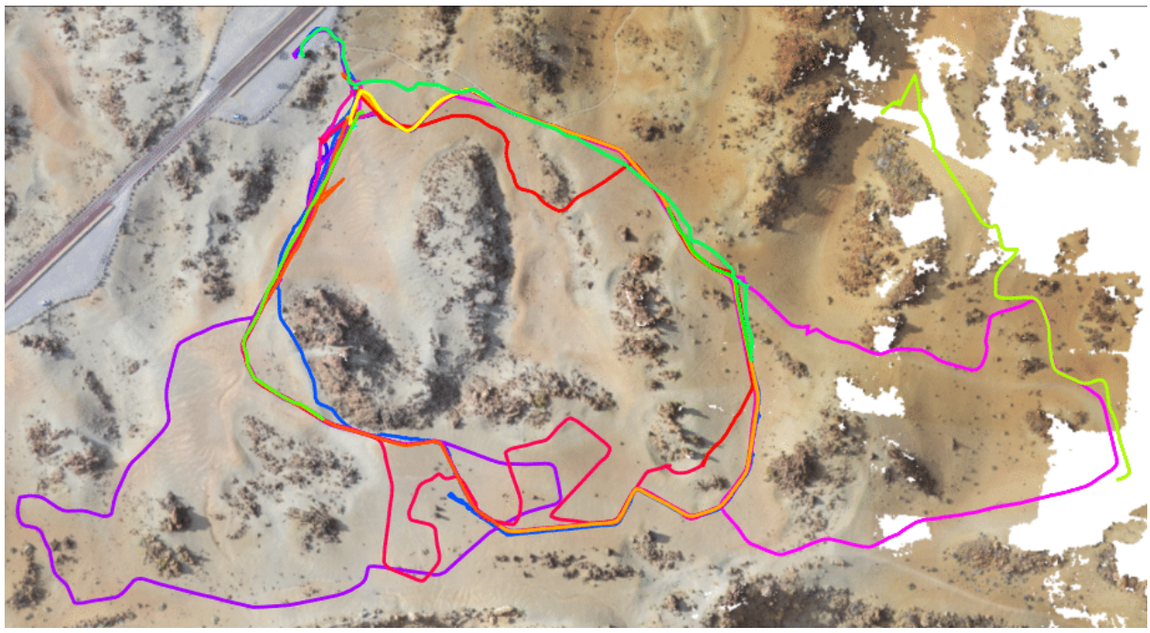
Improving Global Localization Algorithms for Mars Rovers with Neural Networks



Master Thesis Report

Iñigo Moreno i Caireta

Aalborg University
Robotics MSc.

This report was written on the Overleaf website using the LaTeXtypesetting format and the AAU report template v. 1.2.0. The citations are written in the IEEE format. The figures and plots are generated using matplotlib.

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University (AAU)
https://www.aau.dk

Automation and Robotics Section
European Space Agency (ESA)
https://www.esa.int

Title:
Improving Global Localization Algorithms for Mars Rovers with Neural Networks

Theme:
Localization and Machine Learning

Project Period:
Spring Semester of 2021

Project Group:
1069

Participant(s):
Iñigo Moreno i Caireta

Supervisor(s):
Chris Holmberg Bahnsen (AAU)
Levin Gerdes (ESA)
Martin Azkarate (ESA)

Copies: 1

Page Numbers: 38

Date of Completion:
2021/06/03

Abstract:

This thesis investigates the possibility of using a Siamese Neural Network in order to create an algorithm for global localization in the context of Mars rovers. The thesis details the whole process of creating the Neural Network: from the acquisition and processing of two datasets to the creation of the model and the tuning of its hyperparameters. Then, the model is tested and its resource usage analyzed. The whole model only takes up to 9 megabytes of space and is able to give predictions in 18 milliseconds. The thesis also shows how the model can be used in a global localization algorithm by implementing a sliding window approach that can be compared with previous works. This sliding window approach shows some promising results and seems to perform better than the previous solution.

Contents

Authors

Iñigo Moreno i Caireta
imoren19@student.aau.dk

Chapter 1

Introduction

1.1 Problem Analysis

Due to the communication constraints, rovers on the surface of Mars can only be controlled once a day and as such they require a high level of autonomy. An important task needed to achieve this autonomy is self-localization. While robotics applications on earth can use external sources of information such as a GPS system or known landmarks in earth, a rover on Mars is mostly on its own and must therefore be able to determine its own location in order to successfully navigate the terrain and perform the scientific missions on the surface. Visual and inertial odometry combined with a Simultaneous Localization And Mapping (SLAM) algorithm can be used by the rover to self-localize. However, these methods only provide the location relative to a previous location. This means that the small errors done by these algorithms will accumulate over time and the rover will lead to large errors. To solve this, the rovers need a way to achieve global localization, where an absolute measurement of the rover's position with respect to the planet can be done.

Another point to consider is that existing rovers in Mars move fairly slowly (around 25 m per martian day), while future missions such as the Sample Fetching Rover (SFR) will need to move at higher speeds (210 m per martian day) [1]. This increase in speed means that these future missions will need to rely less on human intervention and more on automated algorithms in order to perform global localization.

Luckily, there have been a few satellite missions to Mars and rover missions usually put a satellite in an orbit close to the rover to be able to relay data back to earth. The data collected by these satellites can then be used to generate an absolute map of the environment around the rover. Then, the rover can then try to find a match between this global map and the map generated by the SLAM algorithm. If a match is found, the global position can be corrected. This is done in [2] by using a sliding window approach, where a downscaled version of the local map is compared against all possibilities in the global map in order to find the best match. However, this algorithm is time consuming and computationally expensive, which means that it is only used when necessary. This is especially important when considering that the algorithm will eventually need to run on the on-board computers of the rover, which are very limited in terms of computing power and memory.

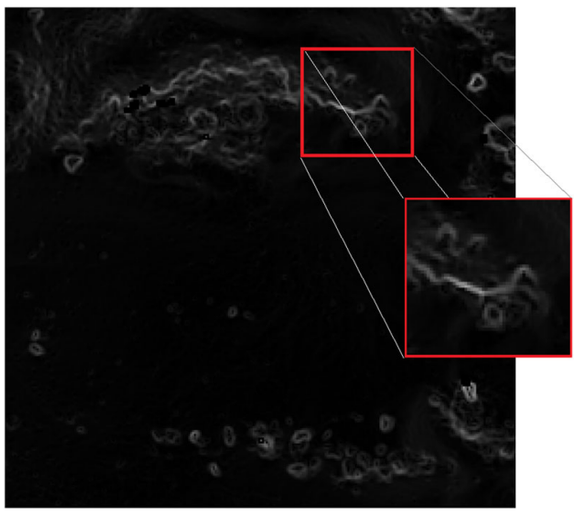


Figure 1.1: Graphical illustration of how template matching is performed in [2]

1.2 State of the Art

There have been many attempts at solving the global localization problem for martian rovers. On one of the earliest examples of this is the Sojourner rover, which landed on Mars in 1997. On this rover, human operators had to manually determine the location of the rover based on the images received from it [3]. However, during that time some early attempts of performing autonomous global localization appeared. Olson and Matthies [4] attempted to locate the rover's position by matching a local map generated from the robot cameras with a global map generated by the lander's cameras using a probabilistic approach.

Since then, a lot of different approaches have appeared to solve this problem. One of them is Doppler tracking, where the Doppler effect in signals sent from the rover and the earth or other spacecraft such as an orbiter is studied in order to estimate the position of the rover [5]. This solution is the least accurate, but it can locate the spacecraft even without any prior knowledge of its position.

Another solution is skyline matching, where the rover uses its cameras to measure the shape of the horizon line (skyline) and comparing it to a set of simulated skylines generated at different template positions [6].

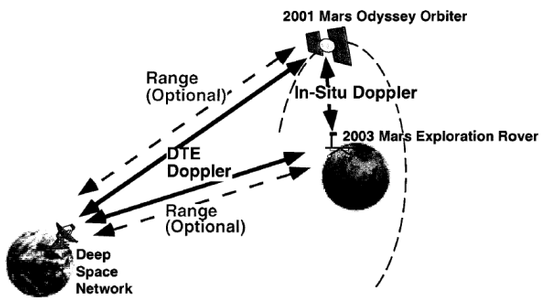
A newer approach is constellation matching. Where sets of features are detected by the rover and the distance between these features is studied to find a matching set of features in a global map. Carle et al. [7] do constellation matching with peaks of the elevation maps as their feature. Boukas et al. [8] do the same but using rocks and outcrops as their features.

A particle filter can be used in conjunction with map matching in order to localize a rover [9]. This works by initializing a set of particles over the possible solution space and then for each particle the correlation score between the local map and the global map cropped to the particle's position is computed. Then, the set of particles is re-sampled close to the particles with the higher score. This is repeated several times until the particles converge to a solution.

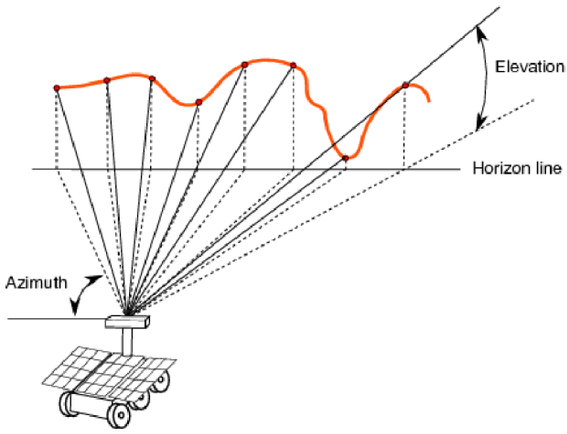
Few approaches exist that use Neural Networks to solve the problem. Naguib et al. [10] try to match images captured by a rover to an expected view of the rover at different locations that are generated using the elevation maps. It uses a Generative Adversarial Network (GAN) to generate fake images and train a Convolutional Neural Network (CNN) discriminator to differentiate between the real and fake images. The output of this discriminator is then used as a score to evaluate different hypotheses.

Wu et al. [11] uses four of each side of the rover and re-projects them onto a ground plane. It then tries to match this re-projection with satellite images by using a Siamese Neural Network (SNN) to classify the images between matching images and non-matching images.

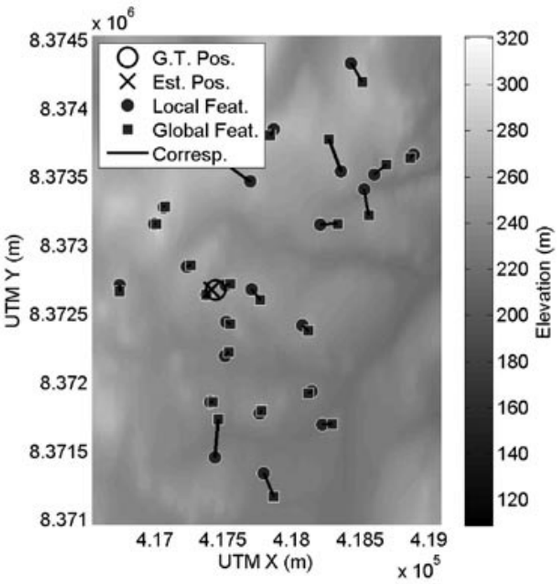
Franchi [12] employs an SNN that tries to match the satellite image directly with a rover image. It couples the output of the SNN with a particle filter in order to converge to a final solution. It also utilizes a GAN to augment the dataset used.



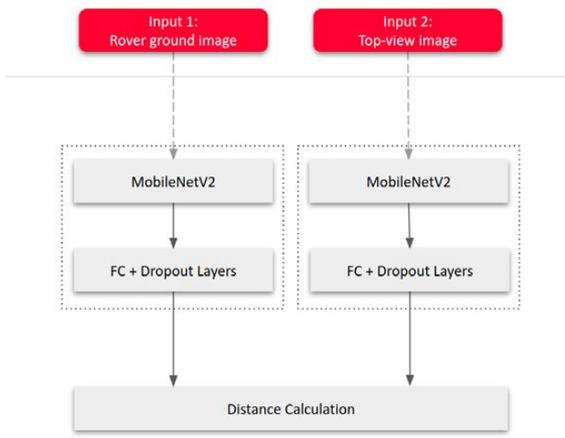
(a) Doppler Effect [5]



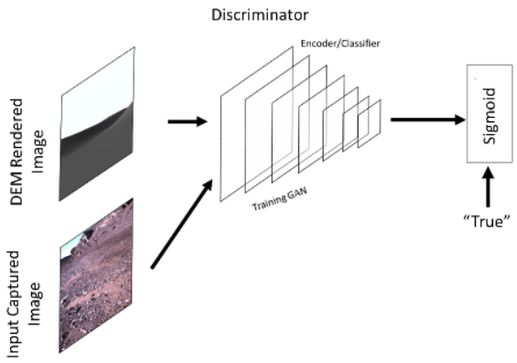
(b) Skyline Measurement [6]



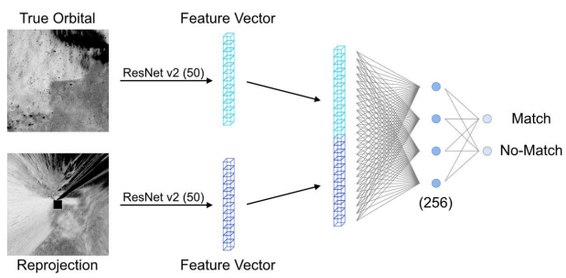
(c) Constellation Matching [7]



(d) SNN classifier [12]



(e) Discriminator CNN [10]



(f) SNN classifier [11]

Figure 1.2: Overview of the different Localization Methods

1.2.1 State of the Art Conclusion

As the State of the Art has shown, while there have been many approaches to this problem, not many of these approaches use Machine Learning. However, from the few Machine Learning approaches found, three methods can be appreciated. In [10] two images are fed into a single CNN to compute the probability. In [12, 11] each image passes through a SNN to extract two embeddings which are then compared. In [12] the embeddings are compared with a distance metric while in [11] the embeddings are compared with a separate Neural Network that tries to compute the match probability.

However, all of these approaches try to take two images and compare them. Something similar can be done for our approach but instead of using raw images, we can use the elevation maps. Therefore our approach will try to compare the local elevation map generated by the rover with a patch of the global elevation map.

1.3 Initial Problem Formulation

The goal of this Thesis is to see if Machine Learning can be used to improve global localization from the sliding window algorithm described in Geromichalos et al. [2].

Is it possible to use a Machine Learning algorithm to perform global localization for a Mars rover while being faster and more resource-efficient than the current solution?

In order to answer this question, these steps will be followed:

- **Data preparation:** All Machine Learning methods need big amounts of good quality data, so the first step will be to prepare the data.
- **Model Implementation:** A Machine Learning Model will be constructed that will try to compare a local elevation map with a global elevation map.
- **Model testing:** The Machine Learning Model will be tested and the best hyper-parameters for the Model will be found.
- **Localization Algorithm Implementation:** If the Machine Learning Model is successful, it will be used to construct a localization algorithm.
- **Algorithm Testing:** The localization algorithm will be tested against other methods.

1.4 Requirement specification

In the Problem Formulation, it is stated that the solution should be "faster and more resource-efficient than the current solution". To specify this, a list of requirements is shown here:

- **Accuracy:** The algorithm should lead to a more accurate solution.
- **Time:** The algorithm should take as little time as possible.
- **Memory Usage:** Memory space in rovers is usually limited, therefore we need to use it as little as possible.
- **Communication:** If data needs to be communicated between the rover and the satellite, this needs to be limited.

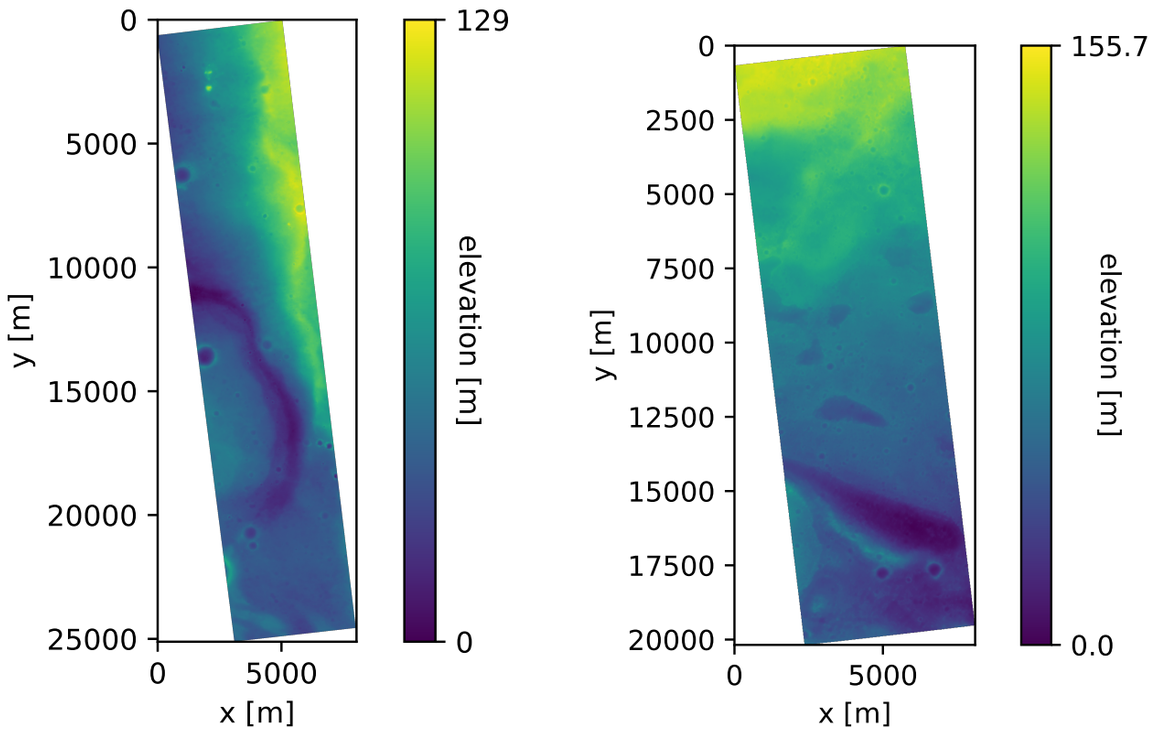
Chapter 2

Data preparation

To train and test any Machine Learning method, big amounts of data are always needed. Luckily, in 2017, European Space Agency (ESA) recorded a dataset with a lot of data from a rover moving in the volcanic landscape of the island of Tenerife, Spain. However, as it took a while to get access to the full dataset, download it and process it, another dataset was also used in the mean time, which contained elevation data from the surface of Mars.

2.1 HiRISE Dataset

High Resolution Imaging Science Experiment (HiRISE) is a very high-resolution camera that rides onboard NASA's Mars Reconnaissance Orbiter (MRO). Its high resolution allows it to get very accurate pictures of the Martian surface. From these images, some algorithms can construct Digital Terrain Models (DTMs) with high accuracy. These elevation maps have a 1–2 m resolution with vertical precision in the tens of centimetres. Many DTMs of different regions of the Martian surface have been already computed by the University of Arizona and they are publicly available on their website[1]. From these, two elevation maps were selected. These maps represent the elevation of regions close to the landing site of two rovers, in order to get representative data of possible traversable zones. These can be seen in figure 2.1.



(a) Oxia Planum (ESA's ExoMars landing site)[2]

(b) Jezero Crater (NASA's Perseverance landing site) [3]

Figure 2.1: HiRISE DTMs

[1] https://www.uahirise.org/dtm/

[1] https://www.uahirise.org/dtm/dtm.php?ID=ESP_037070_1985

[2] https://www.uahirise.org/dtm/dtm.php?ID=ESP_023247_1985

2.1.1 Processing the DTMs

To be able to read the DTMs from python, the `GDAL` library [S1] was used. However, to use these DTMs as training data for our models, some processing needs to be applied to them. Our goal is to first generate a patch that resembles a local elevation map generated by the rover and then a patch of the global map close to the local map.

To generate the local patch a random point within the DTM is selected. Then, a square $80\,\text{m} \times 80\,\text{m}$ patch is extracted from the DTM that represents the elevation of the area around the selected point. If the patch has too many NaN values (because it is outside the elevation map), it is discarded and another point is selected. The local map is then distorted and artificial occlusion is applied to simulate how a local map generated by a rover would look. This will be explained in subsections 2.1.2 and 2.1.3.

To get the global patch, a random point close to the local patch is selected. This shift represents the possible uncertainty of the robot's a priori position. The shift is computed following a two-dimensional normal distribution. As with the local map, a patch of the elevation around this point is extracted from the DTM and it is discarded if it has too many NaN values.

2.1.2 Local Patch Distortion

Local maps will never be the same as the global map. To simulate these differences, once a local patch of the DTM has been extracted, it is distorted by adding some noise to it. Instead of adding random white noise on each pixel, which would cause unnaturally big gradients between individual pixels, simplex noise is used, which has smoother gradients. This is done using the imgaug library [S2]. An example of the simplex noise generated is seen in figure 2.2. This noise has a scale between zero and one and can be multiplied by a scaling factor to decide how much noise to apply. For this thesis, a scale of 0.2 m of noise was used.

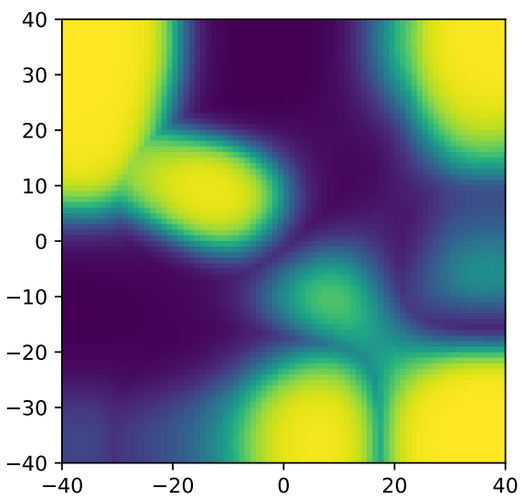


Figure 2.2: Simplex noise example

2.1.3 Artificial Occlusion

Local rovers are not able to see the whole terrain because part of the terrain will be occluded. This means that the local DTM generated by the rover is usually incomplete. To simulate this, an algorithm was written that, for each pixel in the DTM, checks if the pixel would be occluded from the rover.

The algorithm works by, first assuming that the robot is at the middle of the patch, at some fixed elevation above the ground. Then, the algorithm iterates over every pixel of the DTM to see if its three-dimensional position (using the elevation from the DTM) would be visible from the robots point of view. To check this, the algorithm needs to cast a three-dimensional ray from the robot to the terrain and see if there is any obstruction along the way that would cause that part of the DTM to be occluded.

See figure 2.3 for a visual explanation of the algorithm and figure 2.4 to see the algorithm applied to a patch. As this algorithm was very costly (nearly two seconds per patch) it was

optimized to run on the GPU using Numba [S3]. The full code can be seen on GitHub [4].

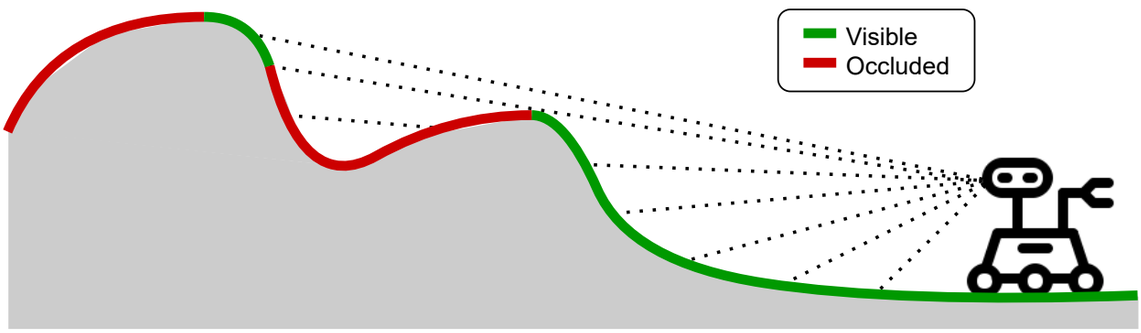


Figure 2.3: Simulated occlusion algorithm

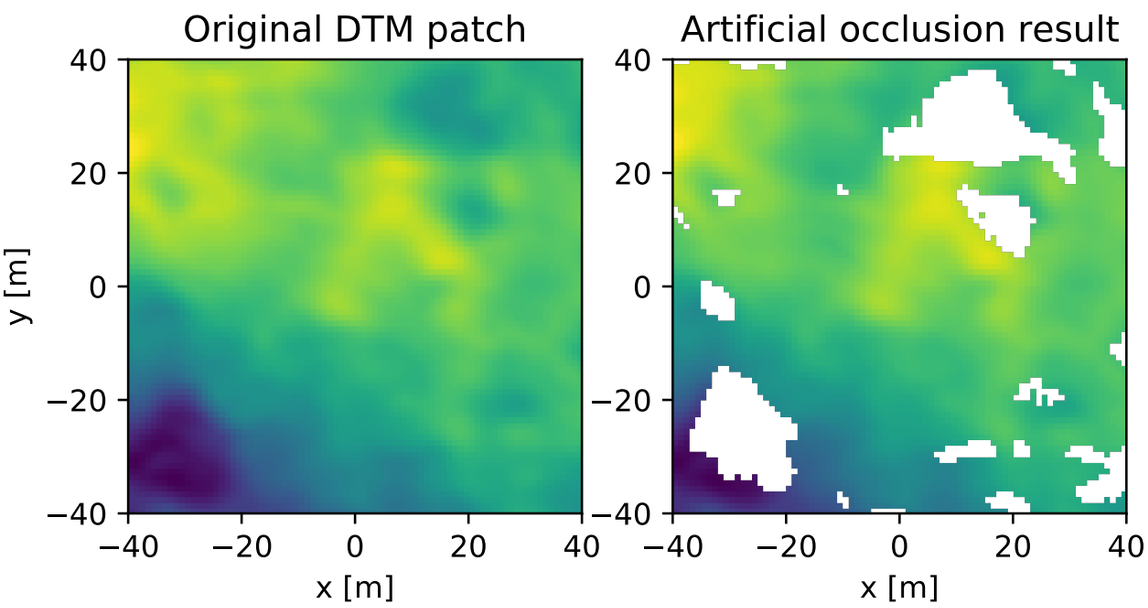


Figure 2.4: Artificial occlusion applied to a random DTM patch

2.1.4 Validation Split

As we have two different DTMs, we will use one of them for the training set (subfigure 2.1a) and another one for the validation set (subfigure 2.1b).

2.2 Tenerife Dataset

As mentioned previously, this dataset was recorded by ESA in 2017 on Tenerife. It consists of data recorded from a field test with the Heavy Duty Planetary Rover (HDPR), a Rover used by ESA's Automation and Robotics (TEC-MMA) section. The field test was performed around the volcanic areas close to the Teide Volcano on the island of Tenerife, Spain. This is a highly representative environment of a planetary exploration mission scenario, due to the barren scenery and the presence of many volcanic rocks.

The data captured during the field test consists of sensor data of the rover during several traverses conducted in the span of a few days. This includes captured images from the stereo cameras, measurements from the inertial measurement unit (IMU), wheel odometry data and a differential GPS system. Figure 2.5 shows the rover and its sensors.

The dataset also includes a georeferenced point cloud of the area included in the field test which was generated from images captured by a drone.

[4] https://github.com/InigoMoreno/deep_ga/blob/18a6518c67968fa4d6e2388c68aa3e6cf9984ea6/deep_ga/patch_generator.py#L9

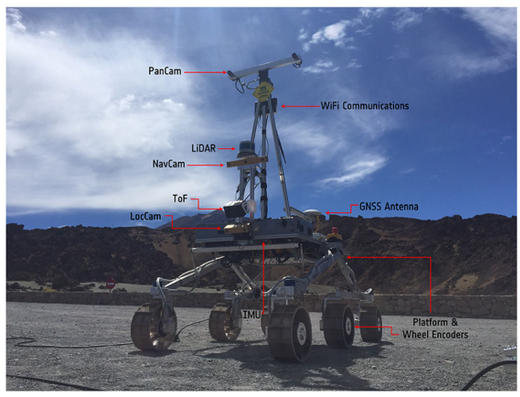


Figure 2.5: HDPR in Tenerife field test at Teide Volcano [2]

2.2.1 Downloading the Dataset

The total size of the Dataset is over 5 TB and it was uploaded to a `ownCloud` online folder. However, the dataset was too big to be downloaded on the laptop used for this thesis. Luckily, the data could be downloaded to another desktop computer which could be accessed via `ssh`. Even so, the Dataset was too big, so only the necessary files were downloaded. This was done using a custom PowerShell script[5] which communicated with the `ownCloud` folder to select the necessary files and download them. This meant that only 2 TB of data needed to be downloaded.

2.2.2 Processing the Point Cloud

The point cloud is stored in a `ply` file and has a resolution of 0.5 meters. To read it in python, the `plyfile` python library [S4] is used.

From the point cloud representing the terrain of the field test, a global elevation map needs to be generated. This can be done by generating a 2D grid and on the XY plane and for each cell of the grid select the points within the cell and average the Z coordinate of the points to get the elevation.

To do this in a more efficient way, we can take advantage of the `histogram2d` function of the `numpy` library [6]. The 2d histogram with the Z coordinate as weights is used to get the sum of the Z coordinates in each cell. Then, the 2d histogram is used again but without weights to get the number of points in each cell. Finally, the algorithm divides the sum of Z by the number of points to get the average elevation. The resulting image can be seen in figure 2.7.

The same process can be repeated with the colours of each point to get a colour image which can be seen in figure 2.6. The code for this can be seen in GitHub [7]

2.2.3 Processing the Rover Sensor Data

Much of the sensor data from the rover is stored as `Rock` log files. `Rock`[S5] is a robotics development environment where the code is split up into libraries and components that interact with each other through data streams. For more information, see the "About Rock" section (4.1) in [13].

In order to read these from python, `pocolog-pybind`[S6] can be used. This is a tool that reads these log files directly from python. However, when doing this a problem was encountered, as the log files were old, the `Rock` version had changed and the files could not be opened. Luckily, `Rock` has a tool for converting old logfiles into the newer version, which was used to recover the old files.

[5] `https://github.com/InigoMoreno/deep_ga/blob/main/rock/download.ps1`

[6] `https://numpy.org/doc/stable/reference/generated/numpy.histogram2d.html`

[7] `https://github.com/InigoMoreno/deep_ga/blob/0aaa42be131563515b872c58dcba5f1c4667cd7a/deep_ga/functions.py#L7`

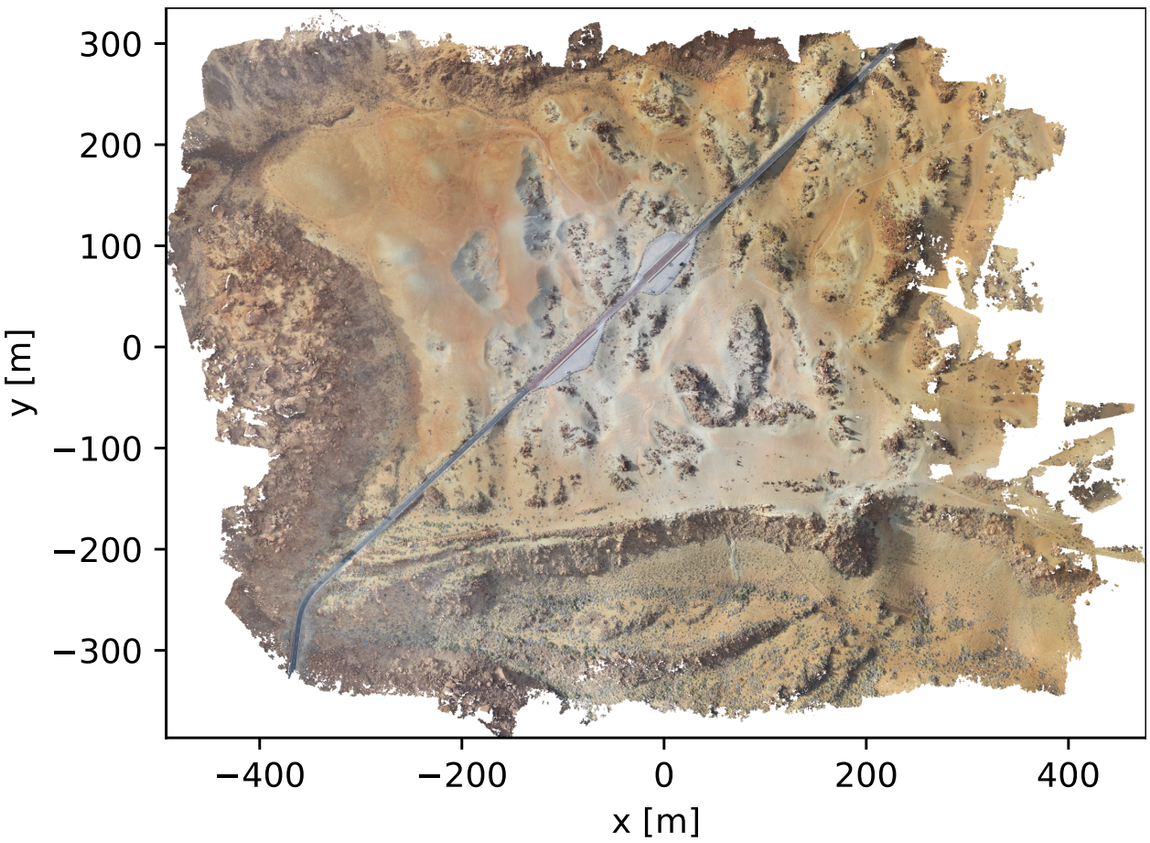


Figure 2.6: Tenerife Color Map

2.2.4 Processing the GPS data

Once the GPS data was opened, it needs to be aligned with the map obtained from the drone data. To properly align the data, there is an offset that needs to be applied added to the GPS data. This offset was already computed in Geromichalos et al. [2], so it was recovered from their code. The resulting GPS traverses are plotted overlaid on top of the Tenerife map in figure 2.8 to show the proper alignment.

2.2.5 Generating the SLAM Local Maps

In order to generate the Local Maps, the Simultaneous Localization And Mapping (SLAM) algorithm from Geromichalos et al. [2] is used. This algorithm is able to generate the local maps based on the images coming from the stereo cameras of the rover. In theory, the algorithm can use all of the cameras at the same time, but due to the blurriness of the other cameras, only the LocCam was used (see figure 2.5). The algorithm is designed to work at any resolution. However, to match the global map, the same resolution of 0.5 meters was used.

This algorithm is available as a stand-alone `C++` library[8] or as a `Rock` library [9]. As all of the data is already stored as `Rock` logfiles, it is more convenient to use the `Rock` library. To be able to use `Rock` installation, a Docker image was used which contained `Rock` and all of the libraries used in the by ESA's Planetary Robotics Lab (PRL).

Once the `Rock` installation was complete, a `Rock` script[10] was written that would replay the data stored in the logfiles and feed it to the SLAM algorithm, storing the generated local maps into new `Rock` logfiles. An example of a few local maps alongside their corresponding global maps can be seen in figure 2.9. These log files were much smaller than the whole dataset, around 5 GB, and could be loaded directly to python for training.

[8] `https://github.com/esa-prl/slam-ga_slam`
[9] `https://github.com/esa-prl/slam-orogen-ga_slam/`
[10] `https://github.com/InigoMoreno/deep_ga/blob/main/rock/tenerife_log_dems.rb`

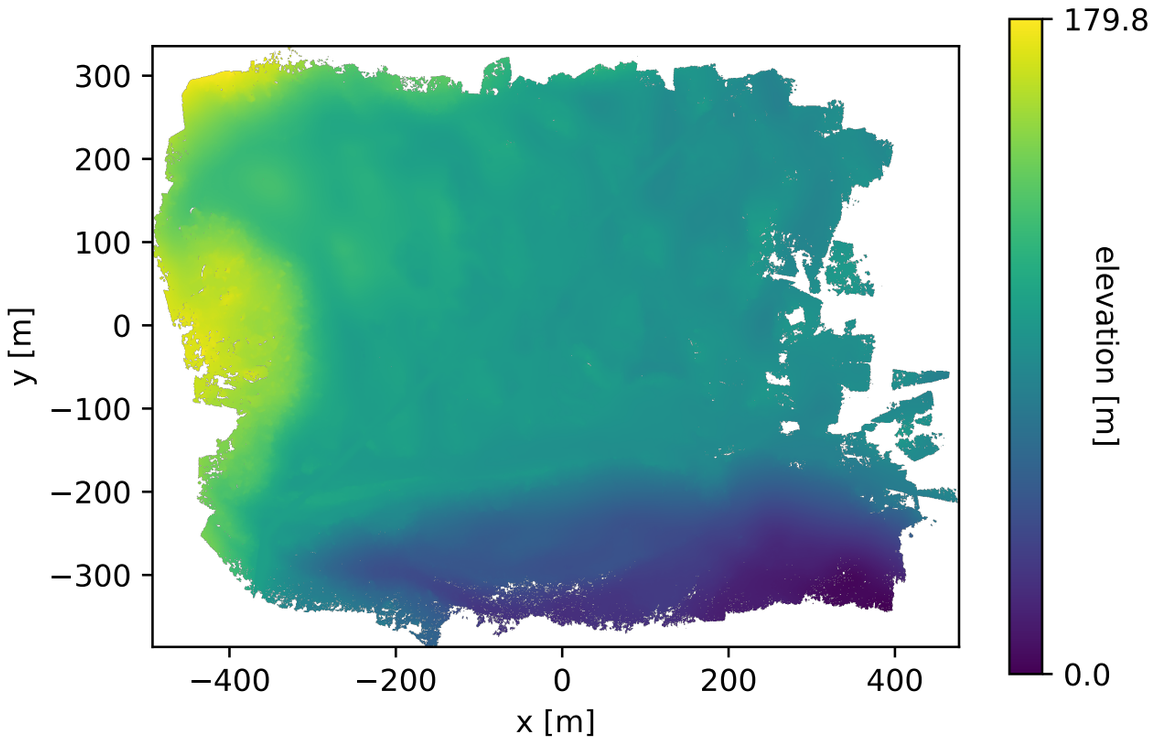


Figure 2.7: Tenerife Elevation Map

2.2.6 Filtering the Tenerife Dataset

The Tenerife Dataset contains many errors. These need to be filtered out of the dataset if possible.

When the algorithm is starting, it has not gathered sufficient stereo images to generate a correct local map. This means that the local map is very empty. To remove these maps, the percentage of the map that is empty (NaN value) is computed and the maps that have more than 65 % empty space are removed.

As it can be seen in figure 2.8, the global map captured by the drone has some data missing. This is only important when the traverses of the rover go through these missing parts. This happens on some parts of the two right-most traverses. To remove these from the dataset, a patch of the global map is computed around the location of the GPS. If this patch has more than 5 % empty space, the point is removed from the dataset.

In the first plot of figure 2.9, some artefacts with a triangular shape can be observed. These artefacts come from errors of the SLAM algorithm when joining together data from some of the stereo images. Unfortunately, this error is quite common and difficult to detect, so it is impossible to remove it from the dataset.

In the second plot of figure 2.9, a small mound can be observed on the left of the elevation maps. One can see that the mound is in a slightly different position on the local map than on the global map. This is because the alignment of the GPS data with the global map explained in subsection 2.2.4 is not perfect. This can be corrected by adjusting the offset manually, but it is difficult to get a perfect alignment.

2.2.7 Validation Split

In order to split the Tenerife dataset, some of the traverses of the rover will be used as validation data and the rest will be used as training data. This ensures that the data from the validation dataset is completely new. The traverses chosen were those of the 10th of June (see figure 2.8), which resulted in a validation split of 15 %.

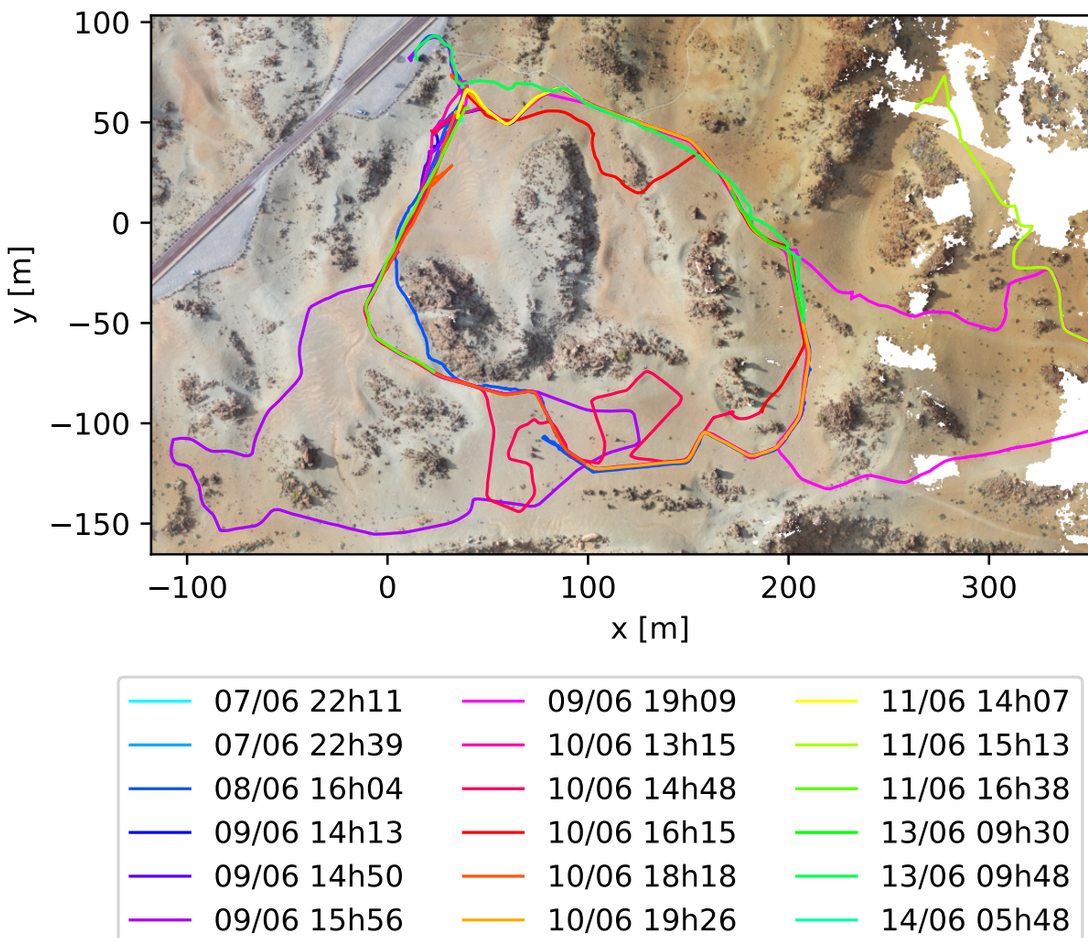


Figure 2.8: Tenerife Traverses overlaid on top of the color map

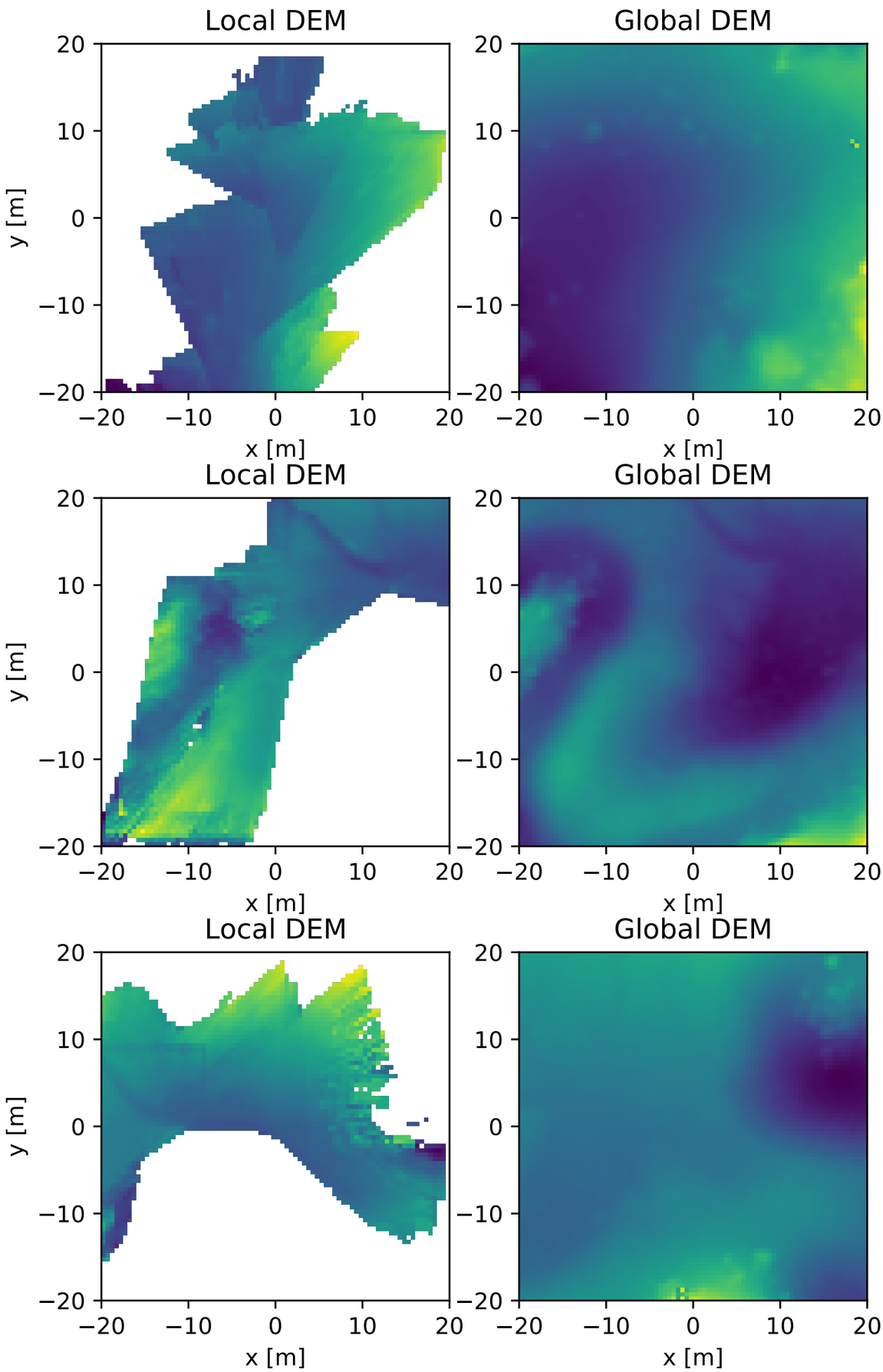


Figure 2.9: Tenerife local and global maps

Chapter 3

Model Implementation

3.1 Framework and Resources

To build and train the Neural Network, the Keras framework was used [S7]. Keras is a high-level API for Deep Learning in python which lets us quickly develop the structure without having to worry too much about the low-level details. Another advantage of Keras is that it is built on top of Tensorflow, which means that tools like Tensorboard or Tensorflow Lite can be used seamlessly.

To develop the Neural Network, it will be necessary to make a lot of decisions about the structure of the network and certain parameters. A tool that can help with this is Talos [S8], a hyperparameter optimization framework for Keras. Talos lets us define a list of parameters and a list of possible values for each of these parameters. Then, Talos will test all of these possible values and save all of the information of each test to be able to decide which value is the best for each parameter.

To be able to train the Neural Networks, access to computational power is required. A very useful service for this is Google Colab [1]. Google Colab is a hosted service that stores and executes Jupyter[2] notebooks using computing resources including GPUs. This is a free service by Google, but it aimed only for interactive use and for shorter runtimes. Therefore, while it is ideal to write code and testing to see if everything works correctly, something more powerful is needed.

Aalborg University (AAU) has a cloud service called CLAAUDIA AI Cloud [3]. This is a cluster of servers with many GPUs where students and researchers from AAU can submit jobs that are run with the GPUs. The jobs are submitted using Slurm[4], a workload manager that manages the order of execution of each job using a queue with priority. Each job can run a Singularity command.

Singularity is a container framework similar to Docker. To create the container with the necessary software to run our code, a Singularity file needs to be written. In this file, all of the dependencies are installed into the container. The file can be found on GitHub[5]. Once the singularity file is created we can use it to run the same Jupyter notebooks developed and tested in Google Colab.

[1] https://colab.research.google.com/
[2] https://jupyter.org/
[3] https://git.its.aau.dk/CLAAUDIA/docs_aicloud/src/branch/master/aicloud_slurm
[4] https://slurm.schedmd.com/
[5] https://github.com/InigoMoreno/deep_ga/blob/main/Singularity

3.2 Structure

As seen in subsection 1.2.1, there are many approaches when it comes to building the structure of the neural network. However, the most common one is to build a Siamese Neural Network (SNN). However, in order to find which structure is better for the SNN a modular approach was chosen. With this approach, the SNN is broken into its most basic components. This can be seen in figure 3.1. Each of these components can be tweaked or interchanged by different versions of the component.

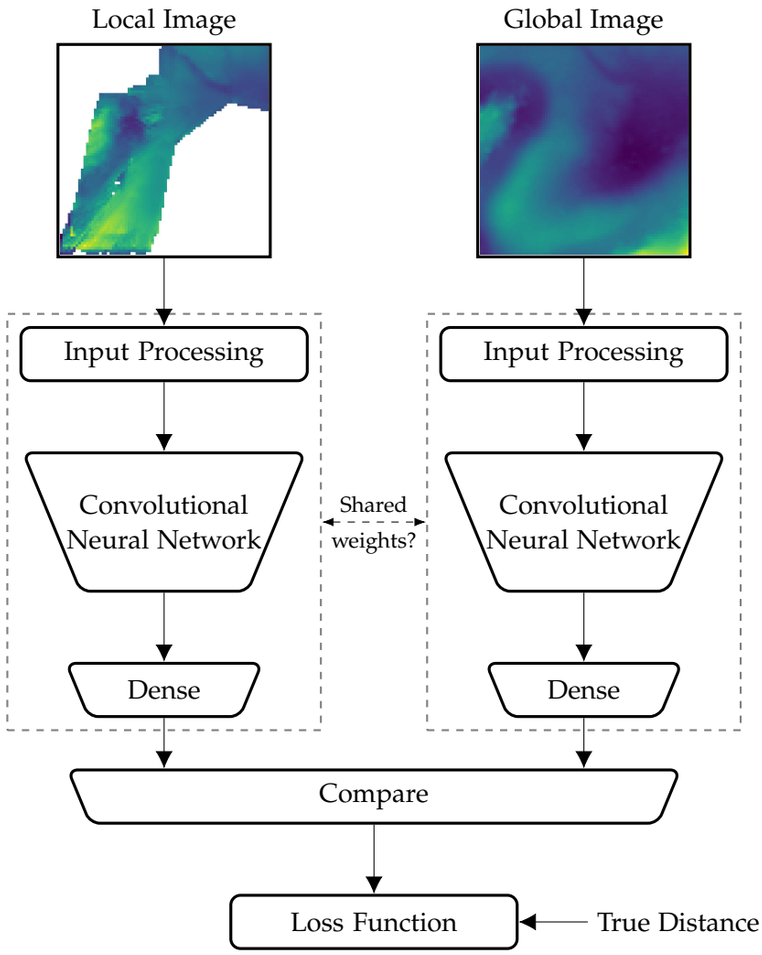


Figure 3.1: Structure of the SNN approach

3.2.1 Input Processing

The input to many pre-built Convolutional Neural Networks (CNNs) is usually a three-layer RGB image without any missing values. The Digital Terrain Models (DTMs) have only one layer (depth) and have missing values. Therefore, before the DTMs to the CNN, some processing needs to be done to remove the missing values and turn one layer into three. There are several ways to do this.

Raw Elevation

The most straightforward way to turn the elevation layer into three is to just make three copies of the elevation layer. To remove the missing values, a custom Keras layer was created that replaces them with zeros. This custom layer can be seen on GitHub[6].

[6] https://github.com/InigoMoreno/deep_ga/blob/a3e62303535aaecc9f4569d7de9487c266690e24/deep_ga/layers.py#L110

Raw Elevation and Mask

A problem with the previous approach is that the data is repeated, and, therefore the Network will have to do more calculations for data that is essentially the same. One think that can be changed from the previous approach is to make only two copies of the elevation layer, and, for the third layer, pass a mask that shows which pixels are missing values. To generate the mask, another custom layer was created[7].

Convolution

The two previous approaches still repeat a lot of data. One approach would be to add a Convolution layer that goes from one layer to three layers. This approach lets the Neural Network control how the Input Processing is done. The convolution layer in Keras is not able to handle missing values. Therefore, we need to set the missing elevation values to zero before passing them to the convolution.

Partial Convolution

Not being able to handle missing elevation values can be a big problem for the network. If we just set them missing values to zero before the convolution (such as in the previous method), the convolution might give unexpected results around the edges of the elevation map.

A type of Convolution that is able to handle missing values is a Partial Convolution [14]. A custom implementation of the Partial Convolution was written and can be seen in GitHub [8].

Sobel Filter

In Geromichalos et al. [2], they mention that there is possible drift that could lead to errors in the z-axis. This means that the absolute elevation values should not be compared, only the gradients of the elevations. One such way of computing (used in [2]) these gradients is to use a Sobel Filter. As we need three layers, we can compute the horizontal, vertical and diagonal Sobel filters:

$$\mathbf{S}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{S}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad \mathbf{S}_{xy} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix} \tag{3.1}$$

We can use these matrices as the kernels of the Partial Convolution described previously.

Skew-Centrosymmetric Convolution

When it comes to computing the gradient of an image, there are many variants of the Sobel filter described above, such as the Sobel-Feldman or the Scharr operators. All of these are matrices of the form:

$$\mathbf{S} = \begin{bmatrix} a & b & c \\ d & 0 & -d \\ -c & -b & -a \end{bmatrix} \tag{3.2}$$

These are called skew-centrosymmetric matrices. For the Sobel filters in equation (3.1), we can verify that they follow the patter in equation (3.2):

	a	b	c	d
$\mathbf{S}_x$	1	0	-1	2
$\mathbf{S}_y$	1	2	1	0
$\mathbf{S}_{xy}$	2	1	0	1

Knowing this, we can create a custom convolution layer that behaves like a partial convolution, but that forces the weights to have the shape described in equation (3.2). This forces the

[7] https://github.com/InigoMoreno/deep_ga/blob/a3e62303535aaecc9f4569d7de9487c266690e24/deep_ga/layers.py#L122

[8] https://github.com/InigoMoreno/deep_ga/blob/a3e62303535aaecc9f4569d7de9487c266690e24/deep_ga/layers.py#L74

convolution to compute a gradient of the elevation, solving the problem of the errors in absolute elevation, while also giving some control to the Neural Network in how to compute these gradients.

3.2.2 Convolutional Neural Network

There are many pre-built CNNs available on Keras[9]. Most of these models are developed and trained for the ImageNet dataset, a huge dataset of images that can be used to train neural networks for image classification. As they are built for such a big dataset, these networks can be very large and have large inference times. As we mentioned in the Problem Analysis (section 1.1), the on-board computers of the rover are very limited. Therefore, we need to use a lightweight Neural Network.

The smallest network available in Keras is MobileNet-V2[15], which is also the network used in Franchi [12]. This network is so small because it was designed to be able to run on low resource platforms such as mobile devices. This makes it a perfect fit for the limited onboard computers of the rover.

MobileNet Width Multiplier

The MobileNet-V2 Network has a width multiplier hyperparameter. This multiplier proportionally changes the number of filters on each layer. This effectively changes the width of the network and lets us do a trade-off between accuracy and performance. In the original MobileNet-V2 paper, Sandler et al. [15], the values of 0.35, 0.5, 0.75, 1 and 1.4 are tested, so these are the ones that we test.

MobileNet Pooling

MobileNet-V2 has another hyperparameter that lets us choose how the pooling is done at the end of the Neural Network. This lets us choose between average, max or no pooling.

MobileNet Weights

The Keras team has trained MobileNet-V2 on ImageNet and has made the weights of these pre-trained networks available on Keras[10]. These are available for several Width Multipliers and input sizes. Using these pre-trained weights gives us the option to reduce the number of trainable parameters, but can also lead to worse results.

To allow for the use of our arbitrarily sized elevation maps as an input for the pre-trained network which is only available for certain input sizes, a resizing layer needs to be added to scale the elevation maps to the available sizes. This is possible thanks to the Resizing layer in the experimental branch of Keras [11].

3.2.3 Dense Layers

After the features are extracted by the CNN, the network can then use up to two dense layers to construct an embedding space from these features. This is an optional phase, which allows for higher complexity of the network and reduces the size of the embeddings. To allow for testing of different shapes of dense layers, four hyperparameters were created.

First and Second Layer Size

These hyperparameters control the number of neurons on each of the dense layers. If the size of a layer is zero, the layer is removed, which means that one can also control how many dense layers there are using these parameters.

[9] `https://keras.io/api/applications/`

[10] `https://github.com/JonathanCMitchell/mobilenet_v2_keras/releases/tag/v1.1`

[11] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/Resizing`

Activation

The dense layers can be configured to have an activation layer using an activation hyperparameter.

Dropout rate

A dropout layer is used between the first and second dense layers. Dropout layers randomly set some of the activations to zero, forcing the network to counteract this by adding redundancy to the network, which can lead to lower over-fitting of the network. This dropout is controlled by the rate hyperparameter, which controls the rate at which the dropout happens.

3.2.4 Weight Sharing

In the model structure shown in figure 3.1 an arrow indicates that the two of the SNN can share weights. This is usually the norm for SNNs, as they reduce the number of parameters that need to be trained and reduce over-fitting. However, due to the differences between the local and the global patch, the network might actually perform better without sharing the weights between the two branches. There will be a hyperparameter to choose if the weights are shared.

3.2.5 Comparation Layer

The computation layer takes the two embeddings and compares them to generate a single output value that represents the difference between the embeddings. In many of the works described in the State of the Art, the Euclidian Norm is used [12, 16, 17]. To implement this into our Network, a custom Keras layer was written[12].

An alternative to this is suggested in Wu et al. [11], where instead of using an Euclidean Distance, a small dense network is used to predict the output of the neural network from the concatenation of the two embeddings. A hyperparameter will be able to choose between these two alternatives.

3.2.6 Loss Functions

From the true distance that separates the local and global elevation maps (d_{true}) and the distance predicted by the network (d_{pred}), the loss function computes the loss. This loss is what the network will try to minimize, so one can think of the loss function as a tool to penalize the errors that the network makes. A good loss function needs to be easy to derive, as the neural network will take many derivatives of the loss function in order to search for the minimum value. There many options to choose from and a hyperparameter will be able to control which of the loss functions is used.

Mean Average Error and Mean Squared Error

Two of the most basic loss functions are Mean Average Error (MAE) and Mean Squared Error (MSE), which just compute the squared and average errors respectively (see equation (3.3)). A plot of these errors can be seen in subfigures 3.3a and 3.3b.

$$\text{MAE} = d_{pred} - d_{true} \qquad \text{MSE} = \left(d_{pred} - d_{true}\right)^2 \tag{3.3}$$

[12] https://github.com/InigoMoreno/deep_ga/blob/fd63a57cbf3199bf2a27ec7fd953d0629104b60d/deep_ga/layers.py#L135

Weighted Mean Squared Error

A problem with MAE and MSE is that they treat errors equally, for example, if $d_{true} = 2$ and $d_{pred} = 12$ the error will be the same as if $d_{true} = 100$ and $d_{pred} = 110$, while the first one should be much more severe. To solve this, we can use Weighted Mean Squared Error (WMSE), which adds a weight component to the MSE. The formula for WMSE can be found in equation (3.4) and it is plotted in subfigure 3.3c.

To choose the weight function, we need something that gives a higher weight to low values, and almost zero weight to values past a certain scale. The function should also be easily derivable. The chosen weight function is equation (3.5), which uses the tanh function, which is very easy to differentiate.

$$\text{WMSE} = \frac{W\left(d_{pred}\right) + W\left(d_{true}\right)}{2}\left(d_{pred} - d_{true}\right)^2 \tag{3.4}$$

$$\text{where,} \qquad W\left(d\right) = 1 - \tanh\frac{d}{scale} \tag{3.5}$$

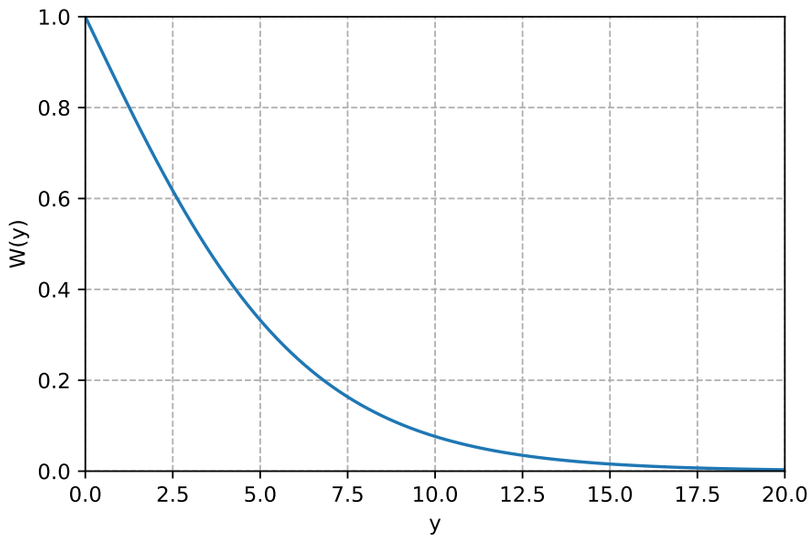
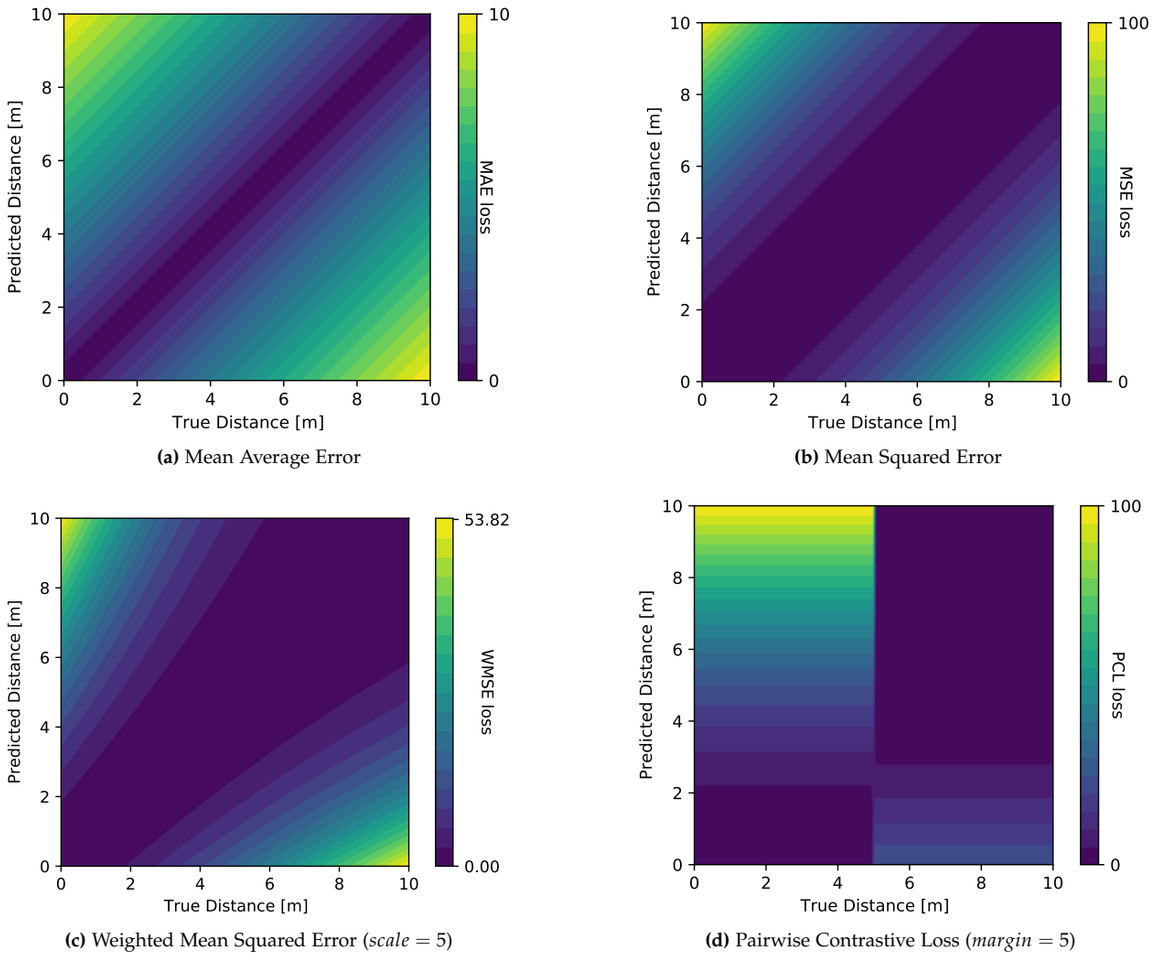


Figure 3.2: Weight function ($scale = 5$)

Pairwise Contrastive Loss

Pairwise Contrastive Loss (PCL) is a loss that is used in some of the works found in the State of the Art [12, 16]. It tries to minimize d_{pred} if d_{true} is below the margin and maximize d_{pred} otherwise

$$PCL = \begin{cases} d_{pred}^2 & \text{if } d_{true} < margin \\ \max(margin - d_{pred}, 0)^2 & \text{otherwise} \end{cases} \tag{3.6}$$



(a) Mean Average Error

(b) Mean Squared Error

(c) Weighted Mean Squared Error ($scale = 5$)

(d) Pairwise Contrastive Loss ($margin = 5$)

Figure 3.3: Losses

3.3 Training

3.3.1 Batch Size

Training on Neural Networks is done in batches, which are groups of training data that are used to compute the loss and the gradient of the loss with respect to the weights of the Neural Network, this gradient is then used to update the weights after each batch. This means that a smaller batch size will train faster, as the network is updated more often. However, a small batch size can lead to instabilities, as the gradient is computed using less samples and is therefore less representative of the gradient of the whole dataset.

3.3.2 Training Optimizer and Learning Rate

When training the model, we need to choose an optimizer. The two most common ones are Stochastic Gradient Descent (SGD) and Adam. Both of these optimizers can also be tuned with a hyperparameter called the learning rate, which controls the speed at which the training occurs. Increasing the learning rate might speed up training but it can also lead to unstable results.

3.3.3 Data Generators

As explained in the previous chapter, the goal of both datasets is to generate pairs of images, one that represents the local elevation map as seen by the rover and another that represents the the global elevation map of the area around the rover, but with a random shift that represents the possible localization error. To generate this, we could generate a lot of random shifts and store each data point in memory, but this would take a lot of memory space during training. An alternative is to use data generators. This is a style of training in Keras where instead of passing an array with all of the training data, we pass a python class that generates each training batch. These generators can be found in GitHub[13]

[13] `https://github.com/InigoMoreno/deep_ga/blob/main/deep_ga/patch_generator.py`

3.4 Python Library and Jupyter Notebooks

As mentioned in section 3.1 (Framework and Resources), all of our code needs to run in jupyter notebooks. These started to get big quickly, so a python library was created that houses the custom Keras layers, the data generators, the model implementation and various other helper functions. This library is available in GitHub[14]. With the library created, the jupyter notebooks are much smaller and more manageable. These are also on GitHub [15].

[14] https://github.com/InigoMoreno/deep_ga/tree/main/deep_ga
[15] https://github.com/InigoMoreno/deep_ga/tree/main/notebooks

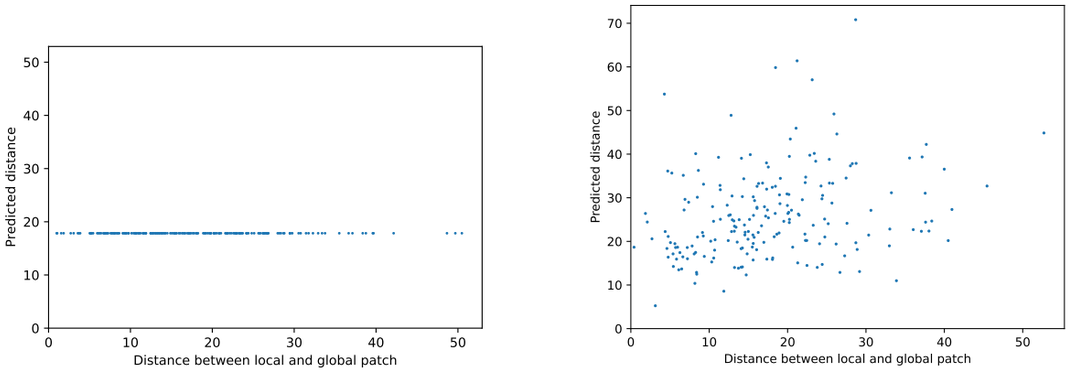
Chapter 4

Model Testing

4.1 Initial Tests

4.1.1 HiRISE Dataset

Initial tests were done using the HiRISE dataset and just guessing some values for the hyperparameters. The first attempts to train the network were unsuccessful, as the training was not actually reducing the loss function much and the results showed that the model was not predicting the distance correctly and was instead predicting only the mean. This can be seen in subfigure 4.1a. However, after a days weeks of debugging, some issues were found in the way the model handled the NaN values, and some results started to appear (see subfigure 4.1b).

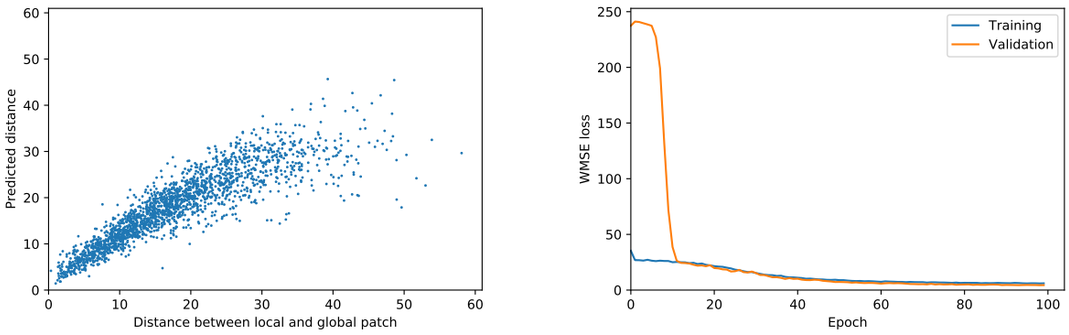


(a) Model just predicting the mean

(b) Model predicting very poorly

Figure 4.1: Failed initial tests with the HiRISE Dataset

However, one can see in the plot of subfigure 4.1b that the results predicted by the model, while they seem to follow the true distance, they are not very accurate. This was later found to be the consequence of an issue with the code that generated the datasets. With the issue fixed, the results became much better. This can be seen in figure 4.2. Subfigure 4.2a shows that the predicted distance is very close to the real distance. Subfigure 4.2b shows how the progress of both the training and the validation loss go down together with each epoch, with the validation loss being even lower than the training loss, which indicates that the model is not overfitting at all. This makes sense, as the HiRISE is almost infinite thanks to the big size of the Digital Terrain Model (DTM) and the fact that data generators are being used to train, which use a new patch of the DTM every time.



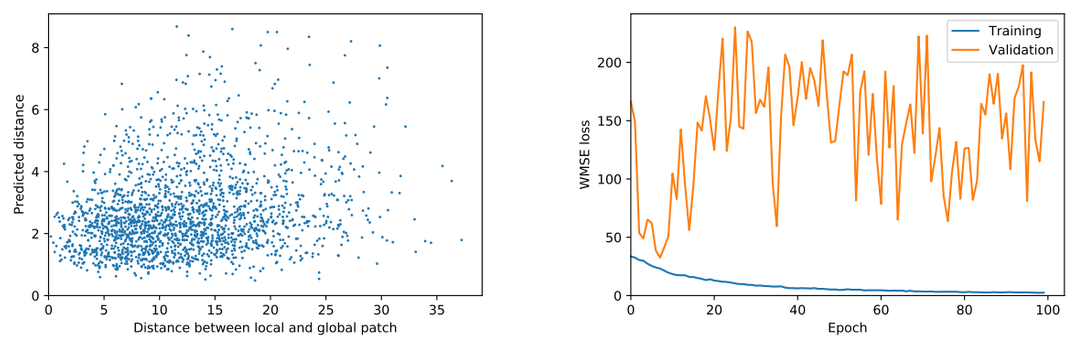
(a) Successful predictions

(b) Training and validation loss history

Figure 4.2: First Success with the HiRISE Dataset

4.1.2 Tenerife Dataset

After the success of the initial tests of the model on the HiRISE Dataset, the model was then trained and tested on the Tenerife Dataset. The first results for this can be seen in figure 4.3, and they show that the model is not training properly on the Tenerife Dataset. The loss history in subfigure 4.3b shows how the training loss is decreasing, but the validation loss stays very high. This means that our model is overfitting heavily on the Tenerife Dataset. This is possibly due to the dataset being too small or to the network having too many weights. From here, the next step was to look for the optimum hyperparameters to use on the Tenerife Dataset.



(a) Predictions

(b) Training and validation loss history

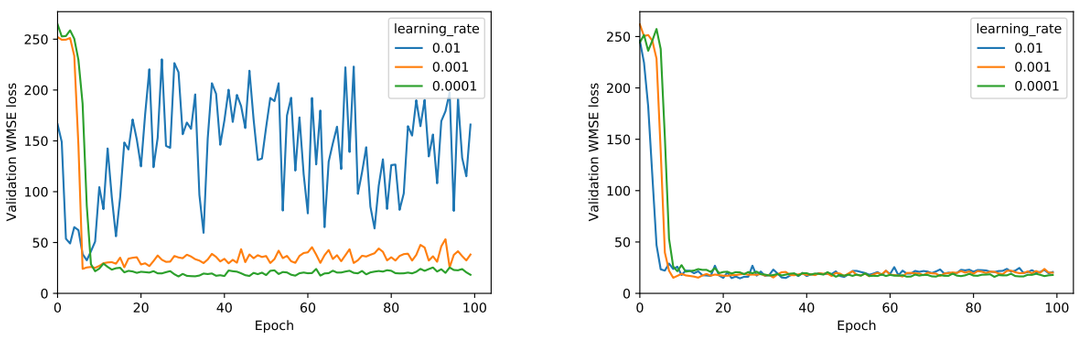
Figure 4.3: First attempt at training on the Tenerife Dataset

4.2 Hyperparameter optimization

As discussed in the previous section, the training on the Tenerife Dataset with the default hyperparameters was not successful, so the next step is to look for the optimal hyperparameters. As mentioned previously, the tool that will be used for this is Talos[S8], which repeats the training process for each new set of parameters. As the training can take very long (up to two hours), each experiment ran with Talos can take days, which limits the number of runs that can be performed. This, coupled with the fact that each training session can lead to different results even if they have the same parameters, means that the results of the Talos experiments can be a bit unreliable. However, they are still a very useful tool to determine which hyperparameter to use.

4.2.1 Optimizer and Learning Rate

One of the things noticeable in subfigure 4.3b is that the validation loss is very erratic, it moves up and down a lot. One of the things that could cause this is the optimizer or the learning rate, as discussed in subsection 3.3.2. Therefore, a Talos experiment was set up where these hyperparameters were tested. The training history of each experiment of the test can be seen in figure 4.4. The results are summarized in subfigure 4.5b with a boxplot of the final validation loss. From these results, one can conclude that the best optimizer for our task is Stochastic Gradient Descent (SGD), with a low learning rate.



(a) Using Adam optimizer

(b) Using SGD optimizer

Figure 4.4: History of validation loss for different values of learning rate and optimizer

4.2.2 Input Processing

In subsection 3.2.1 several ways of processing the input data were discussed, to decide which of them is better, a Talos experiment was run. The experiment tried the different input processing options a total of three times each. The results are shown in subfigure 4.5c. These results indicate that the two best options for input processing are to do a simple convolution or to use the skew-centrosymmetric convolution. From these two, the latter was chosen, as it guarantees that no information about the absolute height of the DTM is used by the network (see subsection 3.2.1).

4.2.3 MobileNet Parameters

The hyperparameters of the MobileNet-V2 Convolutional Neural Network (CNN) were discussed in subsection 3.2.2. These are the width multiplier, the pooling and the weights. As before, a Talos test was run for each of these parameters and the results are shown in subfigures 4.5d to 4.5f. Looking at the results, it seems like the pooling does not have a very noticeable effect, so average pooling will be chosen. As for the width multiplier, it seems like the 1.0 width multiplier is the best one. However, using a lower width multiplier might give us similar results with lower resources, this will be investigated later.

For the MobileNet weights, it seems like using pre-trained ImageNet actually worsens the validation results of the network. This might be because the data from ImageNet is very different from the elevation maps, so the features extracted from ImageNet don't make much sense for elevation maps.

4.2.4 Dense Layers

Subsection 3.2.3 describes the implementation of the dense layers. While doing some manual testing, it was seen that using two dense layers was excessive and just one of them was enough. Talos tests were created for the layer size, the dropout and the activation function. The results can be seen in subfigures 4.5g to 4.5i. The results show that it is better to use the ReLu activation function and to use a large dropout.

As for the layer size, the plot shows that the network performs better without size 0, which means that the layer is removed. This might be because adding a Dense layer adds a lot of extra weights, which can increase the over-fitting of the network. However, an advantage of using the Dense layer is that the size of the embedding vector is reduced from the output size of the MobileNet (1280) to the size of the dense layer. A smaller embedding space might be useful, as it could mean that less data needs to be transmitted between the rover and the satellite.

4.2.5 Weight sharing

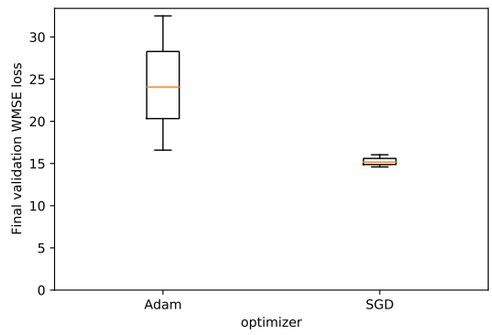
Subsection 3.2.4 discussed the possibility of having the Siamese Neural Network (SNN) not share the weights between the two branches. This was tested with a Talos experiment and its results can be seen in subfigure 4.5j. Surprisingly, the results show how sharing weights actually decreases the performance of the network. This might be because the local and the global elevation maps are very different and not sharing the weights lets the network treat them differently.

4.2.6 Comparation Layer

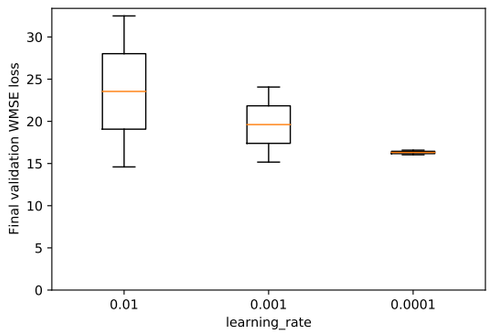
In subsection 3.2.5, two alternatives on how to do the comparison Layer were suggested: computing the euclidean distance between the embeddings or using a small neural network. However, during manual testing of both of these options, it was found that the usage of a small neural network increased over-fitting by quite some margin and made the model un-trainable.

4.2.7 Choosing the Loss Function

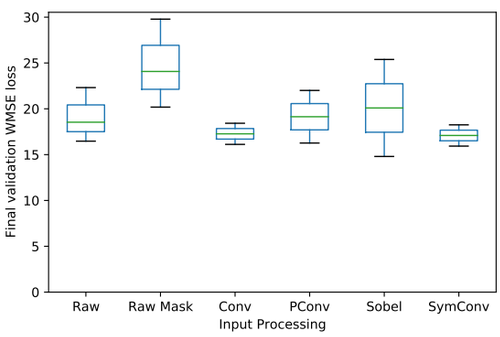
In subsection 3.2.6 several loss functions were discussed as potential candidates for our model. To choose the loss function one cannot simply compare the final loss, as they would be in different units. The only option is to look at plots of the predictions of the model after being trained with each loss function and choose the one that looks better. To get an idea of the best-case scenario, the HiRISE Dataset will be used instead of the Tenerife Dataset for these plots. The plots can be seen in figure 4.6 and they show that, apart from the MAE loss, the other ones seem to perform similarly.



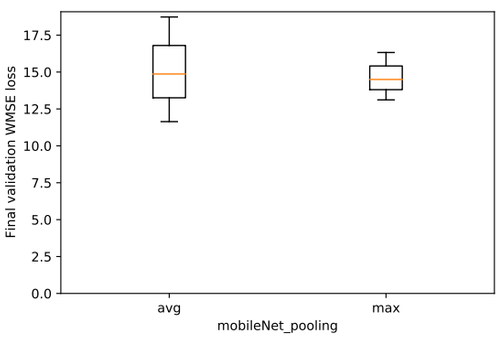
(a) Optimizers



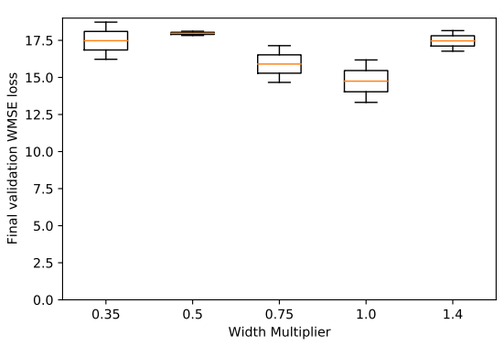
(b) Learning Rate



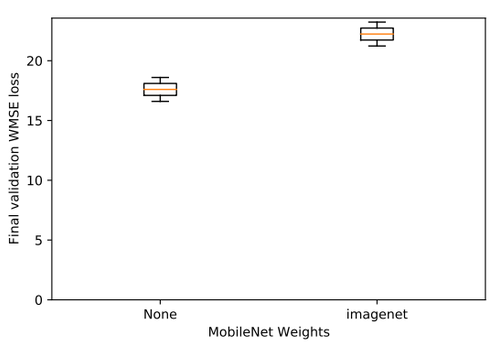
(c) Input Processing



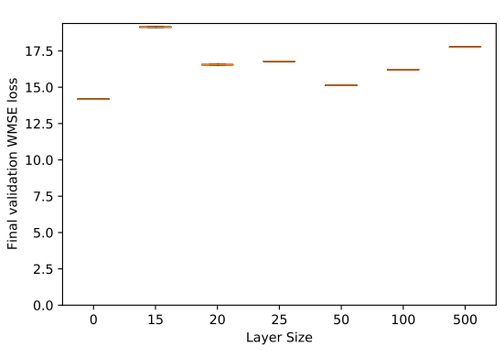
(d) MobileNet Pooling



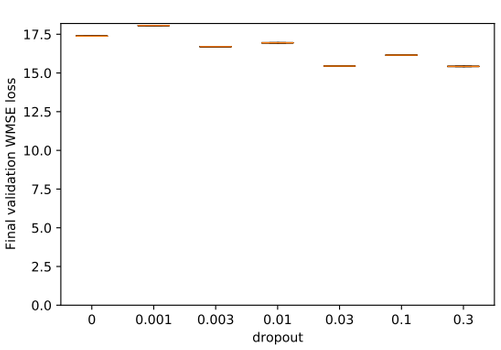
(e) MobileNet Width Multiplier



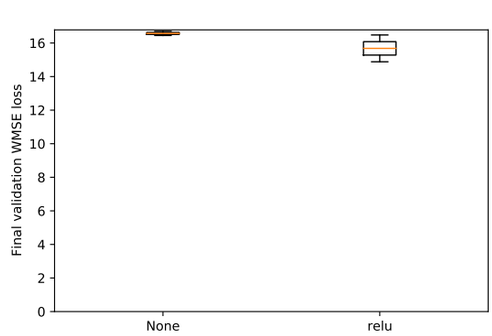
(f) MobileNet Pre-trained Weights



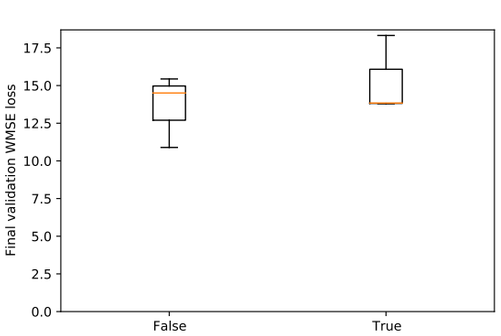
(g) Dense Layer Size (0 means no dense layer)



(h) Dense Layer Dropout

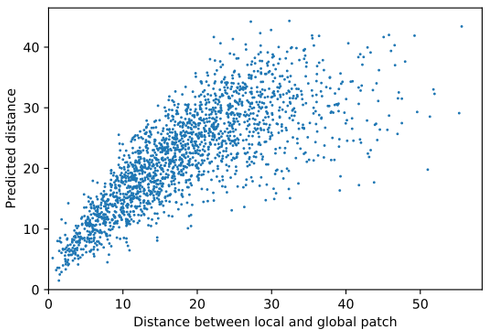


(i) Dense Layer Activation Function

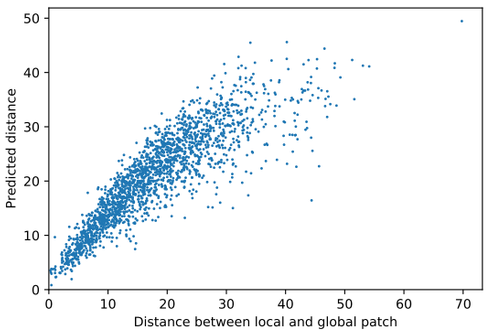


(j) Weight Sharing

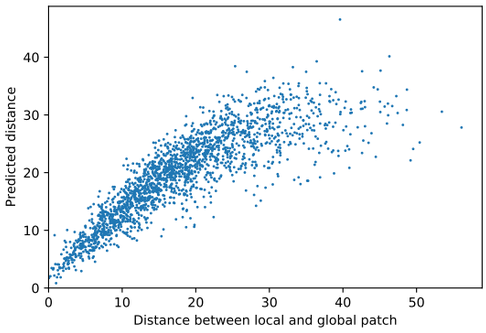
Figure 4.5: Boxplots of the final validation loss depending on different hyper-parameters



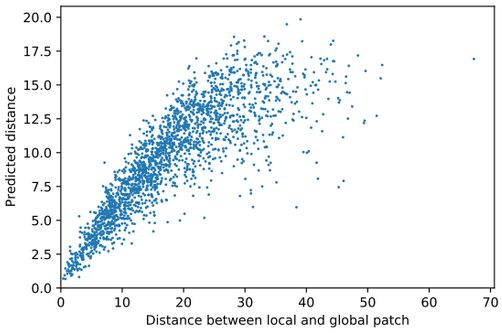
(a) Model trained with MAE loss



(b) Model trained with MSE loss



(c) Model trained with WMSE loss

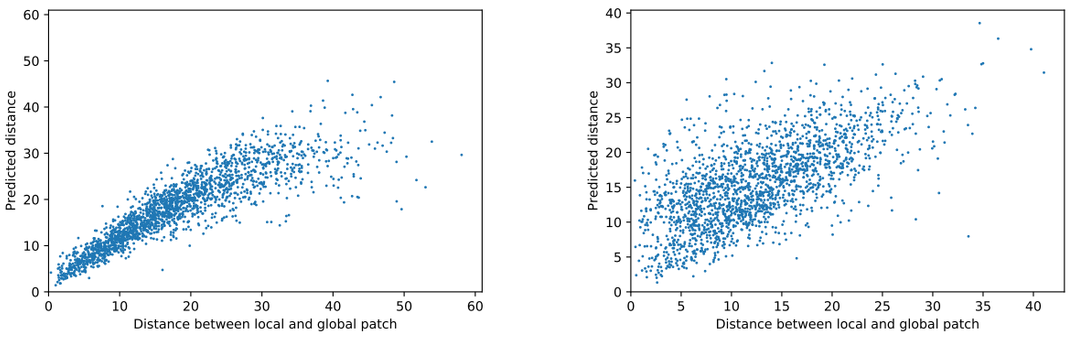


(d) Model trained with PCL loss

Figure 4.6: Predictions of Models trained with different loss functions

4.3 Prediction Results

After all of the experiments to test the hyperparameters, the model that uses all of the best parameters can be constructed and tested on both datasets. These predictions of these models can be seen in figure 4.7. In these plots, one can see that, while the model performs better on the HiRISE Dataset, the predictions on the Tenerife Dataset still show some promising results.



(a) HiRISE Dataset

(b) Tenerife Dataset

Figure 4.7: Prediction Plots

4.4 Resource usage analysis

In this section, the resource usage of the neural network will be analyzed. As discussed in the previous section, one can alter the size of the network with some of the hyperparameters, namely the width multiplier of the MobileNet and the dense layer size. This means that the resource usage will depend on these two parameters. Table 4.1 shows the number of parameters (in millions) of the model, depending on these parameters.

In section 1.4, some of the requirements of the network were introduced, memory usage and prediction time. To estimate the memory usage TensorFlow Lite will be used, a Deep Learning tool that lets us compress our models into a simple C library file that could be used by the rover directly. Table 4.2 shows the size of this library in MegaBytes. To approximate the prediction time in the rover, the C library generated by TensorFlow Lite can be run on Google Colab's CPU environments (without using GPU). The resulting prediction times are shown in milliseconds in table 4.3.

		Layer Size				
		1000	500	100	50	0
Width Mult.	1.4	6.16	5.26	4.54	4.45	4.36
	1.0	3.54	2.90	2.39	2.32	2.26
	0.75	2.66	2.02	1.51	1.45	1.38
	0.5	1.99	1.35	0.83	0.77	0.71
	0.35	1.69	1.05	0.54	0.47	0.41

Table 4.1: Number of parameters (in millions) depending on layer size and width multiplier

		Layer Size				
		1000	500	100	50	0
Width Mult.	1.4	23.57	20.15	17.41	17.07	16.73
	1.0	13.68	11.23	9.28	9.03	8.79
	0.75	10.38	7.94	5.99	5.74	5.50
	0.5	7.86	5.42	3.46	3.22	2.97
	0.35	6.76	4.32	2.37	2.12	1.88

Table 4.2: TF Lite model size in MB. Depending on layer size and width multiplier

		Layer Size				
		1000	500	100	50	0
Width Mult.	1.4	30.53	29.47	29.38	29.25	28.94
	1.0	19.32	19.08	18.57	18.70	18.35
	0.75	16.05	15.83	15.21	15.58	15.20
	0.5	11.84	11.43	10.94	11.57	10.79
	0.35	10.26	10.18	9.55	9.48	9.40

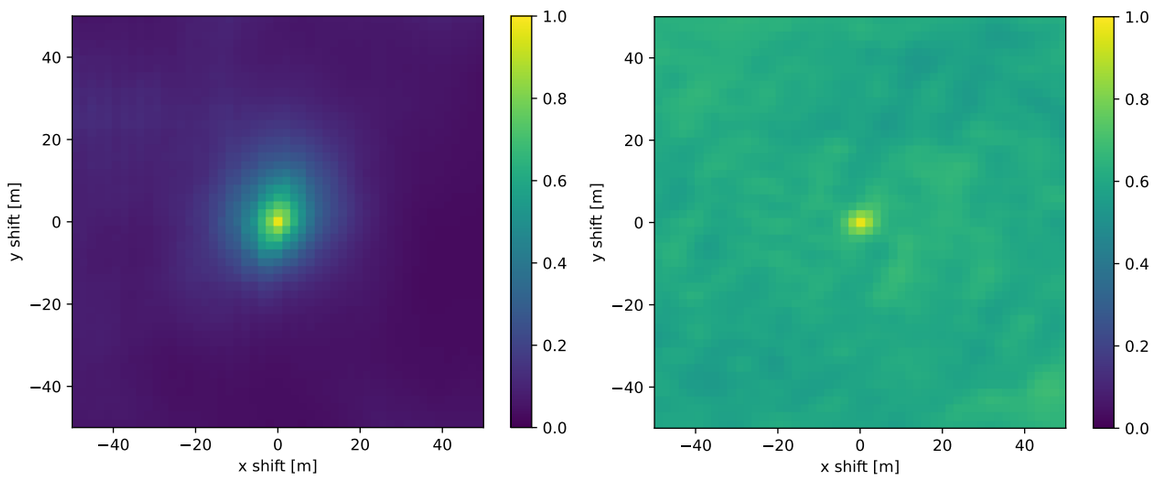
Table 4.3: TF Lite average prediction time in milliseconds. Depending on layer size and width multiplier

4.5 Comparison with Other Methods

In this section, the performance of our model will be compared against the method introduced in Geromichalos et al. [2]. As explained in section 1.1, their method uses a sliding window approach to calculate a correlation score between the local map and all the possible global maps to generate a 100x100 meter grid and select the one with the highest score. Each score is computed in a mere 0.24 ms.

To do the same with our model, a similar sliding window approach is implemented, that compares the local map against all of the possible global maps, computing the predicted distance between the local map and each of the global maps. The chosen model predicts this distance in 18.7 ms, and the whole grid in 3 minutes. In order to transform the predicted distance into something comparable to the score of the other method, the same function used in equation (3.5) will be used.

This can be done for both the HiRISE and the Tenerife Datasets and the results can be seen in figures 4.8 and 4.9 respectively. Note that in these plots one should expect higher values (yellow) in the centre, as it is the true position of the local map and lower values outside.



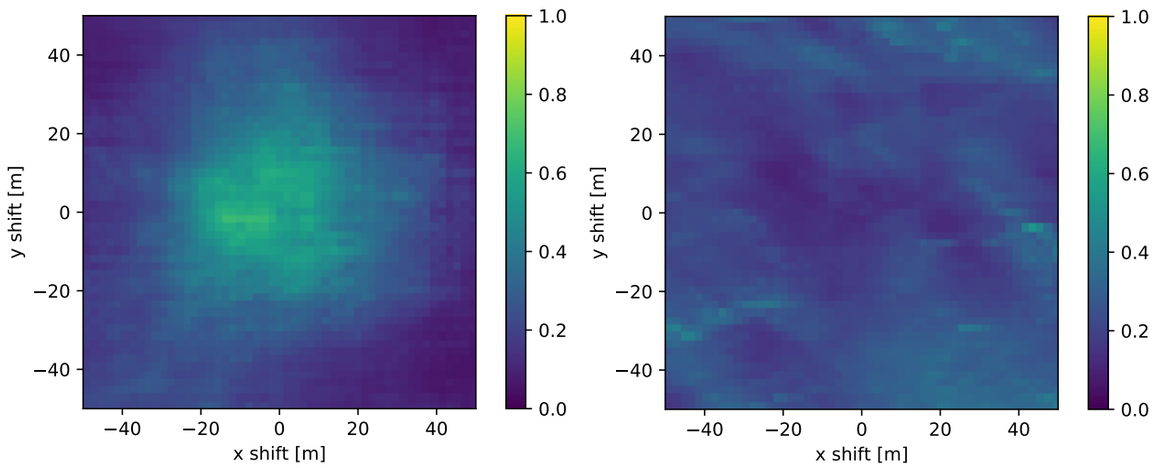
(a) Using our model: $1 - \tanh(d_{pred}/15)$

(b) Using the score in [2]

Figure 4.8: 100x100m grids of results for the HiRISE dataset

These results look very promising. For the HiRISE dataset, figure 4.8 shows that both algorithms correctly identify the centre as being the one with the highest score. However, while the correlation of [2] only marks the immediate centre with a high score and the rest with lower scores, our model is capable of giving high scores to values close to the centre. This means that an algorithm such as Hill Climbing or a Particle Filter could be used in conjunction with our model to find the maximum score without having to compute every value on the grid.

As for the Tenerife dataset, figure 4.9 shows that both methods seem to struggle. The score of [2] seems to not be able to find any correlation between the local map and the global map. With the maximum score being only 0.52 and at a location 44.18 m off the centre. In contrast, our model seems to also struggle, but the plot shows that the values close to the centre have higher values than the rest. The maximum score of our model is 0.68 and is only 14.14 m off the centre. To make sure this is not a fluke, this process was repeated for 20 random local maps from the Tenerife validation dataset, and the error for each method is plotted in figure 4.10.



(a) Using our model: $1 - \tanh(d_{pred}/15)$ **(b)** Using the score in [2]

Figure 4.9: 100x100m grids of results for the Tenerife dataset

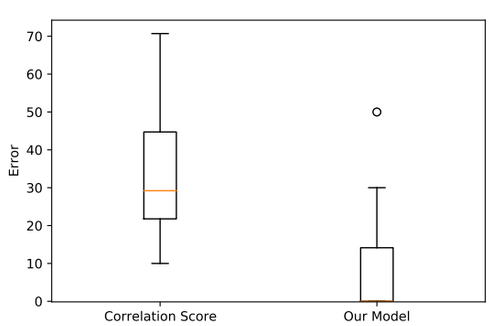


Figure 4.10: Comparison in the error between the two methods for the Tenerife Dataset

Chapter 5

Conclusion

The initial problem formulation of this thesis was to see if it is possible to use a Machine Learning algorithm to perform global localization for a Mars rover. Due to time constraints, a full algorithm could not be written. However, this thesis has shown that a Machine Learning model can be used to estimate the distance between a local and a global elevation map. The thesis has also introduced a preliminary version of a global localization algorithm in the form of a sliding window approach and shows that it can outperform the previous approach in terms of accuracy.

In this thesis, we showed the full process of the development of the Machine Learning model. After analyzing the state of the art and setting out our requirements, two datasets were gathered. The HiRISE dataset utilizes real global maps and artificial local maps generated by adding some simplex noise and simulated occlusion. The Tenerife dataset consists of real local maps captured by a rover in a representative environment and global maps captured by a rover.

After generating the two datasets, the Machine Learning model was constructed in a way that permits a lot of adjustment through the use of hyperparameters. The training on the HiRISE dataset was successful, but the training on the Tenerife dataset was having a lot of over-fitting. In order to reduce the over-fitting and reach a lower validation loss, several experiments were run that would help discover the best hyperparameters to use for the model. With these experiments, we managed to bring the validation loss by quite some margin and got some decent results from the Tenerife dataset. However, these results are still nowhere near the quality achieved using the HiRISE dataset. This might be in part because the Tenerife dataset, while big, might not have enough data to train a Deep Network model. Another reason might be that the dataset contains a lot of errors that cannot be filtered out of the dataset.

After the model was created, the number of resources that it utilizes was analyzed. The model is capable of doing predictions in about 18 milliseconds in a single-core CPU while only taking up around 9 megabytes of memory space. To test the model against the previous algorithm introduced in Geromichalos et al. [2], we created a similar sliding window approach but using our model to predict the score and compared the two. Our method showed promising results and was able to more accurate results.

Computing the whole 100x100 meter grid with the sliding window approach means that we need to run the prediction 10000 times, resulting in 3 minutes of total computation time. While this is high, it is still a manageable amount in a rover scenario. The results could be made even better by the introduction of an algorithm that looks for the maximum value without computing the whole grid, like a Hill-Climbing algorithm or a Particle Filter.

5.1 Future Work

Here are some potential avenues for future work on this topic:

- **Algorithm Implementation:** As mentioned before, it would be possible to write an algorithm that looks for the maximum score without having to compute the whole grid.
- **Different resolutions:** In the data processing, we set the resolution of the Tenerife local maps to be 0.5 meters, the same as the global maps. This was mainly done because the goal was to build a Siamese Neural Network (SNN) with identical branches. However, we saw that having the branches be different actually increased performance. Therefore, we could test if increasing the resolution of the local maps would improve the results.
- **More data:** It would be interested the approach tested with different datasets and see if they perform better.
- **Testing on real rovers:** It would be interesting to see the model being actually implemented and run directly on the rovers.

Bibliography

[1] Andrea Merlo, Jonan Larranaga and Peter Falkner. 'Sample fetching rover (sfr) for msr'. In: *Advanced Space Technologies in Robotics and Automation*. 2013.

[2] Dimitrios Geromichalos et al. 'SLAM for autonomous planetary rovers with global localization'. In: *Journal of Field Robotics* 37.5 (Aug. 2020), pp. 830–847. DOI: 10.1002/rob.21943. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21943.

[3] D. Shirley and J. Matijevic. 'Mars Pathfinder Microrover'. In: *Autonomous Robots* 2.4 (Dec. 1995), pp. 283–289. DOI: 10.1007/BF00710795. URL: https://doi.org/10.1007/BF00710795.

[4] C. F. Olson and L. H. Matthies. 'Maximum likelihood rover localization by matching range maps'. In: *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*. Vol. 1. May 1998, 272–277 vol.1. DOI: 10.1109/ROBOT.1998.676398.

[5] Joseph R. Guinn. 'Mars surface asset positioning using in-situ radio tracking'. In: *American Astronautical Society* (Feb. 2001).

[6] S. Chiodini et al. 'Mars rovers localization by matching local horizon to surface digital elevation models'. In: *2017 IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*. June 2017, pp. 374–379. DOI: 10.1109/MetroAeroSpace.2017.7999600.

[7] Patrick J. F. Carle, Paul T. Furgale and Timothy D. Barfoot. 'Long-range rover localization by matching LIDAR scans to orbital elevation maps'. In: *Journal of Field Robotics* 27.3 (Mar. 2010), pp. 344–370. DOI: https://doi.org/10.1002/rob.20336. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/rob.20336.

[8] Evangelos Boukas, Antonios Gasteratos and Gianfranco Visentin. 'Introducing a globally consistent orbital-based localization system'. In: *Journal of Field Robotics* 35.2 (July 2017), pp. 275–298. DOI: https://doi.org/10.1002/rob.21739. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21739.

[9] Bach van Pham, Artur Maligo and Simon Lacroix. 'Absolute Map-Based Localization for a Planetary Rover'. In: May 2013, p. 1. URL: https://hal.archives-ouvertes.fr/hal-01020989.

[10] Ahmed M. Naguib et al. 'Planetary Long-Range Deep 2D Global Localization Using Generative Adversarial Network'. In: *Journal of Korea Robotics Society* 13.1 (Mar. 2018), pp. 26–30. DOI: 10.7746/jkros.2018.13.1.026. URL: http://jkros.org/_common/do.php?a=full&b=33&bidx=2201&aidx=26169.

[11] B. Wu et al. 'Absolute Localization Through Orbital Maps and Surface Perspective Imagery: A Synthetic Lunar Dataset and Neural Network Approach'. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Nov. 2019, pp. 3262–3267. DOI: 10.1109/IROS40897.2019.8968124.

[12] Valerio Franchi. 'Vision-based localization of a rover on a planetary-like surface using monocular cameras, IMU and a known bird-view map of the environment'. MA thesis. Edinburgh, Scotland: Heriot-Watt University, Dec. 2020.

[13] Iñigo Moreno-Caireta. *Fusion of velocity and relative pose measurements for localization of extraterrestrial rovers*. 3. semester Intenship Report. Jan. 2020, p. 55. URL: https://projekter.aau.dk/projekter/da/studentthesis/fusion-of-velocity-and-relative-pose-measurements-for-localization-of-extraterrestrial-rovers(96ddd0b4-37f5-44f3-b2ea-fc396e4ff62d).html.

[14] Guilin Liu et al. 'Image Inpainting for Irregular Holes Using Partial Convolutions'. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Sept. 2018.

[15] Mark Sandler et al. 'MobileNetV2: Inverted Residuals and Linear Bottlenecks'. In: *arXiv:1801.04381 [cs]* (Mar. 2019). URL: http://arxiv.org/abs/1801.04381.

[16] D. Kim and M. R. Walter. 'Satellite image-based localization via learned embeddings'. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. May 2017, pp. 2073–2080. DOI: 10.1109/ICRA.2017.7989239.

[17] S. Hu et al. 'CVM-Net: Cross-View Matching Network for Image-Based Ground-to-Aerial Geo-Localization'. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. June 2018, pp. 7258–7267. DOI: 10.1109/CVPR.2018.00758.

Software

[S1] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. 2021. URL: https://gdal.org.

[S2] Alexander B. Jung et al. *imgaug*. 2020. URL: https://github.com/aleju/imgaug.

[S3] Siu Kwan Lam, Antoine Pitrou and Stanley Seibert. 'Numba: a LLVM-based Python JIT compiler'. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*. Austin, Texas: ACM Press, 2015, pp. 1–6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: http://dl.acm.org/citation.cfm?doid=2833157.2833162.

[S4] Darsh Ranjan et al. *plyfile*. 2021. URL: https://pypi.org/project/plyfile/.

[S5] DFKI GmbH Robotics Innovation Center. *Rock, the Robot Construction Kit*. 2011. URL: http://www.rock-robotics.org.

[S6] Maximilian Stölzle. *pocolog_pybind, python bindings for rock*. 2020. URL: https://github.com/esa-prl/tools-pocolog_pybind.

[S7] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[S8] Autonomio. *Talos*. 2019. URL: http://github.com/autonomio/talos.

Acronyms

AAU Aalborg University
ESA European Space Agency
HDPR Heavy Duty Planetary Rover
IMU inertial measurement unit
PRL Planetary Robotics Lab
TEC-MMA Automation and Robotics
CNN Convolutional Neural Network
SNN Siamese Neural Network
GAN Generative Adversarial Network
SLAM Simultaneous Localization And Mapping
HiRISE High Resolution Imaging Science Experiment
MRO Mars Reconnaissance Orbiter
DTM Digital Terrain Model
MSE Mean Squared Error
WMSE Weighted Mean Squared Error
MAE Mean Average Error
PCL Pairwise Contrastive Loss
SGD Stochastic Gradient Descent
SFR Sample Fetching Rover

List of Figures

List of Tables