1 Implementation

1.1 Tools

The most important tools and libraries used in our work were:

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

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

1.1.1 ROS Melodic

The Robot Operating System (ROS) [**ROS**] is a flexible framework for writing robot software. It is *de facto* the industry and research standard framework for robotics due to its simple yes effective inferface that facilitates the task of creating robust and complex robot behavior regardless of the platforms. ROS work by generate a peer-to-peer connection where each *node* is to communicate between the others by exposing sockets endpoints to stream data called *topics*.

Each *node* can subscribe to received the incoming messages or publish new data on a specific *topic*. In our case, *Krock* exposes different topics, for example `/pose`, in which we can subscribe in order to get the real time informations about the state of the robot. Unfortunately, ROS does not natively support Python3, so we had to compile it by hand. Since it was not difficult and time consuming operation we decided to share the ready-to-go binaries as docker image.

where should I place the link to docker

1.1.2 Numpy

Numpy is a fundamental packages 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 utilised *Numpy* manipulate matrices in an expressive and efficient way.

scrape some text from the previous section

1.1.3 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 customise every region of the figure. Almost every 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.

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

1.1.5 OpenCV

Open Source Computer Vision Library, OpenCV, is an open source computer vision library with a rich collection of highly optimized algorithms. It includes classic and state-of-the-art computer vision and machine learning methods applied in a wide array of tasks, such as object detection and face recognition. With a huge community of more than fourtyseven 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.

1.1.6 PyTorch

PyTorch is 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 autograd system. Due to its *Python-first* philosophy it has a deep integration into Python allows popular libraries and packages to be used, such as *OpenCV* or *Pillow*.

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

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

1.1.7 FastAI

FastAI is library based on PyTorch that simplifies fast and accurate neural nets training using modern best practices. It provides a high-level API to create train, evaluate and test deep learning models on any type of dataset.

1.1.8 imgaug

Image augmentation (imgaug) is a python library to perform image augmenting operations on images. It provides a variety of methodologies, such as affine transformations, perspective transformations, contrast changes and gaussian noise, to build sophisticated pipelines. It supports images, heatmaps, segmentation maps, masks, keypoints/landmarks, bounding boxes, polygons and line strings.

1.1.9 Blender

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

1.2 Data Gathering

1.3 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 convert each *.bag* file to a Pandas dataframe and cache them into *.csv* files. We used an open source library that converts the bags file to dataframe to Python3 that we ported to Python3.. Then, we load the dataframes with the respective heightmaps and start the data cleaning process. We need to remove the rows in 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 simulation. After clearning the data, we convert *Krock* rotation express as a quaterion to euler angle using the *tf* packes from ROS. Then, we extract the sin and cos from the Euler's last component and store them in the dataframe. Before caching again resulting dataframes into *.csv* files, we convert the robot's position into heightmap's coordinates to use them later to crop the correct region of the map.

library that convert the bags file

link to tf transform

Finally, we load again the last stage and compute the advancement by project 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 seclet two seconds we need to multiple 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 a 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 on. With these informations we can now crop the coresponding region in the heightmap by including the whole *Krock*'s footprint and the maximum advancement. The following figure visualise the patch extraction process.

For each simulation run, a dataframe containing the map coordinates, the advancement and the patch path is created.

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

Once we extract the patches, we can always compute again the advancement without re-runnig the whole pipeline.

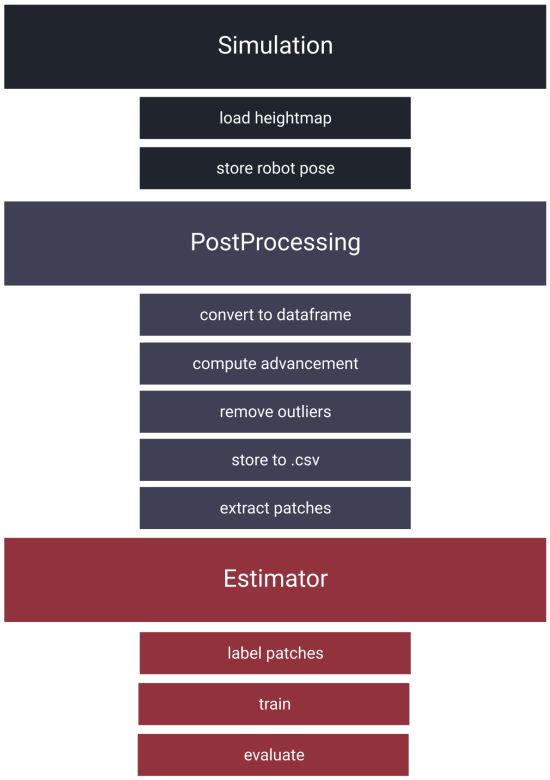


Figure 1: Pipeline

1.4 Estimator

1.4.1 Vanilla Model

1.4.2 ResNet

We decide to use a Residual Network, ResNet [**he2015deep**], variant. Residual network are deep convolutional networks consisting of many stacked R̈esidual Units ∴ Intuitively, the residual unit allows the input of a layer to contribuite to the next layer's input by beeing added to the current layer's output. Due to possible different features dimension, the input must go thought 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 $\mathcal{F}\left(\mathbf{x}, \{W_i\}\right)$ is the residual mapping to be learn and h is the identity mapping. The next figure visualises the equation.

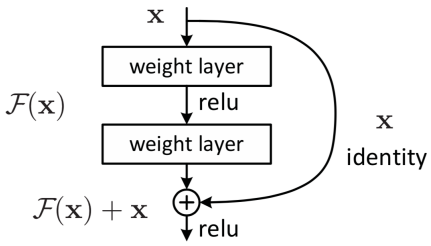


Figure 2: *Resnet* block.

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.

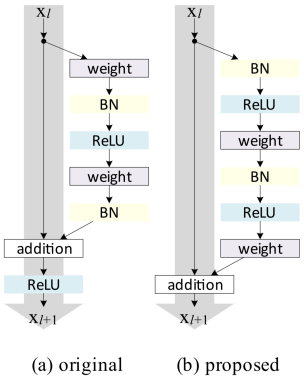


Figure 3: *Preactivation*

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.

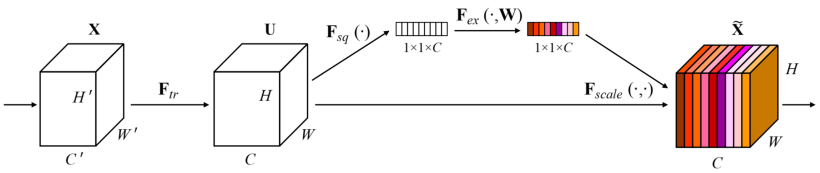


Figure 4: *Preactivation*

add model picture

1.4.3 Normalization

Before feeding the data to the models, we need 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 krock's position height from the patch to correctly center the height.. The following figure shows the normalization process on the patch with the square in the middle.

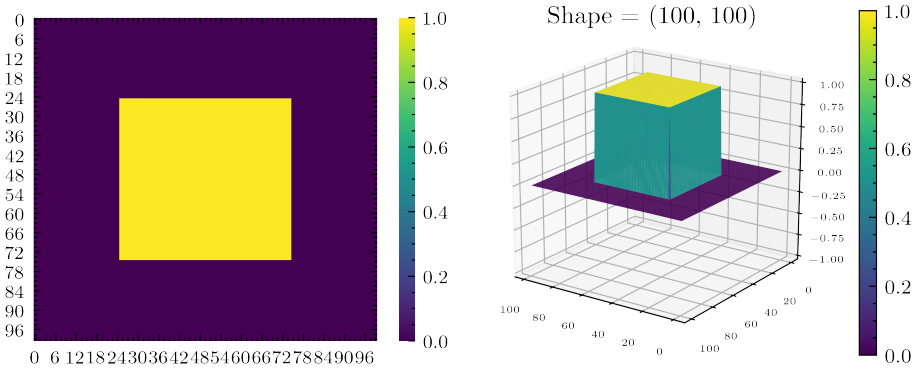
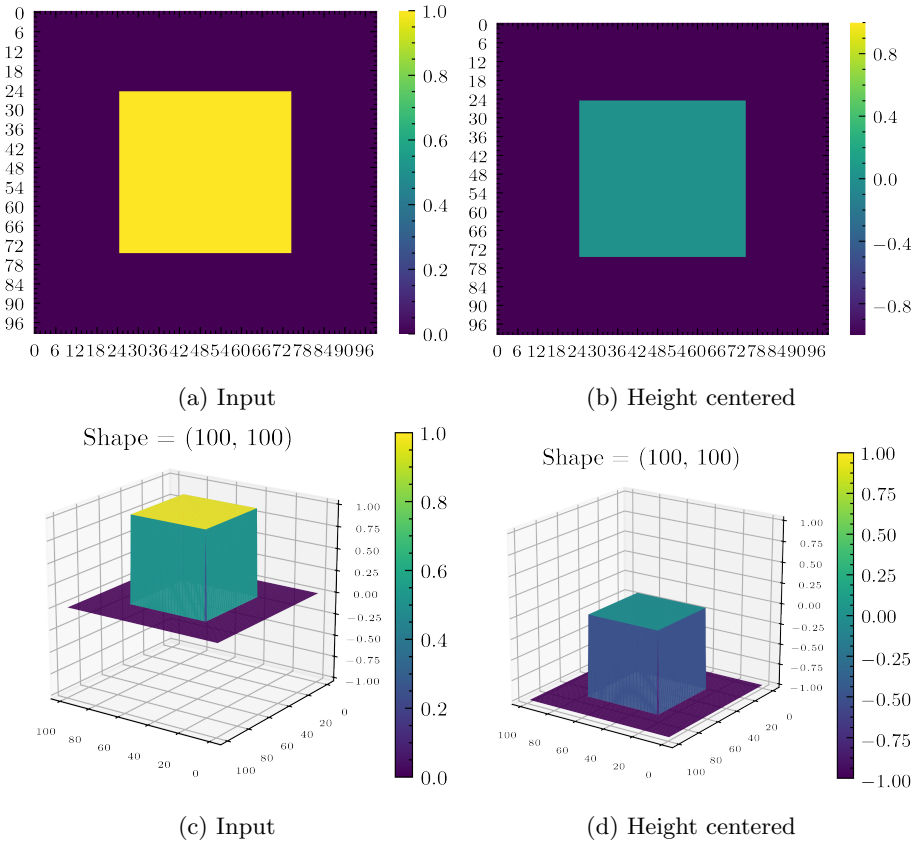


Figure 5: A patch with a square in the middle

1.4.4 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 utilise 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 $100x100$.



(a) Input (b) Height centered

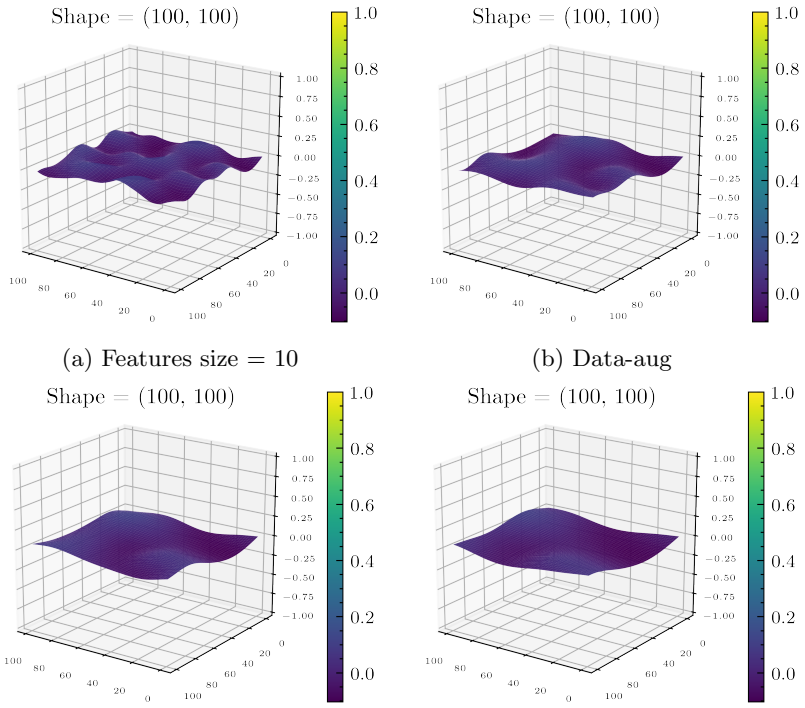
(c) Input (d) Height centered

Figure 6: Normalization process

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 generalise better since lots of training maps have only obstacles in flat ground. Since it is computational 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

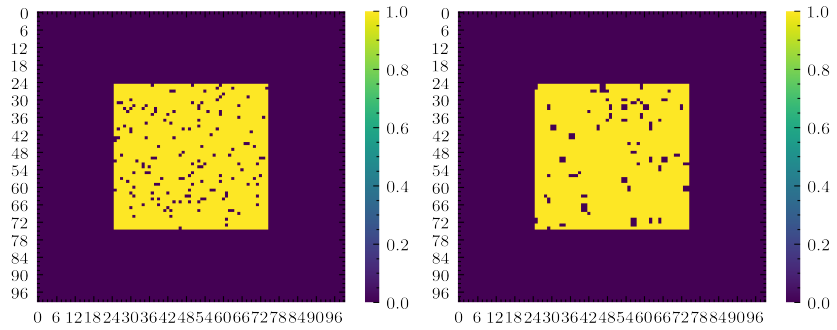
(b) Data-aug

(c) Features size = 30

(d) Features size = 40

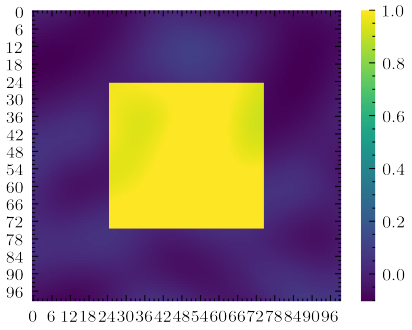
Figure 7: Simplex Noise on flat ground

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

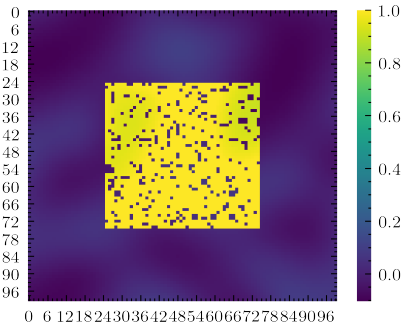


(a) Dropout

(b) Coarse Dropout



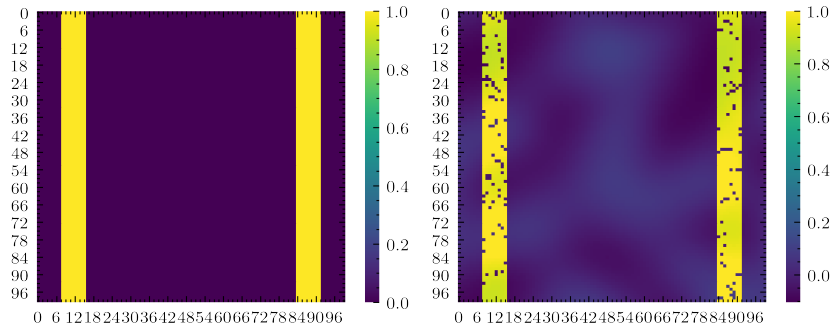
(c) Simplex Noise



(d) Final result

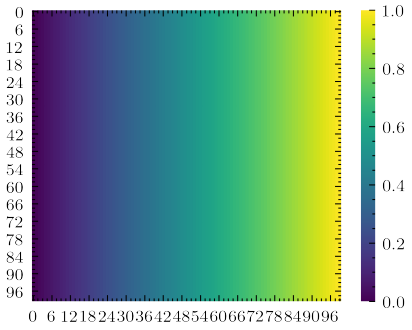
Figure 8: Data augmentation

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

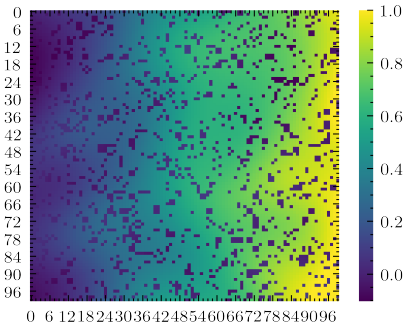


(a) Walls

(b) Walls aug



(c) Ramp



(d) ramp aug

Figure 9: Wall

add the parameters that we set