1 Abstract

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

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

2 Introduction

Effective identification of traversable terrain is essential to operate mobile robots in every type of environment. Today, there two main different approaches used in robotics to properly navigate a robot: online and offline. The first one uses local sensors to map the surroundings "on the go" while the second equip the mobile robot with an already labeled map of the terrain.

In most indoor scenarios, specific hardware such as infrared or lidar sensors is used to perform online mapping while the robot is exploring, this is the case of the most recent vacuum cleaner able to map all the rooms in an apartment. With the recent breakthroughs of deep learning in computer vision, more and more cameras have been used in robotics. For example, self-driving cars utilize different cameras around the vehicle to avoid obstacle using object detection. Indoor scenarios share similar features across different places shifting the problem from which ground can be traversable to which obstacle must be avoided. For instance, the floor is always flat in almost all rooms due to is artificial design. Moreover, usually, traversability must be estimated on the fly due to the high number of possible obstacles and to the layout of the objects in each room may not be persistent in time.

On the other hand, outdoor scenarios may have less artificial obstacle but their homogeneous ground makes challenging to determine where the robot can properly travel. Moreover, a given portion of the ground may not be traversable by all direction due to the not uneven terrain. Fortunately, a height map of the ground can be obtained easily by using third-party services or special flying drones. Those maps are extremely valuable in robotics applications since they provide an efficient way to examine the features of terrains, such as bumps, holes, and walls.

These scenarios have different difficulties. In indoor environments, it is easier to move the robot on the ground since it is designed for humans, but harder to perform obstacle avoidance. While in outdoors scenario it is maybe more challenged to the first estimate where the robot can go due to the huge variety of ground features that may influence traversability. To learn where to move, an artificial controller must be trained to predict the robot interaction with the environment. Such a process requires to collect some data to train the controller.

However, this may not be a straight forward process. In indoors terrain, most of the times, data is collected by driving the robot directly in the environment by a human or an artificial controller. While in the outdoors scenario data is assembled using the simulation for convenience.

Our approach aims to estimate traversability of a legged robot krocodile-like robot called Krock. We generate different uneven grounds in form of height map and let the robot walk for a certain amount of time on each one of them while recording its interaction, position and orientation. After, for each stored robot position we crop the corresponding ground portion in which the robot was during the simulator, those patches composed the training dataset. We select a minimum space in a fixed amount of time that the robot must travel to successfully traverse a ground region and use it to label the dataset. Then, we fit a deep convolutional neural network to predict the traversability probability. Later, we evaluate it using different metrics using real-world terrains.

The report is organized as follow, the next chapter introduces the related work, Chapter 2 describes our approach, Chapter 3 talks in deep about the implementation details, Chapter 4 shows the results and Chapter 5 discuss conclusion and future work.

3 Related Work

The learning and perception of traversability is a fundamental competence for both organisms and autonomous mobile robots since most of their actions depend on their mobility [10]. Visual perception is known to be used in most all animals to correctly estimate if an environment can be traversed or not. Similar, a wide array of autonomous robots adopt local sensors to mimic the visual properties of animals to extract geometric information of the surrounding and plan a safe path through it.

Different methodologies have been proposed to collect the data and then learn to correctly navigate the environment. Most of the methodologies rely on supervised learning, where first the data is gathered and then a machine learning algorithm is trained sample to correctly predict the traversability of those samples. Among the huge numbers of methods proposed, there are two categories based on the input data: geometric and appearance based methods.

Geometric methods aim to detect traversability using geometric properties of surfaces such as distances in space and shapes. Those properties are usually slopes, bumps, and ramps. Since nearly the entire world has been surveyed at 1 m accuracy [8], outdoor robot navigation can benefit from the availability of overhead imagery. For this reason, elevation data has also been used to extract geometric information. Chavez-Garcia et al. [2], proposed a framework to estimate traversability using only elevation data in the form of height maps.

Elevation data can also be estimated by flying drones. Delmerico et al. [4] proposed a collaborative search and rescue system in which a flying robot that explores the map and creates an elevation map to guide the ground robot to the goal. They train on the fly a convolutional neural network to segment the

terrain in different traversable classes.

Whereas appearance methods, to a greater extent related to camera images processing and cognitive analyses, have the objective of recognising colours and patterns not related to the common appearance of terrains, such as grass, rocks or vegetation. Those images can be used to directly estimate the traversability cost.

Historically, the collected data is first preprocessed to extract texture features that are used to fit a classic machine learning classified such us an SVM [10] or Gaussian models [8]. Those techniques rely on texture descriptors, for example, Local Binary Pattern [9], to extract features from the raw data images obtained from local sensors such as cameras. With the rise of deep learning methods in computer vision, deep convolution neural network has been trained directly on the raw RGB images bypassing the need to define characteristic features.

One recent example is the work of Giusti et al. [1] where a deep neural network was training on real-world hiking data collected using head-mounted cameras to teach a flying drone to follow a trail in the forest. Geometric and appearance methods can also be used together to train a traversability classifier. Delmerico et al.[3] extended the previous work [4] by proposing the first on-the-spot training method that uses a flying drone to gather the data and train an estimator in less than 60 seconds.

Data can also extract in simulations, where an agent interacts in an artificial environment. Usually, no-wheel legged robot able to traverse harder ground, can benefits from data gathering in simulations due to the high availability. For example, Klamt et al. [5] proposed a locomotion planning that is learned in a simulation environment.

On other major distiction we can made is between different types of robots: wheel and no-wheel. We will focus on the later since we adopt a legged crocodile-like robot to extend the existing framework proposed by Chavez-Garcia et al. [2].

Legged robots show their full potential in rough and unstructured terrain, where they can use a superior move set compared to wheel robots. Different frameworks have been proposed to compute safe and efficient paths for legged robots. Wermelinger et al. [**wermelinger2016navigation**] uses typical map characteristics such as slopes and roughness gather using onboard sensors to train a planner. The planner uses a RRT* algorithm to compute the correct path for the robot on the fly. Moreover, the algorithm is able to first find an easy local solution and then update its path to take into account more difficult scenarios as new environment data is collected.

Due to uneven shape rough terrain, legged robots must be able to correctly sense the ground to properly find a correct path to the goal. Wagner et al. [6] developed a method to estimate the contact surface normal for each foot of a legged robot relying solely on measurements from the joint torques and from a force sensor located at the foot. This sensor at the end of a leg optically determines its deformation to compute the force applied to the sensor. They combine those sensors measurement in an Extended Kalman Filter (EKF). They

showed that the resulting method is capable of accurately estimating the foot contact force only using local sensing.

While the previous methods rely on handcrafted map's features extraction methods to estimate the cost of a given patch using a specific function, new frameworks that automatise the features extraction process has been proposed recently. Lorenz et al. [**wellhausen2019where**] use local sensing to train a deep convolutional neural network to predict terrain's properties. They collect data from robot ground interaction to label each image in front of the robot in order to predict the future interactions with the terrain showing that the network is perfectly able to learn the correct features for different terrains. Furthermore, they also perform weakly supervised semantic segmentation using the same approach to divide the input images into different ground classes, such as glass and sand, showing respectable results

Add mirko's paper

4 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.

4.1 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.

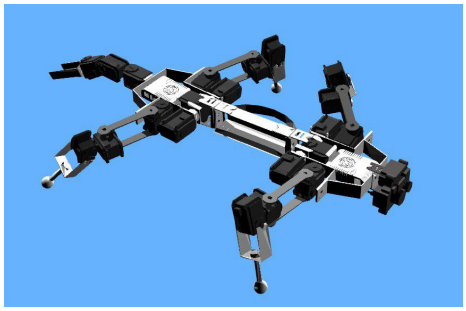


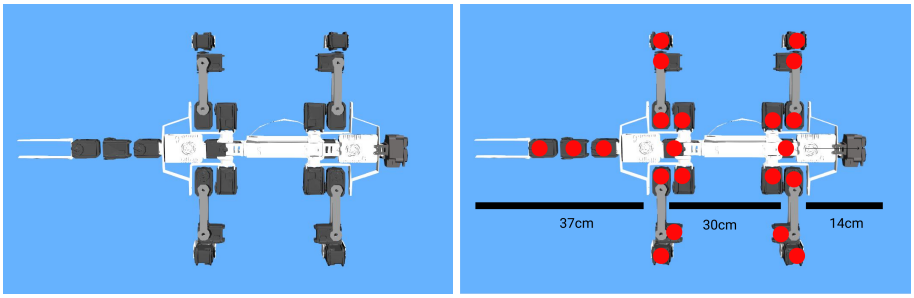
Figure 1: *Krock*

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

??

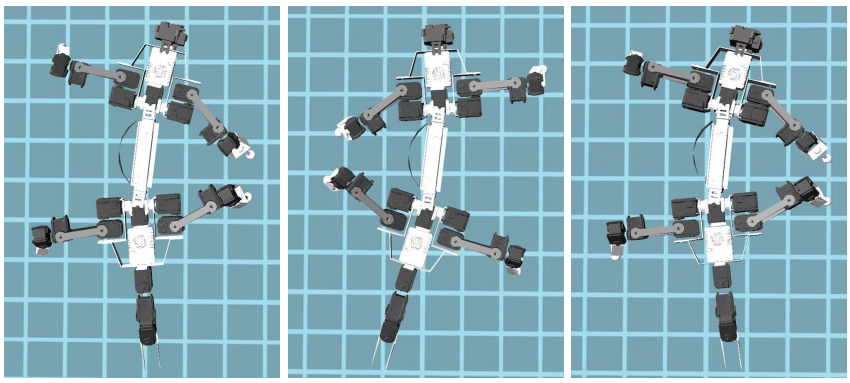
Figure 4: *Krock* differents gait configuration.

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.



(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.

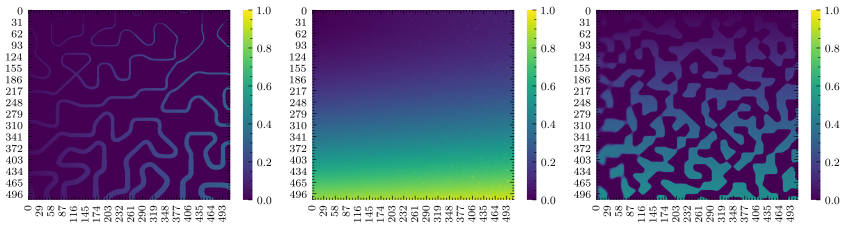
Krock can also raise its own body in three different configurations. Figure

Add Krock gait picture

4.2 Simulation

Our approach relies on simulations to collect the data needed to train the estimator. We used Webots [7], 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 [2]. 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.

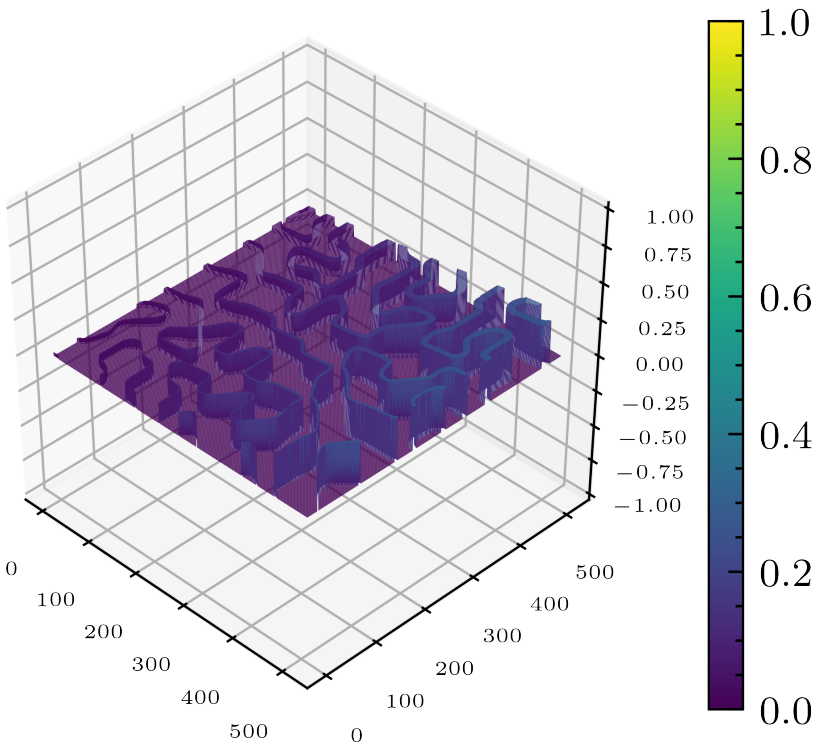
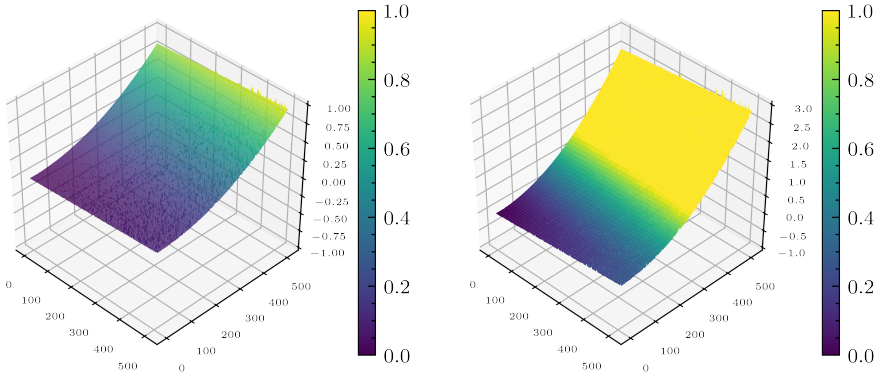


Figure 6: *Bars1* 3d map

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 = 3

Figure 7: *Slope_rocks2* 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.

Update this image, maybe a blender render?

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.

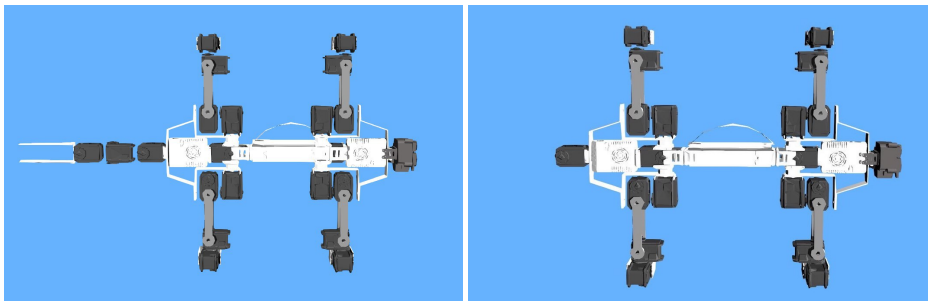


Figure 9: *Krock* with and without tail

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.

link to the project

4.3 Postprocessing

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.

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

4.4 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.

5 Implementation

5.1 Tools

The most important tools and libraries adopted in this project are:

- ROS Melodic
- Numpy
- Matplotlib
- Pandas
- OpenCV
- PyTorch
- FastAI
- imgaug
- Blender

The framework was entirely developed on Ubuntu 18.10 with Python 3.6.

5.1.1 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.

5.1.2 Numpy

Numpy is a fundamental package for any scientific use. Thanks to its powerful N-dimensional array object with the sophisticated broadcasting functions, it is possible to express efficiently any matrix operation. We utilized Numpy almost everywhere to manipulate matrices in an expressive and efficient way.

5.1.3 Matplotlib

Matplotlib is a widely used Python 2D plotting library which generates high-quality figures in a variety of hardcopy formats and interactive environments across platforms. It provides a similar functional interface to MATLAB and a deep ability to customize every region of the figure. All the figures made in this report were produced using Matplotlib. It is worth citing *seaborn* a data visualization library that we inglobate in our work-flow. It is based on Matplotlib and it provides a high-level interface for drawing attractive and informative statistical graphics.

5.1.4 Pandas

Pandas is a Python library providing fast, flexible, and expressive data structures in a tabular form. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python. Today, it one of the most flexible open source data manipulation tool available. 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. It provides to two primary data structures, `Series` and `DataFrame` for representing 1 dimensional and 2 dimensional data respectively.

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.

5.1.5 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. With a huge community of more than forty-seven thousand people, the library is a perfect choice to handle image data. In our framework, OpenCV is used to pre and post-process the heightmaps and the patches.

5.1.6 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 has a deep integration into Python allows popular libraries and packages to be used, such as OpenC or Pillow.

Due to its simple, expressive and beautiful object-oriented API it has been adopted by a huge number of researches and enthusiastic all around the world creating a flourishing community. Its main advantages over other mainstream frameworks such as TensorFlow are a cleaner API structure, better debugging, cite TF

code shareability and an enormous number of high-quality third-party packages. All the neural network proposed in this project are built using Pytorch.

5.1.7 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.

5.1.8 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.

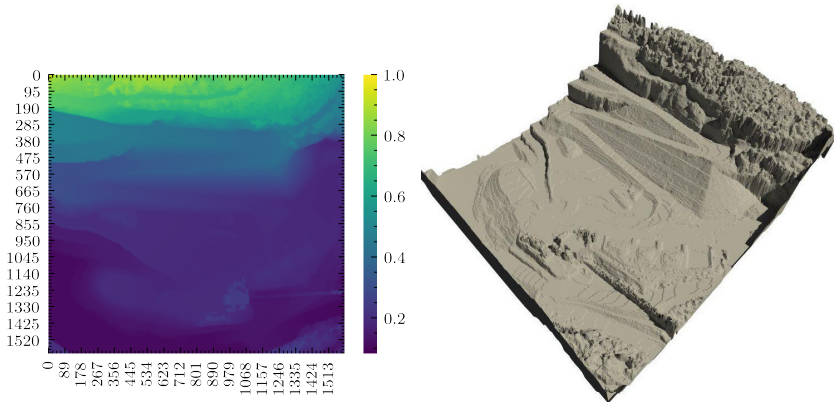
5.1.9 Blender

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.

5.2 Data Gathering

5.2.1 Heightmap generation

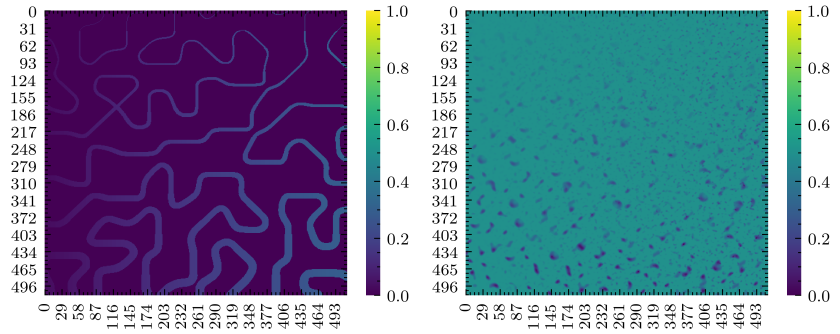
A heightmap is a 2D array, an image, where each pixel's value represents the terrain height.



(a) *Quarry* heightmap (b) *Quarry* rendered

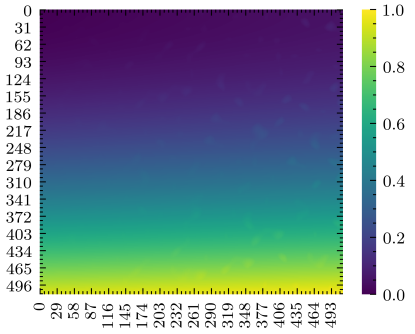
We generate fifteen terrain using random 2D simplex noise [**simplex**], a variant of Perlin noise [**perlin**], a widely used technique in terrain generation litterature.

3d rendered heightmap from Blender should visualize better the idea

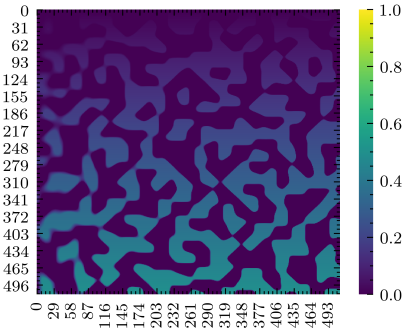


(a) *bars1*

(b) *holes1*



(c) *slope_rocks1*



(d) *steps1*

Figure 12: Some of the synthetic heightmaps

5.2.2 Simulator

We used Webots to move *Krock* on the genereted terrain. The robot controlled was implemented by EPFL and handed to IDSIA . The controller implements a ROS' node to publish *Krock* status including its pose at a rate of $250hz$. We decide to reduce it $50hz$ by using ROS build it `throttle` command. To load the map into the simulator we had to convert it to Webots's *.wbt* file. Unfortunately, the simulator lacks support for heightmap so we had to use a script to read the image and perform the conversion.

cite them?

cite?

To communicate with the simulator, Webots exposes a wide number of ROS services, similar to HTTP endpoints, with which we can communicate. 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 perform this call automatically making the code cleaner and more intuitive.

link

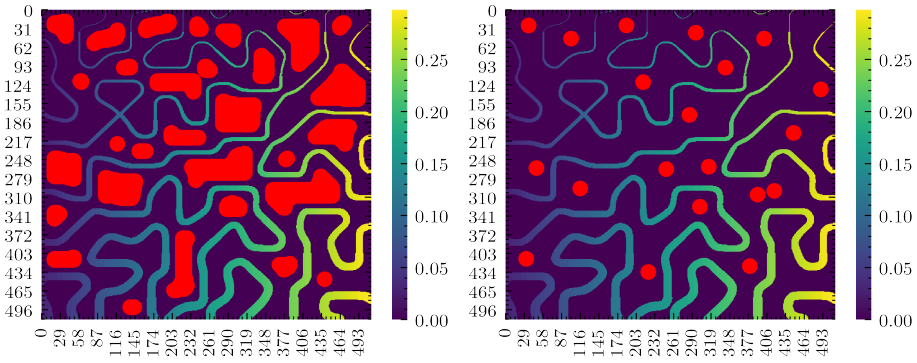
We also implement one additional library called *agent* to create reusable robot's interfaces independent from the simulator. The package supports callbacks that can be attached to each agent adding additional features. Finally, we used *Gym* [**gym**], a toolkit to develop and evaluate reinforcement 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 code.

link

5.2.3 Simulation

To collect *Krock*'s interaction with the environment, we spawn it on the ground and let it move forward for 20 seconds. We repeat this process fifty times per each map. Unfortunately, spawning the robot is not a trivial task. In certain maps, for instance, *bars1*, we must avoid spawning on an obstacle otherwise the run will be ruined because *Krock* will be stuck form the start and we will introduce unwanted noise in the dataset. To solve the problem, we define a random spawn strategy used in most of the maps without big obstacles such as $slope_rocks$, and a flat ground spawn strategy for the others. The random spawn just selects a random position and rotation for the robot. On the other hand, the flat ground strategy first selects suitable spawn positions by using a sliding window on the heightmap of size equal *Krock*'s footprint and check if the mean pixel value is lower than a small threshold. If so, we store the center coordinates of the patch. Intuitively, if a patch is flat the mean valu will be close to zero.

We clustered those points with K-Means in k clusters where k is the number of spawning points we need, in our case fifty. By clustering, we guarantee to cover all region of the map. The following picture shows this strategy on *bars1*.



(a) Flat regions

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

Figure 13: Selecting 20 spawning points in *bars1*

5.3 Postprocessing

We now need to extract the patches for each pose p_t of *Krock* and compute the advancement for a given time window. To create the post-processing pipeline, we create 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

link to the project

First, we turn each *.bag* file to a pandas dataframe and cache them into *.csv* files. We used an open source library we ported to python3 to perform the conversion.. Then, we load the dataframes with the respective heightmaps and start the data cleaning process. We remove the rows corresponding tp the first second of the simulation time to account for the robot spawning time. Then we eliminate all the entries where the *Krock* pose was near the edges of a map, we used a threshold of 22 pixels since we notice *Krock* getting stuck in the borders of the terrain during a simulation. After cleaning the data, we convert *Krock*quaternion rotation to Euler notation using the *tf* package from ROS. Then, we extract the sin and cos from the Euler orientation last component and store them in a column. Before caching again the resulting dataframes into *.csv* files, we convert the robot's position into heightmap's coordinates used later to crop the correct region of the map.

library that converts the bags file

link to tf transform

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

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

figure that shows Krock somewhere with the patch bounding boxed

. Lastly, we create a final dataframe containing the map coordinates, the advancement, and the patches paths for each simulation and store them to disk as *.csv* files.

The whole pipeline takes less than one hour to run the first time with 16 threads, and, once it is cached, less than fifteen minutes to extract the patches.

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

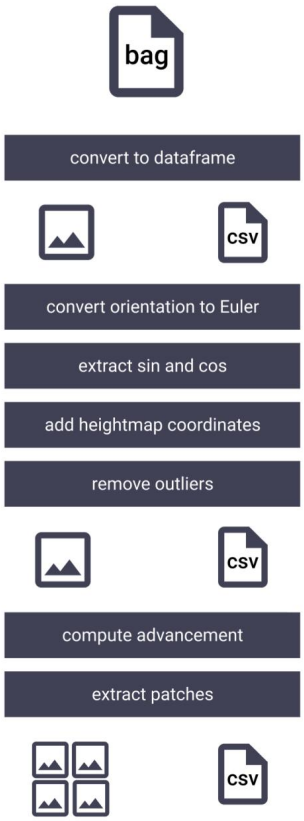


Figure 14: Postprocessing

5.4 Estimator

5.4.1 Vanilla Model

5.4.2 ResNet

We decide to use 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, it is defined as:

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

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

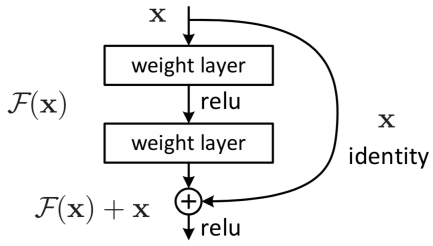


Figure 15: *Resnet* block.

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.

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

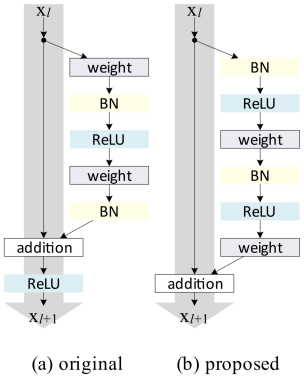


Figure 16: *Preactivation*

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. The next figure visualises the SE module.

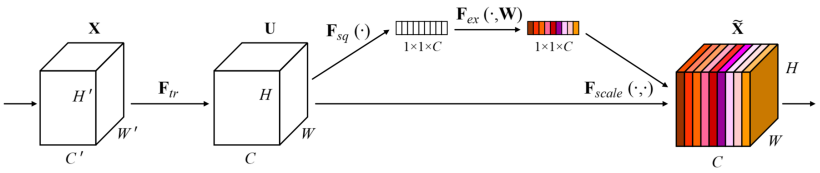


Figure 17: *Preactivation*

Our network differs is composed by n ResNet blocks, a depth of d and a channel incrementing factor of 2. We select $n = 1$ and $d = 3$ with a starting channel size of 16, we called this model architecture *micro-resnet*.

micro resnet
depth = 3
n = 1
3 x 3, 16 *stride* 1
2 x 2 max-pool
$\begin{bmatrix} 3\ x\ 3, & 16 \\ 3\ x\ 3, & 16 \end{bmatrix}$ x 1
SE-module
$\begin{bmatrix} 3\ x\ 3, & 32 \\ 3\ x\ 3, & 32 \end{bmatrix}$ x 1
SE-module
$\begin{bmatrix} 3\ x\ 3, & 64 \\ 3\ x\ 3, & 64 \end{bmatrix}$ x 1
SE-module
average pool, 1-d fc, softmax

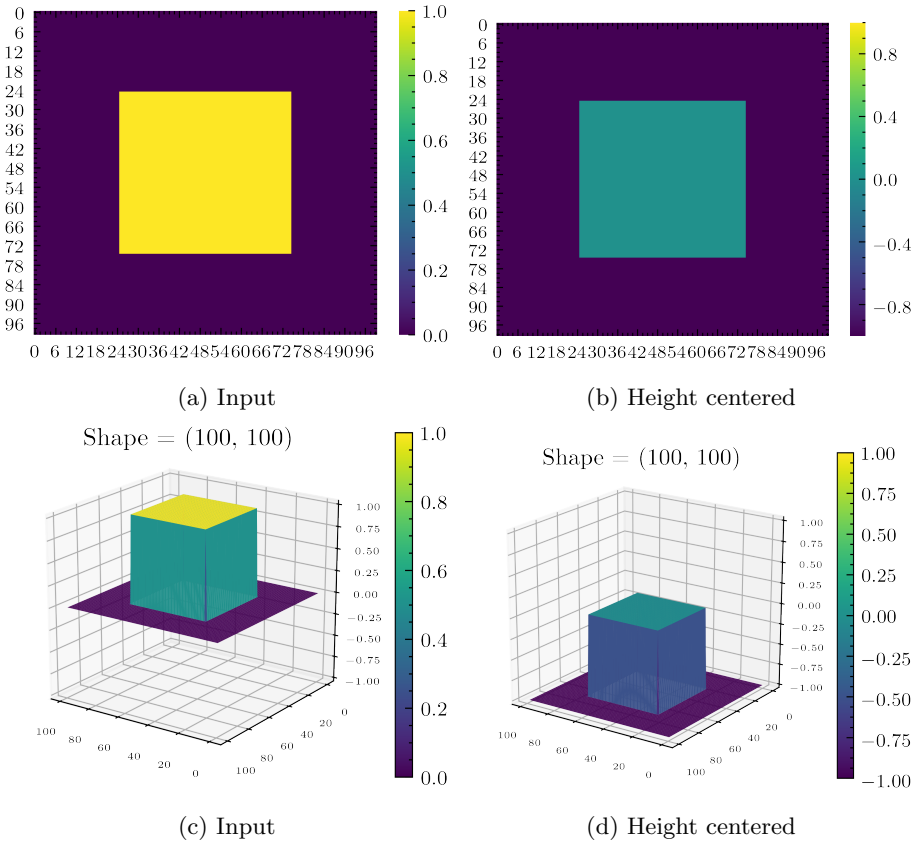
Table 1: *micro-resnet* architecture

add model picture

5.4.3 Normalization

Before feeding the data to the models, we need to make the patches height invariant. This must be done to correctly normalize different patches taken from different maps with different height scaling factor. We subtract the height of the map corresponding *Krock*'s position from the patch to correctly center

it. The following figure shows the normalization process on the patch with the square in the middle.



(a) Input (b) Height centered

(c) Input (d) Height centered

Figure 18: Normalization process

5.4.4 Data Augmentation

Data augmentation is used to change the input of a model using different techniques to change it in order to produce more training examples. Since our inputs are heightmaps we cannot utilize the classic image manipulations such as shifts, flips, and zooms. Imagine that we have a patch with a wall in front of it, if we random rotate the image the wall may go in a position where the patch it is now traversable but its label is still not traversable, we have to be more creative. 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 following example patch of size $100x100$.

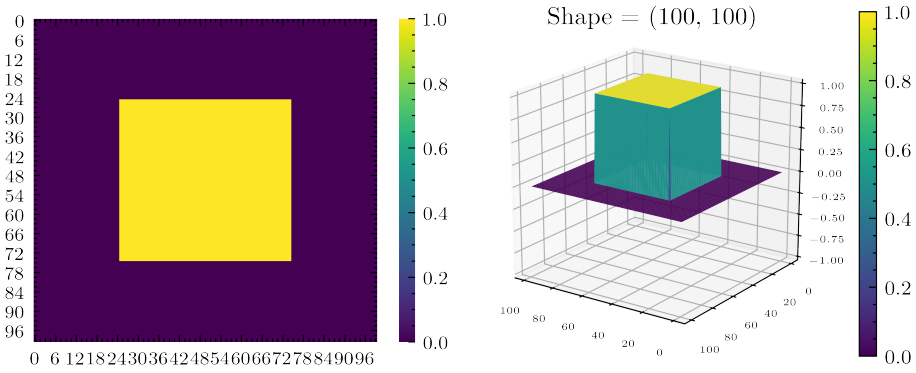
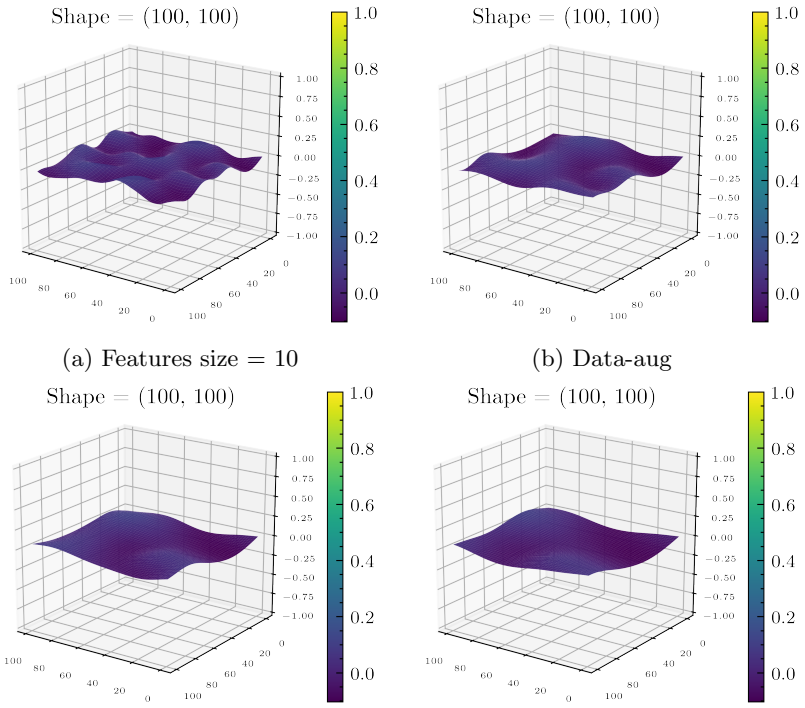


Figure 19: A patch with a square in the middle

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

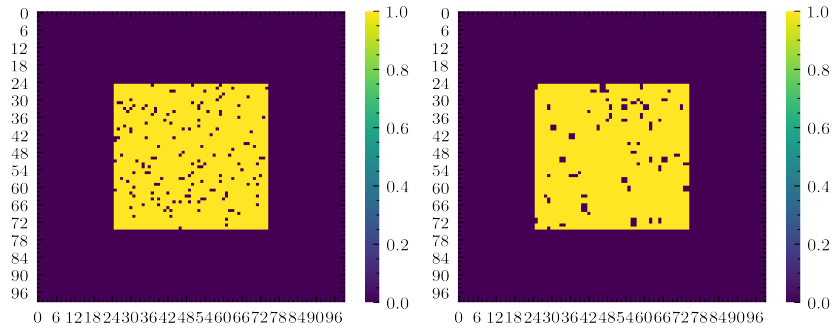
(b) Data-aug

(c) Features size = 30

(d) Features size = 40

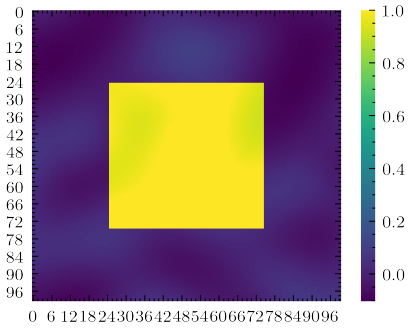
Figure 20: Simplex Noise on flat ground

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

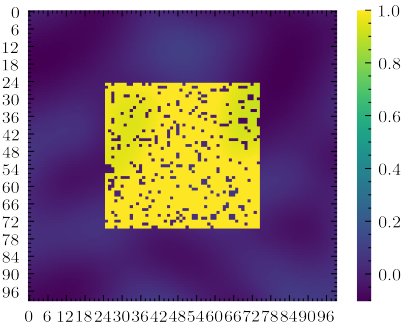


(a) Dropout

(b) Coarse Dropout



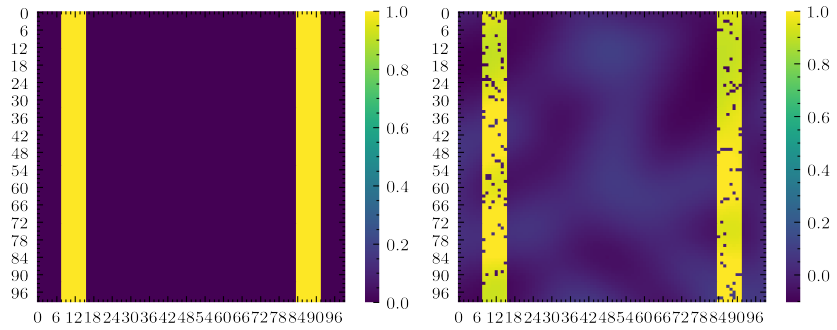
(c) Simplex Noise



(d) Final result

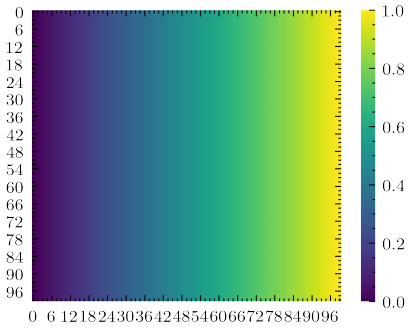
Figure 21: Data augmentation

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

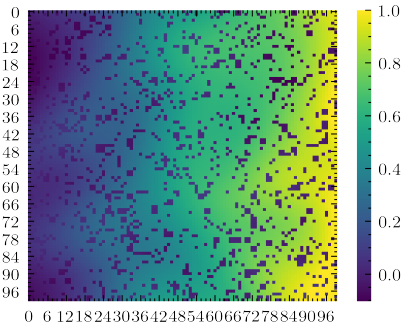


(a) Walls

(b) Walls aug



(c) Ramp



(d) ramp aug

Figure 22: Wall

add the parameters that we set

In all the traning epochs, we apply data-augmentation to each input image x with a probability of 0.8. Dropout has a probability between 0.05 and 0.1. Coarse dropout with a probability of 0.02 and 0.1 with a size of the lower resolution image from which to sample the dropout between 0.6 and 0.8. Simplex noise with a feature size between 1 and 50 with a random scaling factor between 6 and 10. factor between 6 and 10.

6 Results

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

6.0.1 Experiment Setup

We run all the experiment on a work station with Ubuntu 18.10 operating system. The machine is equipped with a Ryzen 2700x, a powerful CPU with 8 cores and 16 threads, and a NVIDIA 1080 GPU with 8GB of dedicated RAM. For the classification task, we select a threshold of $0.2m$ on a time window of two seconds to label the patches, meaning that a patch with an advancement less than 20 centimeters is labeled as *no traversable* and viceversa. We minimise the binary Cross Entropy. On the other hand, for regression, we did not label the patch and directly regress on the advancement for a given time window while minimising the Mean Square Error (MSE). To train the network we follow the best practice on residual network [**he2015deep**] using Standard Gradient Descent with momentum set to 0.95 and weight decay to $1e-4$ with an initial learning rate of $1e-3$. We fix the maximum number of train epochs to 30 and reduce the learning rage on platuoe by a factor of 0.1 with a patience of 4. We used early stopping to stop the training if the validation accuracy does not increase in 6 epochs.

fix this typo

6.0.2 Experimental validation

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

add more maps if we add them to the test set

6.0.3 Metrics

To evaluate the model's classification performance we used two metrics: *accuracy* and *AUC-ROC Curve*. Accuracy scores the number of correct predictions made by the network while AUC-ROC Curve represents degree or measure of separability, informally it tells how much model is capable of distinguishing between classes. For each experiment, we select the model with the higher AUC-ROC Curve during training to be evaluated. For the regression we just rely on the loss value.

6.1 Quantitative Results

The following table shows the final results on various test dataset made by using real-world heightmaps.

table with results

Moreover, we would like to also show the different steps we made to reach this result. The following table shows the metric's score without any data-augmentation. Adding dropout increases the results.

table with results

With dropout plus coarse dropout.

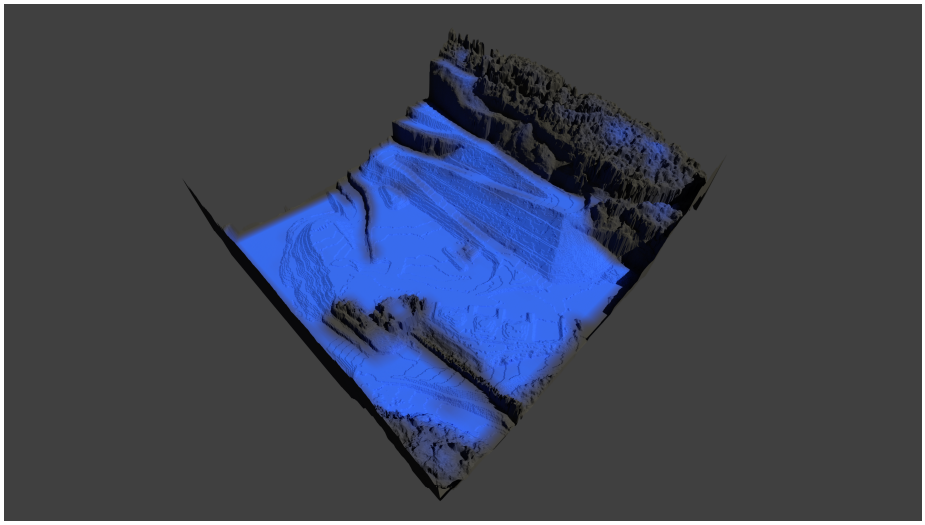
table with results

6.2 Qualitative results

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

add quarry textures from bottom to top

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



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

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

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

7 Interpretability

References

[1] Jérôme Guzzi Alessandro Giusti et al. "A Machine Learning Approach to Visual Perception of Forest Trails". In: (2016).

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

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

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

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

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

[7] Olivier Michel. "Cyberbotics Ltd. WebotsTM: Professional Mobile Robot Simulation." In: ().

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

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

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