1 Model

This section gives a high-level overview of the steps in our approach. We will first describe the data gathering, then the post-processing pipeline used to generate the dataset. After, we will introduce the model used to fit the data and then show the results with different test environments.

1.1 Robot: Krock

Our task was to apply the approach proposed in [**omar@traversability**] with a legged robot developed at EPFL named *Krock*. Figure **??** shows the robot in the simulated environment.

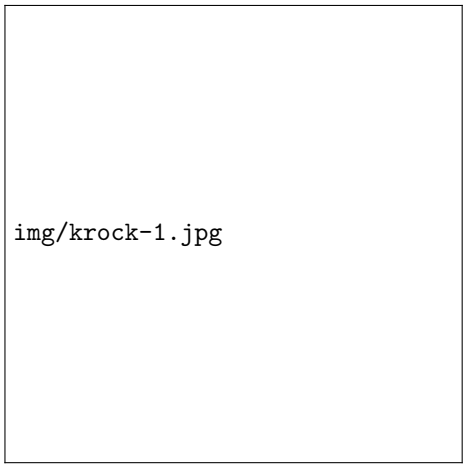


Figure 1: *Krock*

Krock has four legs, each one of them is equipped with three motors in order to rotate in each axis. The robot is also able to raise itself in three different configurations, gaits, using the four motors on the body connected to the legs. In addition, there are another set of two motors in the inner body part to increase *Krock*'s move set. 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 and weights around $1.4kg$. The next figure **??** shows *Krock* from the top helping the reader understanding its composition and the correct ratio between its parts. Also, each motor is highlighted with a red marker.

??

Figure 4: *Krock* differents gait configuration.

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

Krock's moves by lifting and moving forward one leg after the other. The following figure shows the robots going forward.

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

Figure 3: *Krock* moving forward.

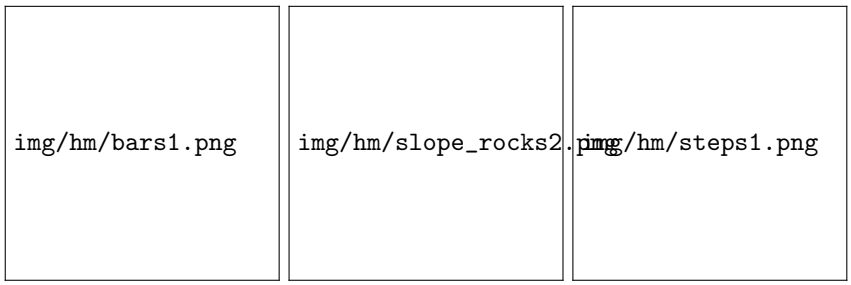
Krock can also raise its own body in three different configurations. Figure

Add Krock gait picture

1.2 Simulation

Our approach relies on simulations to collect the data needed to train the estimator. We used Webots [**webots**], a professional mobile robot simulator, as the backbone for our framework.

We generate fifteen synthetic $10x10$ meters long heightmaps with different features, such as walls, bumps and slopes using the same techniques described in [**omar2018traversability**]. A heightmap is a gray image, formally a 2D array, where each element, pixel, represents a height value. Since each image has a range from $[0, 255]$, we also need to scale them when needed. For this reason, we associated a height scaling value for each height map that is used to increase or decrease the steepness. The next figure shows some of the maps used.



(a) *bars1* (b) *slope rocks 2* (c) *steps 1*

Figure 5: Some of the heightmaps used in the simulation.

The following figure shows the 3d rendered version of the bars heightmap.

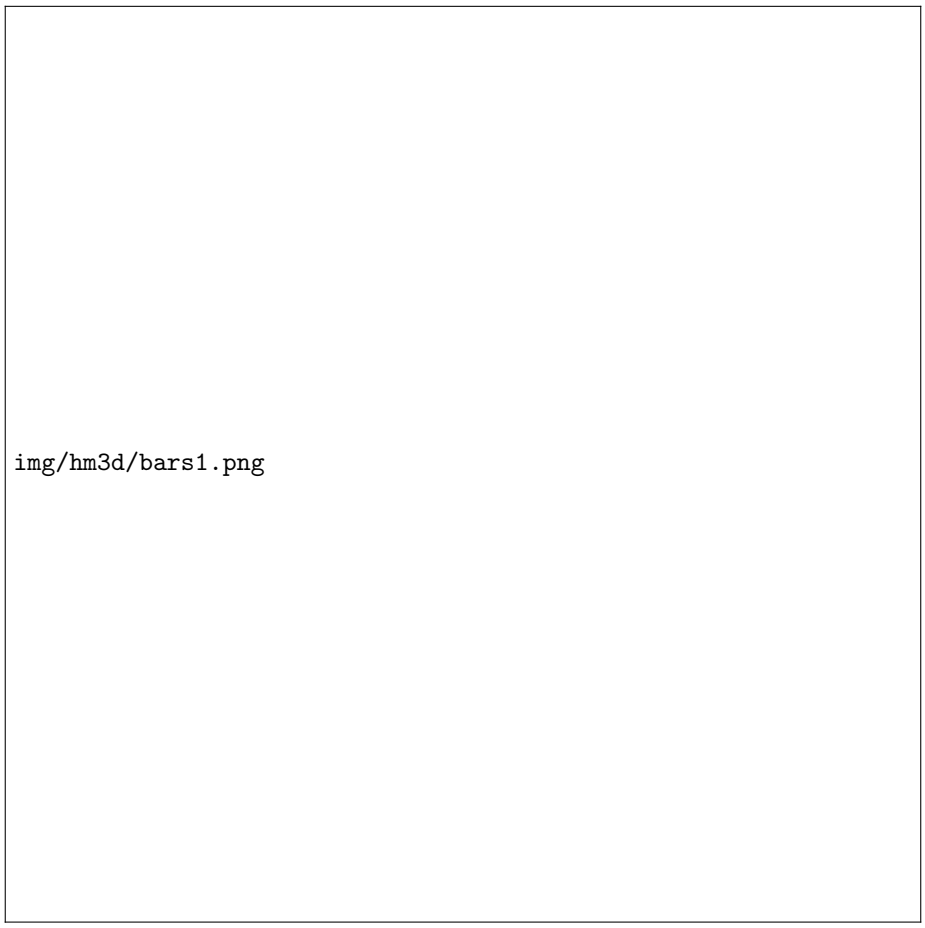


Figure 6: *Bars1* 3d map

To generate the train data, each map is loaded into the simulator, then the robot is spawned in the world and we let it walked forward for a certain amount of time or until it reaches the edge of the map. While moving, we store his pose, that contains position and orientation, with a rate of $50hz$.

We fix the robot's gait in its standard configuration and run fifteen simulations for each map, where one simulation one robot spawns and death, with a height scale factor of one. We also decide to include steepest training samples by using one of the maps with a slope with heights factors from $[1, 7]$. The following pictures show this specific map with different scaling factors.

(a) height = 1 (b) height = 2

Figure 7: *Slope rocks 2* map with two different height scaling factor.

Krock is equipped with a controller implemented with the Robotic Operating System (ROS) software that is used as a communication channel between our framework and the simulator. Basically, ROS exposes nodes, called *topics*, in which we can listen for incoming messages. Thus, *Krock* is able to stream its pose to all the clients connected to its pose topic.

We also generate the test set, by running the robot in real-world environment, such as *Quarry* a cave map, that we later use to test the fitted model.

Figure 8: *Quarry* map.

By observing the simulations we notice two problems. First, *Krock*'s tail sometime gets under the body, second, if a random spawn strategy is used, in certain maps the agent may directly fell on an obstacle.

To overcome the first problem, we decide to remove the tail after we ensure by asking the developers that this will not compromise the traversability. We immediately observe less noisy data.

Figure 9: *Krock* with and without tail

The second issue was more challenging. If the robot spawns on an obstacle then it will be stuck for all the simulation compromising the quality of the data. This is very important for the *bars* map, where there are lots of wall of different sizes and the probability to fall on one of them is very high. So, for some maps, we guarantee that *Krock* does not directly spawn into obstacles by only spawning the robot in the flat ground first. The following figures show the result of this strategy on the *bars1* map where the robot was spawned where there is no obstacle. The following image shows different *Krock*'s spawn position in the simulator on the *bars1* map, the reader can notice that the robot is always placed where there are no obstacles under it.

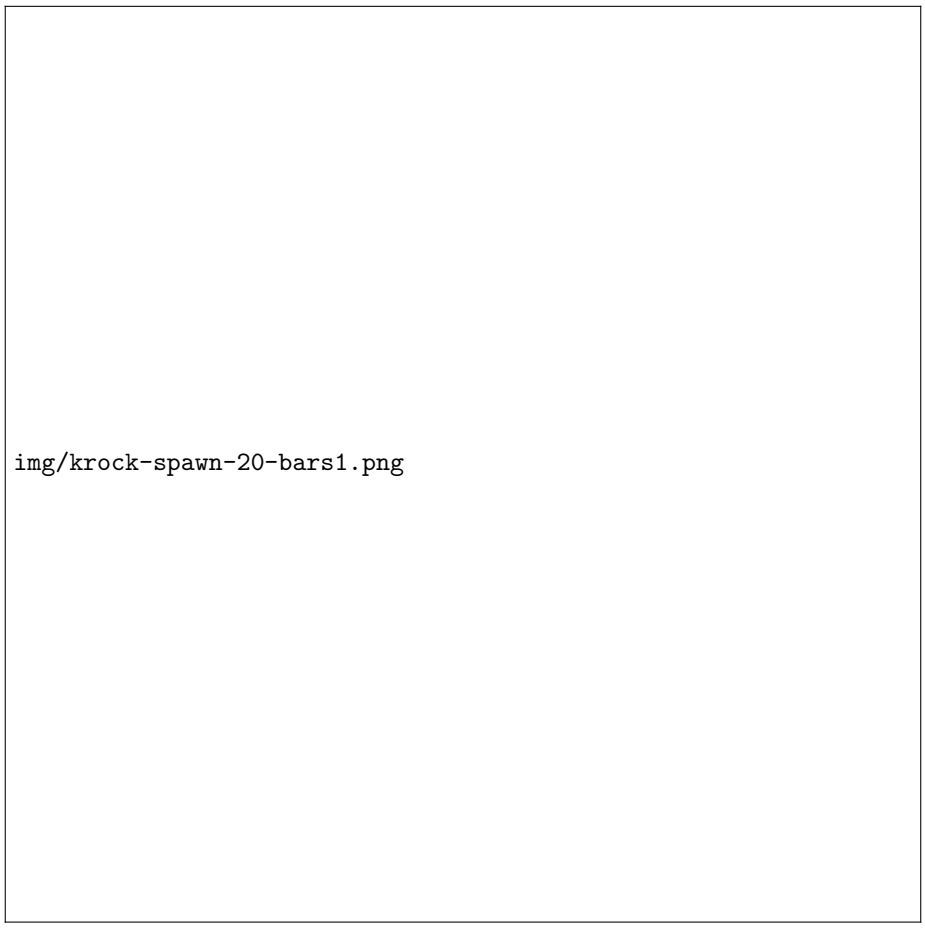


Figure 10: *Krock*'s spawns on the *bars1* map.

We run a total of storing the results using ROS's *.bag* files for a total of .

add number of simulations

add size of all bags.

2 Postprocessing

In order to create the training dataset, we need to post process the data stored during the simulation. To train our classifier we need to first compute the advancement for each pose at each time for a specific time window, $pose_{it}$. Then, we can use it to crop a patch from the heightmap that was used in the simulation of the correct size. Such patch must include the whole *Krock* footprint and the maximum possible advancement in a given time window. By keeping this final goal in mind, we quickly describe each step taken to reach it. Figure **??** shows each step applied in the pipeline.

Figure 11: Pipeline

We implemented an easy to use API to define a cascade stream of function that is applied one after the other using a multi-thread queue to speed-up the process. First, we convert each *.bag* file into a *.csv* file and store it since this operation must be done only once. Then, we define a time window t and for each entry in one simulation run we compute the future advancement using booth position and angle from the pose, we also store the resulting data frame into a *.csv* file. During the process, we also clean the data by removing the samples in which the robot was upside down or out of the map.

link to the project

Finally, we crop from each heightmap the patches corresponding to each data point. To compute the patch size need to know the maximum possible advancement of *Krock* for a specific time window. In order to find it, we run *Krock* on a flat ground map and we take the mean across all of them. We observer a mean advancement of $33cm$ per second.

For a given advancement, we ensure the whole *Krock* footprint is included on the left part to take into account the situations in which an obstacle is directly under the robot. To clearly show the process, we used a simulated environment in which *krock* has a big wall in front of him and one small under his legs. The following picture shows the terrain.

add figure with the two walls

Clearly, assuming *Krock* is able to traverse 33cm of flat ground per second, in one second it won't be able to reach the obstacle, this is shown in the first picture. Increasing the time window, the taller wall appears while the small one under the robot is always correctly included.

add figure with the patch computation

3 Estimator

To classify the patches we need a class associated to each one of them, thus we must label each patch as *traversable* or *not traversable*. So, we selected a threshold, tr, and labeled each patch by $patch_{advancement} > tr$. In other words, if *Krock* was able to advance in a patch more than the threshold, then that patch is label as *traversable* and vice-versa. The pair of patches and labels represent the final dataset.

add an image with a trivial traversable and not traversable patch

We used a deep convolutional neural network to fit the training set. Convolutional Neural Network is able to map images into a features space and then to target class. Before feeding the patches into the model, we subtract in each patch the value in the middle in order to normalize them by making height invariant. Visually,

add figure with a patch not centered and centered

We evaluate different models with different hyperparameters to select the best one using the metrics score on the test set as guidelines. We then visually evaluate the network by running it on the *Quarry* map for different orientations. In order to do so, we extract patches form the ground using a sliding window and feed them to the model. Then, we create a texture where each pixel represents the traversability probability, the brighter the higher.