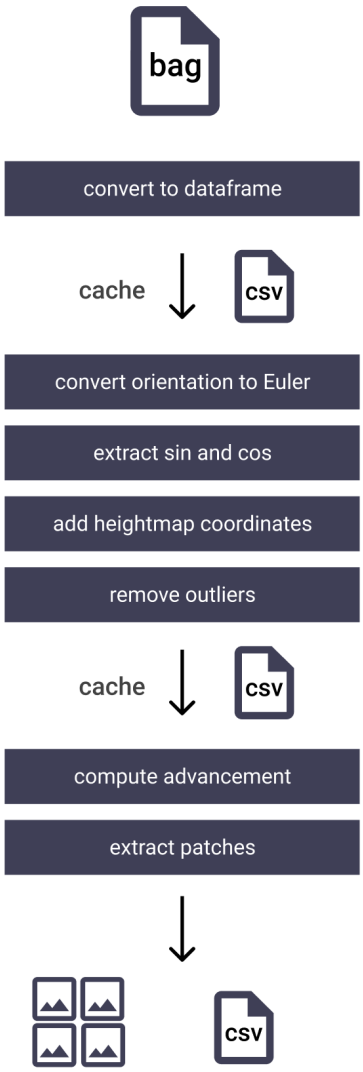


Figure 11. Postprocessing Pipeline flow graph, starting from the top.

The following figure shows the mean advancement across all the maps used to train the model in a range of ± 0.71cm, the maximum advancment the used time window, $\Delta t = 2$s.

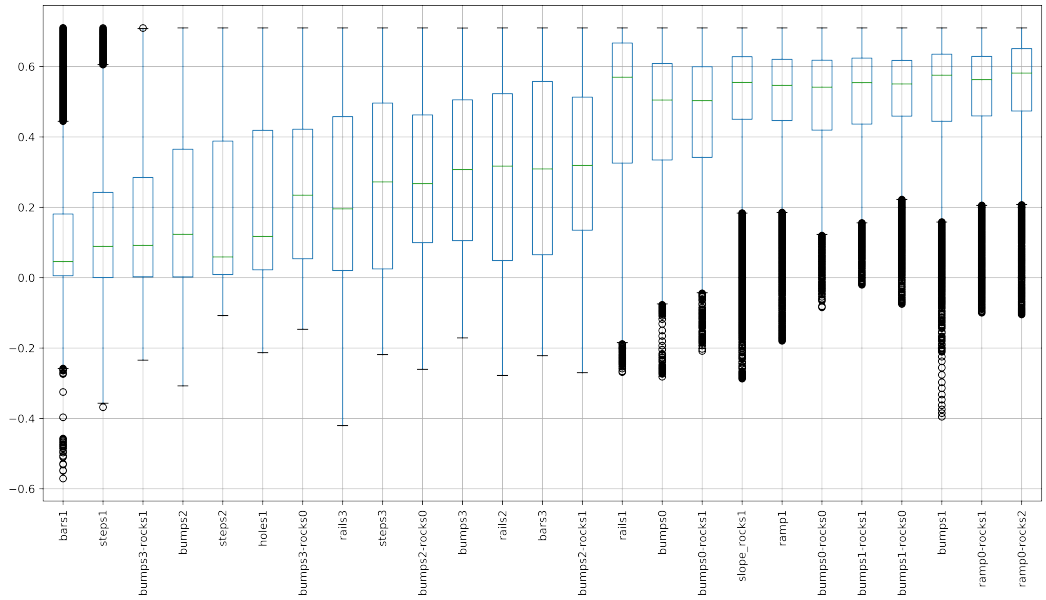
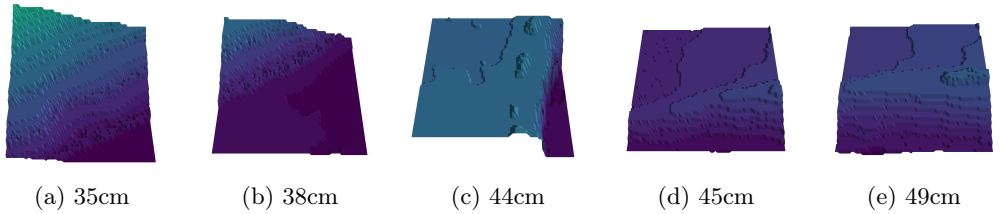


Figure 12. Advancement on each map with a $\Delta t = 2$s in ascendent order.

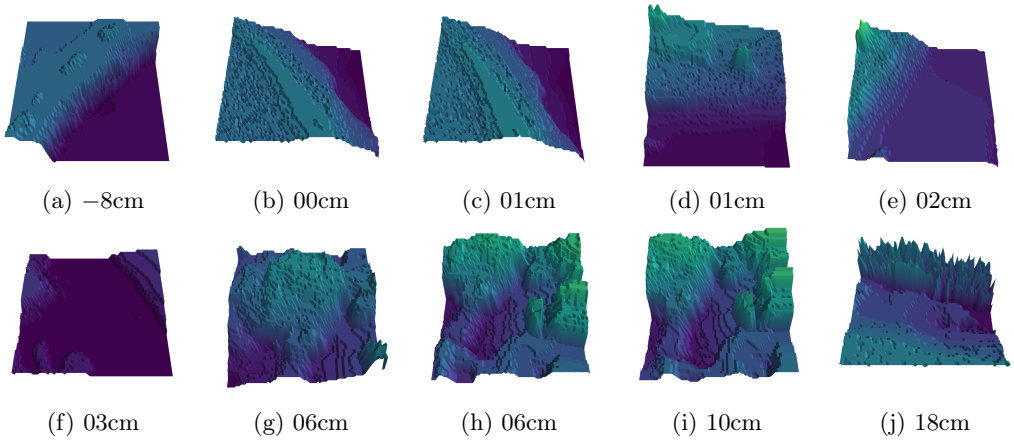
To give the reader a better idea of how the patches looks like, the next figures shows the 3d render of some patches extracted form *Quarry* ordered by advancement.



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

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

Figure 13. 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 14. Patches with low advancement Quarry a $\Delta t = 2$s.

Corretly, the patches with a lower advancement values have some ostacle in front of the robot, while the other set of inputs are almost flat.

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.

Label the 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 $tr_{\Delta t=n}$

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.

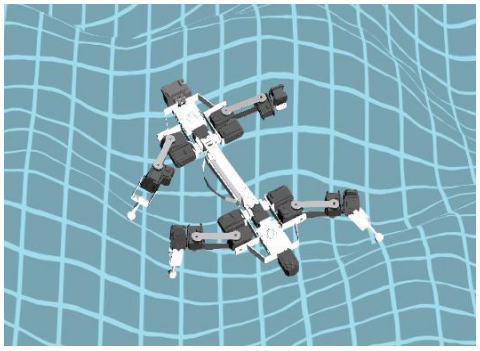
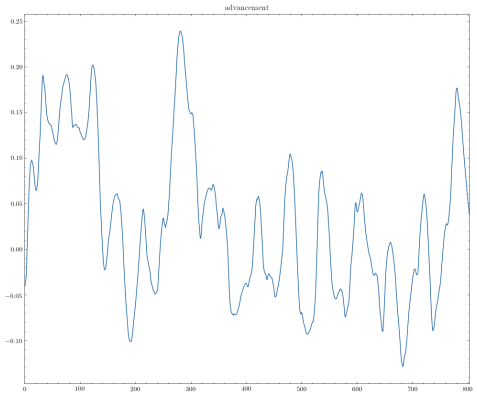


Figure 15. Krock tries to overcome an obstacle in the bumps3 map.

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.

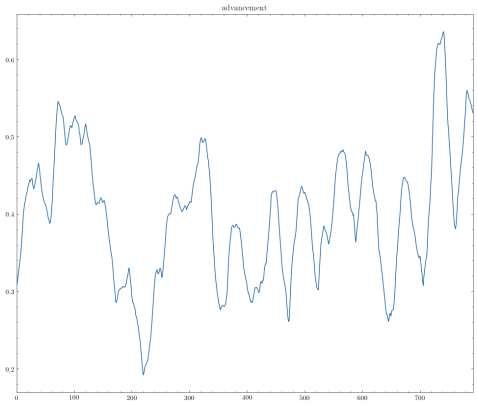


(a) $\Delta t = 2$s.

Figure 16. Advancement over time with different Δt on bumps3.

A wheel robot, on the other hand, will not produce such graph since it cannot free itself easily from an obstacle.

Ramps To correctly decide the threshold we must be sure it will not create false negative. For this reason, we check the advancement on the $slope_rocks1$ map with a height scaling factor set to 5.



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

Figure 17. Advancement over time with different Δt on slope_rocks3.

We know that those patches are traversable even if sometimes the robot may find it hard to climb them. So, we want to choose a threshold that is beetween the upper bound

of *bumps* and beetween the lower bound of *slope_ rocks1*. Thus, good thresholds value is $tr_{\Delta t=2s} = 20cm$.

0.3.4 Estimator

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

Vanilla Model

The original model proposed by Chavez-Garcia et all [6] 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.

MicroResNet

We adopt a Residual network, ResNet [8], 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* [11] and a *Relu* blocks. Formally defined as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + h(\mathbf{x}) \tag{1}$$

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

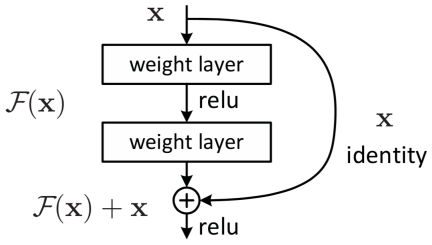


Figure 18. Resnet block [8]

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.

Preactivation

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

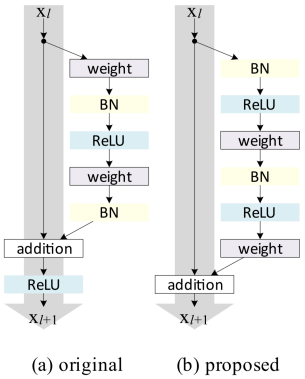


Figure 19. Preactivation [9]

Sqeeze 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 $\boldsymbol{F}_{tr} : \boldsymbol{X} \mapsto \boldsymbol{U}$, $\boldsymbol{X} \in mathbbR^{H' \times W' \times C'}$, $\boldsymbol{U} \in \mathbb{R}^{H \times W \times C}$, the SE module first squeeze the information by using average pooling, $\boldsymbol{F}_{sq}$, the it excitates them using learnable weights, $\boldsymbol{F}_{ex}$ and finally, adaptive recalibration is performed, $\boldsymbol{F}_{scale}$. The next figure visualises the SE module.

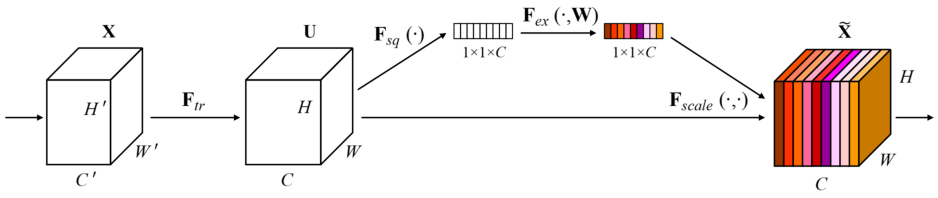


Figure 20. Squeeze and Excitation [10]

Micro resnet

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 [4] with a negative slope of 0.1 instead of ReLU to allow a better gradient flow during backpropagation. LeakyRelu is defined as

ref to Resnet Table

follows

$$LeakyRelu(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1x & \text{otherwise} \end{cases} \tag{2}$$

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.

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 2. MicroResNet architecture from top to bottom. Some part of the architecture are equal across models, this is shows by sharing columns in the table.

this table sucks

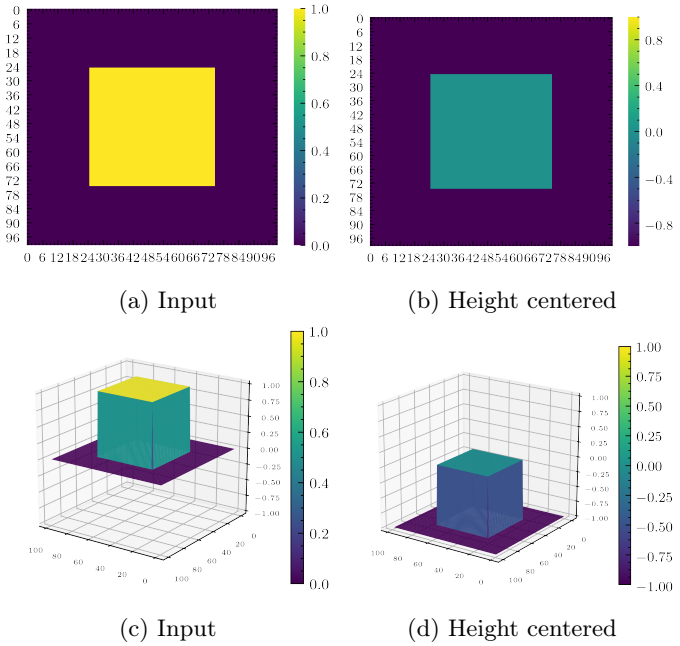
Our models have approximately 35 times less parameters than the smalles ResNet model, ResNet18, that has 11M parameters. To simplicity we will use the following notation to describe each architecture variant: `MicroResNet-3x3/7x7-/SE`.

ask omar help to add margin in the rows

add model picture

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.



(a) Input (b) Height centered

(c) Input (d) Height centered

Figure 21. Normalization process

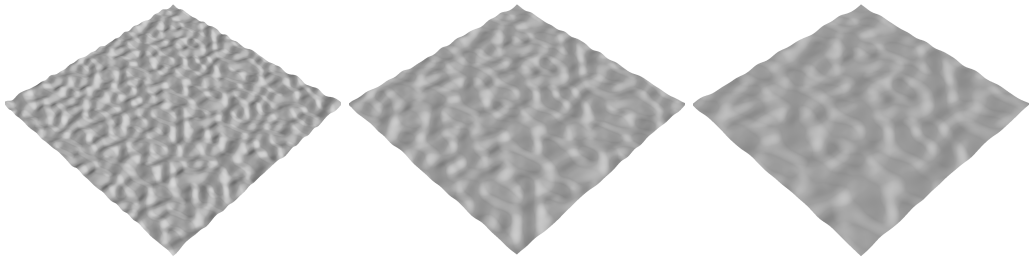
Data Augmentation

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 21.

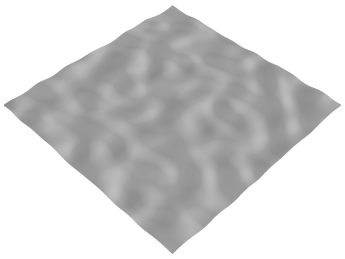
Dropout is a technique to randomly set some pixels to zero, in our case we flat some random pixel in the patch.

Coarse Dropout similar to dropout, it sets to zero random regions of pixels.

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.



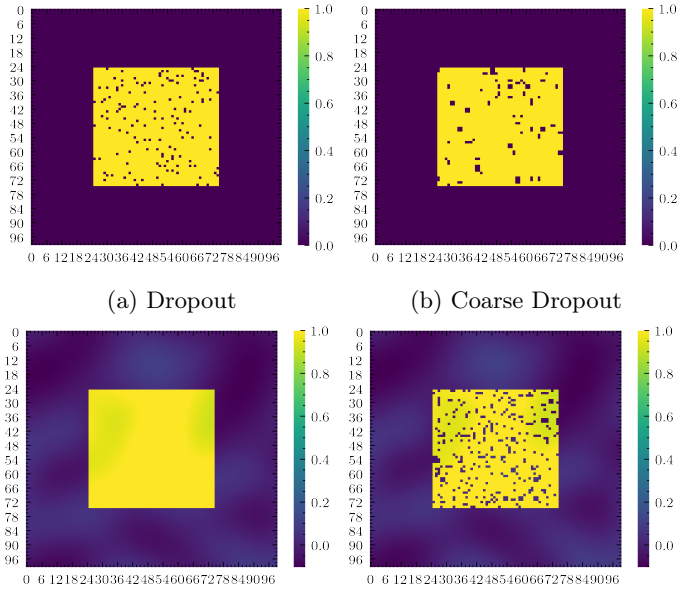
(a) Features size = 10 (b) Features size = 20 (c) Features size = 30



(d) Features size = 40

Figure 22. Simplex Noise on flat ground

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

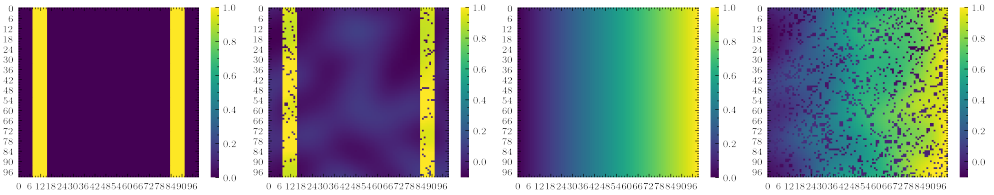


(a) Dropout (b) Coarse Dropout

(c) Simplex Noise (d) Final result

Figure 23. Data augmentation applied on a patch.

It follows an other set of figures that shows the data augmentation applyed on different inputs.



(a) Walls (b) Walls aug (c) Ramp (d) Ramp aug

Figure 24. Data augmentation applyed first on a patch with two walls, then on a ramp.

In all the traning epochs, we data augment each input image x with a probability of 0.8. Dropout has a probability between 0.05 and 0.1. Coarse dropout with 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 with a feature size between 1 and 50 with a random scaling factor between 6 and 10.

0.4 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.

0.4.1 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.

0.4.2 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. The 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 and weight decay to $1e-4$ with an initial learning rate of $1e-3$ as was originally proposed to train residual network [8]. However, we later utilize Leslie Smith's 1cycle policy [18] 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.

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

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.

0.4.3 Quantitative Results

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.

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

		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 3. Model comparison on the test set.

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 [6].

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 4. 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.

0.4.4 Final results

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

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. Final results on different datasets.

we would like to also show the different steps we made to reach this result. The following table shows the metric's score without any data-augmentation. Adding dropout increases the results. With dropout plus coarse dropout.

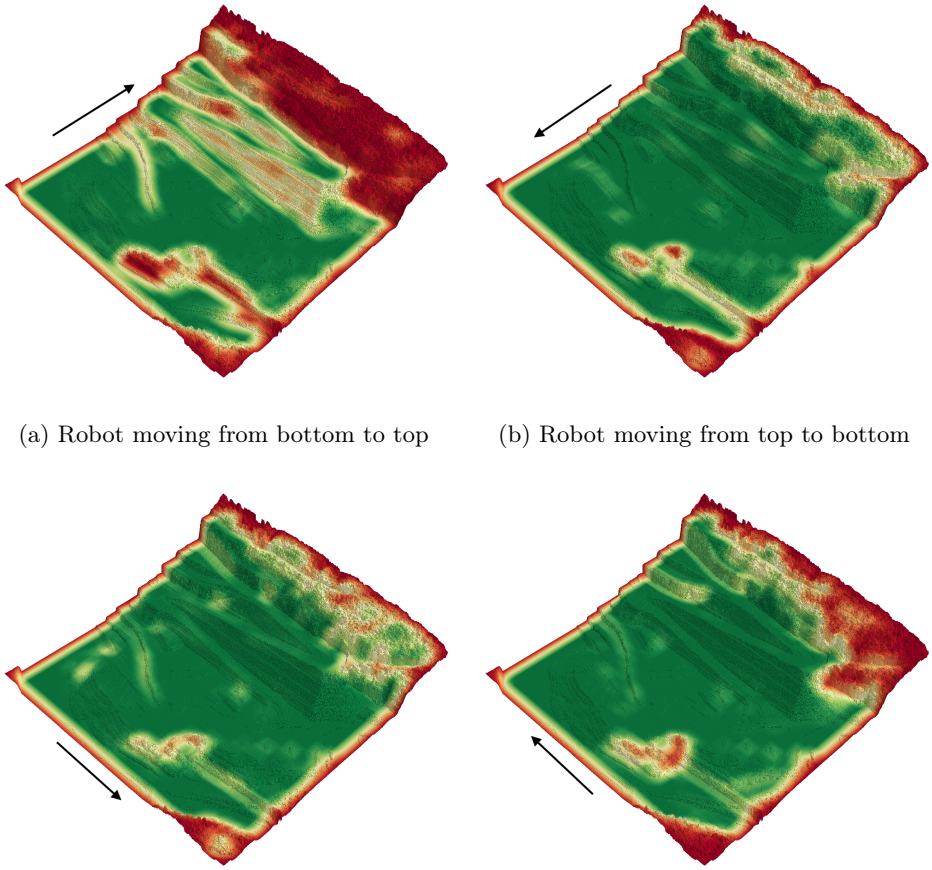
0.4.5 Qualitative results

We qualitative evaluate the model predictions by showing the traversability probability on different maps in 3D as a texture. Specifically, we used a sliding window to extract the patches from the heightmaps and create a texture based on the model's output for the traversable class. Then, we apply this texture on the image and colour using a colormap. For each map we show the traversability from bottom to top, top to bottom, left to right and right to left since those are the most human understandable.

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



(a) Robot moving from bottom to top

(b) Robot moving from top to bottom

(c) Robot moving from left to right

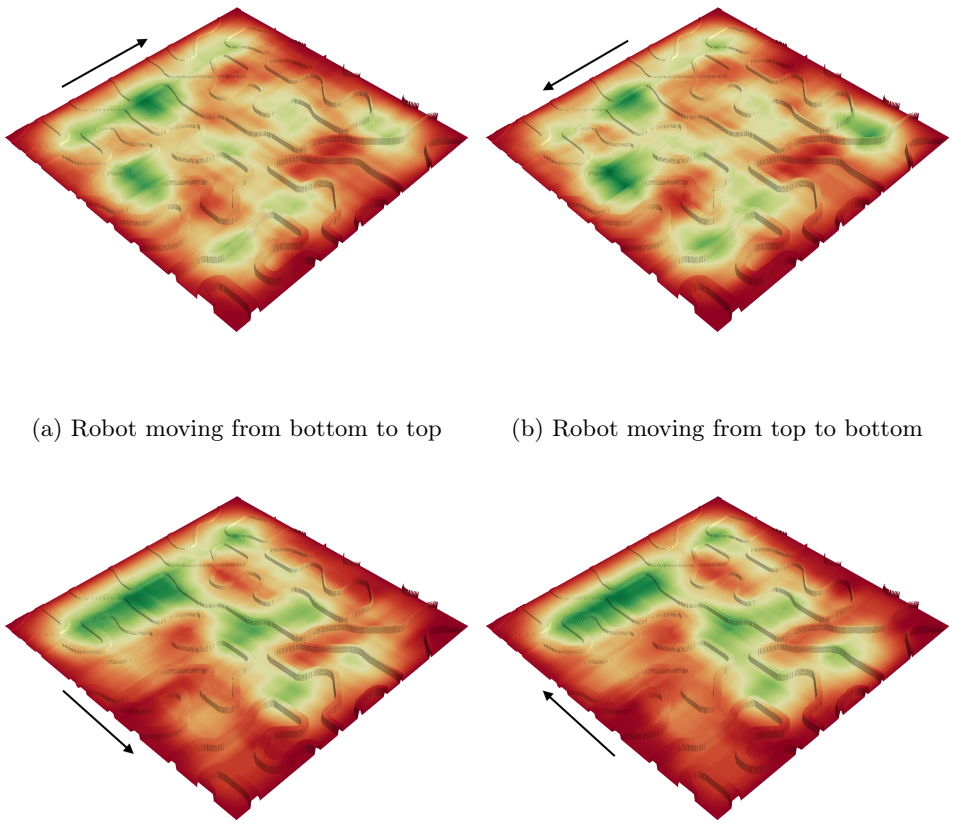
(d) Robot moving from right to left

Figure 25. 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.

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.

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 26. 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.

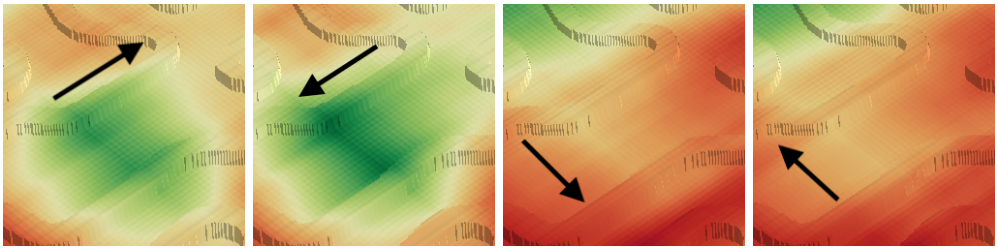


Figure 27. Detail of a region in the bars map where there are two walls forming a tunnel. Correctly, when the robot is travel following the trail the region is label as traversable.

Sullen

The following map is a small village downloaded from sensefly online datasets hub. Since the map is huge, we crop a

0.5 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 introduce on 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 were 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.

0.5.1 Grad-CAM

Informally, Grad-CAM [17] is a technique to produce "visual explanations" for convolutional neural networks. It uses the gradient flowing into the last convolution layer to highlight a regions of interested in the image that contribute the most to the final prediction.

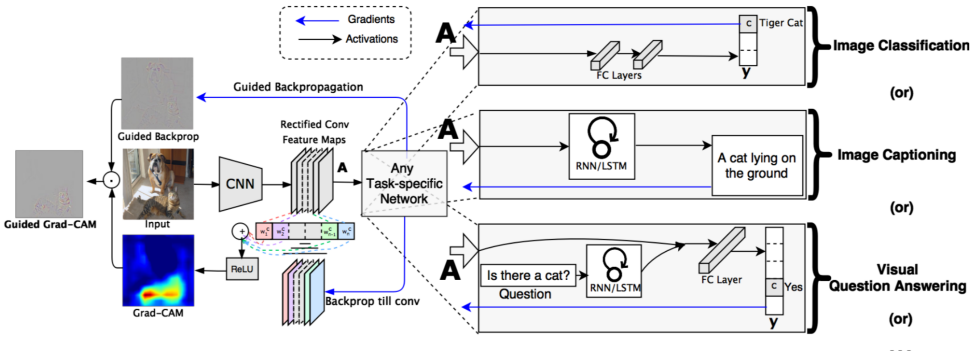


Figure 28. Grad-CAM procedure on an input image [17].

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 applyed 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 multiplyed with the averaged gradients to create the grad cam. This whole procedure is fast and it is architecture independent.

0.5.2 Robustness

To test the model's robustness we createed custom patches with different features, walls/bumps/ramps, and test the model prediction against the real robot advancement obtained from the simulator. According to the previous experiments, we used a threshold of 20cm and a time window of two seconds.

Wall in front of Krock

The most trivial test is to place a not traversable wall in front of *Krock* at an increasing distance from the its head. We will expect to reach a point where the model predict 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 the wall 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.

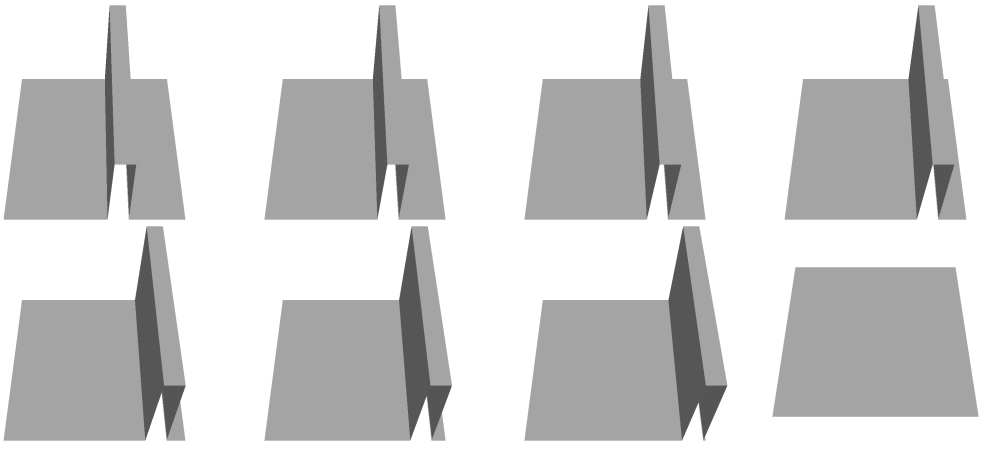


Figure 29. Some of the tested patches with a non traversable wall at increasing distance from Krock.

Given those inputs to the model, we get the following predictions.

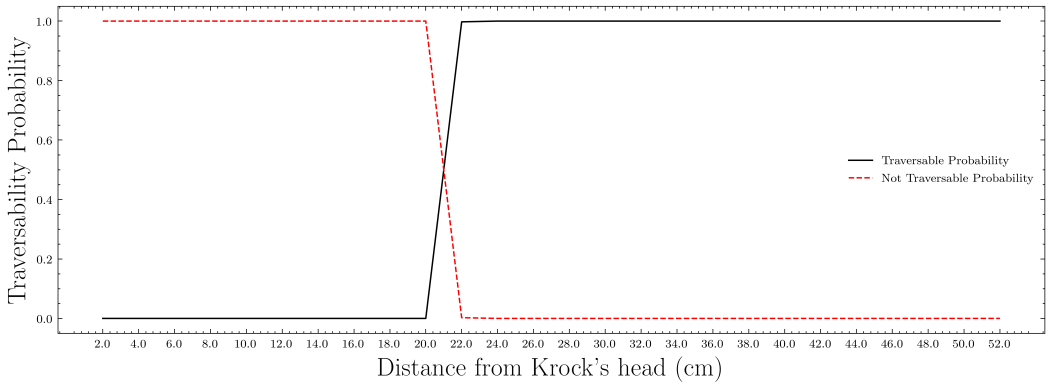


Figure 30. Traversability probabilities against wall distance from Krock's head.

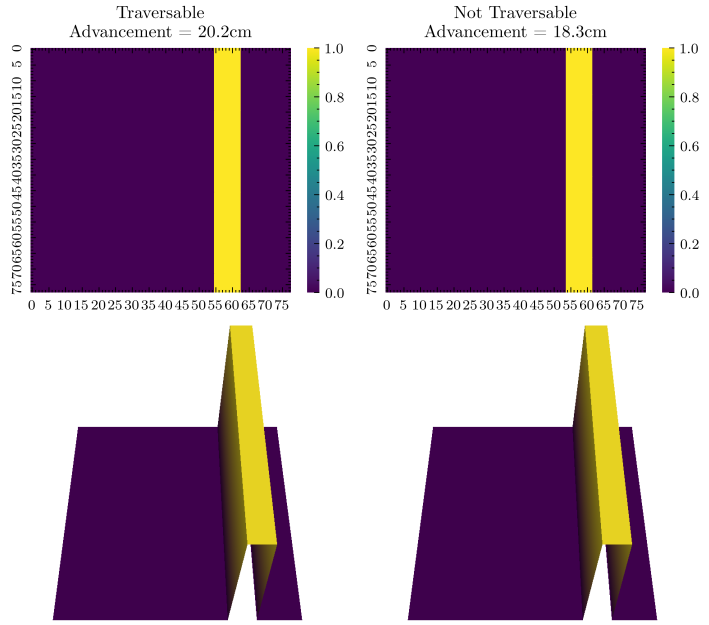
graph too tall, adjust the figure size

Summarized by the following table:

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

Table 6. 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 19.9cm on the not traversable patch (a) where the wall is at a distance of 19.6cm from the head. While, on the first traversable patch in which the wall is at a distance of 20.5cm, the robot was able to travel for 22.4cm. This shows the network ability to correct understanding that distance from the obstacle is more relevant than its height.

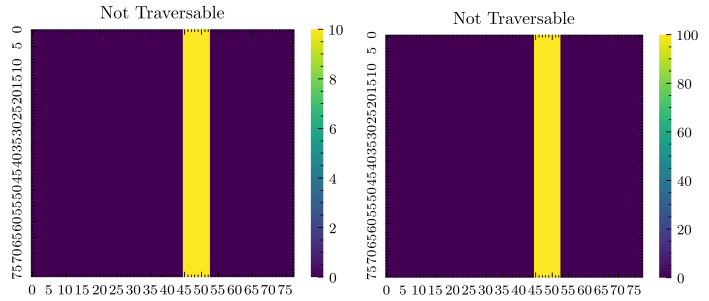


(a) Distance 20cm (b) Distance 22cm

Figure 31. The last non traversable and the first traversable patches with wall at a distance $< tr$ and $>$ tr.

Due to the patch resolution and minor changes in the spawing position, the robot may advance more than the distance between its head and the wall.

Furthermore, we can increase the wall size of the first traversable patch, (b), to 10 and to 100m to stress even more the ability of the model to look at the distance and not at the height.



(a) height = 10m (b) height = 100m

Figure 32. Two patches with a very tall wall at a distance $> tr$.

remove title, second colormap is wrong

Correctly, the model classifies the patches as traversable and was not confused by the enourmous height of the wall.

Increasing height walls

We can also evaluate the model's robustness by placing different wall of increasing heights in front of the robot 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.

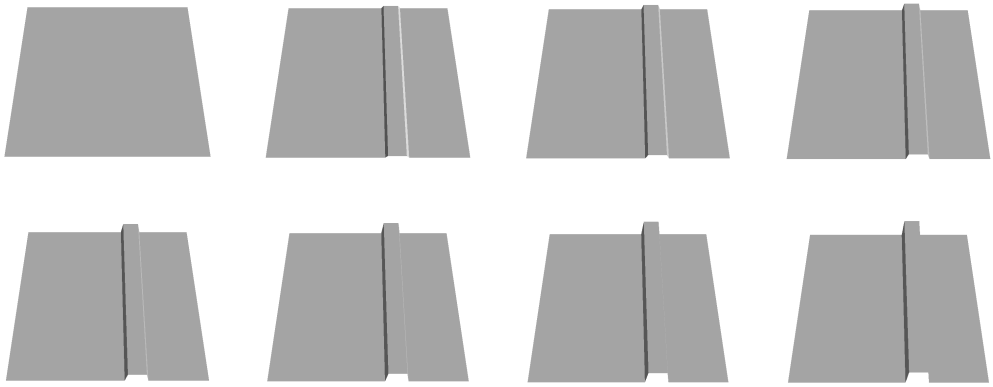


Figure 33. Some of the tested patches with a wall at increasing height ahead of Krock.

The models predicts that the walls under 10cm are traversable.

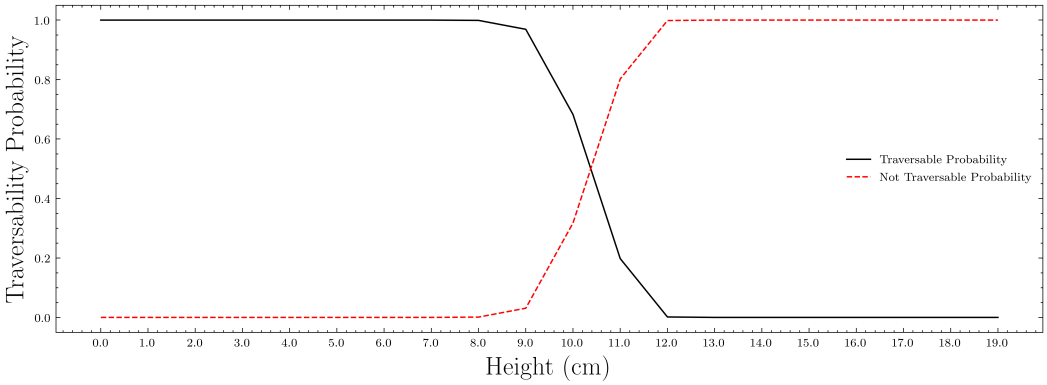


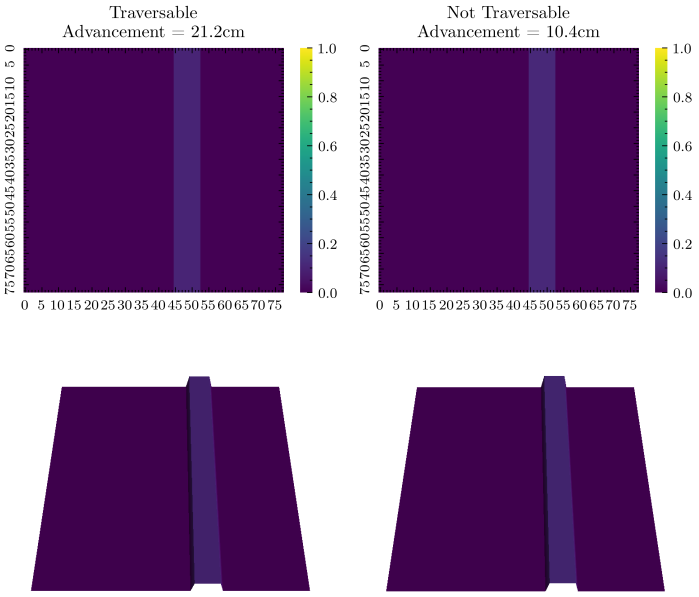
Figure 34. Traversability probabilities against walls height in front of Krock.

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

Table 7. 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.



(a) height 9.7cm (b) Height 10cm

Figure 35. The last traversable and the first non traversable patches with a increasing height wall ahead of Krock.

In the first case, the simulator outputs and advancement of 39.5cm meaning that Krock was able to overcome the obstacle, while it fails in the second case. Correctly, the predictions match the real data.

Increasing height and distance walls

We can combine the previous experiments and test the model prediction against the ground truth for each height/distance combination. To reduce the number of samples and improve readability, we limit ourself 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. The following image shows the traversability probability for each patch.

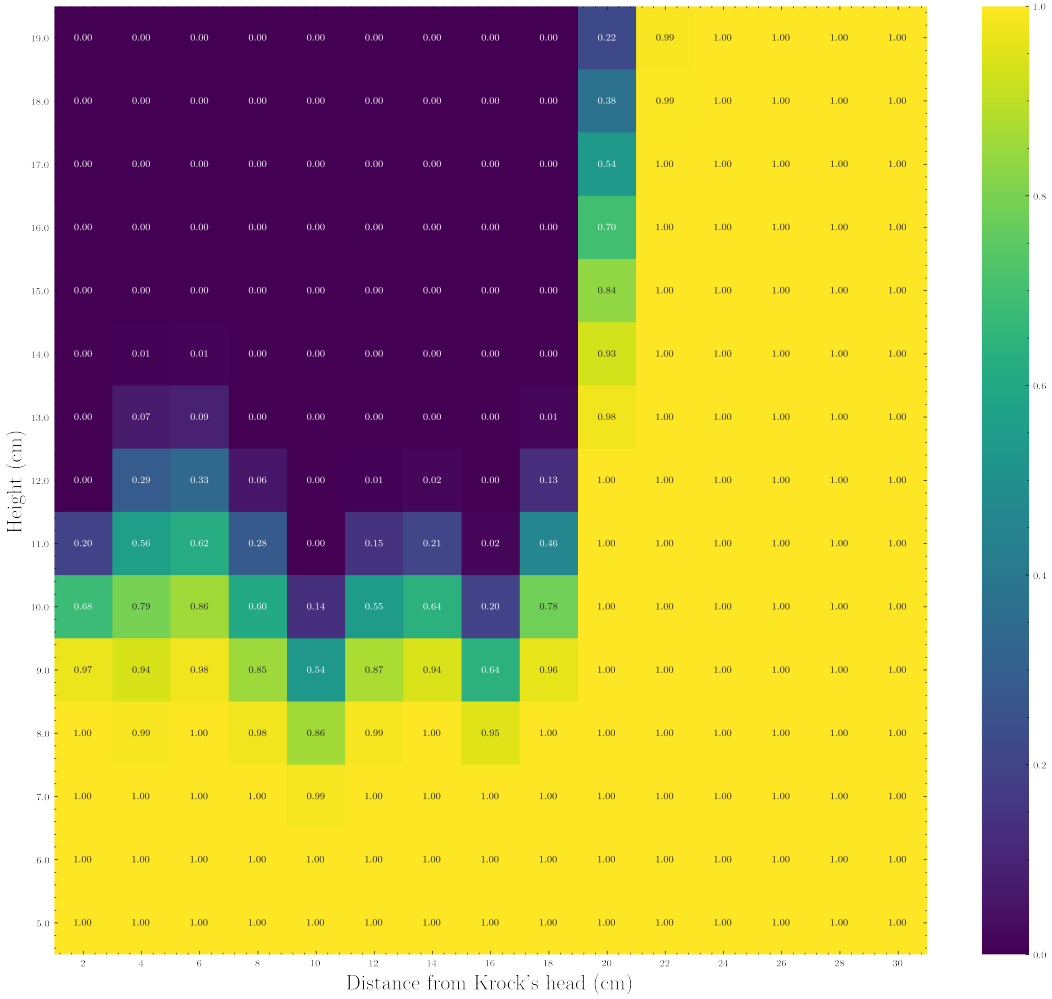


Figure 36. Traversability probabilities for patches with a wall of increasing height and distance form Krock's head.

missing ground truth heatmap

FINISH

Tunnel

do it

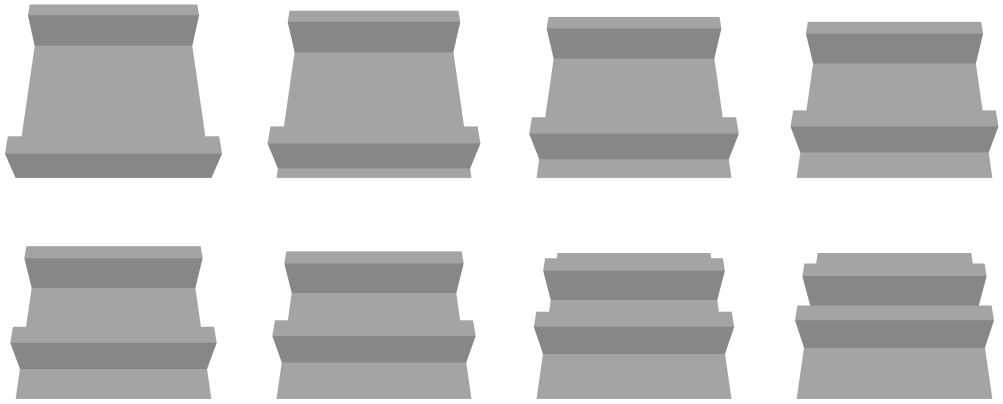


Figure 37. Some of the tested patches with tunnel at different distances.

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.

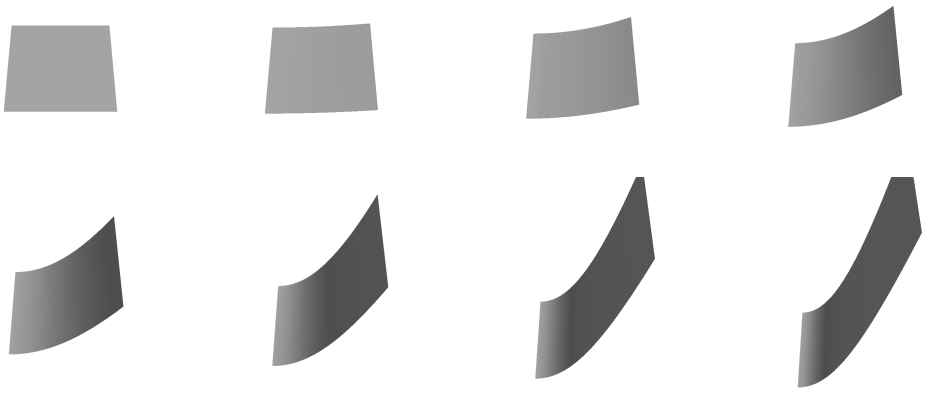


Figure 38. Some of the tested patches with steep ramps.

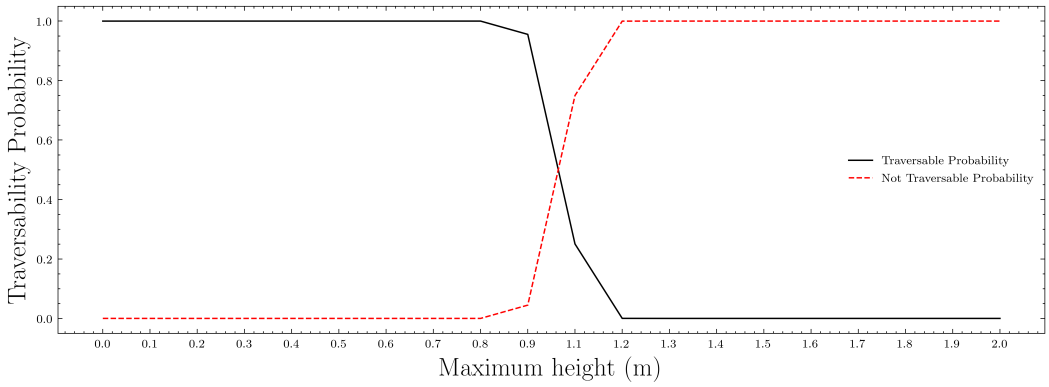


Figure 39. Traversability probabilities against maximum height of each ramp.

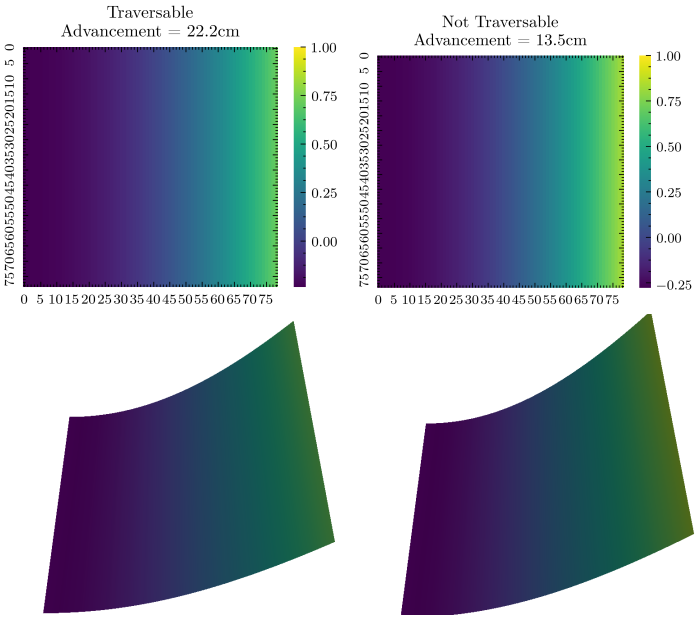
x labels are wrong, why?

The following table summarizes the results.

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

Table 8. 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.



(a) Height 0.94cm (b) height 1.1m

Figure 40. 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

Holes

do it

0.5.3 Quarry Dataset

We start evaluating our model by using the *Quarry* map. We expect the model to correctly classify the patches with easy to see features such as big obstacles, steep ramps and holes. Unfortunately, the dataset is not trivial and most of the patches are challenging to classify even to human eye.

For instance, if look at patches, for some of them is not so easy to estimate the advancement by human eye. This is due to the specific robot locomotion that depends on the starting pose. Our goal in this section is to explain the model predictions on different inputs.

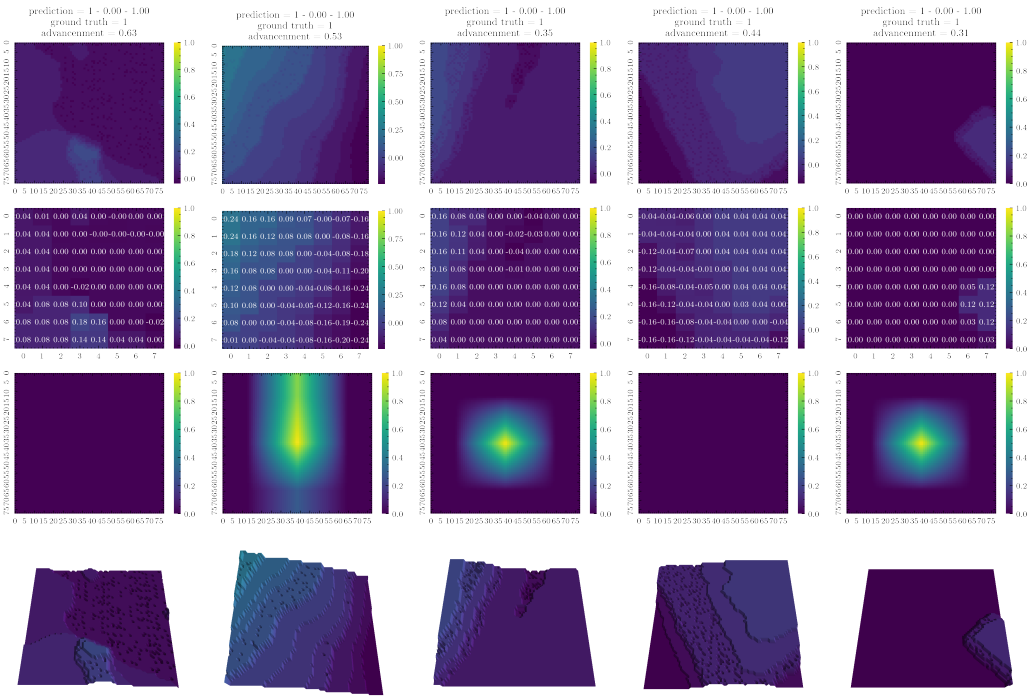


Figure 41. Best

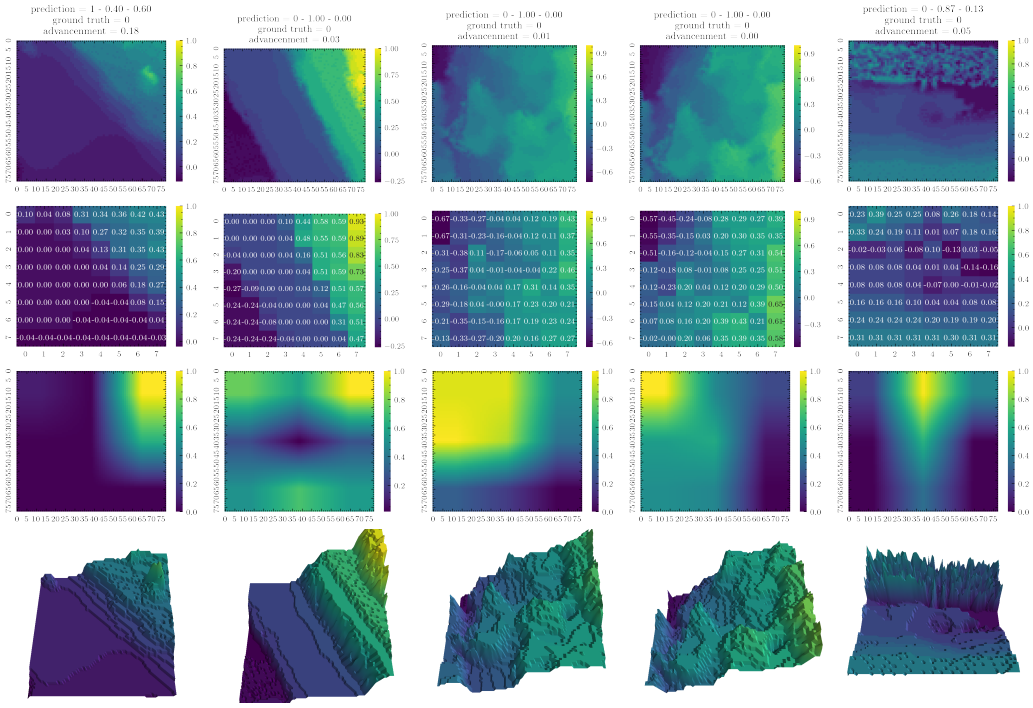


Figure 42. Worst

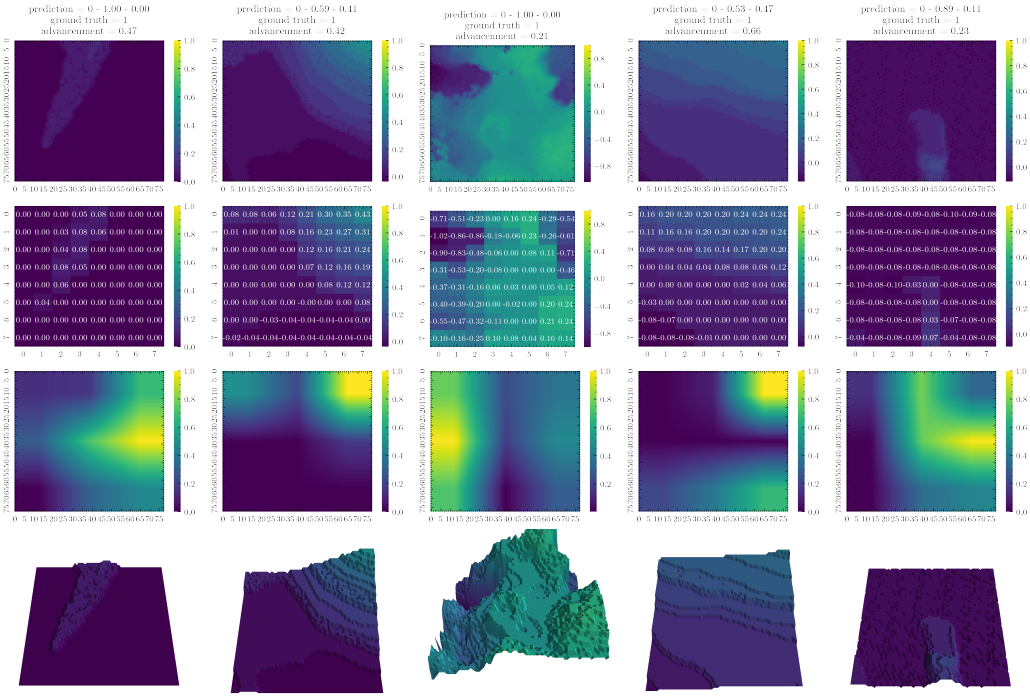


Figure 43. False Positive

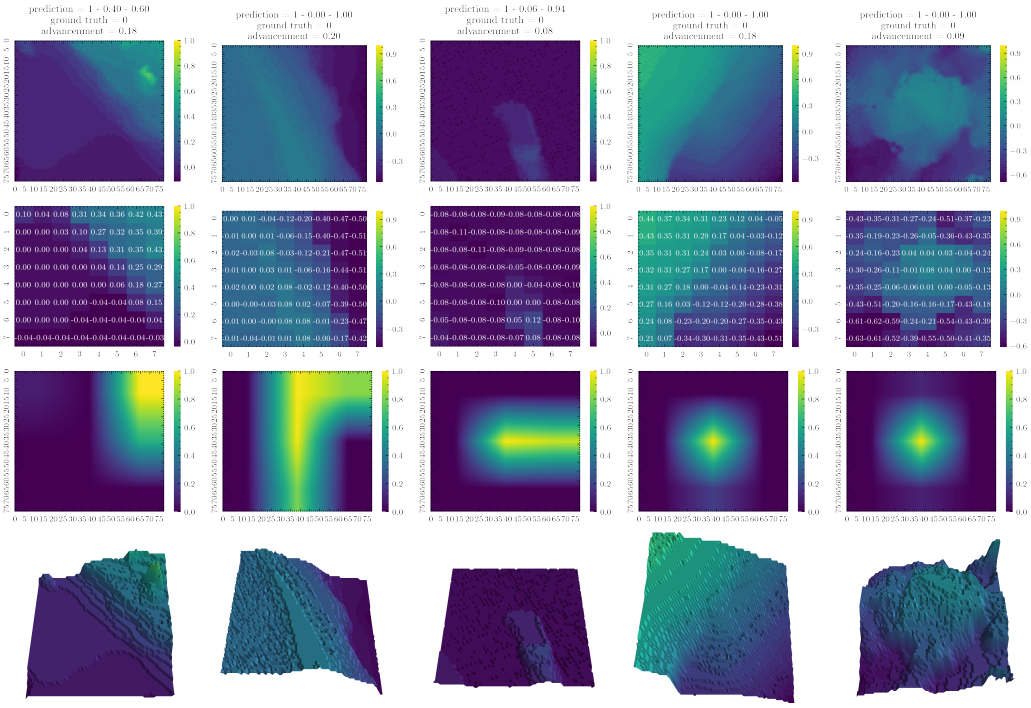


Figure 44. False Negative

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.

[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.

[6] R. Omar Chavez-Garcia et al. “Learning Ground Traversability from Simulations”. In: (2017). doi: 10.1109/LRA.2018.2801794. eprint: arXiv: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.

[9] Kaiming He et al. *Identity Mappings in Deep Residual Networks*. 2016. eprint: arXiv:1603.05027.

[10] Jie Hu et al. *Squeeze-and-Excitation Networks*. 2017. eprint: arXiv:1709.01507.

[11] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: arXiv: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] Wellhausen Lorenz et al. “Where Should I Walk? Predicting Terrain Properties from Images via Self-Supervised Learning.” In: (2019).

[16] K. Perlin. “Improving noise”. In: *ACM Transactions on Graphics* ().

[17] Ramprasaath R. Selvaraju et al. *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization*. 2016. eprint: arXiv:1610.02391.

[18] 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`.

[19] Boris Sofman et al. "Improving Robot Navigation Through Self-Supervised Online Learning". In: (2006).

[20] Matti Pietikainen Timo Ojala and Topi Maenpaa. "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns." In: (2002).

[21] E. Ugur and E. Sahin. "Traversability: A case study for learning and perceiving affordances in robots". In: (2010).

[22] Martin Wermelinger et al. "Navigation Planning for Legged Robots in Challenging Terrain." In: (2016).