Chapter 1

Implementation

This section talks in detail about the implementation of the framework. We will first describe the utilized robot, then discuss each step in the dataset generation. After, we describe five convolutional neural network architectures. We conclude by listing the employed software.

1.1 Robot

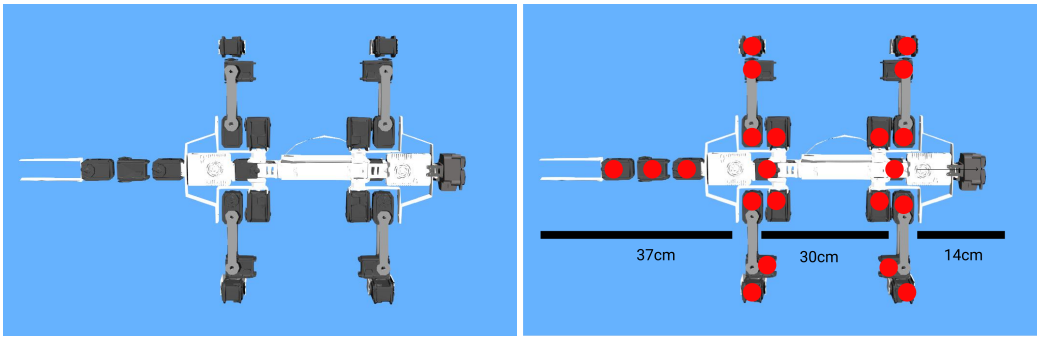
We test our framework on a legged crocodile-like robot called *Krock* developed at EPFL. Figure **??** shows the robot in real life with a cloak and in a simulated environment.

(a) Krock in real life. (b) Krock in the simulator.

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. In addition, there is another set of two motors in the torso to increase Krock's mobility. Those motors can change the robot's gait in three different configurations. 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

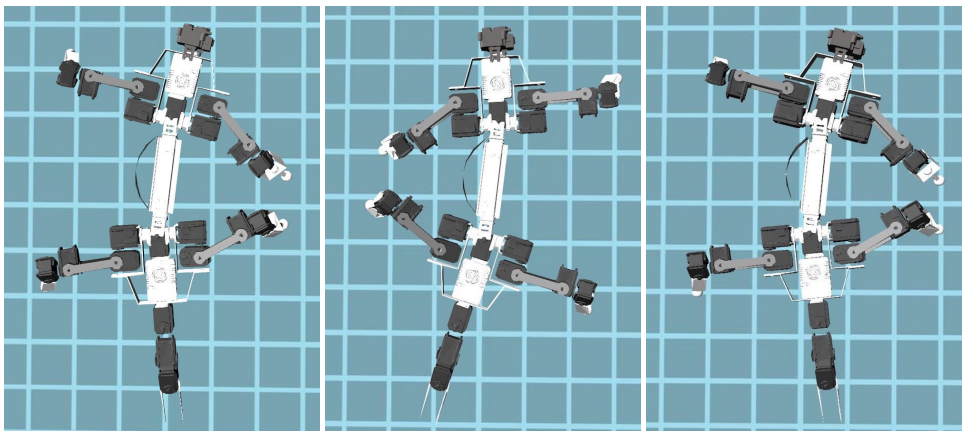
gait configuration its distance from the ground is 16cm. Figure 1.2 shows a topview of the robot. *Krock*'s moves by lifting and moving forward one leg after the other. The following figure shows



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

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

the robots going forward. In our framework we fix the gait configuration to normal, showed in



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

Figure 1.3. *Krock* moving forward. In this gait configuration, the robot distance from the ground is 16cm. Approximately every second the robot moves one leg.

figure 1.1, where the body is at the same legs' motors height.

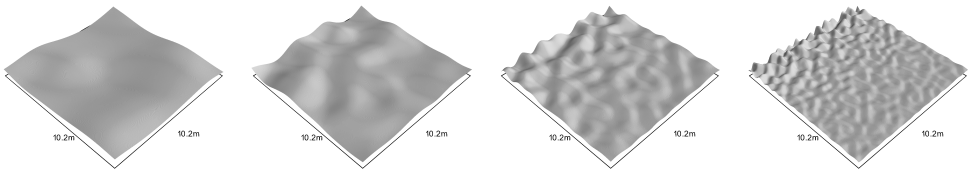
1.2 Data gathering

This section describes in detail how we create, collect and process the synthetic dataset used to train the traversability estimator with supervised learning.

1.2.1 Ground generation

We create thirty 10×10 maps with a resolution offset $0.02cm/pixel$ 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 $holes$.

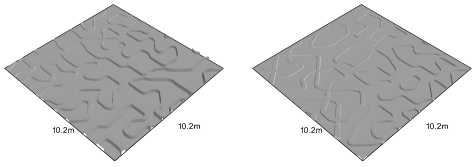
Bumps: We generete four different maps with increasing bumps' height using simplex noise.



(a) *bumps0* (b) *bumps1* (c) *bumps2* (d) *bumps3*

Figure 1.4. Bumps maps.

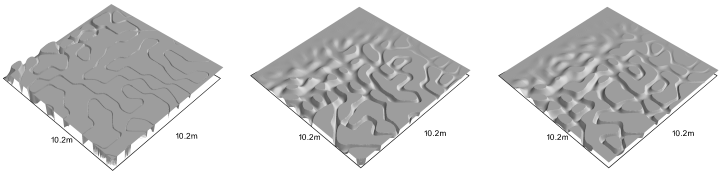
Bars: In these maps there are wall with different shapes and heights.



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

Figure 1.5. Bars maps.

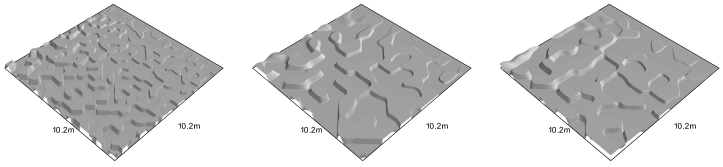
Rails: Flat grounds with slots.



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

Figure 1.6. Rails maps.

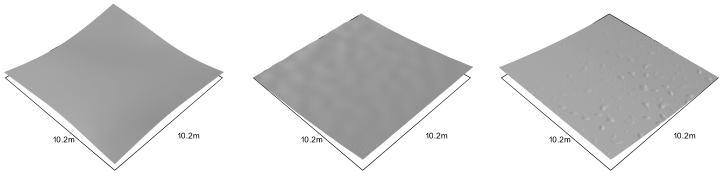
Steps: Maps with various steps at increasing distance and frequency.



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

Figure 1.7. Steps maps.

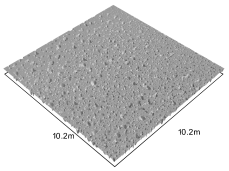
Slopes/Ramps: Maps composed by uneven terrain scaled by different height factors from 3 to 5 used to include samples where *Krock* has to climb.



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

Figure 1.8. Slopes maps.

Holes We also included a map with holes



(a) *holes1*

Figure 1.9. Holes map.

Arc rocks We also crafted an other 10×10m map with a big rocky bump in the middle.

(a) *holes1*

Figure 1.10. Arc rocks map.

1.2.2 Real world maps

We also include real word terrains. We gather two heightmaps produced by ground mapping flying drones from sensefly's dataset, a quarry and a small village. Figure 1.11 shows the original and a 3D render of the heightmap for each terrain.

1.3 Simulation

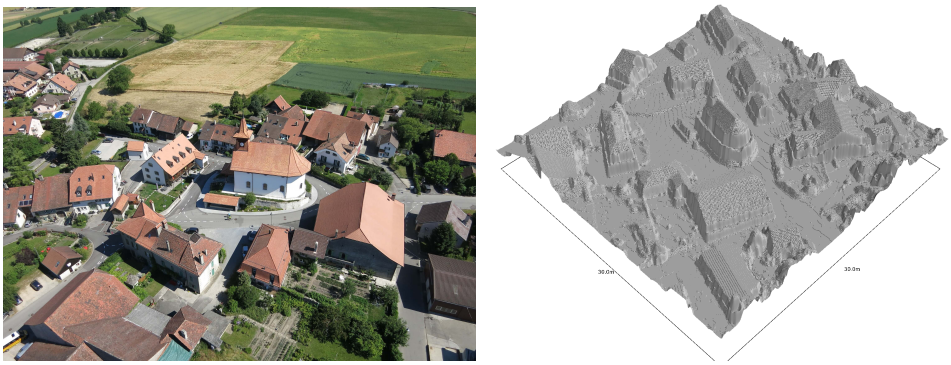
1.3.1 Setup

We use Webots as the simulation platform for the robot model and the generated terrains. The robot's controller is implemented using the Robotic Operating System, it sends data using special nodes called topics. A ROS' node embedded in the controller publishes Krock status, including its pose, at a rate of $250hz$. Due to the high value, we decide to reduce it to $50hz$ by using ROS build it `throttle` command.

1.3.2 Spawing

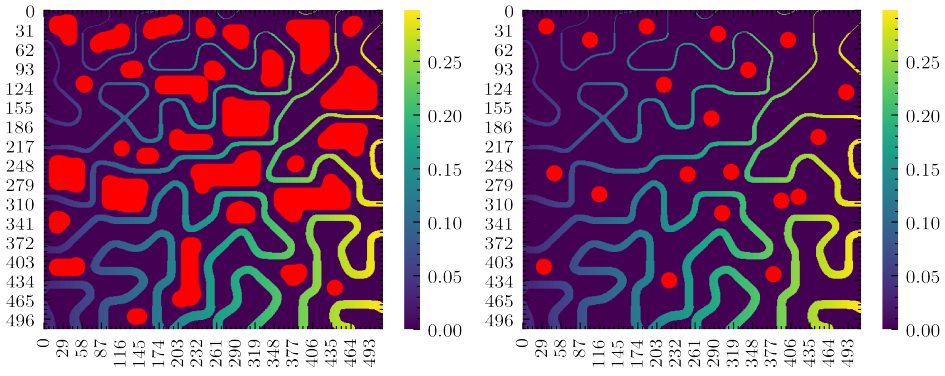
To collect Krock's Interactions with the environment, we spawn the robot on the ground and let it move forward for t seconds. We repeat this process n times per map. Unfortunately, spawning the robot is not a trivial task. For instance, in certain maps we must avoid spawning Krock on an obstacle otherwise the robot will be immediately stuck introducing noise in the dataset. To solve this problem, we define two spawn strategies, a random spawn, and a flat ground spawn strategy. The first one is employed in most of the maps without big obstacles. This strategy just spawns the robot in a random position and orientation. 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*.

(a) Quarry 32×32m



(b) A small village. 20×20

Figure 1.11. Real world maps obtained from sensefly's dataset. The left images show the real world location, the right images a render in 3D of theirs heightmaps. Both images have a maximum height of 10m.



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

Figure 1.12. An exampls of the flat 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 select ten or twenty seconds depending on the map. Every five simulations we completely regenerate the robot's body to counter possible damages caused the hard traversable ground. We mov 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 configuration of the maps used in the simulator. For some terrain, 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 convert 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 load the stored dataframes with the respective terrains and start the data cleaning process. We remove the first simulation's second to account for the robot spawning lag. Then, we eliminat all the entries where a part of Krock was outside the edges of the map. After the data cleaning process, we convert the robot's quaternion rotation to Euler notation using the `tf` package from ROS. Then, we extract the $\sin$ and $\cos$ from the Euler's last components and store them in one dataframe's 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 done once and it is cached to speed up latter runs.

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

Table 1.1. Maps configuration used in the simulator.

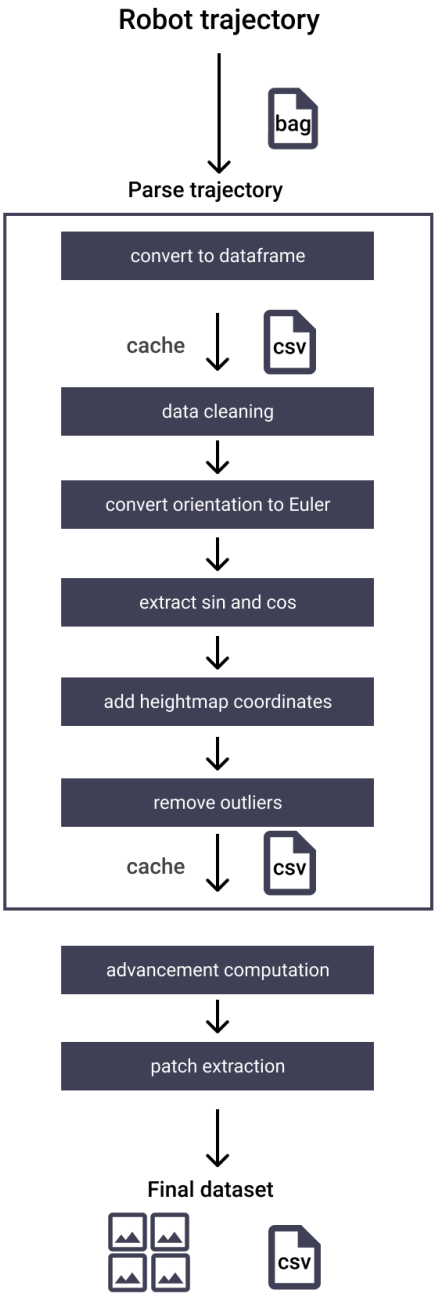


Figure 1.13. Postprocessing pipeline flow graph, starting from the top.

1.4.2 Advancement computation

To compute the robot's advancement we have to to define a time window, Δt, and consider how much the robot traveled between the currenct pose, $X(t)$ and the future one $X(t + \Delta t)$. We empirically observed in the simulator that the robot moves one leg every second. Thus, we select a time window of two seconds to both left and right legs of Krock to move once.

1.4.3 Patch extraction

Each patch must contain both Krock's footprint, to account for the ground under the robot, and a certain amount of ground region in front of it. This corresponds to the maximum possible ground Krock can traverse in a selected time window.

To discover the correct value, we must compute the maximum advancement on 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 70cm in our $\Delta t = 2$s.

Each patch must include Krock's footprint and the maximum possible distance it can travel in from the current pose $X(t)$ to the future position $X(t + \Delta t)$. Since Krock's pose was stored from the 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 $\Delta t = 2$s plus the removed offset. The final patch size is 156×156cm that is encoded as a 78×78 px 2D gray image with a resolution of $2cm/px$. Figure **??** visualizes the patch extraction process. Lastly, we create a final dataframe containing the map coordinates, the advancement, and the patches paths per simulation and store them to disk as *.csv* files.

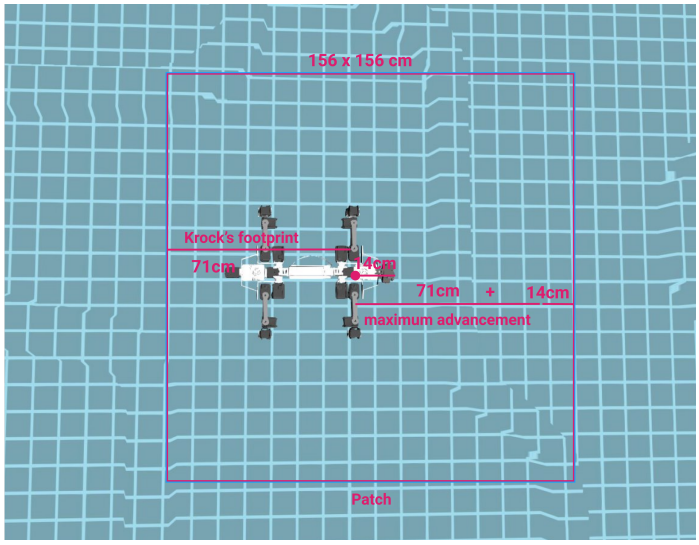
1.4.4 Final dataset

The final dataset is composed of ≈ 470K and ≈ 37K 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

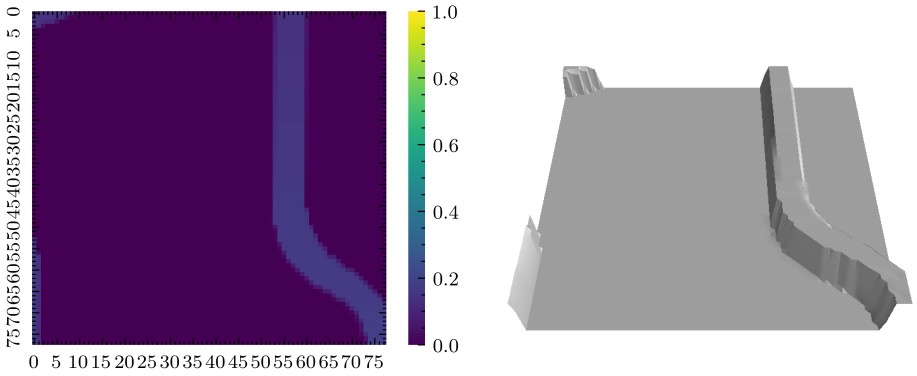
We also generate an other dataset, ≈ 38K patches, using the *arc rocks* map , to further evaluate the model. The whole pipeline took less than one hour to run the first time with 16 threads, and, once it is cached, less than fifteen minutes to extract all the patches. For completeness, we plotted in figure 1.15 the mean advancement over each map in the training set. As sanity check, we visualized some of the patches from the train set ordered by advancement, to conclude the data generation process was correct. Figure 1.17 shows a sample of fourty 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.



(a) Robot in the simulator.



(b) Cropped patch in 2d.

(c) Cropped patch in 3d.

Figure 1.14. Patch extraction process for $\Delta t = 2$s. The final patch covers a region of 156×156cm and is encoded as a 78×78px 2D gray image with a resolution of $2cm/px$.

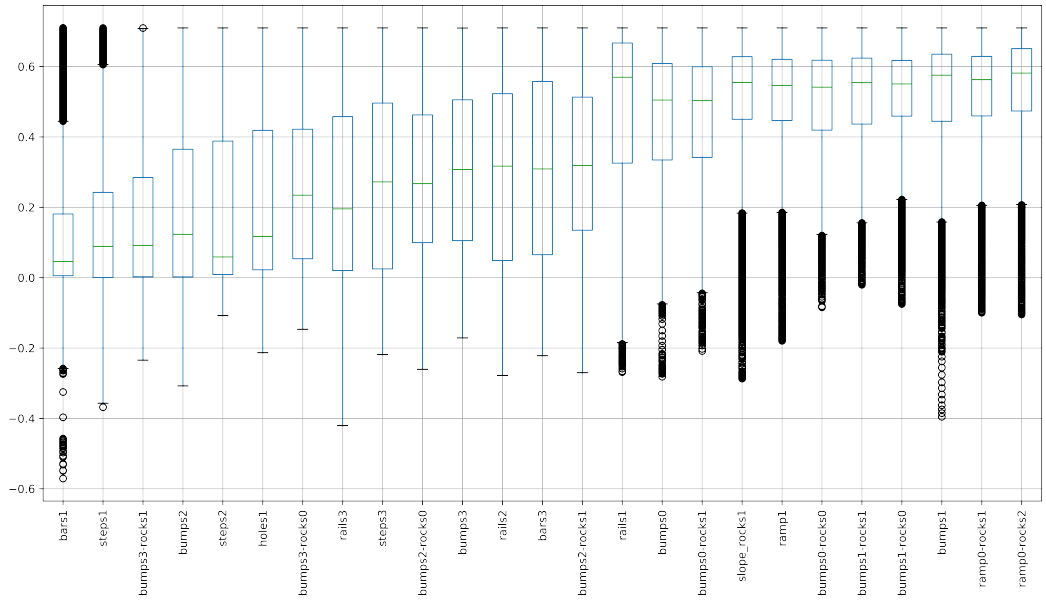
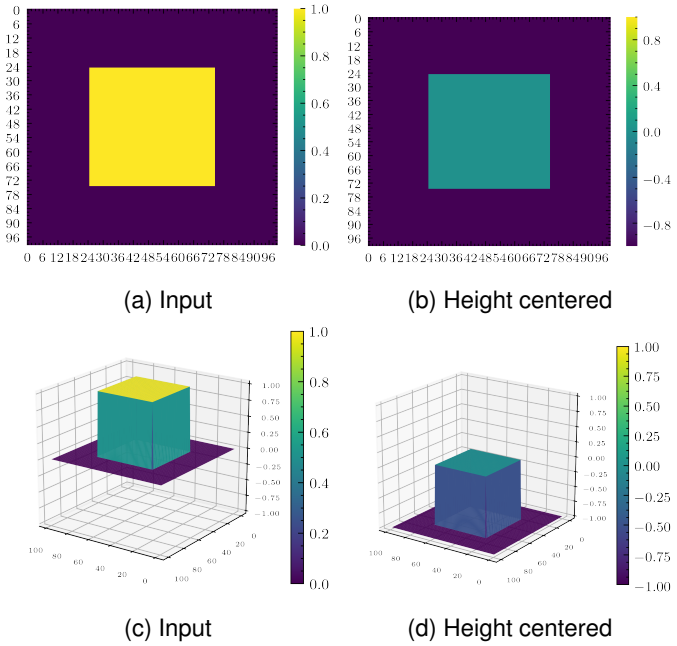


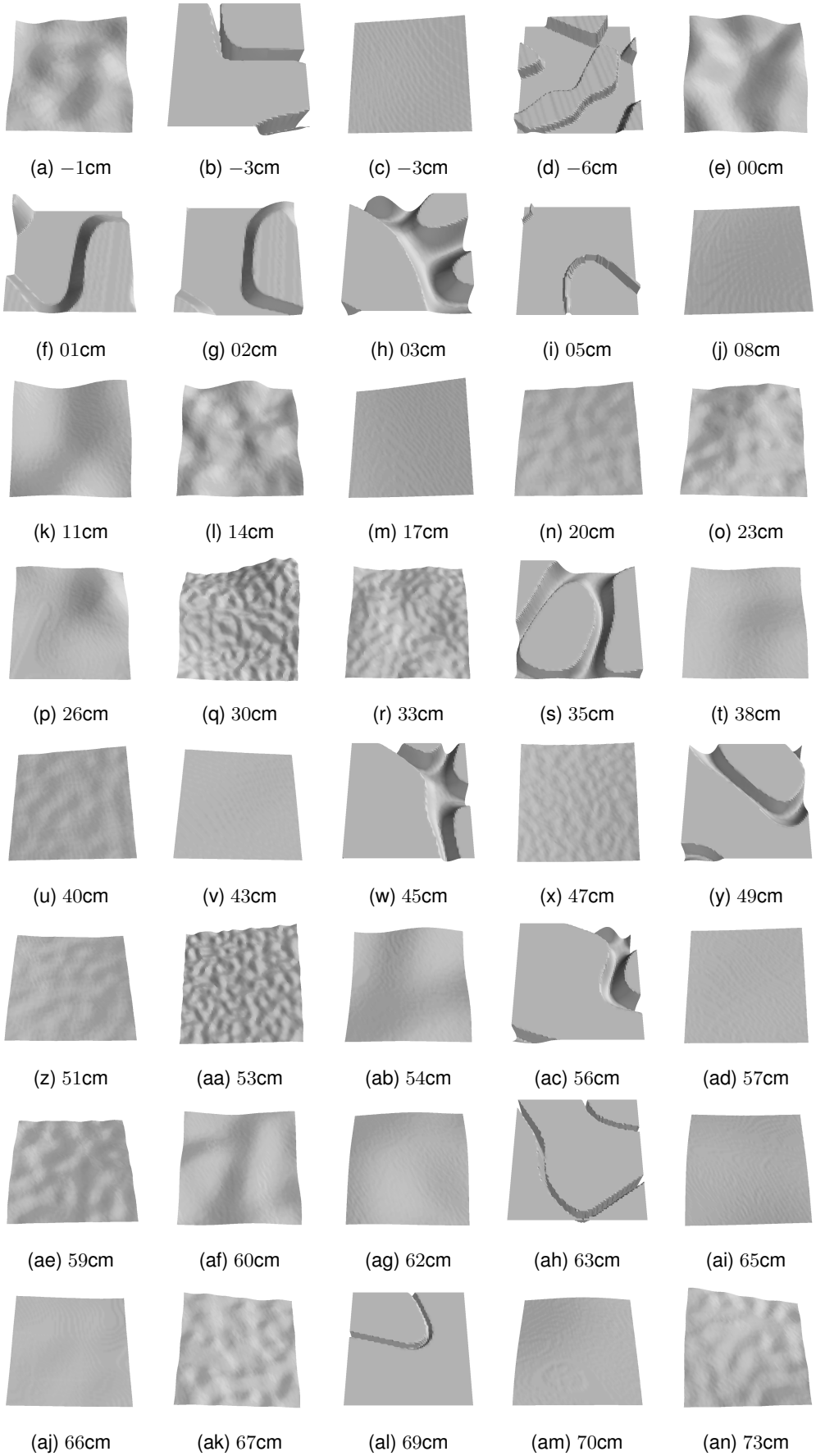
Figure 1.15. Advancement on each map with a $\Delta t = 2$s in ascendent order.



(a) Input (b) Height centered

(c) Input (d) Height centered

Figure 1.16. Normalization process. Each patch is normalized by subtracting the height value corresponding to the robot position to center its height.



(a) -1cm (b) -3cm (c) -3cm (d) -6cm (e) 00cm

(f) 01cm (g) 02cm (h) 03cm (i) 05cm (j) 08cm

(k) 11cm (l) 14cm (m) 17cm (n) 20cm (o) 23cm

(p) 26cm (q) 30cm (r) 33cm (s) 35cm (t) 38cm

(u) 40cm (v) 43cm (w) 45cm (x) 47cm (y) 49cm

(z) 51cm (aa) 53cm (ab) 54cm (ac) 56cm (ad) 57cm

(ae) 59cm (af) 60cm (ag) 62cm (ah) 63cm (ai) 65cm

(aj) 66cm (ak) 67cm (al) 69cm (am) 70cm (an) 73cm

Figure 1.17. 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 smooter surfaces.

1.5 Label patches

To classify the patches we first need to label them. We need to select a threshold to consider a patch traversable if the advancement is greater or not traversable if it is lower. 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 spawning Krock in front of bumps and a ramp and let it walk.

Bumps We place the robot on the *bumps3* map close to the end and let it go for 20 seconds. Figure 1.18 shows Krock in the simulated environment.

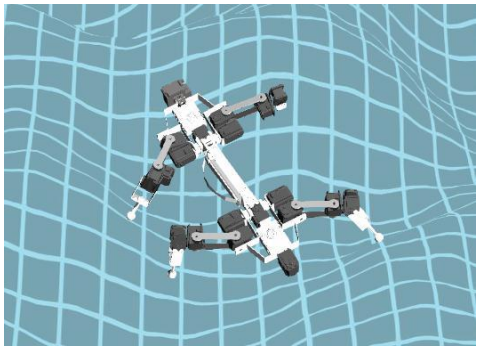


Figure 1.18. Krock tries to overcome an obstacle in the *bumps3* map.

Due to its characteristic locomotion, Krock tries 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 1.19 shows the advancemnet on this map with $\Delta t = 2$, we can notice the noisy output.

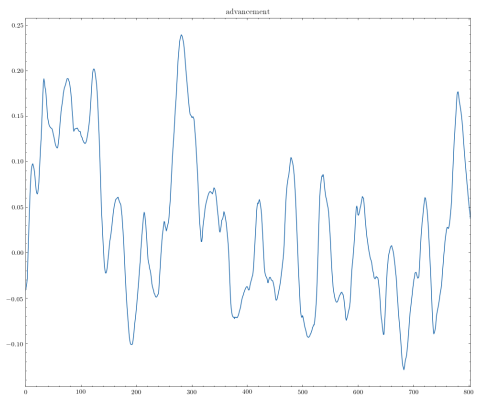


Figure 1.19. Advancement over time with $\Delta t = 2$ on *bumps3*.

A wheeled robot, on the other hand, will not produce such a graph since it cannot free itself easily from an obstacle. Image a wheeled robot moving forward, once it hits an obstacle it stops moving. So, 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 also small enough to not create any false positive, patches that are traversable but were classified as not. So, we let the robot walk uphill on the $slope_rocks1$ map with a height scaling factor set to 5. We knew from empirical experiments that the robot is able to climb this steep ramp, the advancement is showed in figure 1.20. The mean ≈ 0.4cm over a

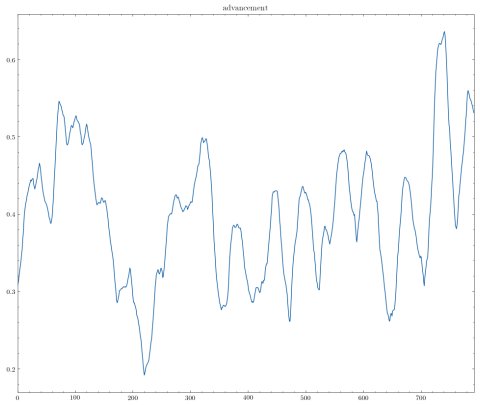


Figure 1.20. Advancement over time with $\Delta t = 2$s on *slope_rocks3*.

$\Delta t = 2$, an impressive value considering the steepest of the surface. We used this information combined with the previous experiment to choose a threshold that is beetween the upper bound of *bumps* and beetween the lower bound of *slope_rocks1*. This ensured to minimize the false positive. A good value is $tr_{\Delta t=2s} = 20cm$.

1.6 Estimator

In this section, we described the convolutional neural network utilized in the original paper. Then, we introduce residual networks and new techniques used to improve their performance. In the end, we proposed four residual network variants to be evaluated.

1.6.1 Original Model

The original model proposed by Chavez-Garcia et al. **?** 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. We considered this the baseline, Figure 1.21 visualizes the architecture.

1.6.2 ResNet

We adopt a Residual Network, ResNet **?**, variant. Residual networks are deep convolutional networks consisting of many stacked R̈esidual Units ¨. Intuitively, the residual unit allows the input of a layer to contribute to the next layer's input by being added to the current layer's output. Due to possible different features dimension, the input must go through and identify the map to make the addition possible. This allows a stronger gradient flows and mitigates the degradation problem. A R̈esidual Units ïs composed by a two $3x3$ *Convolution*, *Batchnorm* **?** and a *Relu* blocks. Formally

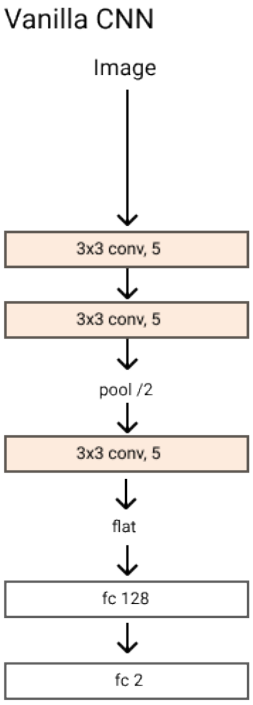


Figure 1.21. original model proposed by Chavez-Garcia et al. **?**. The rectangle represents the building block of each layer. Convolutional layers are described by the kernel size, the number of filters and the stride.

defined as:

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

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 mistmatch, the *identity map* is applyed to the input as

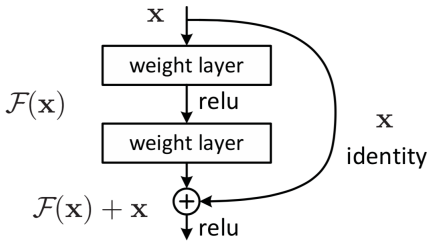


Figure 1.22. *Resnet* block **?**

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.

1.6.3 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.

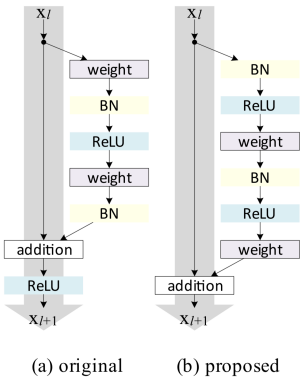


Figure 1.23. *Preactivation* **?**

1.6.4 Sqeeze and Excitation

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

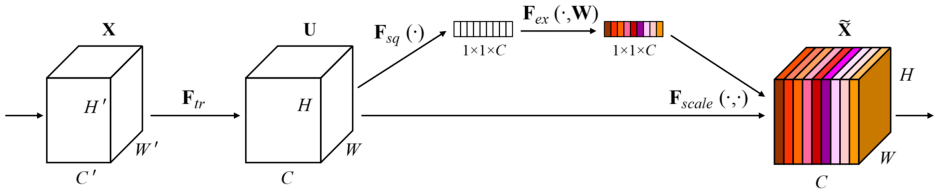


Figure 1.24. *Squeeze and Excitation*. Figure from the original paper. **?**

1.6.5 MicroResNet

The proposed network's architecture is composed by n ResNet blocks and channel incrementing factor of 2. ResNet is used to classify RGB images with 224×224 pixels so it used an aggressive convolution and max-polling operation in the first layer to reduce the input size. We decide to adopt a less aggressive convolution motivated by the smaller size of our inputs. We test two kernel sized: 7×7 and 3×3 with stride of 2 and 1 respectively. Lastly, we use LeakyReLU **?** with a negative slope of 0.1 instead of ReLU to allow a better gradient flow during backpropagation. LeakyRelu is defined in equation 1.2 .

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

We call this architecture *MicroResNet*. We propose four networks layouts: with $n = 1$, two first layers setups, 3×3 stride $= 1$ and 7×7, with and without the Squeeze and Excitation module. Those architectures can be used both for regression and classification by only changing the output size of the fully connected later, 2 for classification and 1 for regression. All other layers are equals. Figure 1.25 shows the models architecture while table 1.2 adds more information. Our models have 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-3x3/7x7-/SE`.

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 a dummy patch with a cube in the middle, figure 1.16.

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.

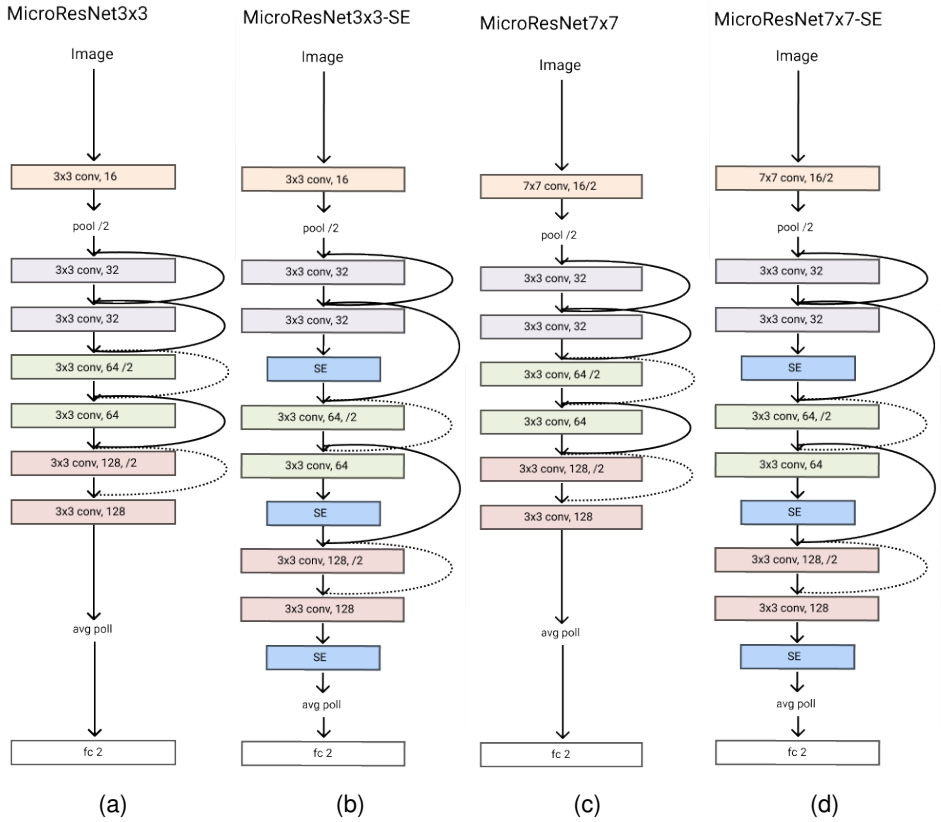


Figure 1.25. MicroResNet architectures for classification. All models have an initial features size of 16 and $n = 1$ with and without the Squeeze and Excitation module. The rectangle represent the building block of each layer. Convolutiol layers are described by the kernel size, the number of filters and the stride. Lines and dashed lines between layers represent the residual operations and the shortcut respectively.

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			
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.

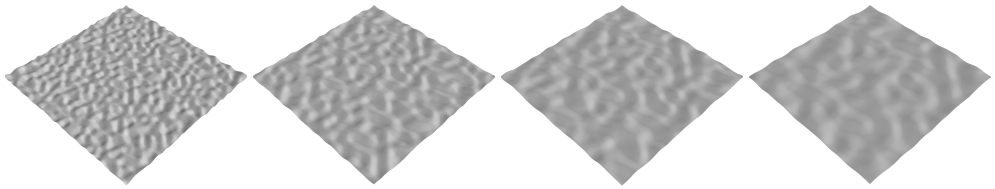
1.7.2 Coarse Dropout

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

1.7.3 Simplex Noise

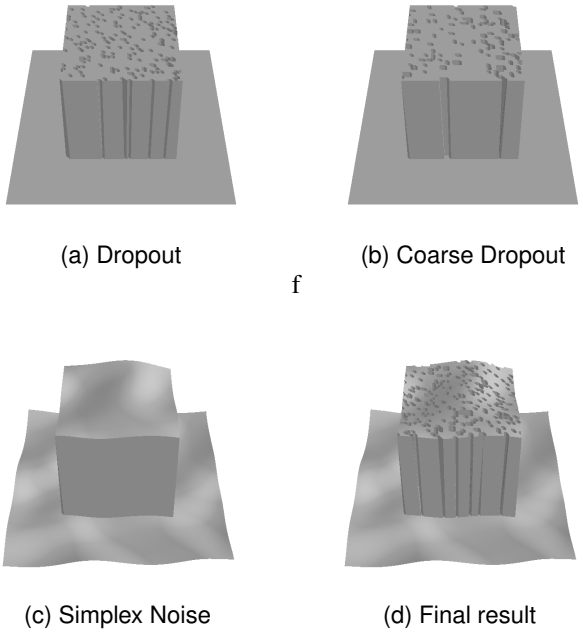
We added some noise the inputs to slightly changed the ground in order to help the network generalize better. Since it is computationally expensive, we first randomly applied the noise to five hundred images with only zeros, flat grounds. We cached them and then we randomly scaled them and added to the 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 1.26 clearly shows the grounds generated by different sizes.

Figure **??** shows the tree data augmentation techniques applied the the dummy patch. 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 is applied to the 70% of the training data samples.



(a) Noise size = 10 (b) Noise size = 20 (c) Noise size = 30 (d) Noise size = 40

Figure 1.26. Simplex noise on flat ground. The feature size corespond to the amount of noise we introduce in the surface. A lower value produces noisy grounds, while a bigger one smoother terrains.



(a) Dropout (b) Coarse Dropout

f

(c) Simplex Noise (d) Final result

Figure 1.27. 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 Experiments

1.8.1 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 the 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 selected 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.

1.9 Training setup

Initially, to train the models we first use Standard Gradient Descent with momentum set to 0.95, weight decay to $1e-4$ and an initial learning rate of $1e-3$ as proposed in He et al. **?**. Then, we switch to Leslie Smith's 1 cycle policy **?** that allow us to train the network faster and with higher accuracy. The average training time is $\approx$ 10 minutes. We minimized the Binary Cross Entropy (BCE) for the classifier and the Mean Square Error (MSE) for the regressor.

1.10 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.

1.10.1 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.

1.10.2 Data processing

Numpy Numpy is one of the most popular packages for any scientific use. It allows expressing 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 the SQL table or Excel spreadsheet, time series and matrices. We take advantages of the relational 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 b̈ig data ̈o other frameworks such as Spark or Hadoop. We used Pandas to store the results from the simulator and inside a Thread Queue to parse each *.csv* file efficiently.

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

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

1.10.3 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 a high-level interface.

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