1 Abstract

With this project, we estimate ground traversability for a legged crocodile-like robot. We generate different synthetic grounds and let the robot walk on them in a simulated environment to collect its interaction with the terrain.

Then, we train a deep convolutional neural network using the data collected through simulation to predict wether a given ground patch can be traverse or not. Later, we highlight the strength and weakness of our method by using interpretability techniques to visualise the network's behaviour in interesting scenarios.

2 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.

3 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 [18]. 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 correctly 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 [16], 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. [5], 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. [11] 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 [18] or Gaussian models [16]. Those techniques rely on texture descriptors, for example, Local Binary Pattern [17], 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.[6] extended the previous work [11] 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. [12] 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. [5].

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. [**wermelinger2016navigation**] 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. [13] 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. [**wellhausen2019where**] 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.

Add mirko's paper

4 Implementation

In this section we show the reader all the details about the implementation starting from the employed software. We will first present all the main tools used in the implementation and then describe in detail each pipeline's step.

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 Matplotlib

To create almost all the plots in this report we used 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.

4.1.4 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 ¨big data¨to other frameworks such as Spark or Hadoop. We used Pandas to store the results from the simulator and inside a Thread Queue to parse each *.csv* file efficiently.

4.1.5 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.

4.1.6 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 [2] are a cleaner API structure, better debugging, code shareability and an enormous number of high-quality third-party packages.

4.1.7 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.

4.1.8 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 4.4.7

4.1.9 Blender

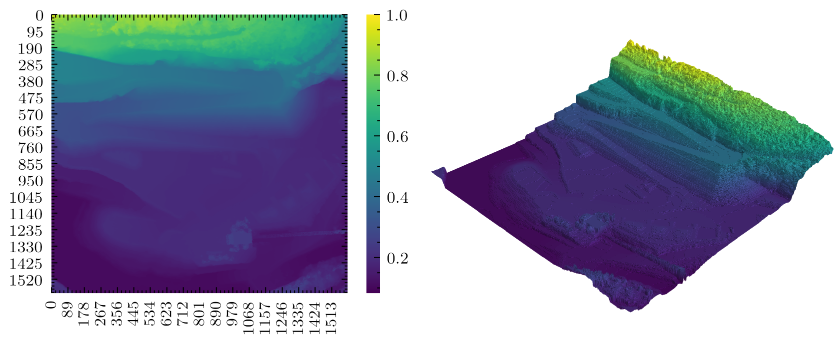
Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation. We used Blender to render some of the 3D terrain utilized to evaluate the trained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

4.2 Data Gathering

In this section we first describe how we generated the synthetic maps, how we let the robot intereact with them and all the postprocessing needed to generate the final dataset.

4.2.1 Heightmap generation

To collect the data throught simulation we first need to generate meaningful terrain to be explored by the robot those maps are create using heightmaps. A heightmap is a 2D array, an image, where each pixel's value represents the terrain height. The following figure shows an heightmap and the relative terrain.

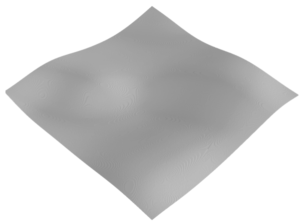


(a) *Quarry* heightmap (b) *Quarry* map

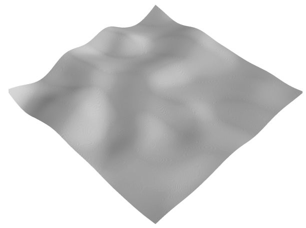
Figure 1: A heightmap

We used thirty maps of 513×513 pixel with a resolution of $0.02cm/pixel$ in order to represent a $10 \times 10m$ region. We generated them by 2D simplex noise [14], a variant of Perlin noise [1], a widely used technique in terrain generation litterature. We created four main categories of terrains: *bumps*, *rails*, *steps* and *slopes/ramps*. For each map we add three different rocky texture to create even more different situations.

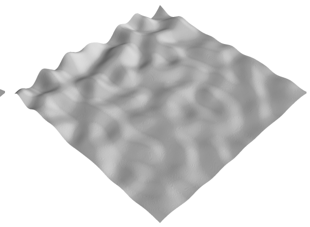
Bumps: We genered four different maps with increasing bumps using simplex noise with features size $\in [200, 100, 50, 25]$.



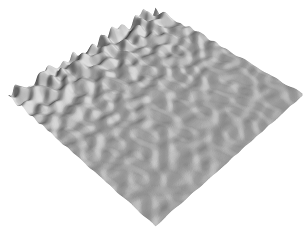
(a) *bumps0*



(b) *bumps1*



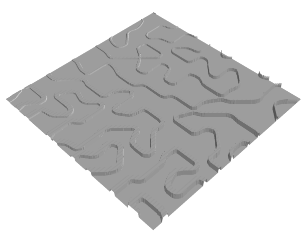
(c) *bumps2*



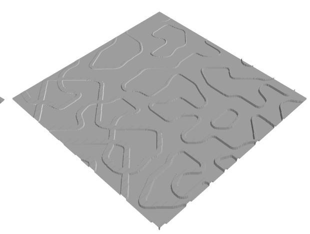
(d) *bumps3*

Figure 2: Bumps maps (10×10m).

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



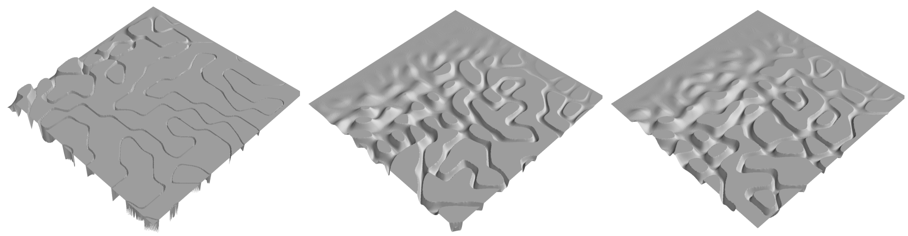
(a) *bars1*



(b) *bars3*

Figure 3: Bars maps (10×10m).

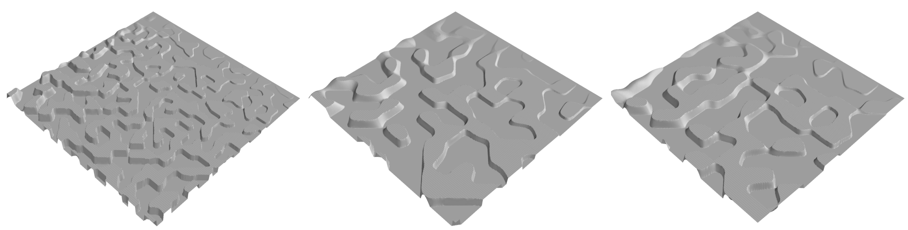
Rails: Flat grounds with slots.



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

Figure 4: Rails maps (10×10m).

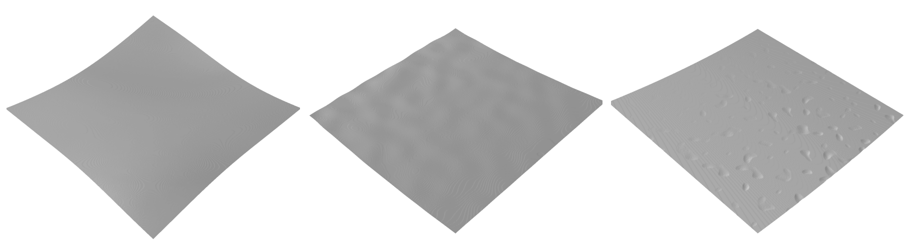
Steps: These are maps have huge wall and holes



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

Figure 5: Steps maps (10×10m).

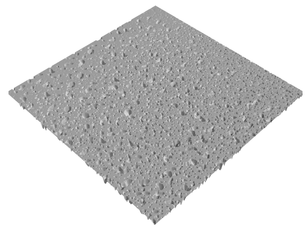
Slopes/Ramps: Maps composed by uneven terrain scaled by different height factors from 3 to 5 used to include samples where *Krock* has to climb.



(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).

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 $50hz$ by using ROS build it `throttle` command. To load the map into the simulator we had to convert it to Webots's *.wbt* file. Unfortunately, the simulator lacks support for heightmap so we had to use a script to read the image and perform the conversion.

cite them?

cite?

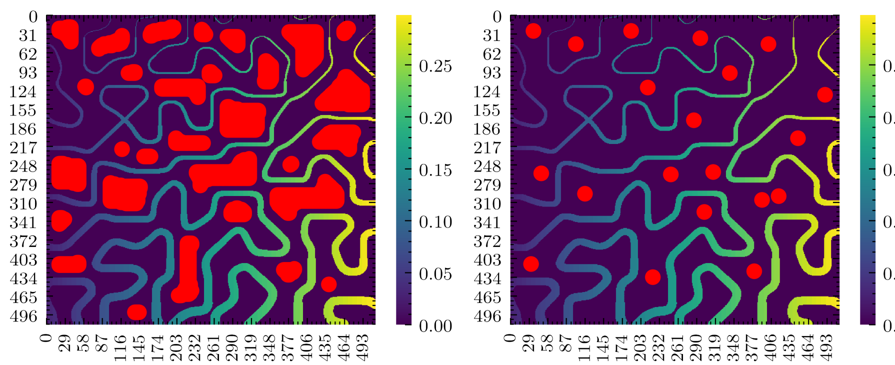
To communicate with the simulator, Webots exposes a wide number of ROS services, similar to HTTP endpoints, with which we can communicate. The client can use the services to get the value of a field of a Webots' Node, for example, if we want to get the terrain height, we have to ask for the field value `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* [4], 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 code.

4.2.3 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*, we must avoid spawning on an obstacle otherwise the run will be ruined by *Krock* getting stuck at the beginning. To solve the problem, we define a random spawn strategy used in most of the maps without big obstacles such as *slope_rocks*, and a flat ground spawn strategy for the others. The random spawn just selects a random position and rotation for the robot. On the other hand, the flat ground strategy first 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. Intuitively, if a patch is flat the mean value will be close to zero.

Since there may be more flat spawing positions than simulations we need to run we have to reduce the size of the candidate points. To maintaings the correct distribution on the map to avoiding spawing the robot always in the same cloud of points, we used K-Means with k clusters where k is of simulations we wish to run. By clustering, we guarantee to cover all region of the map avoiding adding bias. The following picture shows this strategy on *bars1*.



(a) Flat regions

(b) K-Means with $k = 20$

Figure 8: 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.

4.3 Postprocessing

We now need to extract the patches for each pose p_t of *Krock* and compute the advancement for a given time window. All the handles are available as a python package. 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

4.3.1 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 used later to crop the correct region of the map.

Finally, we load again the last stage and compute the advancement by projecting the pose's position, x and y, in the current line. Then, we select a time window according to the store rate, for if we select two seconds we need to multiply the rate by two, so $50 * 2 = 100$ since *Krock* published with a $50hz$ frequency. Once the time window is defined, we project x and y on the current line 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*. used the sin and cos values calculated before to get the advancement.

4.3.2 Extract patches

To create the patches, we first discover the maximum advancement for one second by running some simulations of *Krock* on flat ground and averaging the advancement. For our robot, the maximum speed is $33cm/s$ Then, we multiply the maximum displacement by the number of seconds we are interested in. We can now crop the corresponding region in the heightmap by including the whole *Krock*'s footprint and the maximum advancement. The following figure visualizes the patch extraction process.

figure that shows Krock somewhere with the patch bounding boxed

. 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 the patches.

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

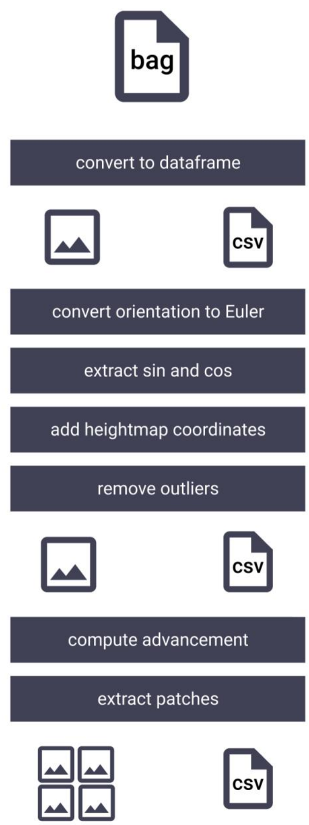


Figure 9: Postprocessing pipeline flow from top to bottom

add arrows!

Here on in interpretability???

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 on this time window.

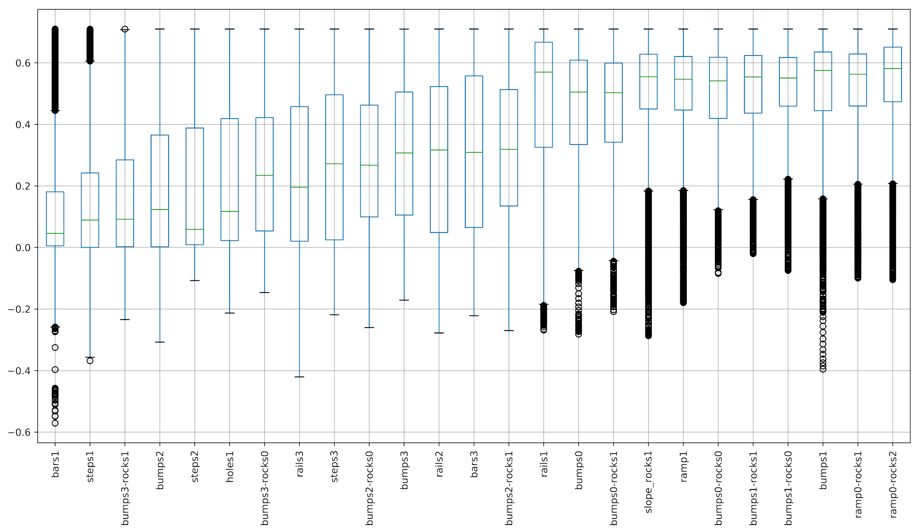
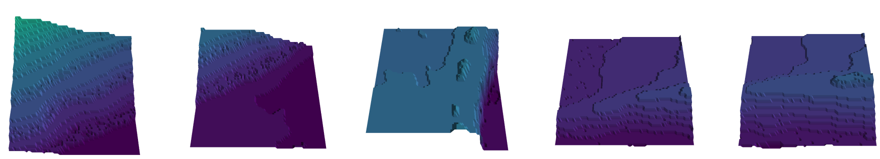
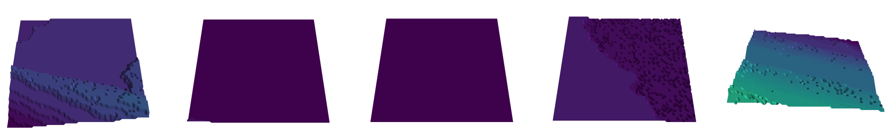


Figure 10: Advancement on each map with a time window of 2s in ascendent order.

The following picture shows some traversable an not traversable patches in 3d sorted by advancement taken directly from the *Quarry* dataset to help the reader visualize better the generated dataset.

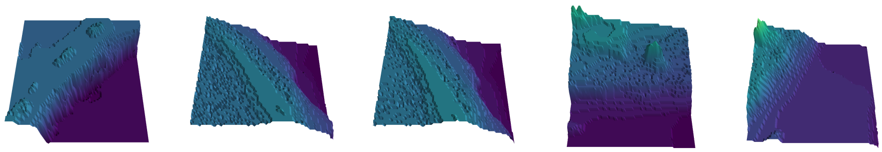


(a) 35cm (b) 38cm (c) 44cm (d) 45cm (e) 49cm

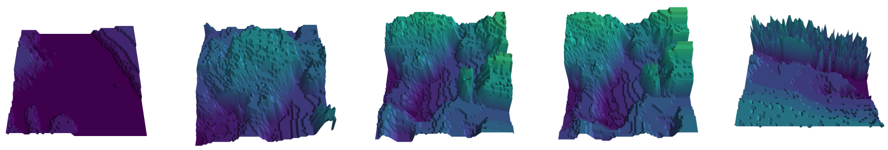


(f) 51cm (g) 65cm (h) 67cm (i) 67cm (j) 72cm

Figure 11: Traversable patches from *Quarry*



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



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

Figure 12: Non traversable patches from *Quarry*

4.4 Estimator

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

4.4.1 Vanilla Model

4.4.2 ResNet

We decide to use a Residual Network, ResNet [7], 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 ̈is composed by a two 3x3 *Convolution*, *Batchnorm* [10] and a *Relu* blocks. Formally, it is defined as:

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

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

add resnet image or table

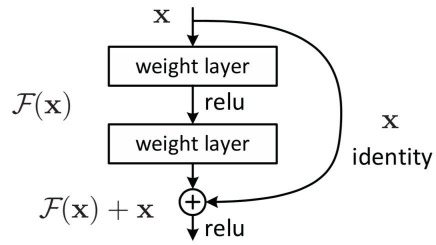


Figure 13: *Resnet* block [7]

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.

4.4.3 Preactivation

Following the recent work of He et al. [8] we adopt *pre-activation* in each block.*Pre-activation* works by just reversing the order of the operations in a block.

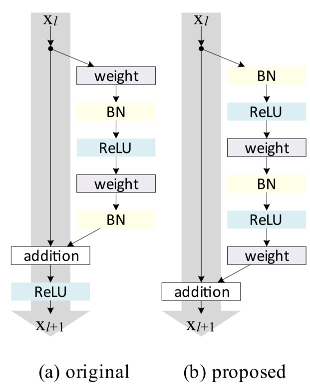


Figure 14: *Preactivation* [8]

4.4.4 Sqeeze and Excitation

Finally, we also used the *Squeeze and Excitation* (SE) module [9]. 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.

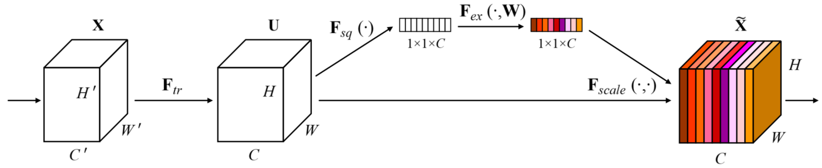


Figure 15: *Squeeze and Excitation* [9]

4.4.5 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 with a negative slope of 0.1 instead of ReLU, defined as following. We called this model architecture *micro-resnet*.

ref to Resnet Table

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.

Input	$(1, 79, 79)$			
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

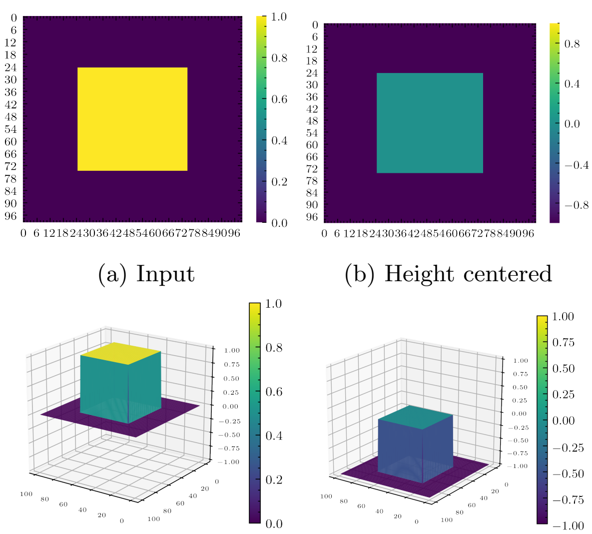
Table 2: Micro-resnet architecture

To simplicity we use the following notation to describe each architecture variant: `micro-resnet-/se-3x3/7x7` ask omar help to add margin in the rows

add model picture

4.4.6 Normalization

Before feeding the data to the models, we need to make the patches height invariant. This must be done to correctly normalize different patches taken from different maps with different height scaling factor. 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 16: Normalization process

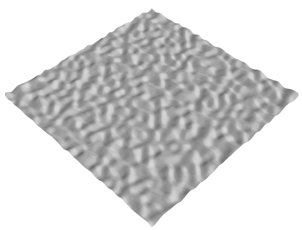
4.4.7 Data Augmentation

Data augmentation is used to change the input of a model using different techniques to change it 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 **??**.

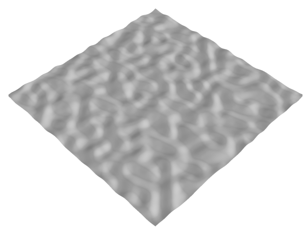
Dropout is a technique to randomly set some pixels to zero, in our case we flat some random pixel in the patch.

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

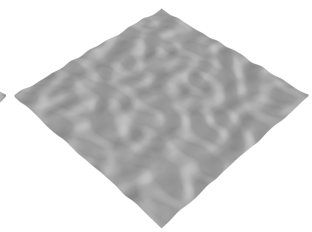
Simplex Noise is a form of Perlin noise that is mostly used in ground generation. Our idea is to add some noise to make the network generalize better since lots of training maps have only obstacles in flat ground. Since it is computationally expensive, we randomly fist apply the noise to five hundred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



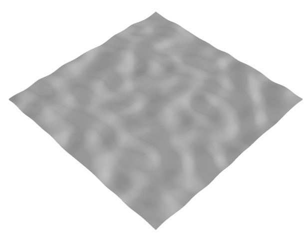
(a) Features size = 10



(b) Features size = 20



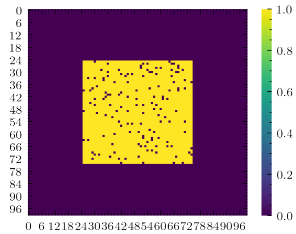
(c) Features size = 30



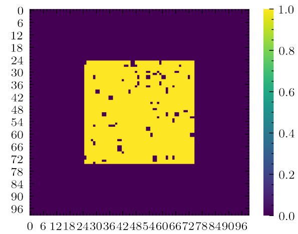
(d) Features size = 40

Figure 17: Simplex Noise on flat ground

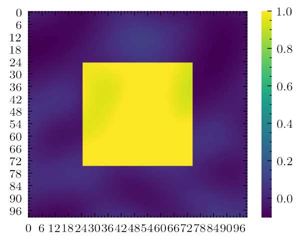
The following images show the tree data augmentation techniques used applied the input image.



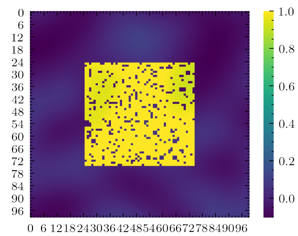
(a) Dropout



(b) Coarse Dropout



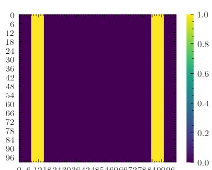
(c) Simplex Noise



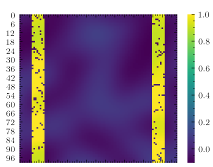
(d) Final result

Figure 18: Data augmentation

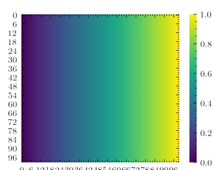
It follows an other set of figures that shows the data augmentation applyed on different inputs.



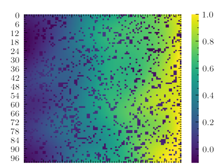
(a) Walls



(b) Walls aug



(c) Ramp



(d) Ramp aug

Figure 19: Wall

In all the traning epochs, we apply data-augmentation to 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.

5 Results

In the section we show and evaluate the models results. We will start by presenting to the reader the networks score on each metric, then we will use the best models to predict the traversability of real world terrain. Finally, we will use handcrafted patches, for example a wall of a certain height in a specific position, to test the robustness of the network by trying to highlighting its behaviour.

5.1 Experiment Setup

5.1.1 Hardware

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

5.1.2 Dataset

For the classification task, we select a threshold of $0.2m$ on a time window 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. We minimise the binary Cross Entropy.

On the other hand, for regression, we did not label the patch and directly regress on the advancement for a given time window while minimising the Mean Square Error (MSE). To train the network we follow the best practice on residual network [7] using Standard Gradient Descent with momentum set to 0.95 and weight decay to $1e{-}4$ with an initial learning rate of $1e{-}3$. We fix the maximum number of train epochs to 30 and reduce the learning rage on platuoe by a factor of 0.2 with a patience of 4. We used early stopping to stop the training if the validation accuracy does not increase in 6 epochs. fix this typo

I am actually using acycling learning or something like it I don't rememeber the paper's name

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 [7]. However, we later utilize Leslie Smith's 1cycle policy [15] that allows us to trian the network faster and with an higher accuracy.

5.1.3 Experimental validation

We select as *validation* ten percent of the training data. We remain to the reader that we store each run of *Krock* as a *.csv* file. So, to avoid any biases, we used completely different dataframes, meaning that train and validation sets are composed by non overlapping data from the simulations.

add more maps if we add them to the test set

5.1.4 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.

5.2 Quantitative Results

The following table shows the final results on various test dataset made by using real-world heightmaps. We select *micro-resnet* with squeeze and excitation and a starting convolution's kernel size of 7×7 with stride of 2.

maybe add model comparison

Dataset			micro-resnet		Size	Resolution(cm/px)
Type	Name	Samples	ACC	AUC		
Synthetic	Training	429312	-	-		2
	Valdiation	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 3: Todo

I have actually never talk about surf rocks

Moreover, 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.

add result with and without data agu

Adding dropout increases the results.

table with results

With dropout plus coarse dropout.

table with results

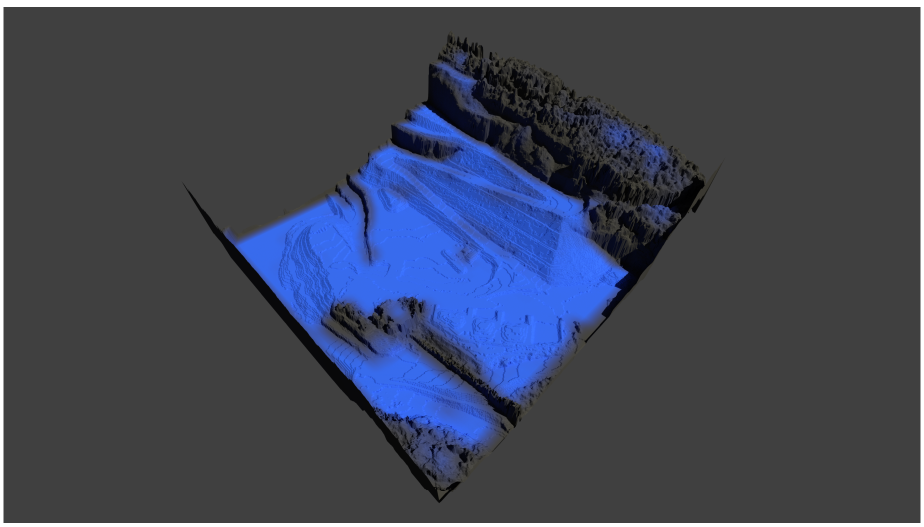
5.3 Qualitative results

THIS PLOT IS OLD!

We qualitative evaluate the models in real world scenarios by computing the traversability probability for each map with different rotation. Specifically, we used a sliding window to extract the patches from the heightmaps and colour by blue the relative region with the corresponding traversability probability. A brighter colour yields an higher probability. 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. We will start by showing the traversability probability on the *Quarry* assuming *Krock* is walking from bottom to top.

add quarry textures from bottom to top

Thanks its special locomotion, *Krock* can traverse the big slopes in the top part of them map while obliously it is stock by big bumps near the bottom as show in the next figures.



add figure of krock traversing the big slopes and getting stop near the bottom

To convince the reader that those slopes can be traverse, we run *Krock* on them directly from the simulator.

image of one extracted patch from quarry and one run on the simulator

6 Interpretability

z

References

[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] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[5] R. Omar Chavez-Garcia et al. “Learning Ground Traversability from Simulations”. In: (2017). DOI: `10.1109/LRA.2018.2801794`. eprint: `arXiv:1709.05368`.

[6] Jeffrey Delmerico et al. “On-the-spot Training” for Terrain Classification in Autonomous Air-Ground Collaborative Teams”. In: (2017).

[7] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. eprint: `arXiv:1512.03385`.

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

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

[10] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: `arXiv:1502.03167`.

[11] Julia Nitsch Jeffrey Delmerico Elias Mueggler and Davide Scaramuzza. “Active Autonomous Aerial Exploration for Ground Robot Path Planning”. In: (2016).

[12] Tobias Klamt and Sven Behnke. “Anytime Hybrid Driving-Stepping Locomotion Planning”. In: (2017).

[13] M. Bloesch L. Wagner P. Fankhauser and M. Hutter. “Foot Contact Estimation for Legged Robots in Rough Terrain.” In: (2016).

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

[15] 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`.

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

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

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