

# Chapter 1

## Implementation

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

### 1.1 Robot

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

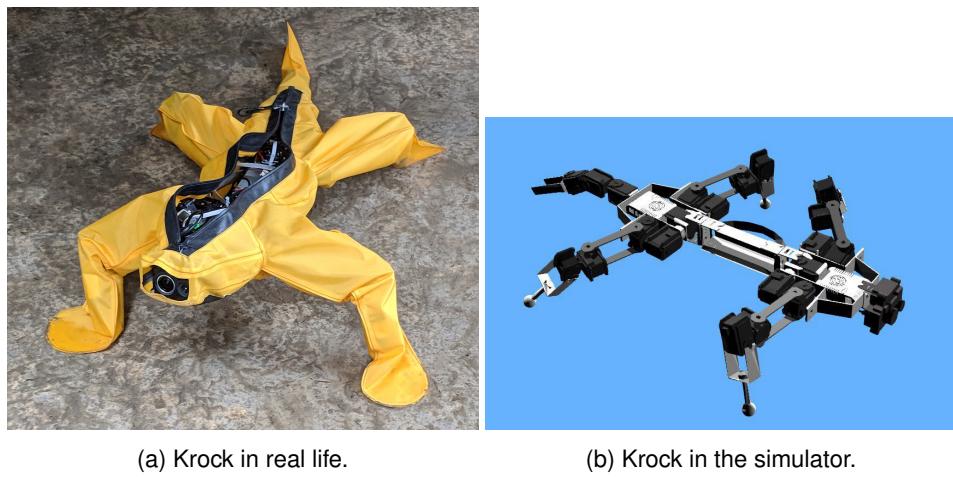


Figure 1.1. *Krock*

K-rock robot was created for the purpose of monitoring real crocodiles, hence it must walk and behave realistically enough to fool the real crocodiles. Krock has four legs, each one of them is equipped with three motors in order to rotate in each axis. The robot can raise itself in three different configurations, gaits, using the four motors on the body connected to the legs. In addition, there is another set of two motors in the torso to increase Krock's mobility. The tail is composed by another set of three motors and can be used to perform a wide array of tasks. The robot is 85cm

long, weights around  $1.4kg$  and in its standard gait configuration its distance from the ground is 16cm. Figure 1.2 shows a topview of the robot. *Krock*'s moves by lifting and moving forward one

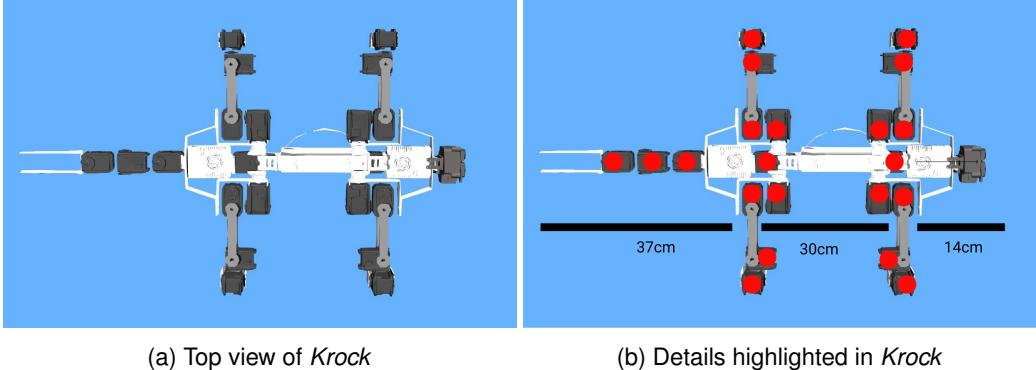


Figure 1.2. Top view of *Krock* to help reader understanding its composition and the correct ratio between its parts. Each motor is highlighted with a red marker.

leg after the other. The following figure shows the robots going forward. In our framework we fixed

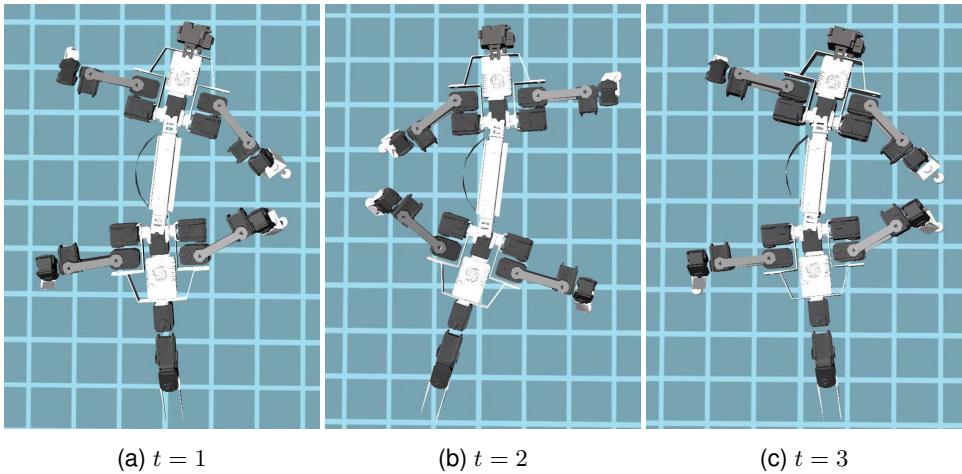


Figure 1.3. *Krock* moving forward. In this gait configuration, the robot distance from the ground is 16cm.

the gait configuration to normal, showed in figure 1.1, where the body is at the same legs' motors height.

## 1.2 Data Gathering

This section describes in detail how we created, collected and processed the synthetic dataset used to train the traversability estimator in supervised way.

### 1.2.1 Ground generation

We created thirty  $10 \times 10$  maps with a resolution offset  $0.02\text{cm/pixel}$  and a height of 1m using 2D simplex noise variant of Perlin noise ?, a widely used technique in the terrain generation litterature. We divide the maps into five main categories based on ground features: *bumps, rails, steps, slopes/ramps* and *holes*.

**Bumps:** We generated four different maps with increasing bumps' height using simplex noise.

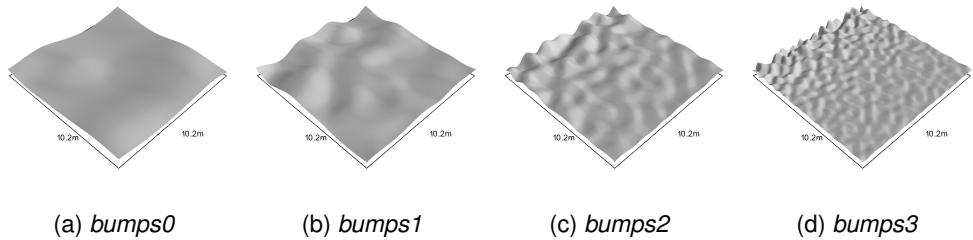


Figure 1.4. Bumps maps.

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

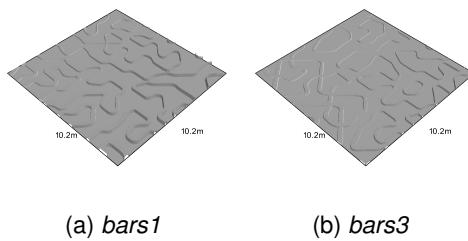


Figure 1.5. Bars maps.

**Rails:** Flat grounds with slots.

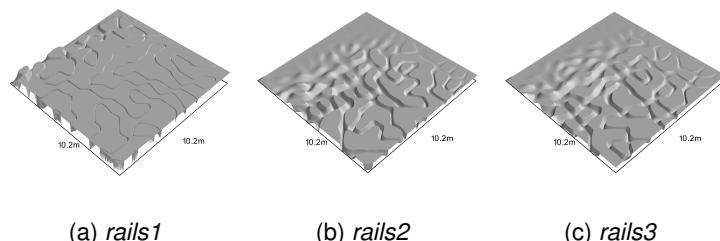


Figure 1.6. Rails maps.

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

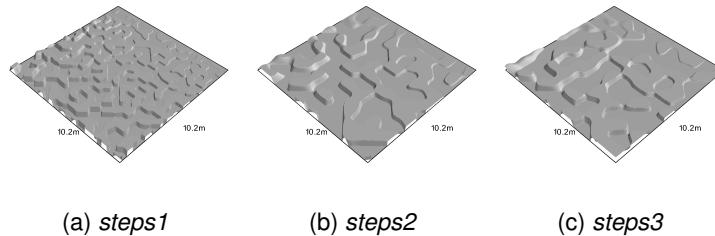


Figure 1.7. Steps maps.

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

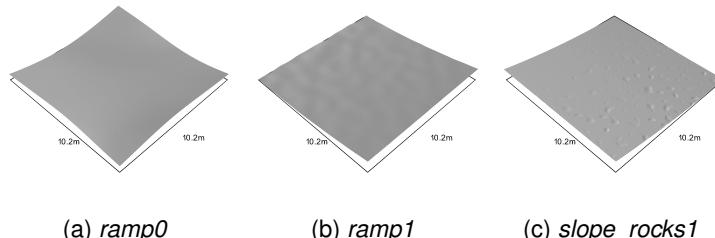


Figure 1.8. Slopes maps.

**Holes** We also included a map with holes

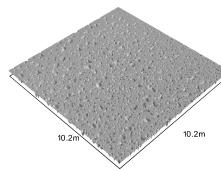
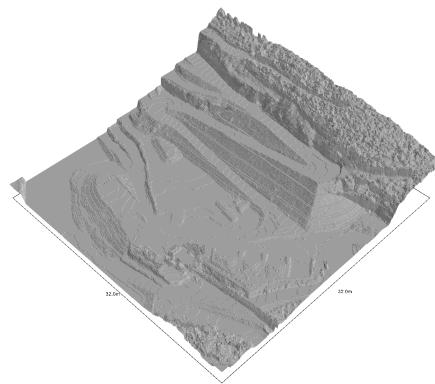


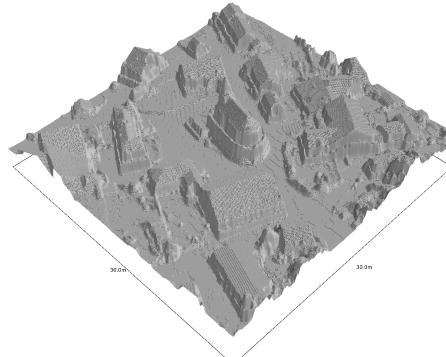
Figure 1.9. Holes map.

## 1.2.2 Real world maps

We also included real word terrains. We gather two heightmaps produced by ground mapping flying drones from sensefly's dataset, a quarry and a small village. Figure 1.10 shows the original and a 3D render of the heightmap for each terrain.



(a) Quarry



(b) A small village.

Figure 1.10. Real world maps obtained from sensefly's dataset. The left images shows the real world location while on the right a render in 3D of the coresponding heightmaps. both images have a maximum height of 10m.

## 1.3 Simulation

### 1.3.1 Setup

We used Webots as the simulation platform for the robot model and the generated terrains. The controller was implemented using the Robotic Operating System, it sends data using a special nodes called topics. The controller implements a ROS' node to publish Krock status including its pose at a rate of 250hz. We decided to reduce it to 50hz by using ROS build it `throttle` command.

### 1.3.2 Spawning

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 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 strategy just spawn the robot on random position and rotation. 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 as a candidate spawning point. Intuitively, if a patch is flat then its mean value will be close to zero. Since there may be more flat spawning positions than simulations needed, we have to reduce the size of the candidate points. To maintain the correct distribution on the map to avoid spawning the robot always in the same cloud of points, we used K-Means with  $k$  clusters where  $k$  is equal to the number of simulations we wish to run. By clustering, we guarantee to cover all region of the map removing any bias. Picture ?? shows this strategy on *bars1*.

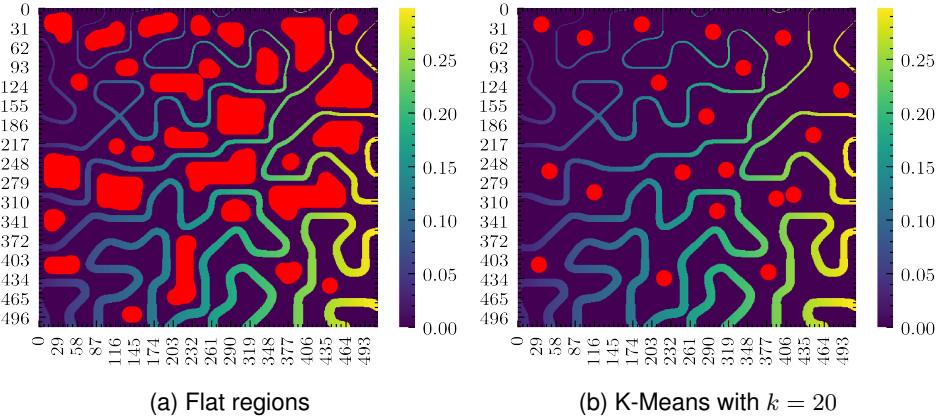


Figure 1.11. Examples of the spawning selection process (marked as red blobs) for the map *bars1*

### 1.3.3 Interactions

In each simulation, the robot is spawned using the two spawning strategies and it walks forward for a fixed amount of time. We selected ten or twenty seconds depending on the map. Every five simulations we completely regenerate the robot's body to counter possible damages during the hard traversable ground. We moved the robot only forward with a fixed controller since we cannot estimate if the robot can traverse a patch while moving sideways. Table 1.1 shows the maps configuration used in the simulator. For some map we add three different rocky textures to create more complicated situations. For example, adding some rocks in the ramps map allowed the robot to anchor the legs to the rocks and climb them easily.

## 1.4 Postprocessing

To generate the dataset, we need to extract we had to extract the patches at each robot's pose of the trajectory generated at simulation time. Figure ?? shows each step in the postprocess pipeline.

### 1.4.1 Parse trajectories

First, we converted the robot's trajectories, stored as `.bag` files, to a more convenient data structure, pandas' dataframes. Then, we cache them into `.csv` files. This operation is done only once. After, we loaded the stored dataframes with the respective terrains and started the data cleaning process. We removed the first second of the simulation time to account for the robot spawning time. Then, we eliminated all the entries where a part of Krock was outside the edges of the map. After the data cleaning process, we converted quaternion rotation to Euler notation using the `tf` package from ROS. Then, we extracted the sin and cos from the Euler orientation last component and stored them in a column. In the simulator, the position (0, 0) correspond to the center of the map, while on the heightmap, like all images, (0, 0) is the top left corner. To later crop the the correct region from the map, we needed to convert the robot's position into the heightmap's coordinates. This part is also run once and it is cached to speed up latter runs.

### 1.4.2 Advancement computation

To compute the robot's advancement we needed to define a time window,  $\Delta t$ , and consider how much the robot traveled between the current pose,  $X(t)$  and the future one  $X(t+\Delta t)$ . We observed from the simulator than the robot moves each leg every second. Thus, we selected a time window of two seconds to allow Krock to move both legs.

### 1.4.3 Patch extraction

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

To find out the correct value, we must compute the maximum advancement on a flat ground for the  $\Delta t$  and use it to calculate the final size of the patch. We compute it by running some simulations of *Krock* on flat ground and averaging the advancement getting a value of 70cm in our  $\Delta t = 2\text{s}$ .

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

Table 1.1. Maps configuration used in the simulator.

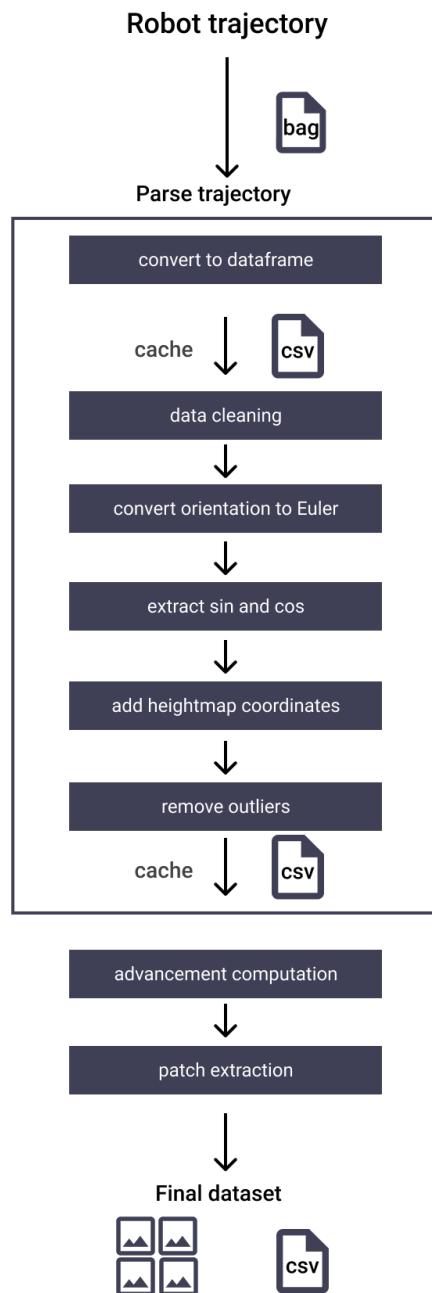
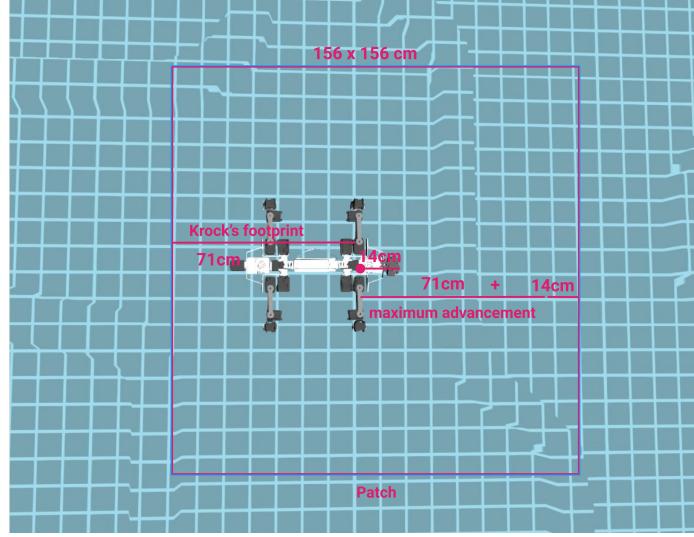
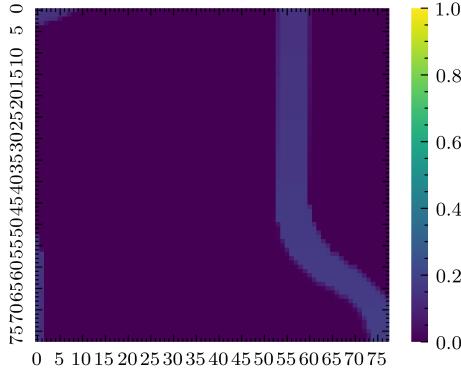


Figure 1.12. Postprocessing pipeline flow graph, starting from the top.

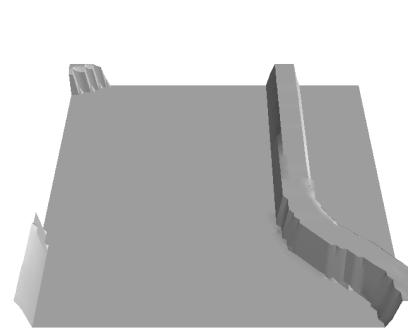
Each patch must include Krock's footprint and the maximum possible distance it can travel is a  $\Delta t$ . So, since Krock's pose was stored from IMU located in the juncture between the head and the legs, we have to crop from behind its length, 85cm minus the offset between the IMU and the head, 14cm. Then, we have to take 70cm, the maximum advancement with a  $\Delta t = 2\text{s}$  plus the removed offset. The following figure visualizes the patch extraction process. Lastly, we



(a) Robot in the simulator.



(b) Cropped patch in 2d.



(c) Cropped patch in 3d.

Figure 1.13. Patch extraction process for  $\Delta t = 2\text{s}$ .

create a final dataframe containing the map coordinates, the advancement, and the patches paths per simulation and store them to disk as .csv files.

#### 1.4.4 Final dataset

The final dataset is composed by  $\approx 470\text{K}$  and  $\approx 37\text{K}$  patches for the train and test set respectively. A small part of the train set, 10% is used as validation. Train and validation set's trajectories do not

overlap, each set has entirely independent simulation's runs

We also generated  $\approx 38K$  patches using the *arc rocks* map to further evaluate the model. The whole pipeline took 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. For completeness, we plotted in figure 1.14 the mean advancement over each map in the training set. As sanity check, we visualized some of the

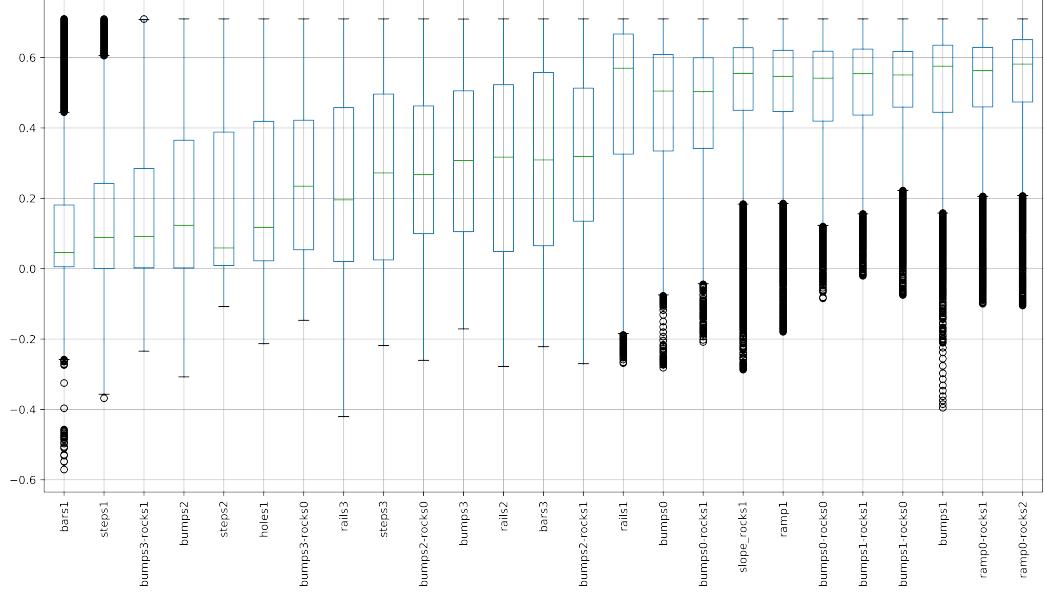


Figure 1.14. Advancement on each map with a  $\Delta t = 2s$  in ascendent order.

patches from the train set ordered by advancement, to conclude the data generation process was correct. Figure 1.16 shows a sample of forty patches.

#### 1.4.5 Normalization

Before feeding the data to the models, at runtime, we needed 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 to Krock's position from the patch to correctly center it. Figure ?? shows the normalization process on a dummy patch.

### 1.5 Label patches

To classify the patches we first need to label them. We defined a threshold to represent and label each patch traversable if the advancement is greater or not traversable if it is lower. Ideally, the threshold should be small enough to include as many false positives as possible and big enough to cover all the cases where Krock gets stuck. We empirically compute the threshold's value by spawning Krock in front of a bump 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. Figure 1.17 shows Krock in the simulated environment.

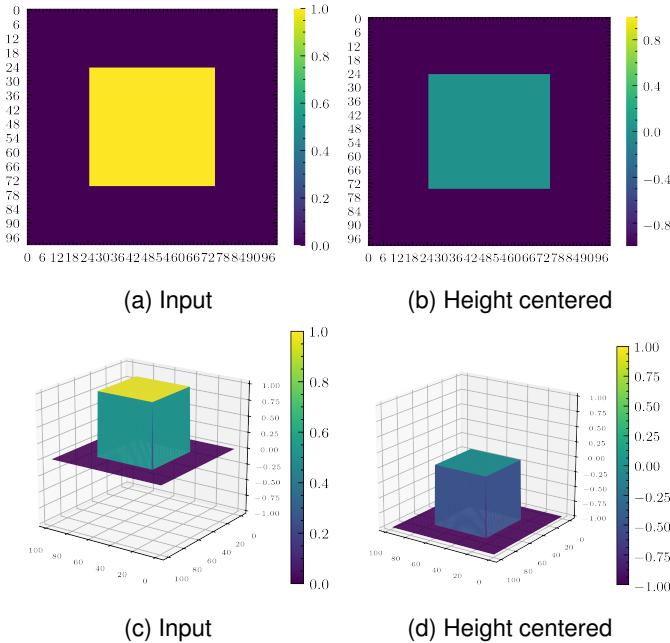


Figure 1.15. Normalization process. Each patch is normalized by subtracting the height value corresponding to the robot position to center its height.

Due to its characteristic locomotion, Krock tried to overcome the obstacle using its legs to move itself to the top but it fell backs producing a spicky advancement where first it is positive and then negative. Figure 1.18 shows the advancemetn on this map with  $\Delta t = 2$ , we can notice the noisy output.

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

**Ramps** The threshold should be small enough to not create any false positive, patches that are traversable but were classified as not. So, we let the robot walk up hill on the *slope\_rocks1* map with a height scaling factor set to 5. We knew from empirical experiment that the robot is able to climb this steep ramp, the advancement is showed in figure 1.19. The mean  $\approx 0.4\text{cm}$  over a  $\Delta t = 2$ , an impressive value considering the steepest of the surface. We used this information combined with the previous experiment to choose a threshold that is between the upper bound of *bumps* and between the lower bound of *slope\_rocks1*. This ensured to minimize the false positive. A good value is  $tr_{\Delta t=2s} = 20\text{cm}$ .

## 1.6 Estimator

In this section we described the convolutional neural network utilized in the original paper. We then introduced residual network and new techniques used to improve their performance. In the end, we proposed four residual network variant to be evaluated.

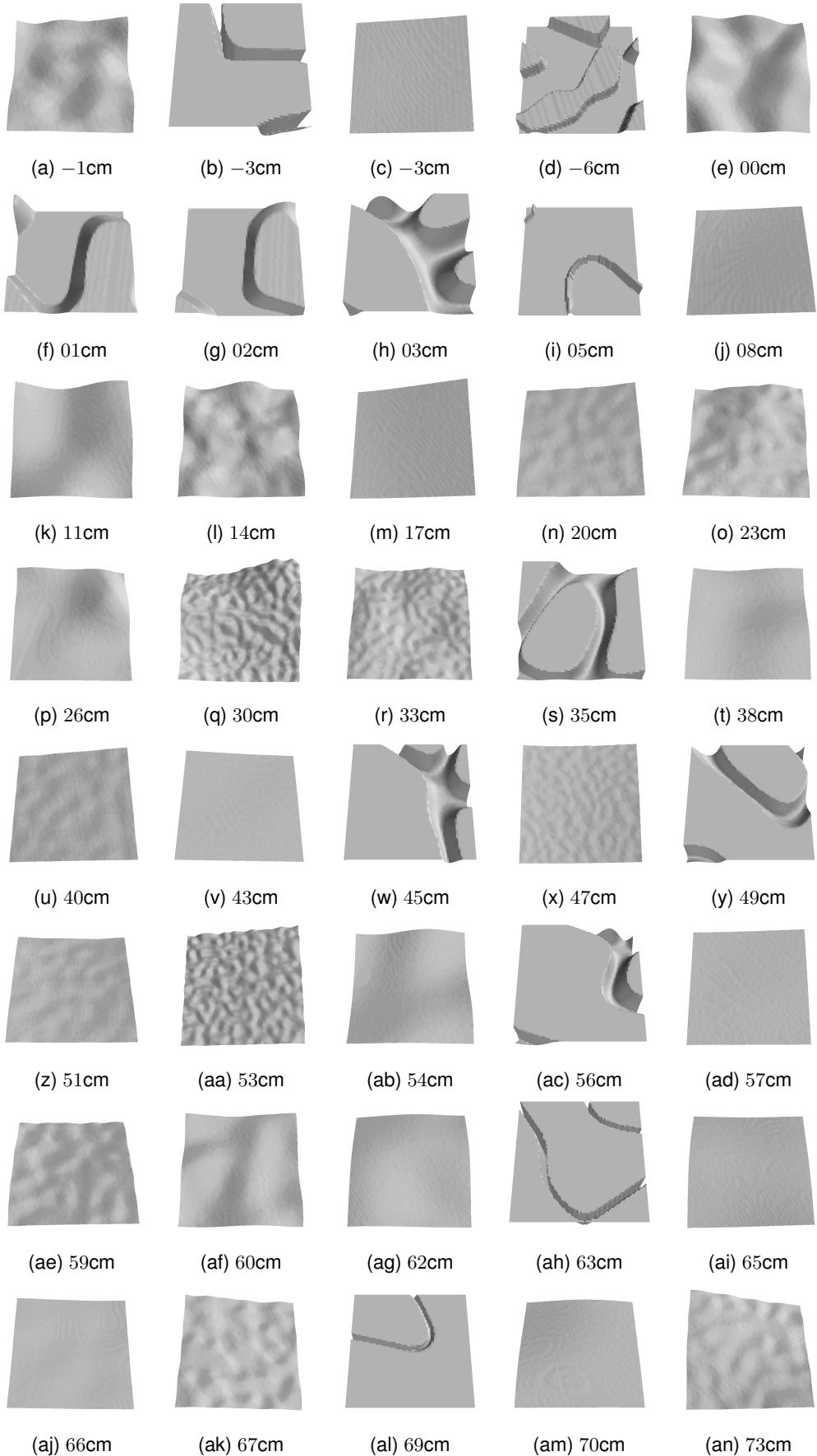


Figure 1.16. Sample of extracted patches from the train set ordered by advancement. The first patches have mostly obstacle close to the robot's head, while the latter have smoother surfaces.

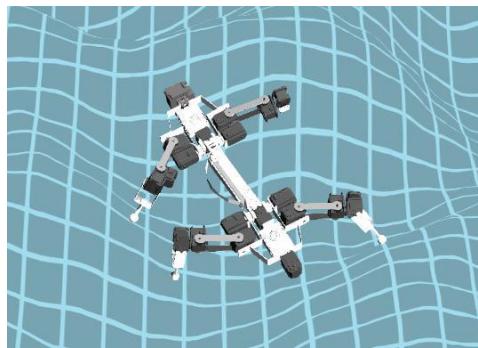


Figure 1.17. Krock tries to overcome an obstacle in the *bumps3* map.

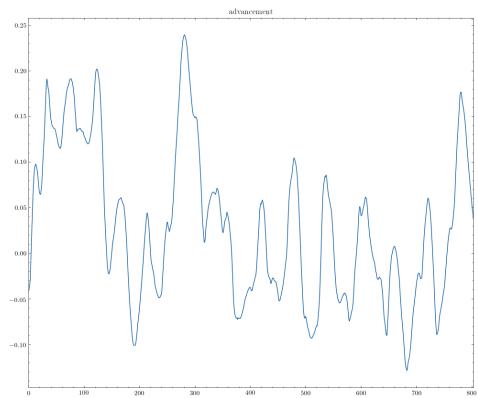


Figure 1.18. Advancement over time with  $\Delta t = 2$  on *bumps3*.

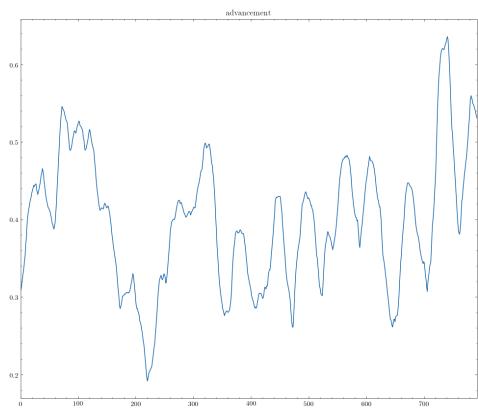


Figure 1.19. Advancement over time with  $\Delta t = 2s$  on *slope\_rocks3*.

### 1.6.1 Original Model

The original model proposed by Chavez-Garcia et al. ? is a CNN composed by a two  $3 \times 3$  convolution layer with 5 filters;  $2 \times 2$  Max-Pooling layer;  $3 \times 3$  convolution layer with 5 filters; a fully connected layer with 128 outputs neurons and a fully connected layers with two neurons. We considered this the baseline, Figure 1.20 visualizes the architecture.

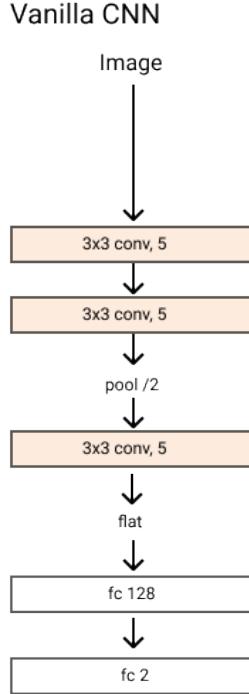


Figure 1.20. original model proposed by Chavez-Garcia et al. ?. The rectangle represent the building block of each layer. Convolutiol layers are described by the kernel size, the number of filters and the stride.

### 1.6.2 ResNet

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

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + h(\mathbf{x}) \quad (1.1)$$

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. When the input and output shapes mismatch, the *identity map* is applied to the input as

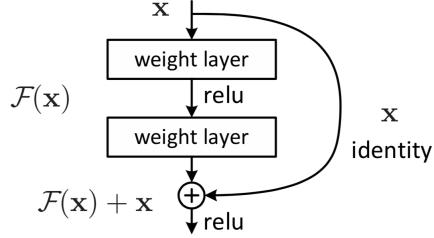


Figure 1.21. Resnet block ?

a  $3 \times 3$  Convolution with a stride of 2 to mimic the polling operator. A single block is composed by a  $3 \times 3$  Convolution, Batchnorm and a *Relu* activation function.

### 1.6.3 Preactivation

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

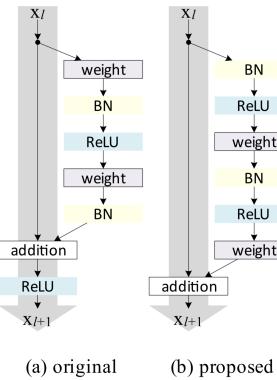


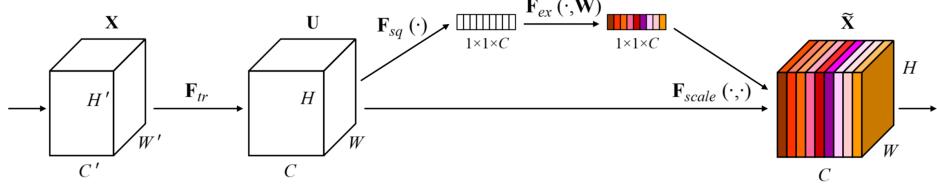
Figure 1.22. Preactivation ?

### 1.6.4 Squeeze and Excitation

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

### 1.6.5 MicroResNet

The proposed network's architecture is composed by  $n$  ResNet blocks and channel incrementing factor of 2. Originally, ResNet was used to classify RGB images with  $224 \times 224$  pixels and used an

Figure 1.23. *Squeeze and Excitation ?*

aggressive convolution and max-polling operation in the first layer to reduce the images's size. We decided to adopt a less aggressive convolution motived by the smaller size of our inputs. We tested two kernel sized:  $7 \times 7$  and  $3 \times 3$  with stride of 2 and 1 respectively. Lastly, we used LeakyReLU ? with a negative slope of 0.1 instead of ReLU to allow a better gradient flow during backpropagation. LeakyRelu is defined as follows

$$\text{LeakyRelu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1x & \text{otherwise} \end{cases} \quad (1.2)$$

We called this model architecture *MicroResNet*. We proposed four networks layouts: with  $n = 1$  and two fist layers setups,  $3 \times 3$  stride = 1 and  $7 \times 7$ , with and without the Squeeze and Excitation module. Those architectures can be used both for regression and classification by only chaning the output size of the fully connected later, 2 for classification and 1 for regression. All others layers are equals. Figure 1.24 add additional informations. Our models have

Input	(1, 78, 78)			
	$3 \times 3, 16$ stride 1 $7 \times 7$ 16 stride 2			
	2 $\times$ 2 max-pool			
	$\begin{bmatrix} 3 \times 3, & 16 \\ 3 \times 3, & 32 \end{bmatrix} \times 1$			
Layers	SE	-	SE	-
	$\begin{bmatrix} 3 \times 3, & 32 \\ 3 \times 3, & 64 \end{bmatrix} \times 1$			
	SE	-	SE	-
	$\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 128 \end{bmatrix} \times 1$			
	SE	-	SE	-
	average pool, 1-d fc, softmax			
Network Size (params)	313,642	302,610	314,28	303,250
Size (MB)	5.93	5.71	2.41	2.32

Table 1.2. MicroResNet architecture. First layer is on the top. Some architecture's blocks are equal across models, this is shows by sharing columns in the table.

approximately 35 times less parameters than the smalles ResNet model, ResNet18 ?, that has 11M

parameters. To simplicity we used the following notation to describe each architecture variant: MicroResNet- $3 \times 3 / 7 \times 7$ -/SE.

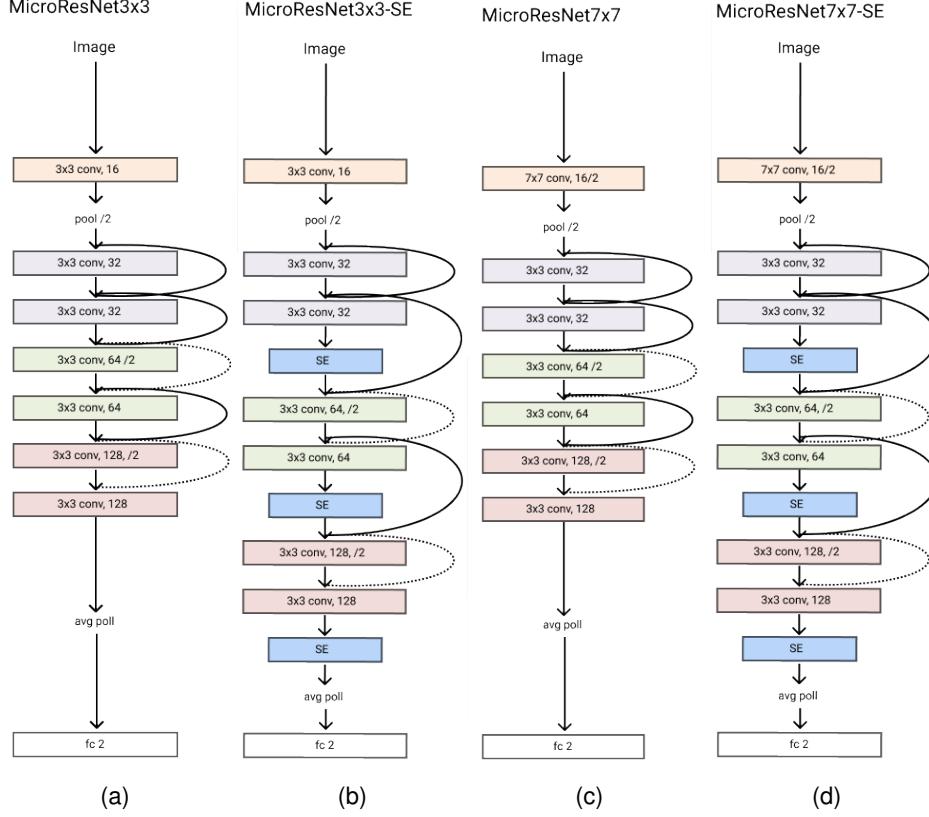


Figure 1.24. MicroResNet architectures for classification. All models have an initial features size of 16 and  $n = 1$  with and without the Squeeze and Excitation module. The rectangle represent the building block of each layer. Convolutiol layers are described by the kernel size, the number of filters and the stride. Lines and dashed lines between layers represent the residual operations and the shortcut respectively.

## 1.7 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 in the correct degree to make them traversability invariant. To illustrate those techniques we used the dummy patch same cude in the middle, figure 1.15.

### 1.7.1 Dropout

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

### 1.7.2 Coarse Dropout

Similar to dropout, it sets to zero random regions of pixels with defined boundaries. To ensure we do create untraversable patches from traversable patch we limited the coarse dropout region to only few centiments.

### 1.7.3 Simplex Noise

We added some noise to the inputs to slightly change the ground in order help the network generalize better. Since it is computationally expensive, we first randomly applied the noise to five hundred images with only zeros, flat grounds. We cached them and then we randomly scaled them and added to the input image. To ensure traversability invariant we applied simplex noise to the inputs based on their classes. For traversability grounds we used a noise feature size 15 – 25 while, for not traversable samples a noise size between 5 – 15. Figure 1.25 clearly shows the grounds generated by different sizes.

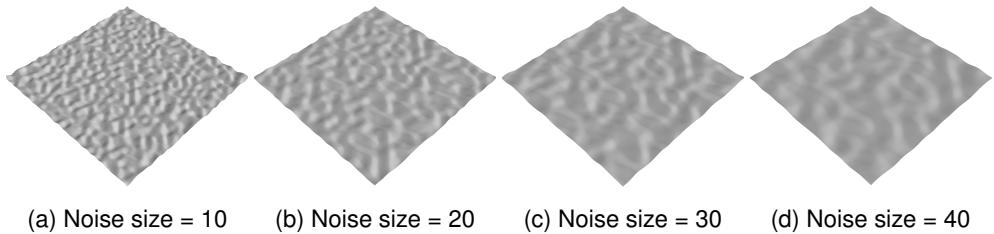


Figure 1.25. Simplex noise on flat ground. The feature size corresponds to the amount of noise we introduce in the surface. A lower value produces noisy grounds, while a bigger one smoother terrains.

Figure ?? shows the three data augmentation techniques used applied to the input image. We augmented 80% of the input images with dropout and coarse dropout. Dropout has a probability between 0.05 and 0.1 and coarse dropout has a probability of 0.02 and 0.1 with a size of the lower resolution image from which to sample the dropout between 0.6 and 0.8. Simplex noise was applied on the 70% of the training data samples.

## 1.8 Tools

We briefly list the most important tools and libraries adopted in this project. The framework was entirely developed on Ubuntu 18.10 with Python 3.6.

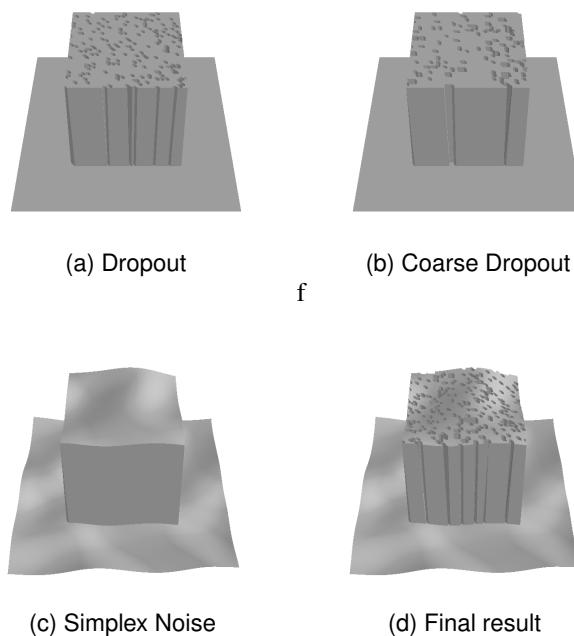


Figure 1.26. Data augmentation applied on a patch. First we randomly set to flat some small regions of the ground. Then, we slightly mutate the surface using simplex noise.

### 1.8.1 Software for simulation

**ROS Melodic** The Robot Operating System (ROS) ? is a flexible framework for writing robot software. It is *de facto* the industry and research standard framework for robotics due to its simple yet effective interface that facilitates the task of creating a robust and complex robot behavior regardless of the platforms. ROS works by establishing a peer-to-peer connection where each *node* is to communicate between the others by exposing sockets endpoints, called *topics*, to stream data or send *messages*. Each *node* can subscribe to a *topic* to receive or publish new messages. In our case, *Krock* exposes different topics on which we can subscribe in order to get real-time information about the state of the robot. Unfortunately, ROS does not natively support Python3, so we had to compile it by hand. Because it was a difficult and time-consuming operation, we decided to share the ready-to-go binaries as a docker image.

### 1.8.2 Data processing

**Numpy** Numpy is a one of the most popular packages 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.

**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 outliers and computing the advancement.

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

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

### CNN training

**Pytorch** PyTorch is a Python open source deep learning framework. It allows Tensor computation (like NumPy) with strong GPU acceleration and Deep neural networks built on a tape-based auto grad system. Due to its Python-first philosophy, it is easy to use, expressive and predictable it is widely used among researches and enthusiast. Moreover, its main advantages over other mainstream frameworks such as TensorFlow ? 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 1.7

### 1.8.3 Visualization

**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.

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