1 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.

1.1 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.

1.1.1 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.

1.1.2 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.

1.1.3 Matplotlib

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

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

1.1.5 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.

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

1.1.7 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.

1.1.8 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 1.4.7

1.1.9 Blender

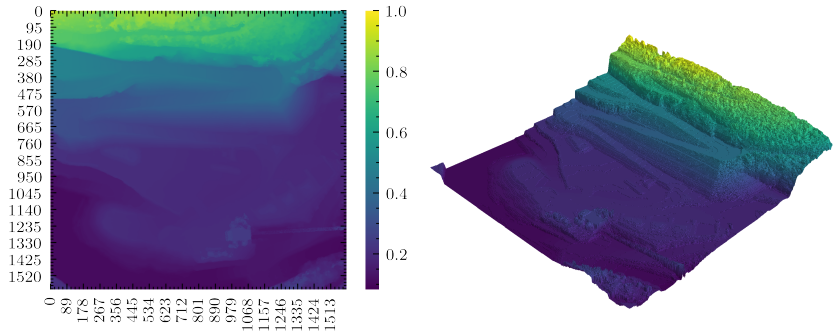
Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation. We used Blender to render some of the 3D terrain utilized to evaluate the trained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

1.2 Data Gathering

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

1.2.1 Heightmap generation

To collect the data throught simulation we first need to generate meaningful terrain to be explored by the robot. Each map 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.

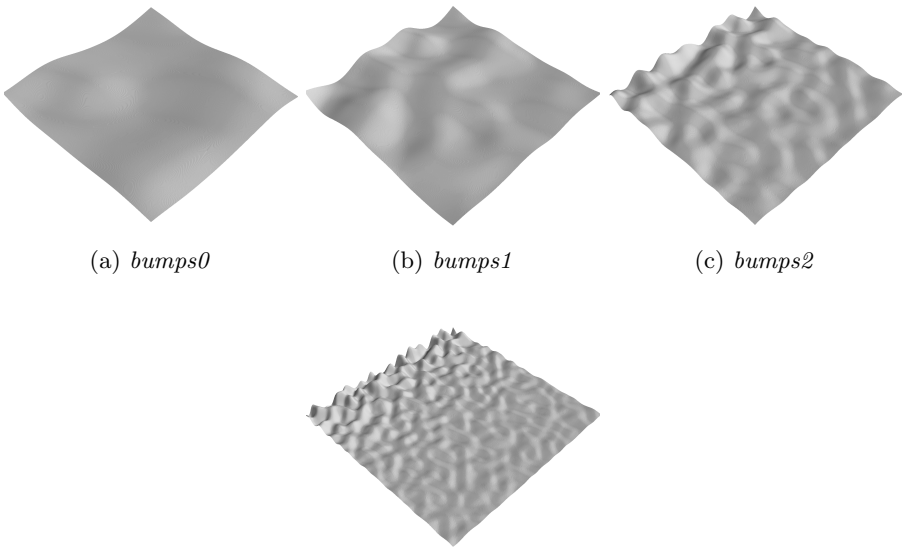


(a) *Quarry* heightmap (b) *Quarry* map

Figure 1: A heightmap

We generated thirty maps of 513×513 pixel with a resolution of $0.02 cm/pixel$ in order to represent a $10 \times 10 m$ region. We created them by 2D simplex noise [**simplex**], a variant of Perlin noise [**perlin**], a widely used technique in the terrain generation litterature. We created 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.

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

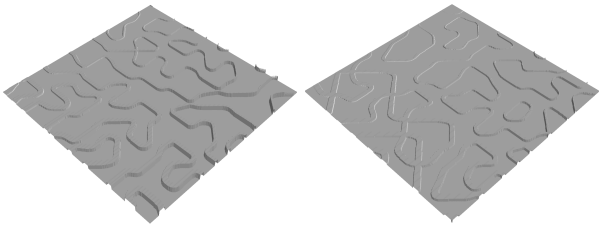


(a) *bumps0* (b) *bumps1* (c) *bumps2*

(d) *bumps3*

Figure 2: Bumps maps (10×10m).

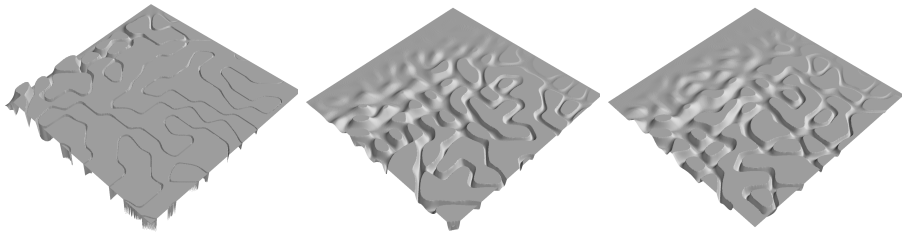
Bars: In these maps there are wall with different shapes and heights. In



(a) *bars1* (b) *bars3*

Figure 3: Bars maps (10×10m).

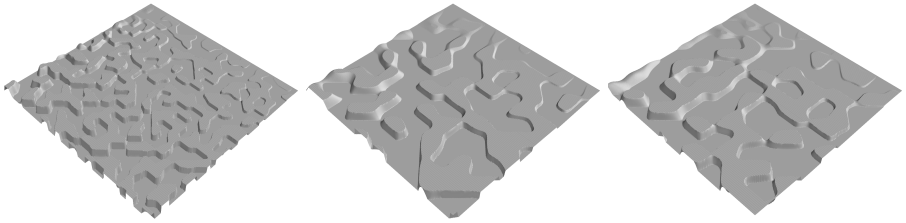
Rails: Flat grounds with slots.



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

Figure 4: Rails maps (10×10m).

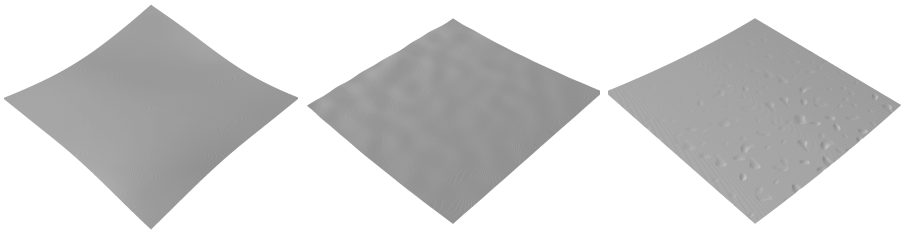
Steps: These are maps have huge wall and holes



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

Figure 5: Steps maps (10×10m).

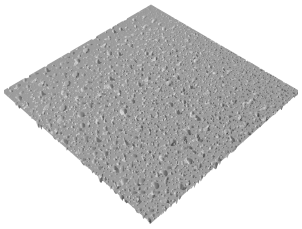
Slopes/Ramps: Maps composed by uneven terrain scaled by different height factors from 3 to 5 used to include samples where *Krock* has to climb. Those maps were scaled up with different height factors.



(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).

1.2.2 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 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.

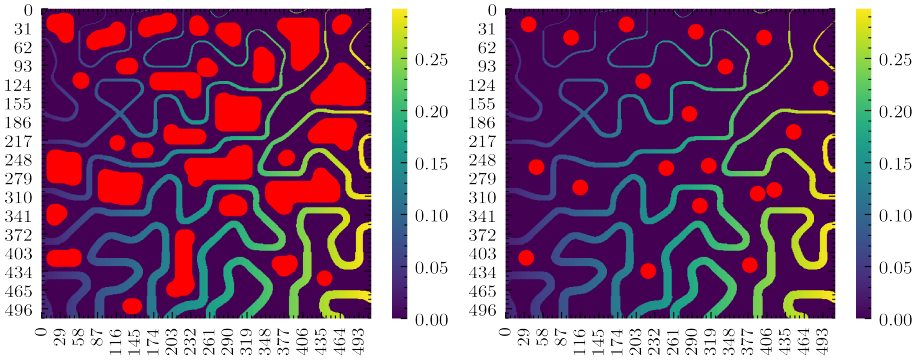
We also implement one additional library called `Agent` to create reusable robot's interfaces independent from the simulator. The package supports callbacks that can be attached to each agent adding additional features. Finally, we used *Gym* [**gym**], a toolkit to develop and evaluate reinforcement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

1.2.3 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*, 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 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 spawn the robot on random position and rotation. While, 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 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 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	random	- rocks1 rocks2	50	10
	2		-		
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	flat	-	50	10
	2		-		
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.

1.3 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.

1.3.1 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 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 booth its legs and does not flat the advancement too much.

1.3.2 Extract patches

Each patch must contain booth 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.

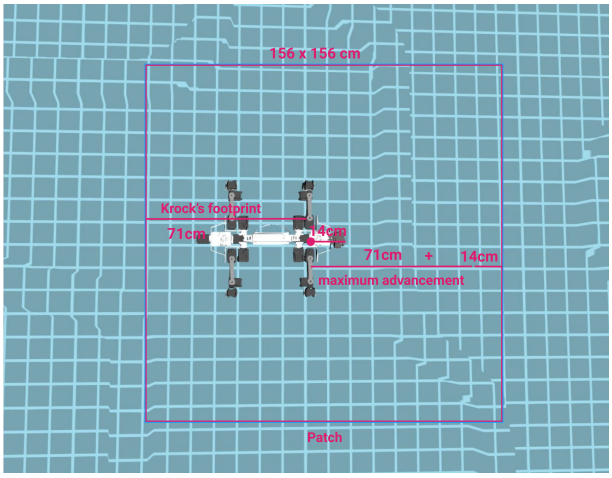


Figure 9: Patch extraction process for $\Delta t = 2$s.

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.

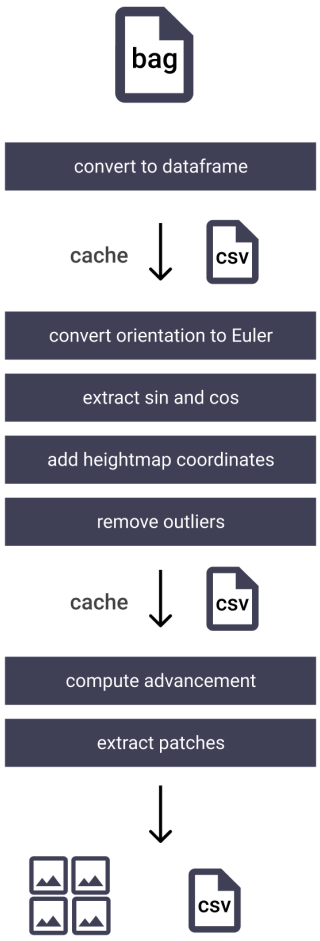


Figure 10: Postprocessing Pipeline flow graph, starting from the top.

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.

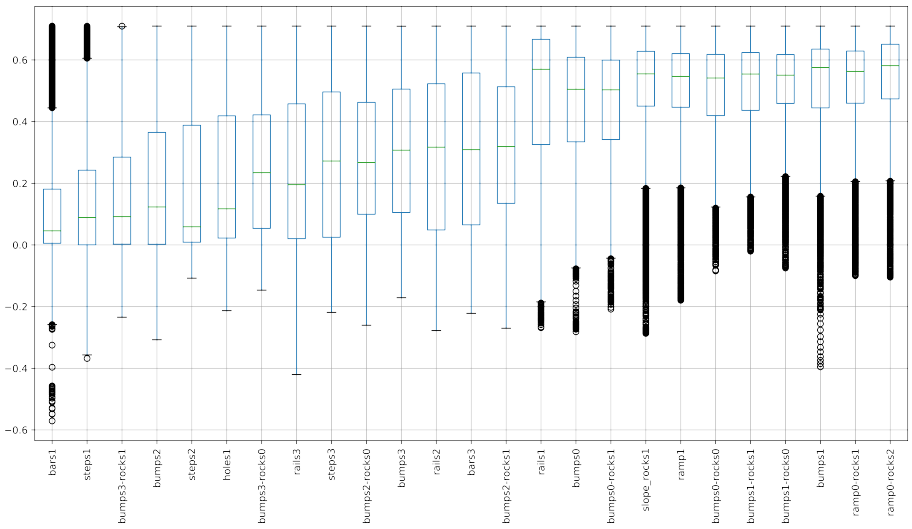
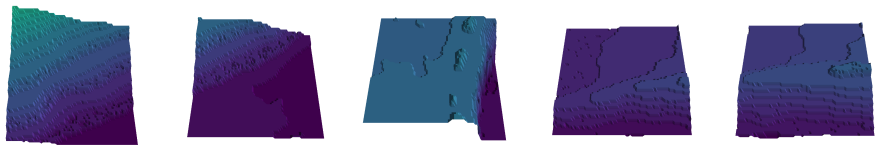


Figure 11: Advancement on each map with a $\Delta t = 2$s in ascendent order.

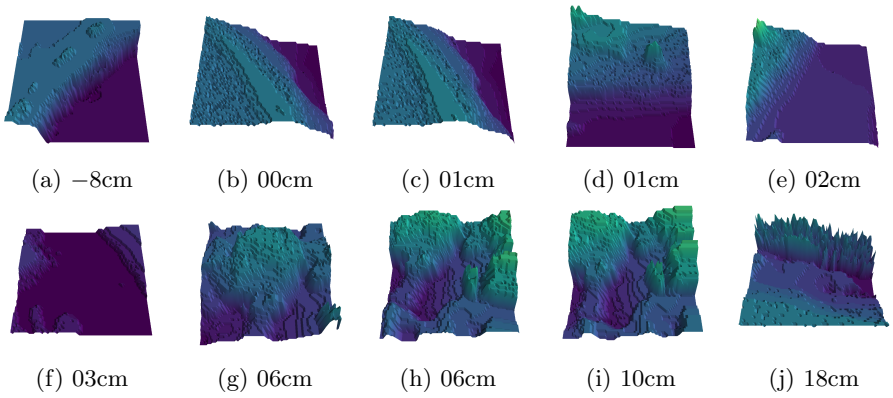
To give the reader a better idea of how the patches looks like, the next figures shows the 3d render of some patches extracted form *Quarry* ordered by advancement.



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

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

Figure 12: 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 13: 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.

1.3.3 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.

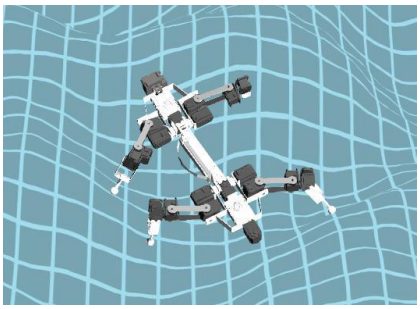
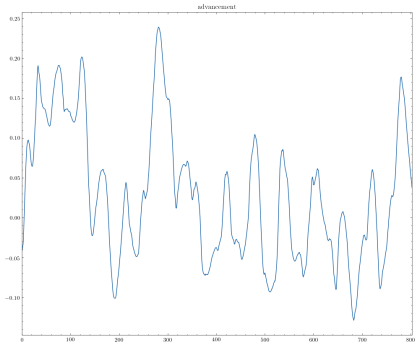


Figure 14: Krock tries to overcome an obstacle in the *bumps3* map.

Due to its characteristic locomotion, Krock tries to first overcome the obstacle using the legs to move itself to the top and then 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.

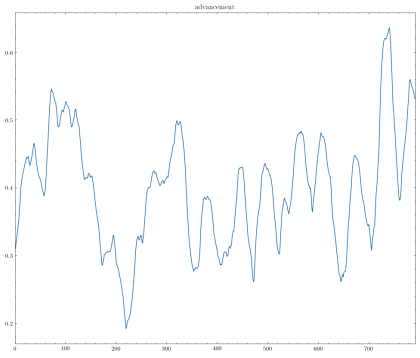


(a) $\Delta t = 2$s.

Figure 15: 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 16: 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 $tr_{\Delta t=2s} = 20cm$.

1.4 Estimator

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

1.4.1 Vanilla Model

The original model proposed by Chavez-Garcia et all [**omar2018traversability**] 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.

1.4.2 MicroResNet

We adopt a Residual network, ResNet [**he2015deep**], 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* [**ioffe2015batch**] 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

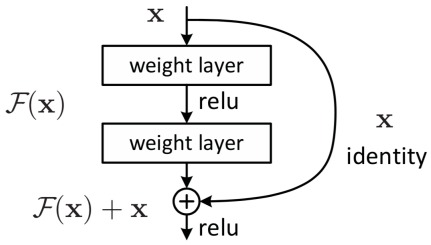


Figure 17: *Resnet* block [**he2015deep**]

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.

1.4.3 Preactivation

Following the recent work of He et al. [**he2015identity**] we adopt *pre-activation* in each block.*Pre-activation* works by just reversing the order of the operations in a block.

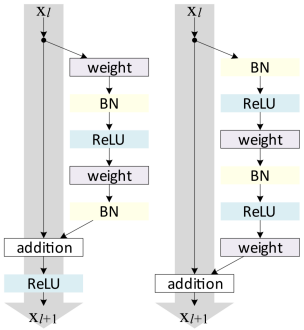


Figure 18: *Preactivation* [**he2015identity**]

1.4.4 Sqeeze and Excitation

Finally, we also used the *Squeeze and Excitation* (SE) module [**hu2017squeeze**]. 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.

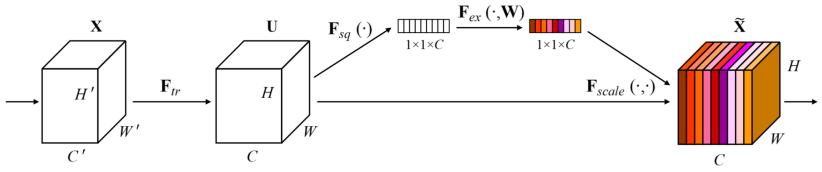


Figure 19: *Squeeze and Excitation* [**hu2017squeeze**]

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

ref to Resnet Table

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

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

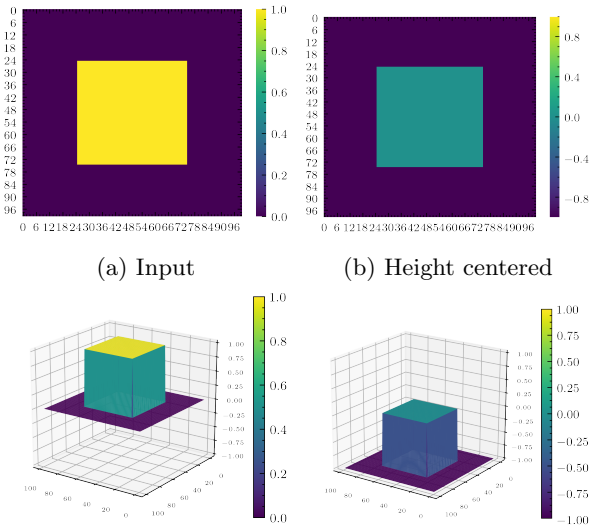
Our models have approximately 35 times less parameters than the smalles ResNet model, ResNet18, that has 11M parameters. To simplicity we will use the following notation to describe each architecture variant: `MicroResNet-3x3/7x7-/SE`.

ask omar help to add margin in the rows

add model picture

1.4.6 Normalization

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



(a) Input (b) Height centered

(c) Input (d) Height centered

Figure 20: Normalization process

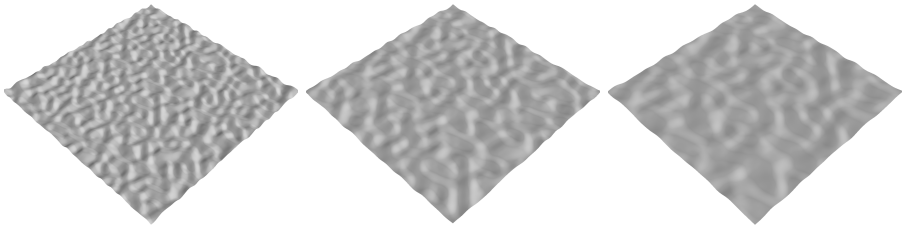
1.4.7 Data Augmentation

Data augmentation is used to change the input of a model in order to produce more training examples. Since our inputs are heightmaps we cannot utilize the classic image manipulations such as shifts, flips, and zooms. Imagine that we have a patch with a wall in front of it, if we random rotate the image the wall may go in a position where the wall is not facing the robot anymore, making the image now traversable with a wrong target. We decided to apply dropout, coarse dropout, and random simplex noise since they are traversability invariant. To illustrate those techniques we are going to use the same square patch showed before 20.

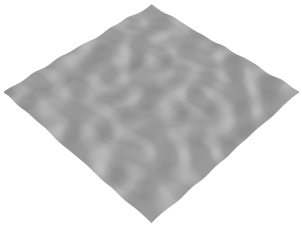
Dropout is a technique to randomly set some pixels to zero, in our case we flat some random pixel in the patch.

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.



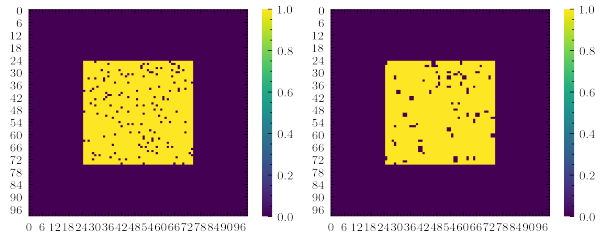
(a) Features size = 10 (b) Features size = 20 (c) Features size = 30



(d) Features size = 40

Figure 21: Simplex Noise on flat ground

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

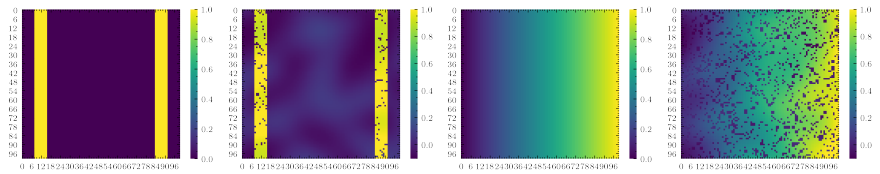


(a) Dropout (b) Coarse Dropout

(c) Simplex Noise (d) Final result

Figure 22: 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 23: 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.