1 Implementation

1.1 Tools

The most important tools and libraries adopted to build this project 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 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.

where should I place the link to docker

1.1.2 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.

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 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.

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 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.

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 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.

1.1.6 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 *OpenCV* 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.

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 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, key points/landmarks, bounding boxes, polygons, and line strings.

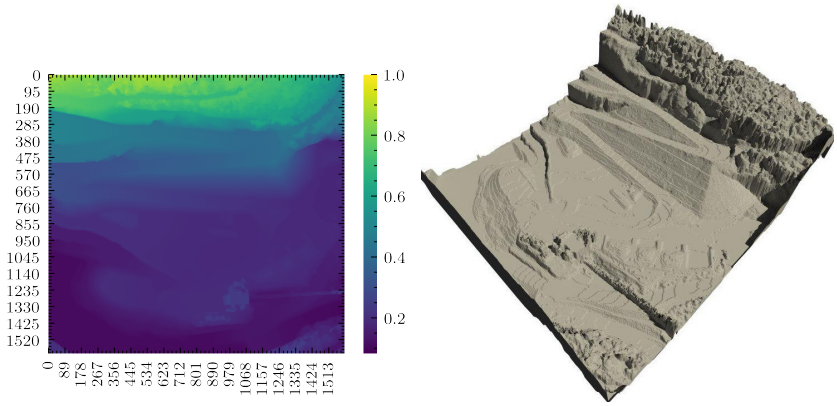
1.1.9 Blender

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 utilized to evaluate the trained model.

1.2 Data Gathering

1.2.1 Heightmap generation

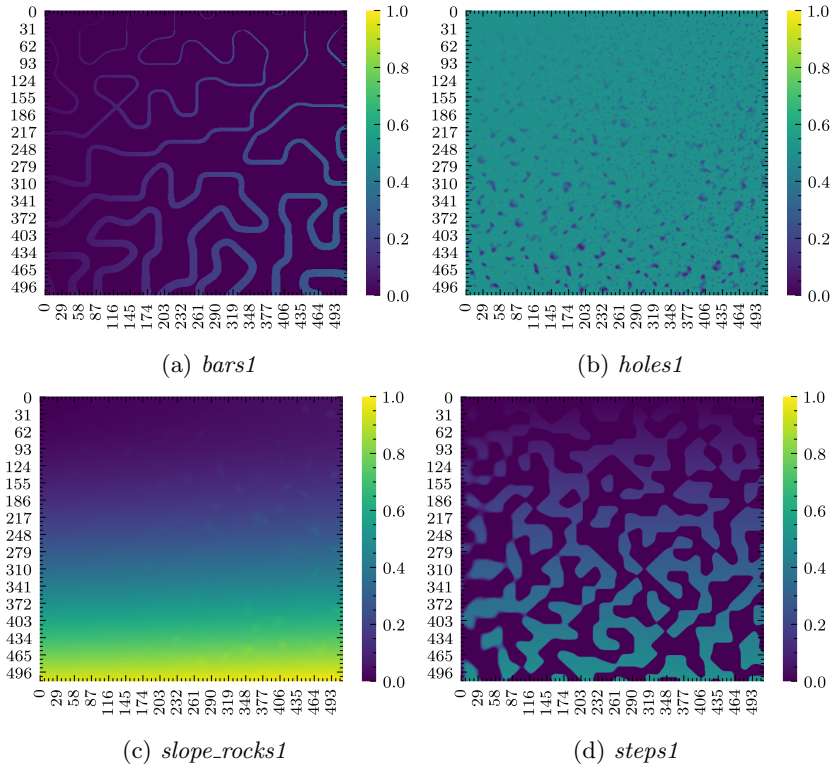
A heightmap is a 2D array, an image, where each pixel's value represents the terrain height.



(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 2: Some of the synthetic heightmaps

1.2.2 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

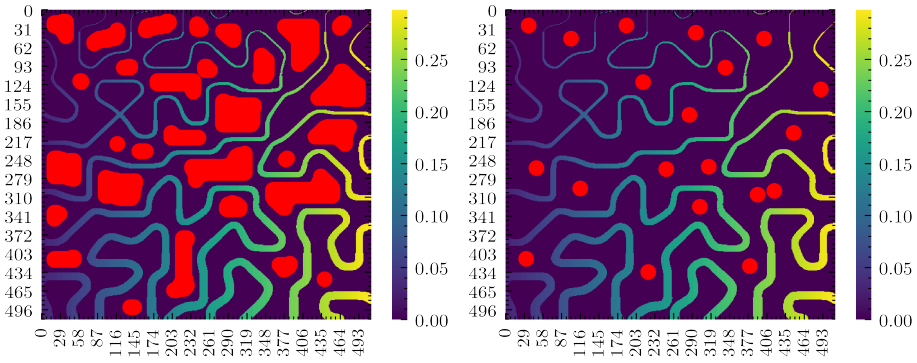
We also implement one additional library called *agent* to create reusable robot's interfaces independent from the simulator. The package supports callbacks that can be attached to each agent adding additional features. Finally, we used *Gym* [**gym**], a toolkit to develop and evaluate reinforcement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

1.2.3 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 3: Selecting 20 spawning points in *bars1*

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 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.

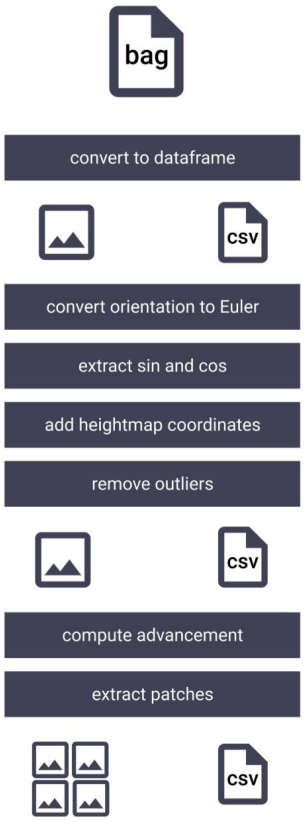


Figure 4: Postprocessing

1.4 Estimator

1.4.1 Vanilla Model

1.4.2 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.

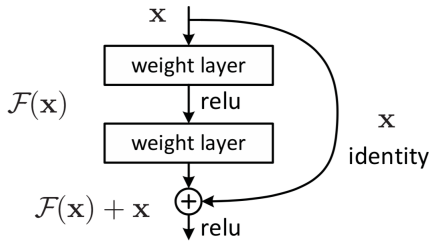


Figure 5: *Resnet* block.

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.

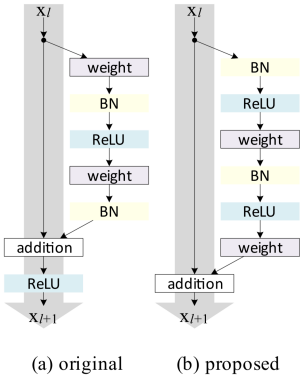


Figure 6: *Preactivation*

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.

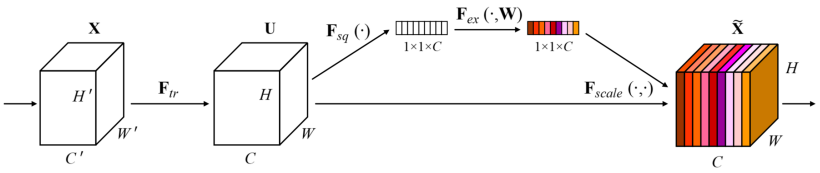


Figure 7: *Preactivation*

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

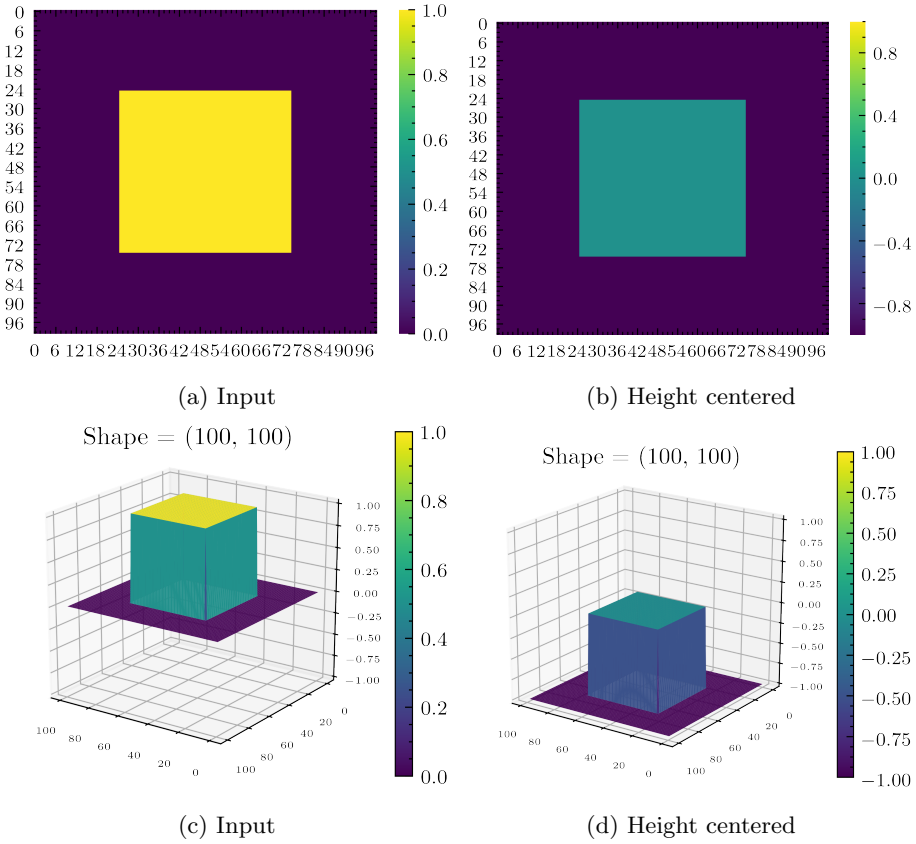
Table 1: *micro-resnet* architecture

add model picture

1.4.3 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 8: Normalization process

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 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 $100x100$.

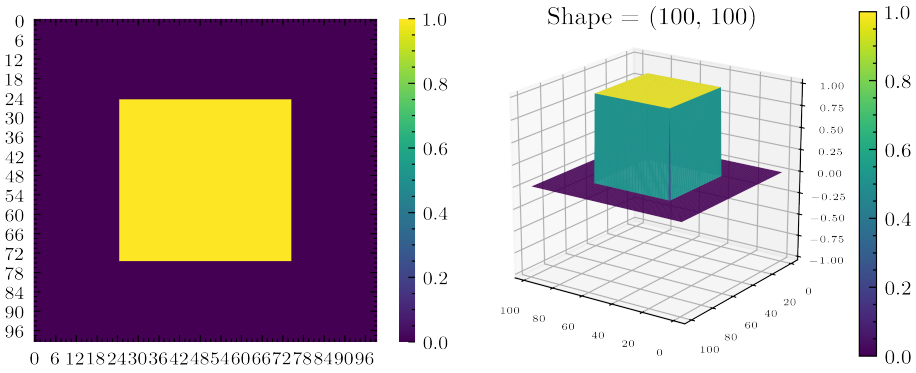
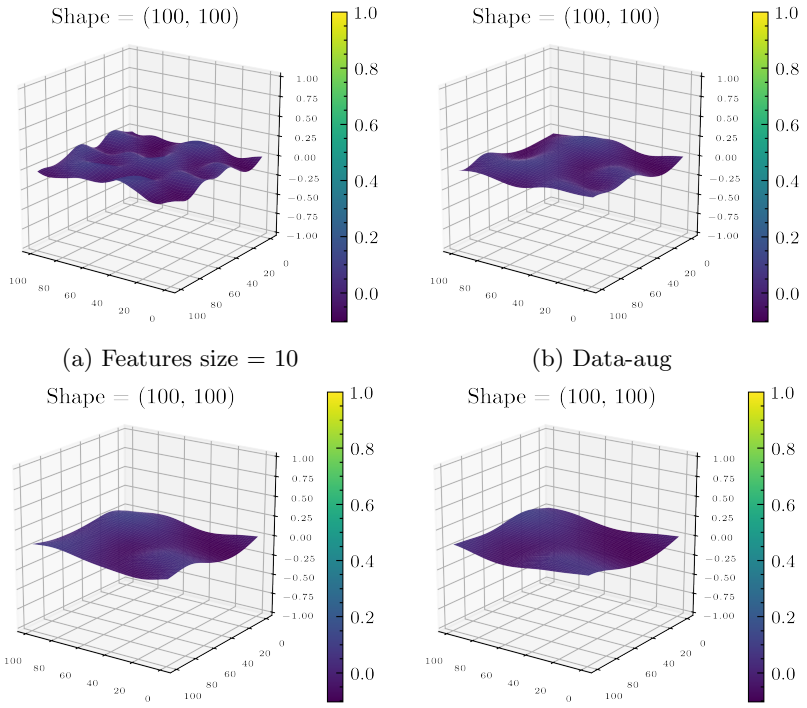


Figure 9: A patch with a square in the middle

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

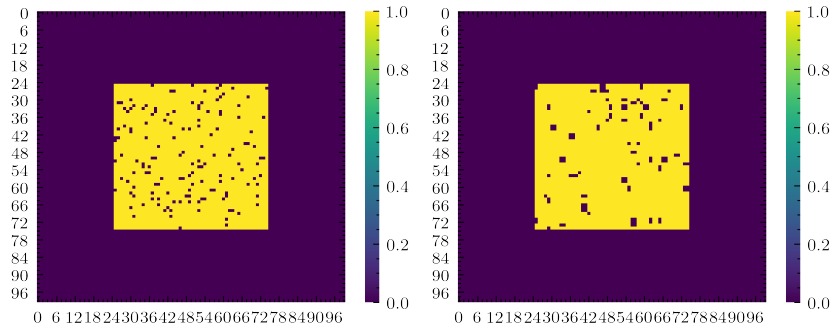
(b) Data-aug

(c) Features size = 30

(d) Features size = 40

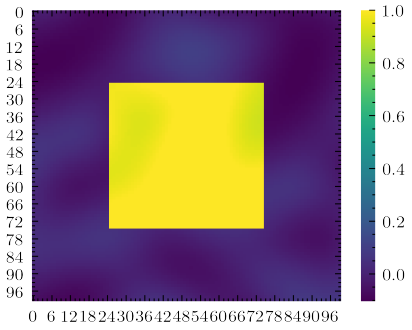
Figure 10: Simplex Noise on flat ground

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

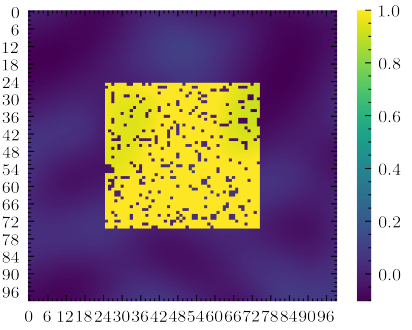


(a) Dropout

(b) Coarse Dropout



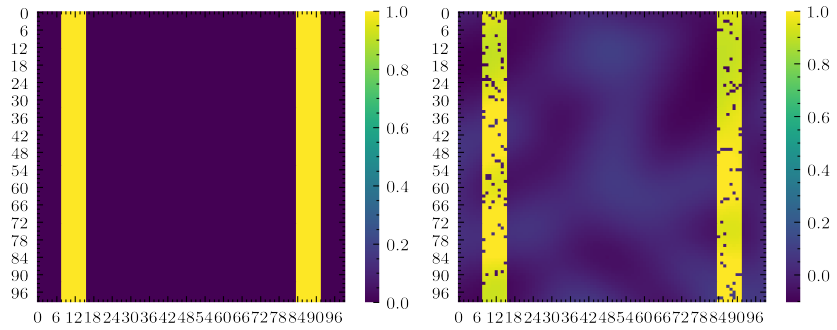
(c) Simplex Noise



(d) Final result

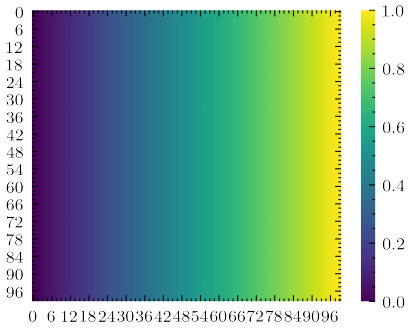
Figure 11: Data augmentation

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

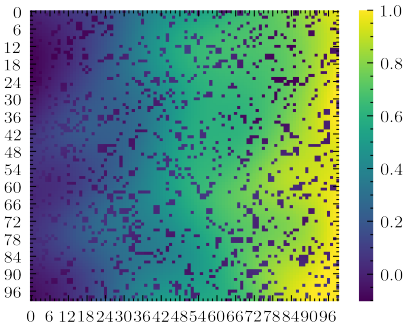


(a) Walls

(b) Walls aug



(c) Ramp



(d) ramp aug

Figure 12: 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.