POLITECNICO DI TORINO

Master's Degree in

Computer Engineering – Artificial Intelligence and Data Analytics

A.A. 2022/2023

A Machine Learning-Based Approach for Traversability Analysis and Path Planning in Martian Rovers

April 2023

Supervisor: Stesina Fabrizio **Candidate: Gigante Samuele**

Company Mentor: Leuzzi Chiara

Abstract

Artificial intelligence is considered one of the major breakthroughs in data analysis and decision-making of the last decades. It is one of the scientific disciplines that have most attracted the attention of the public and the scientific community, with applications ranging from the medical field to cybersecurity. With the advent of Internet-of-Things and the increasing number of high quality sensors present inside most of market-dominating smartphones, the need for a class of methods capable of extracting fundamental information from an unprecedented volume of data was, and still is, very high.

Machine and Deep Learning methods are the main instruments used nowadays to perform such intricate analyses, allowing to establish relationships between input features and output predictions in a similar way to the human brain, without requiring extensive training of human operators to repeat the same analysis on new data.

Interplanetary exploration through autonomous rover systems is a perfect example of a scientifically challenging task that can greatly benefit from the application of Machine Learning algorithms within its workflow.

While new solutions are being devised in order to allow humans to travel, land and conduct scientific research on the Martian soil, Martian rovers represent one of the essential elements that enable the exploration of the red planet surface. One of the main factors of improvement for the Martian rover missions is the velocity of exploration, i.e. to move faster while avoiding obstacles and unstable terrains. To meet this objective, drones and satellites can provide significant information for the rover navigation in terms of maps definition and planning of the path.

This mission poses many technical challenges such as autonomous terrain labeling, traversability analysis and path planning. Solutions for the two latter tasks have been studied and implemented in this Thesis.

Traversability analysis is the quest of examining the data related to a portion of terrain in order to establish which parts of it are safe for the rover to traverse, considering different

constraints regarding terrain composition, slope percentage and hazard presence, in order to output a traversability map that can be fed to the planning subsystem.

Path planning, instead, is the task of searching for the optimal sequence of valid configurations that makes the system reach the goal point, given its current location, a representation of the environment which includes the hazards (i.e. the result of traversability mapping) and the heuristic that we want to use to establish what it means for a set of configurations to be optimal.

Both of these tasks have been performed with a learning-based approach, as the main assumption behind this work is the fact that machine learning algorithms can be very helpful to carry out such analyses, that require using vast amounts of data that are difficult and time consuming for humans to manipulate.

The research regarding traversability mapping was carried out within the SINAV project in collaboration with Altec S.p.A. and sponsored by the Italian Space Agency, under the supervision of Chiara Leuzzi, while the path planning part was conducted within the research environment of the Politecnico di Torino and supervised by Professor Fabrizio Stesina.

Table of Contents

List Of Figures

List Of Tables

List Of Algorithms

List Of Equations

Thesis Structure

This work is organized as follows: in Chapter 1, a concise overview of the SINAV project is presented, along with a brief introduction to Robotic Systems, Traversability Analysis, Path Planning and Machine Learning applications within the scope of autonomous rovers development. In Chapter 2, existing approaches for traversability analysis and path planning tasks in the context of autonomous driving planetary rovers will be examined. Particular emphasis will be placed on past attempts that utilized Machine Learning models to enhance these functions. In Chapter 3, the selected methodologies for traversability estimation and path planning, based on Machine Learning algorithms, are introduced. Chapter 4 contains a description of the dataset used to conduct the experiments and thoroughly exposes the results of the application of the previously described methods. Chapter 5 is dedicated to a brief comparison of the obtained results with those derived from different studies, a discussion about the future possibilities for these works, and conclusions. Chapter 6 provides a comprehensive bibliography.

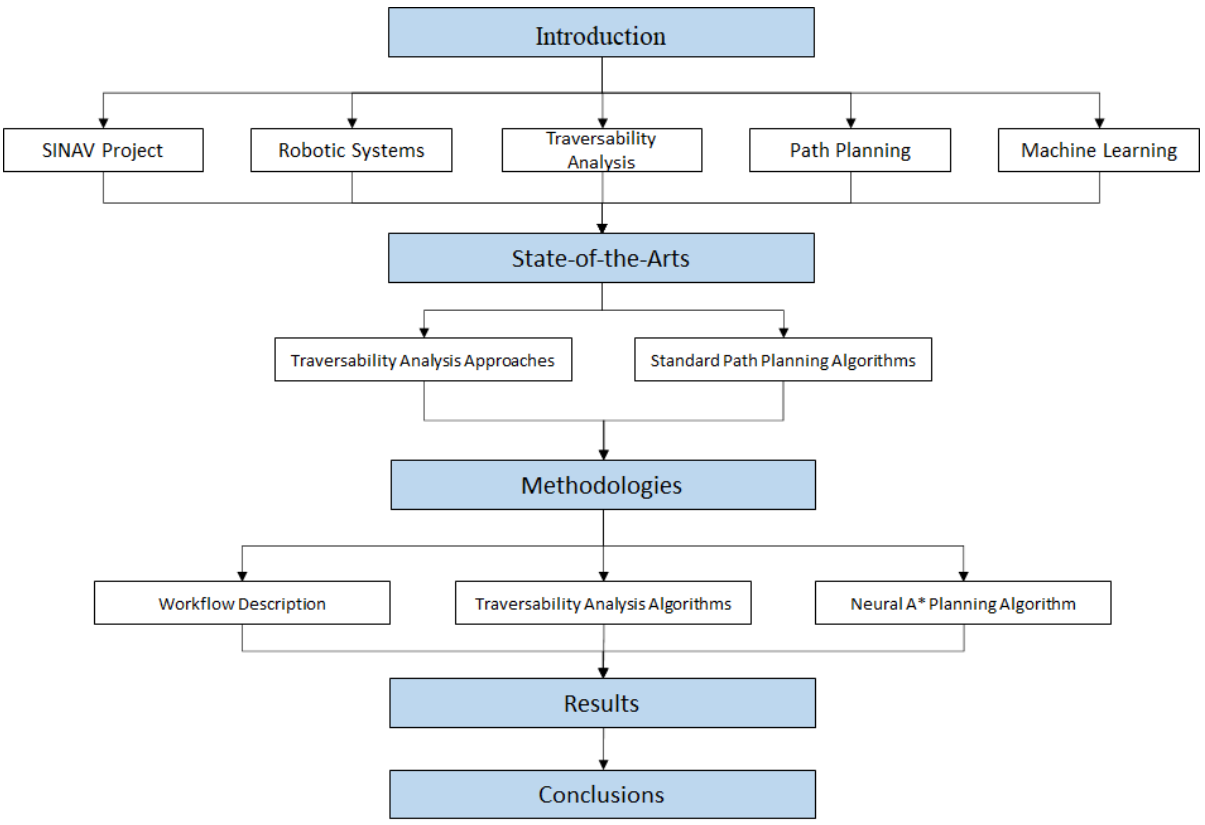

Figure 1: Thesis Structure

1. Introduction

The research presented in this Thesis work was conducted within the scope of the SINAV project ("Soluzioni Innovative per la Navigazione Autonoma Veloce"), in collaboration with the Polytechnic of Turin, Altec S.p.A, Aiko S.r.l., Thales Alenia Space Italia S.p.A. and sponsored by the Italian Space Agency (ASI), and it deals with the issues related to the application of machine learning algorithms to autonomous planetary rover systems, more specifically for the definition of traversability maps and, consequently, the implementation of dynamic path planning strategies.

The aim of the first paragraph of this Chapter is to provide an in-depth explanation of the SINAV project mission, in order to specify what are its key objectives and which of them have been addressed by this Thesis work. In the second paragraph, a high-level overview of robotic systems is presented; this is crucial for the scope of further introducing the traversability analysis and path planning problems, as an at least basic understanding of how the different components are correlated is essential. The following two paragraphs deal with the definition of traversability analysis and path planning tasks in the context of robotic systems, more specifically for autonomous-driving planetary rovers. Finally, the last paragraph is dedicated to a brief introduction to Machine Learning algorithms, highlighting their relevance and potential with respect to the tasks described in the previous sections.

1.1. The SINAV Project

The SINAV project is a major initiative being led by the Italian Space Agency with the aim of creating a new generation of autonomous planetary exploration rovers. The key objective of this project is to develop an advanced autonomous navigation system that would allow rovers to explore planetary surfaces with greater efficiency and effectiveness with respect to the currently employed planetary rovers. For this purpose, the project involves the development of advanced technologies and algorithms for visual perception, object recognition,

localization and mapping, as well as real-time decision-making and planning. This implies a high level of interaction among different complex systems, such as the rover, the lander, drones and satellites; Figure 2 contains a schema that describes a reference operational sequence, where all the main activities of such components are highlighted (MCC refers to the Mission Control Centre).

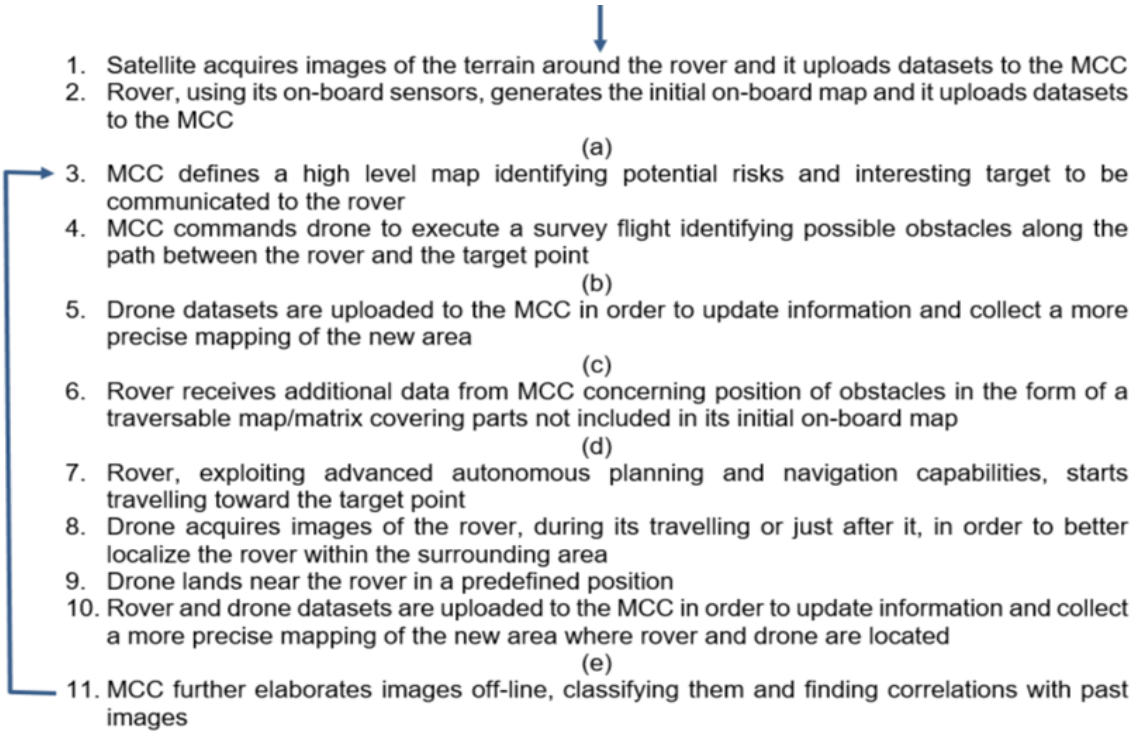

Figure 2: Reference operational scenario for the SINAV project

This Thesis work aims to address the issues related to tasks 5 and 7, that concern the development of a traversability mapping and path planning algorithm in order to enable the rover to traverse a region with very limited human interaction, by only using the monocular pictures of the area that are provided by the drone, together with the data gathered by the analyses that had been conducted before these steps.

The main source of information that has to be exploited for the purpose of traversability analysis and path planning within this part of the SINAV project is the data obtained from the support drone. This device produces images of the environment with a range of about 100-300 m and a 5-50 cm resolution, thus enabling the establishment of a mid-resolution

traversability map that will then be fed to the path planning algorithm, which will subsequently select the optimal route to make the rover reach the goal state. Fellow researchers that took part in the SINAV project also developed deep learning algorithms in order to produce terrain labels for drone imagery, segmenting the different areas; this is another important piece of information that is used as input for the research conducted throughout this Thesis.

The SINAV project also makes an effort to incorporate Artificial Intelligence and Machine Learning techniques to enhance the rover's ability to operate in challenging environments with limited prior knowledge. The same approach has been adopted in this Thesis work, as every analysis step within the aforementioned tasks exploits Machine Learning models for its purpose.

This research will now focus on a brief overview of robotic systems. This is done in order to gain a better understanding of the context in which the traversability analysis and path planning tasks will be defined, and to consequently establish the appropriate workflow required to solve these problems using the available data and learning-based algorithms.

1.2. Robotic Systems Overview

In the rest of this work, one shall benefit from the description first formulated by M. W. Otte [1] of general-purpose hypothetical robotic systems, which can be broken down into four main subsystems:

- *Sensing*, subsystem that involves the sensors and algorithms used to detect and interpret information about the robot's environment;
- *Representation*, method used to organize the pieces of information coming from the sensing subsystem, in order to create and maintain a model of the environment that the robot is operating in;
- *Planning*, subsystem in charge of deciding what actions have to be taken based on the information in the representation and the system’s goals;

- *Actuation*, method used by the robot to perform an action on the environment, which involves controlling the robot's actuators to execute the planned tasks.

These subsystems are interconnected and interdependent, and are essential for enabling a robotic system to operate autonomously and perform a wide range of tasks. However, as it will be exposed in the next chapters, this division between subsystems is often not so clear-cut; indeed, all of the relationships that are present within these components shape the definition and implementation of the different functions inside the system.

In the context of this study, the most important correlation is the one between *representation* and *planning*, as both traversability analysis and path planning are strongly related to these two subsystems.

The next section will now focus on the description of these two specific tasks in the context of autonomous-driving planetary rovers.

1.3. Traversability Analysis Introduction

For autonomous robotic systems, the ability to distinguish between *traversable* and *untraversable* (or hazardous) terrains is fundamental for both system safety and mission accomplishment.

In order to perform an accurate and safe planning activity, the data obtained by the sensing subsystem have to be processed and fed in a specific way to the planning algorithms; the elaborations required for these data depend on the type of data themselves, the specifics of the planning algorithm, and the kind of information we want to embed in the modeling of the environment, that primarily have to specify where the rover can or cannot go.

Traversability analysis is the task of gathering information about the *hazards*, i.e. elements or parts of the environment that cannot be passed over by the rover, in order to unequivocally describe their location and size. This, *as any other function to be implemented within the robotic system*, can be carried out in many different ways that one may, in the first instance, divide into the following categories: “static” analysis, “learning-based” analysis, and a

mixture of both; the first refers to methods such as thresholding, i.e. choosing a threshold value for certain features and using those values to discriminate between traversable and non-traversable regions, and the second alludes to models that use any kind of machine learning algorithm to develop an implicit method to differentiate the areas as accessible or not.

While the previously described operation is essential, it is not sufficient to provide the planning framework the information it needs to pursue the search of a feasible path, as besides an explicit annotation of obstacles this search also requires a geometric and/or topological representation of the environment, i.e. a *map,* whose generation is assigned to the *traversability mapping* task, hereby considered as a component of the broader traversability analysis problem.

The world map should be capable of fully describing the different locations in terms of traversability, that is the capability of the given robotic system to traverse the region given its specifics; the process takes, as input, the results of the traversability analysis and outputs the data structure on which to perform the planning activity.

Traversability maps can take many forms, from the very simple (and commonly used) binary matrix representation of the environment, which just tells us if an area is navigable or not, to a very complex data structure that is able to give profound insights about the region that is being examined. This choice mainly depends on the path planning algorithm's requirements, based on the type of operations it carries out.

In general, environmental models are usually referred to as *state space* when discussing a discrete model of the world, and as *configuration space* (*C-space*) for continuous models [1]; these are basically a set of all possible configurations of the robot. Within these spaces, it is possible to further define the *free space*, which is the set of all the robot's arrangements that avoid collision with obstacles, and its complement, the *obstacle space*, which usually also includes the regions that are forbidden (for example, the regions outside a specific area). *Target space*, instead, is a subset of the free space that we set as the goal configuration for the robot's mission.

The part of the SINAV project dedicated to traversability mapping aims to output a binary traversability matrix starting from a single RGB image of the terrain, taking into consideration the factors introduced in Chapter 2.1 that represent the limitations of the rover's navigability; it's important to acknowledge that this is a highly complex task, and the techniques adopted in this work were intended to approximate, with as much detail as these methods can provide, the results that are typically achieved through the use of more complex and comprehensive techniques, which also rely on a wide range of sensors that are not utilized in this study.

1.4. Path Planning

Path planning is a critical component of many robotics and autonomous systems, and is essential to achieve safe and efficient navigation in complex environments. This section provides a formal problem statement for the path planning task. It is important to specify that path planning and path searching are two related concepts, but although the terms have been used interchangeably in many contexts and the same will be done in this work, it's worth mentioning this distinction in the first introduction of these problems, in order to avoid confusion later on.

Within a graph representation of the state space, path searching is a well-studied computational problem defined as the task of identifying a sequence of nodes that connects the start and goal nodes while trying to minimize certain metrics, such as the distance traveled or the time spent traveling from the starting node to the final one. Path planning, instead, involves not only finding the optimal path between two points but also considering other factors, such as the definition of the start and goal states, the robot's dynamics limitations and the presence of hazards. For the sake of readability, in this work all the problems related to the organization and analysis of the environmental data for the final purpose of defining a traversability map, that also takes into account the navigation system's technical constraints, are considered part of the traversability analysis task; instead, the problem of finding a specific path between the starting position and the goal state is referred to as path planning; while it's already been said that the latter definition is more formally

representative of the path searching task then the path planning one, throughout this work the two terms are used as synonyms, in accordance to what the Neural A* researchers did in their paper [2], because the distinction does not have practical implications for this Thesis purposes.

As in the case of traversability analysis, there is no standard way to perform path planning within a robotic system, nor within a planetary rover system, because many factors have to be considered when approaching this kind of issue. One of the most important aspects of this problem is the type of representation that is used, i.e. the pieces of information it is granted access to and their organization; in the case of *graph-searching algorithms*, the world is described through a set of vertices and edges, where a vertex represents a given pose of the robot and an edge represents a connection between two states, i.e. an action.

Every graph-search algorithm assumes the existence of V, E, $v_{start,}$ $v_{goal,}$ which are, respectively, the set of all vertices, the set of all edges, the starting position and the target position; in general, these algorithms do not require for the graphs to be directed or undirected, so, using the approximation of an undirected edge as a couple of opposite-pointing directed edges, it is assumed that all edges are directed. A node is considered unexpanded, at the time of a specific algorithm's step, if it still hasn't been reached by the searching algorithm; if it has been reached but at least one of its neighbors have not, it's said to be open; if the node has been reached and the same can be said about its neighbors, it's then said to be expanded. During the path searching procedure, the open-list contains all the nodes that are open at a certain step. The search is considered as successfully terminated if the goal node is expanded, and it's said to be unsuccessfully terminated if the open list is empty but the goal node is not expanded.

1.5 Machine Learning Overview

Machine learning algorithms have revolutionized the field of artificial intelligence, enabling computers to learn from data and make predictions or decisions based on that learning. These algorithms are often broadly classified into five categories, based on the type of training data available and the type of feedback they receive from the environment: supervised learning,

unsupervised learning, self-supervised learning, semi-supervised learning, and reinforcement learning. A short introduction to each of these techniques is now presented, in order to then provide a more detailed description of the basic mechanisms employed in the approach that has been mainly utilized within this work.

Supervised learning [3] is among the most common forms of machine learning, drawing inspiration from the way human beings gain information from their experiences in order to make predictions in new and unknown situations. In supervised learning, the algorithm is trained on a labeled dataset, where the correct output is already known. During training, the algorithm learns to map inputs to outputs by adjusting its parameters based on the difference between its predictions and the true output. The trained model can then be applied to make predictions on unseen data. The main issue regarding these algorithms is due to the lack of annotated datasets, whose absence or scarcity can greatly impact the results of said methods.

Unsupervised learning [4], on the other hand, involves training an algorithm on an unlabeled dataset. The models in question try to identify patterns and structures in the data without any explicit feedback from the environment; they are often used for tasks such as clustering, dimensionality reduction and anomaly detection, and their main limitations derive from the difficult interpretation of the results, as the quantitative and qualitative evaluation of the output is not always straightforward.

Self-supervised learning [5] methods actually belong to the class of unsupervised learning; they differ from the latter as they involve using a proxy task to create labels from the data itself, whereas unsupervised learning does not use any explicit labels. These algorithms are trained to predict some aspect of the input data, such as predicting the next frame in a video, filling in missing parts of an image, or predicting the context of a word in a sentence. These predictions serve as labels for the data, which can then be used to train the model in a supervised manner. The new representations that are learned by these models are usually exploited in downstream tasks, in cases where transfer learning is adopted.

Semi-supervised learning [6] is a combination of supervised and unsupervised learning. In semi-supervised learning, the algorithm is trained on a partially labeled dataset, where only a subset of the data has labels available. The algorithm tries to use these labeled data to learn the structure of the data, in order to then apply this knowledge to the unlabeled data.

Finally, reinforcement learning [7] is a type of learning where the agent (i.e. an entity that interacts with its surroundings) learns to make decisions based on rewards or punishments it receives from the environment. The algorithm tries to maximize its reward over time by learning to take actions that lead to positive outcomes, and avoiding actions that induce negative outcomes. Reinforcement learning is often used in robotics, game theory, and other applications that require optimal decision making in complex environments.

Overall, the choice of which type of machine learning algorithm to use depends on the nature of the task at hand, the type and abundance of data that are available, and the resources and constraints of the specific application.

In the rest of this work, the emphasis will be mainly on supervised approaches, so a deeper explanation of these models' basic components and their meanings is now proposed. Supervised learning models can be described in terms of:

- Input data: this is the labeled data on which the model is trained. It consists of a set of features or attributes that describe the characteristics of each example in the dataset from which the algorithm can learn to make predictions.
- Output data: this is the set of values that the model is trained to finally predict. It typically consists of target values, i.e. labels or numbers, that correspond to the input data.
- Model architecture: This refers to the specific type of model that is being used for the task, such as a decision tree, support vector machine or neural network. The architecture determines the type of mathematical functions that the model can learn. Later in this paragraph, a more in-depth description of the neural networks architectures' components can be found.
- Loss function: This is a mathematical function that measures the difference between the predicted output and the actual output for a given input example. The goal of the model is to minimize this function, which is achieved by adjusting the model's parameters during training.

- Optimization algorithm: This is an algorithm that is used to minimize the loss function by adjusting the model's parameters during training. Examples of optimization algorithms include gradient descent and stochastic gradient descent.

As this Thesis work relies on neural networks models for the most part, a more specific dissertation on such models and their architecture is now submitted.

Neural networks are generally composed by a series of interconnected layers of nodes, each of which performs a linear transformation followed by a non-linear activation function. The input layer is the first one of the network, as it receives the raw input data, while the output layer is the last and produces the final output; between the input and output layers there are typically many hidden ones, each of which performs a non-linear transformation on its input. Generally, each node in a layer is connected to every node in the previous one, and these connections are represented by weights that also reflect its strength. During the training process, the network learns to adjust these weights in order to minimize a loss function, which measures the difference between the network's output and the expected one. The backpropagation algorithm is typically used to update the weights in the network during training; it computes the gradients of the loss function with respect to the weights, and then uses these gradients to update the weights in the opposite direction of the gradient. This process is repeated iteratively until the network's performance on the training data reaches an acceptable level.

Different types of hidden layers can be used in a neural network, each with its own specific function. The most common types include fully connected layers, convolutional layers, and recurrent layers. Fully connected layers connect every neuron in the current layer to those of the previous ones, serving as a way to extract higher-level features from the input data; care must be taken in order to avoid overfitting, as fully connected layers can easily lead to a large number of learnable parameters. Convolutional layers are used for image and video processing, and are designed to identify local patterns in the input data; they are mainly described in terms of:

- *kernel size*, where each kernel (or filter) corresponds to a feature map in the output;

- *stride,* that represents the step size of the kernel as it moves across the input data, with higher values being used to reduce the dimensions of the output feature map;
- *padding*, which is a technique used to preserve the spatial dimensions of the feature map by adding zeros to the edges of the input data;
- *activation function*, which introduces non-linearity into the output and it's fundamental when trying to learn complex behaviors and patterns within the input data.

Batch normalization layers, instead, are used to improve the training speed and stability by normalizing the output of a previous layer; these were introduced in 2015 by [8], and basically, for each dimension of the output, they subtract the mean and divide by the standard deviation of the activations in the mini-batch, in order to center the distribution of the output around zero and scale it to have unit variance, and finally apply shift and scale to the data to bring them in a more useful range. This technique has shown great success in improving the performance and stability of different neural networks by reducing the internal covariate shift, and is widely used in many state-of-the-art models, as those introduced in Chapter 2 and furtherly employed inside the workflow described in Chapter 3.

2. State-of-the-Arts

This Chapter presents some of the most notable approaches and algorithms that have been adopted for the tasks of traversability analysis and path planning in the literature, whose insights have been fundamental in the research conducted in this Thesis work.

The research regarding traversability analysis, at first, focused on finding an approach that had already been used in the context of autonomous driving martian rovers, in order to define the main elements that can affect the capability of a rover to traverse a given area; subsequently, another effort has been done to analyze the methods currently used for the purpose of obtaining data related to the aforementioned elements, as the mission required an end-to-end model capable of obtaining a traversability map starting from the monocular images collected from the drone, in conjunction with the terrain labels that were produced through algorithms developed by other researchers in preceding stages.

For path planning, instead, the literature work firstly concentrated upon the traditional path searching algorithms, in order to find out what were the classical approaches to this problem and their limitations; lastly, learning-based algorithms have been examined, for the purpose of understanding how these methods could benefit the path planning task and also which of them could be useful for the objectives of this Thesis.

2.1 Traversability Analysis Approaches

In this paragraph, an effort is made toward the definition of the state-of-art techniques employed for traversability assessment and traversability maps definition. It is although dutiful to state that no such thing as a “standard” way to perform these analyses exist, as these problems and their solutions greatly depend on the kind of systems that are involved and on the type of data one has access to. For this reason, the first part of the literature study conducted for this work concerned the establishment of a precise workflow that had to be followed in order to obtain a reliable traversability map, starting from the data that were

available; consequently, the study [9] is hereby presented, which was lead by the Jet Propulsion Laboratory researchers and had a starting point that was similar to the SINAV project problem statement, as will be furtherly discussed in the next sections. However, the specific methods employed in their study to obtain the data required for traversability mapping fairly differ from those used in this Thesis work; if the first part of the literature study focused on understanding what kind of data are needed to estimate the traversability of an area, the second part focuses on the tools that can be used in order to obtain those data.

For the Mars 2020 mission, which was part of NASAs Mars Exploration Program, in [9] the researchers addressed the challenge of selecting the best candidate landing site, out of eight proposals, in terms of:

- Safety of the landing procedure;
- Possibility of exploring two scientifically diverse Regions of Interest, within 1.25 Mars years (i.e. 836 sols, Martian days), starting from the landing site;
- Traversing time and distance optimality;

The eight possible landing spots are very different with respect to both the ROIs that lay near them and engineering viability. In order to estimate mission design and rover capabilities prior to a detailed analysis of each landing site, the chosen method involves the exploitation of a baseline reference scenario. Such scenario foresees the rover traversing 6 km to get from the landing site to the first ROI; then, another 1.5 km will be traversed within the ROI to collect 10 samples for a (notional) later sample return mission; the rover will then traverse 6 km to reach the second ROI, where it will finally traverse 1.5 km to collect other 10 samples. To address this issue, which also includes multi-ROI planning, the researchers conducted their analyses using the very popular HiRISE dataset, composed by images taken by Mars Reconnaissance Orbiter with 0.3 meter/pixel resolution; they developed a set of automated traversability analysis methods called Mars 2020 Traversability Tools (MTTTT), which are schematized in Figure 3.

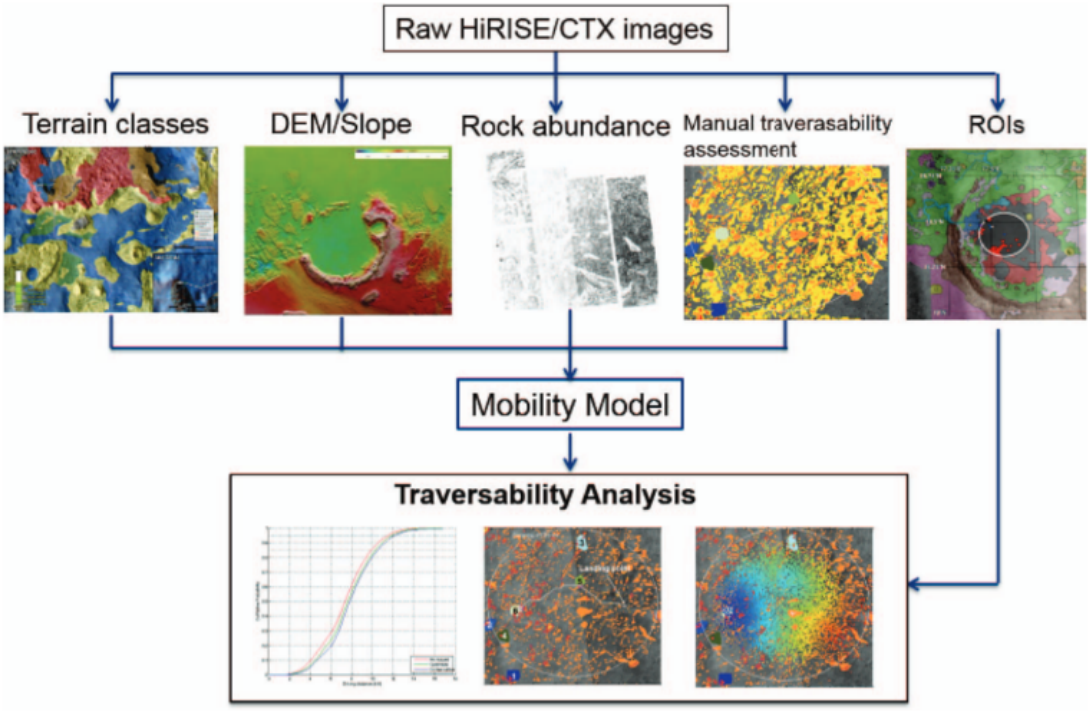

Figure 3: MTTTT schema

Researchers state that the MTTTT includes the following:

- Automated terrain classifier, which assigns a label to each pixel of an HiRISE image based on the type of terrain it classifies;
- DEM generation algorithm, which produces a Digital Elevation Model through stereo processing, in order to then derive a slope map;
- Automated rock detection, which performs rocks segmentation based on a fracture and fragmentation algorithm, in order to then output a Cumulative Fractional Area (CFA) map;
- SILT, which is a labeling tool used to collect manually-generated information about hazards near the ROIs;
- Mobility Model, which is a cost function for route planning and takes as input terrain type map, slope map, CFA map and hazard map and outputs the cost map. It is split

between a binary model and a continuous-valued model, the first is used for distance-minimal planning and simply sets thresholds for both CFA and slope to understand if an area is traversable or not, the other is used for time-optimal planning and maps both the values about CFA and slope to the expected rover traversability speed;

- Multi-ROI Route Planning, which outputs a route in order for the rover to visit the selected ROIs, minimizing the distance traversed or the time consumed.

For the purpose of this Thesis, which corresponds to the traversability estimation and mapping task of the SINAV project, we shall focus mainly on the DEM generation algorithm, the automated rock detection algorithm and the binary Mobility Model.

The DEM generation algorithm is based on stereo imagery taken by the HiRISE camera on the MRO spacecraft, and includes the following processing steps:

- Stereo correlation between HiRISE image pairs;
- Triangulation for correlated pixels in order to generate a point cloud in a martian coordinate frame;
- Ortho-projection of the point cloud into a DEM;

Within the actual HiRISE products, one can also find a satisfying amount of already-generated Digital Terrain Models (DTMs) examples, which can be treated in the same way with respect to the DEMs (these two types of data are, in fact, considered as equivalent from now on) in order to extract the values that we can use to compute information regarding slope, i.e. slope maps; this is crucial for traversability assessment, as it is fair to say that the slope of an area is one of the factors that have the most impact on the viability of that area for a planetary rover.

The steps required by the DEM generation algorithm described above, however, are all time-consuming and also require a certain degree of familiarity with this type of processing, which is not trivial; both constraints led to the conclusion that a learning-based model that could be able to approximate a DEM, given as input a simple RGB or grayscale image, would be the best choice in order to avoid the technical difficulties that come with a "static" non-learning-based analysis, and also could greatly serve the purpose of showing how machine learning models could be useful for such missions.

Regarding the automated rock detection algorithm, researchers used the method first described by Golombek et al. [2] that is based on a fracture and fragmentation theory; this algorithm consists in the usage of a static image processing technique in order to enhance the contrast of the areas recognized as shadow, and then using the sun incidence inclination to derive rock height and width based on the shadow's dimensions. Having access to such pieces of information enabled the researchers to use statistical exponential models to then estimate the size-frequency distribution of rocks. This method's ultimate purpose is obtaining the Cumulative Fractional Area (CFA) values, from which to elaborate a CFA map, which is a measure of rock abundance and it is of great interest for the objective of traversability assessment. These values are derived from the general size-frequency distribution power law:

$$F_k(D) = ke^{-q(k)D}$$

Equation 1: General Size-Frequency distribution power law

$F_k(D)$ is the area populated by rocks with a diameter at least equal to D, D is the rock diameter (they considered only rocks with a diameter within 1.5m and 2.5m), $q(k)$ is a function that controls the decay speed with respect to an increasing diameter, k is the fraction of total area covered by all rocks, which is basically the CFA or rock-abundancy score they want to estimate.

The same considerations made regarding the DEM generation algorithm can be made on their automated rock detection algorithm, as also this approach is both time-consuming and technically challenging, for someone who is not familiar with fracture and fragmentation

algorithms and geological analysis, so other options have been considered for the purpose of this study within the SINAV project mission and will be furtherly described in the next Chapters.

Their binary Mobility Model finally uses terrain slope, CFA values and terrain type to establish a (binary) traversability map, that basically tells if an area is either traversable or hazardous by setting a different threshold for both CFA values and slopes for each terrain type, then checking that the thresholds are respected for both parameters.

This study shows how, in order to establish the traversability of a given area, the main variables that have to be taken into consideration are the slope of the area and its CFA score; in this Thesis work, the methods used to obtain such data fairly differ from those adopted by the JPL researchers, as an effort has been made in order to use learning-based models to avoid the tedious, geology-related and time-consuming analyses that are normally required for deriving these data.

Given that one of the objectives of the SINAV project was to establish a traversability map starting from a monocular RGB image of the Martian terrain, in order to derive an approximation of the slope of the area it was necessary to first obtain information about the depth of the scene; this task is generally referred to as depth estimation, and in the case of a single RGB image as input it's called monocular depth estimation.

For generic depth estimation tasks, a wide body of literature [10] [11] [12] dating back to the 1970s focused on developing algorithms to extract depth information from stereo image pairs, basically attempting to mimic the way the human brain perceives depths by exploiting the differences between what is seen by the left eye and what is seen by the right one. This process is called *stereopsis*, and in stereo vision algorithms it is usually implemented by using two (or more) cameras to capture different viewpoints of a scene, in order to then compare them and extract depth information.

The step required to obtain the information regarding the slope of the scene for traversability mapping in this Thesis work, instead, deals with the inference of depth information from a single RGB image of the simulated Martian terrain taken from the drone, so it falls within the

category of monocular depth estimation. It should be noted that this is actually an ill-posed problem, as the goal to recover the 3D geometry of a scene from a single 2D image is inherently ambiguous and under-constrained, because given a monocular picture of an area there could be different possible depth maps capable of explaining the observed image. Nonetheless, many studies addressing this topic, in an attempt to estimate with acceptable precision the depth information, can be found in literature, and some remarkable approaches are hereby introduced, including the two methods that have been most useful for the research presented in this Thesis work.

The standard (i.e. non-learning based) methods used for monocular depth estimation mainly rely on one of the three oldest approaches, or on improved versions of the latter:

- "Shape from shading" [13], where the intensity of the image is used to estimate the surface normals of the object or scene, which can then be used to compute the depth of the surface points.

- "Depth from Defocus" [14] , which uses the blur (i.e. the loss of sharpness or detail) of the image, caused by the distance of the surrounding area from the point of focus, to estimate the depth of the scene.

- "Depth from Motion" [15], that leverages on the presence of moving objects within the scene to derive the distance of the points from the camera.

Even though during the years these models have been refined and enhanced, certain limitations regarding these methods still endure and consequently paved the way for learning-based approaches, that are nowadays the most frequently adopted tools for such analysis. These constraints include, but are not limited to, the assumptions that the surface of the object or the scene is Lambertian (i.e. the surface reflects light equally in all directions), either the camera or the scene have to be stationary, certain parameters have to be known precisely, and the results are very sensitive of different lighting and noise conditions.

Machine learning models effectively try to tackle these limitations, often producing better results while still being less time consuming, as they can learn to leverage large amounts of

data to infer the depth information by capturing complex and non-linear relationships between image features and depth cues. For these reasons, two very different machine learning models have been applied and compared on the available dataset for the objective of estimating the depth of the scene, in order to then evaluate the results and finally proceed with the next steps required for traversability mapping.

In the first stage of the research, the focus has been laid on a lightweight model that could be trained and fine-tuned on the dataset that was available of simulated martian terrain images; while selecting the following algorithm, the main criteria that have been taken into account are the availability of computational resources, the quality and quantity of training data, and whether or not the model had already obtained reasonable results when tested on relatively small (~10^3 samples) datasets.

The model in question is the official implementation of Monocular Depth Estimation from the Keras.io “Computer Vision Examples” [16], that yielded acceptable results when trained on a dataset with the same order of magnitude as ours and contains 2 million learnable parameters, so it was actually suitable for our system.

This algorithm uses a U-Net [17] architecture with batch normalization and additive skip connections; more specifically, the encoder-decoder architecture of this model is composed of three main blocks:

- Downscale block, which consists of two convolutional layers with leaky ReLU activation and batch normalization, followed by max pooling; this is used for feature extraction.

- Bottleneck block, that presents two convolutional layers with leaky ReLU activation; this component outputs the feature maps with the same size as the input, and is used as a bridge between the downscale and the upscale blocks;

- Upscale block, which is composed of an upsampling layer, a concatenation with skip connections between the feature maps from the encoder and the current layer's feature

maps, two convolutional layers with batch normalization and leaky ReLU activation. This finally outputs the depth estimations.

Figure 4 depicts the model architecture. This encoder-decoder structure is very popular in the field of computer vision, as the symmetrical nature of the architecture allows it to capture multi-scale information and details at different levels of abstraction; the encoder part uses convolutional layers to reduce the spatial resolution of the input while increasing the number of feature maps (i.e. channels), in order to serve as a high-level feature extractor; the bottleneck layer further reduces the dimensionality of the input while maintaining the channel size, helping to capture the most important features in a compact representation; the decoder part, instead, uses transposed convolutions in order to upsample the feature maps while reducing their channel dimensionality. The skip connections between the encoder and the decoder try to preserve the fine-grained details of the input and to reduce the vanishing gradient problem during training.

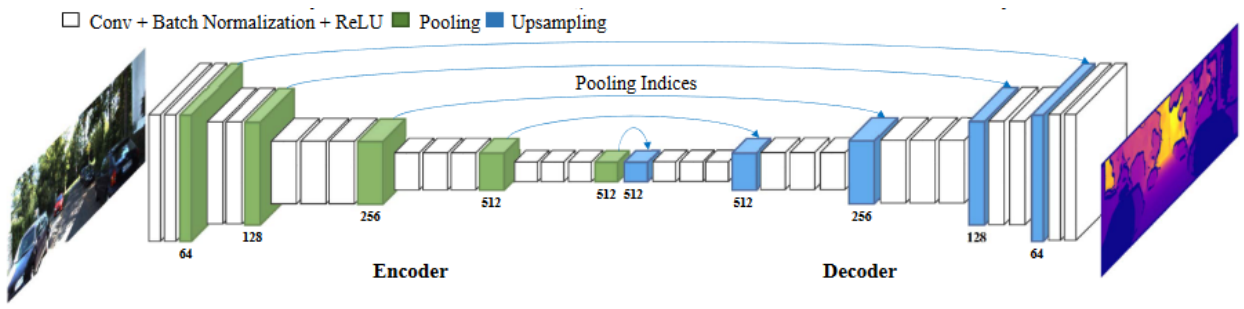

Figure 4: U-Net architecture of the original Keras MDE Model

In this model, the loss is computed as a weighted sum between three functions:

- Structural similarity index(SSIM).
- L1-loss.
- Depth smoothness loss.

The first term measures the perceptual difference between two images, with values between -1 (perfectly dissimilar) and 1 (same images); this metric gives profound insights on the degree of similarity between the ground truth depth map and the one predicted by the model, as it is also scale and shift invariant (i.e. it does not take into consideration different uniform scales and translations of the data for the purpose of measuring their similarities). Its mathematical definition can be found in Equation 2.

$$SSIM = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x + \sigma_y + c_2)}$$

Equation 2: Structural Similarity Index Measure[1]

The second term of the loss is the mean absolute error, which measures the average absolute difference between the corresponding pixels of the two depth maps. The MAE gives an indication of how different the two images are in terms of their actual pixel values, with lower values indicating better agreement between the images; it should be noted that the MAE does not take into account any perceptual differences between the images, such as changes in contrast or brightness, and for this reason it should not be the main criteria to evaluate how similar two generic images are, but in the case of monocular depth estimation for autonomous vehicles even small differences between the estimated value and the real one can have significant impact. Equation 3 contains the formula used for this purpose.

$$MAE = \frac{\sum_{i=1}^{n}|y_i - x_i|}{n} = \frac{\sum_{i=1}^{n}|e_i|}{n}$$

Equation 3: Mean Absolute Error

The third is the sum of the averaged gradients of the estimated depth maps, in order to encourage the "smoothness" of the prediction. Algorithm 1 contains the steps used to calculate said element, using the Tensorflow library.

[1] Here, μ refers to the average of the image, σ² refers to the standard deviation, c_1 and c_2 are two variables that are used to stabilize the division with weak denominator and are computed as $c_1=(k_1L)^2$ $c_2=(k_2L)^2$, where $k_1=0.01$, $k_2=0.03$, L=dynamic range of the pixel values.

```
dy_true, dx_true = tf.image.image_gradients(target)
dy_pred, dx_pred = tf.image.image_gradients(pred)
weights_x = tf.exp(tf.reduce_mean(tf.abs(dx_true)))
weights_y = tf.exp(tf.reduce_mean(tf.abs(dy_true)))
# Depth smoothness
smoothness_x = dx_pred * weights_x
smoothness_y = dy_pred * weights_y

depth_smoothness_loss = tf.reduce_mean(abs(smoothness_x)) +
tf.reduce_mean(abs(smoothness_y))
```

Algorithm 1: Computing the Depth Smoothness Loss

This firstly involves the computation of the gradients of both the ground truth and the estimated depth maps in x and y directions, then it calculates the weights for the x and y gradients using the exponential of the mean absolute value of the corresponding gradients in the ground truth depth map, afterwards the depth smoothness components are derived by the element-wise product between the predicted gradient and its component's weights, so finally the loss is computed as the sum of the absolute values of the depth smoothness terms in the x and y directions.

Originally, the weight of these components inside the overall training loss was, respectively, of 0.85, 0.1 and 0.9, meaning that the authors valued the prediction smoothness and its visual resemblance to the ground truth as more important with respect to the point-wise depth loss.

Other important hyperparameters such as the resized input dimensions, learning rate, the number of epochs and the batch size are further discussed in Chapter 3, where it will also be shown how these have been changed in an attempt to better fit our case-study. It should be taken into consideration, in fact, that these parameters and the weights of the terms inside of the loss function have been adopted while training the model on the DIODE [18] dataset validation set (a subset of the original dataset, which originally contained more than 25000

images), exploiting 1402 examples of both indoor and outdoor images with a ground truth depth precision of ±1 mm, with a maximum range of 350 m and a minimum range of 0.6 m.

Another state-of-the-art algorithm that caught our attention when approaching the monocular depth estimation task is MiDaS [19] (Mixed Dense and Sparse), and it achieved state-of-the-art performance on a variety of popular benchmarks such as KITTI [20] and Make3D [21] datasets.

The researchers that developed this model highlight the need for large and diverse training sets in order to obtain satisfying results for the task of monocular depth estimation. In their paper, their initial assumption is that a model trained on such a rich dataset with an appropriate training loss is capable of delivering state-of-the-art results even on unseen environments; they proved this intuition using what they refer to as “zero-shot cross dataset transfer”, a protocol that basically consists in testing the model as-is on a completely new dataset after an extensive training procedure on a variety of mixed datasets.

They trained different backbones on a fusion of 12 different datasets, identifying three main challenges that had to be solved as to establish a robust training procedure to handle such diversity of data:

- different representations of depth (direct or inverse)
- scale ambiguity
- shift ambiguity

Scale and shift ambiguity are two common issues encountered in the task of monocular depth estimation; even if the relative distance between objects in the scene can be estimated accurately, it’s impossible to determine the actual distance from the camera, as also adding a constant to all the values of a depth map would result in a shifted depth map that would look identical to the original one, so the results of these methods are often scaled and shifted arbitrarily.

They addressed all of these through the choice of applying the inverse depth representation, as it is generally more robust to depth outliers, and the definition of a final loss function that is actually scale and shift-invariant, i.e. it’s insensitive to different depth scales and shifts.

The final loss formulation is shown in Equation 4, where N_l is the training set size, α is empirically set to 0.5, and $\hat{d}$, $\hat{d}_s$ are, respectively, the scaled and shifted versions of the predicted and ground truth depth values.

$$L_l = \frac{1}{N_l} \sum_{n=1}^{N_l} L_{ssitrim}(\hat{d}^n, (\hat{d_s})^n) + \alpha L_{reg}(\hat{d}^n, (\hat{d_s})^n)$$

Equation 4: Final MiDaS Loss

The overall loss is computed as the weighted sum of two terms for all the samples in the training set: the first is a robust loss, denoted by $L_{ssitrim}$, that measures the mean absolute error between the scaled and translated versions of the ground truth and predicted depth values, whose particularity is that it doesn't consider the 20% largest residuals to mitigate the influence of outliers in the training process, as it's shown in Equation 5;

$$L_{ssitrim}\left(\hat{d}, \hat{d_s}\right) = \frac{1}{2M} \sum_{j=1}^{U} \rho_{mae}\left(\hat{d}_j, \hat{d}_{sj}\right)$$

Equation 5: First component of the final MiDaS loss[2]

the second term is the regularization loss, denoted by L_{reg}, which penalizes large changes in depth values and encourages smoothness in the predicted depth map. The formula is shown in Equation 6, where $R_i = \hat{d} - \hat{d}_s$ and R^k denotes the difference between the two maps at scale k (here, K is set to 4, halving the image resolution at each step).

$$L_{reg}\left(\hat{d}, \hat{d}_s\right) = \frac{1}{M} \sum_{k=1}^{K} \sum_{i=1}^{M} \left(\left|\nabla_x R_i^k\right| + \left|\nabla_y R_i^k\right|\right)$$

Equation 6: Second component of the final MiDaS loss

[2] Here, M is the number of pixels in each image, U=0.8M, and j is an index that enables the tuples of ground truth depth values and predicted ones to be accessed sequentially, ordered in a way that at j=0 there is the couple with the least mae score and at j=M there's the highest one.

The two pre-trained models that have been used for the experiments later presented in Chapter 3 are indicated as “DPT_BEiT_large_512” and “DPT_swin2_large_384”.

"DPT" stands for Dense Prediction Transformers [22], which is a class of models that were first adopted for natural language processing but are now being used for computer vision tasks; these are capable of producing dense outputs thanks to their architecture, which is based on transformer layers, and unlike traditional CNNs (that rely on pooling operations for reducing spatial resolution) they preserve the spatial dimension throughout the network, thanks to the usage of multi-scale feature extractors and dense candidate networks.

“DPT_Beit_L512” is a pre-trained variant of the DPT architecture that uses the BEiT backbone (Bidirectional Encoder Representation from Image Transformers) [23]. The latter uses a masked image modeling task to perform self-supervised pre-training of the vision transformer, tokenizing the input and randomly masking certain patches to be fed to the transformer; in this way, the model learns to recover the missing information and improves its ability to perceive visual features. This is the model with the largest number of parameters (345M) that has been used in the original MiDaS research, which also yielded the best results in terms of improvement from the pre-existing backbones.

“DPT_Swin2_L384” is still another pretrained version of the DPT architecture, which employs the Swin V2 [24] backbone; this network has 213M learnable parameters and shares with the BEiT model the usage of self-supervised pre-training to reduce the need of vast labeled images, and also applies self-attention mechanisms for visual tasks, but their main difference lays in the way the input is processed: in the Swin2 transformer, the input is divided into non-overlapping patches, and self-attention is performed within each patch, which means that each patch attends only to the other patches within its local neighborhood; in BEiT, to compute the self-attention for each token, all the other tokens in the sequence are considered. This means that the computational complexity of the mechanism arises, but it is more capable of capturing long-range dependencies and contextual information.

The size of the input images taken by the models are always reshaped to a 512x512 pixels resolution. In their paper, the authors of the MiDaS model show that their algorithm not only surpassed many state-of-the-art models for monocular depth estimation without any

fine-tuning on the benchmark datasets, but also that performing the fine-tuning on such datasets didn't yield any significant improvement, as the mixed-dataset training procedure produced very stable models that are greatly capable of generalizing the learned features. Further explanation regarding the way these algorithms have been employed and tested inside the proposed pipeline of this work can be found in Chapter 3.

2.2 Traditional Path Planning Algorithms

A non-comprehensive list of the most popular path planning algorithms is now presented, more specifically of standard best-first algorithms as they have laid the foundation for the methods implemented and tested in this work, along with some considerations on the proposed methods.

In some cases, nodes or edges can be associated with a cost that represents the importance of a given node or edge within the path-search; trying to grow the search-tree in order to optimize the total cost of the computed path is considered as a best-first search. The method that has been adopted for this Thesis work and will later be presented is based on this type of searching algorithm. Dijkstra's algorithm [23] is one of the first and most popular optimal best-first searching algorithms that have been formulated. It assumes that the distance $d_{i,j}$ for all edges $e_{i,j}$ is known; the cost for a node c_i is the minimum distance that is required to reach the goal for a given node v_i; the cost for the goal node is $c_{goal} = 0$, while for all the other nodes in the search it is calculated as $c_{i,j} = min_j\, d_{i,j} + c_j$. Algorithm 2 displays the pseudo-code for the Dijkstra's algorithm[3].

The "update" function puts v_i in the open-list, which is implemented as a priority heap, with the priority value of c_i; this is done only if v_i isn't already inside the open-list. In the opposite case, the algorithm updates v_i position in the heap depending on its new priority value c_i. In

[3] In this case, the search is performed in the backwards direction, i.e. from the goal node to the starting node; this is done in order to easily derive the resulting path by following the tree generated by the back-pointers.

this way, the search-tree is reordered every time a cheaper path to the goal is found, thanks to a newly expanded node.

```
Require: G, E, V_start, V_goal

for all v_goal ∈ V_goal do
   insert(v_goal,0,open-list)
while the open-list is not empty do
   v_j = get-best(open-list)
   if v_j ∈ V_start then
      return SUCCESS
   for all v_i such that e_i,j ∈ E do
      if v_i is unexpanded or c_i > d_i,j + c_j then
         c_i = d_i,j + c_j
         update(v_i,c_i,open-list)
         set back-pointer from v_i to v_j
return FAILURE
```

Algorithm 2: Dijkstra's path search

Dijkstra's algorithm is indeed optimal with respect to cost and complete for finite graphs.

In cases where the cost of traveling between two nodes is not defined, a *heuristic* function can provide an estimate of such cost; for two nodes v_i and v_j, it is denoted as $h_{i,j}$. $h_{start,j}$ is instead the estimated cost of traveling from v_{start} to v_j. The A* search algorithm [4] performs a best-first search through the following definition of the cost for a node v_i:

$$c_i = h_{start,i} + d_{i,goal} = h_{start,i} + min_j(d_{i,j} + d_{j,goal})$$

Equation 7: Estimated cost for each node in a standard A* algorithm

here v_j is a neighbor of v_i. The pseudo-code for this algorithm is shown in Algorithm 3.

```
Require: G, E, V_start, V_goal
for all v_goal ∈ V_goal do
   insert(v_goal,0,open-list)
while the open-list is not empty do
   v_j = get-best(open-list)
   if v_j ∈ V_start then
      return SUCCESS
   for all v_i such that e_i,j ∈ E do
      if v_i is unexpanded or d_i,goal > d_i,j + d_j,goal then
         d_i,goal = d_i,j + d_j,goal
         update(v_i,c_i,open-list)
         set back-pointer from v_i to v_j
return FAILURE
```

Algorithm 3: Standard A* path search

The optimality of the path found through the A* algorithm, with respect to cost, is guaranteed under the assumption that $h_{i,j} \leq d_{i,j}$. In this case, the heuristic is called *admissible*. The reason for this behavior is that the algorithm will always find a less expensive path through a node in the open-list, granted that the heuristic does never overestimate the true cost. In the opposite case, possibly better paths could not be explored and the least expensive path could not be found. It's worth noticing that if $h_{i,j} = 0$ for all the nodes in the graph, A* degenerates into Dijkstra's algorithm. In the context of mobile rovers, where the representation of the environment is an occupancy grid, the L^2-norm (Euclidean distance) of the grid centers is considered an admissible heuristic.

Over the last decades, several variants of the A* path searching algorithm have been implemented and applied to many different use-cases, each with its own strengths and weaknesses; some of the most common variants are:

- Weighted A* , that allows to “tune” the algorithm to prioritize either the heuristic or the actual cost to reach the node, depending on the specific application;

- Bidirectional A*, that simultaneously searches forward from the start node and backward from the goal node. BIDA* can be more efficient than regular A* because it reduces the search space by limiting the number of nodes that need to be explored;

- Anytime Repairing A*: Anytime Repairing A* is a variant of A* that works by gradually increasing the search space, while being able to return a solution at any time. This makes ARA* useful for applications where the optimal solution is not required immediately, but it is important to continue searching for a better solution as more time becomes available.

Overall, the different variants of A* offer various trade-offs between speed, memory usage, and accuracy, and the choice of the algorithm strongly depends on the specific needs of the application.

2.2.1 Learning-Based Approaches to Path Planning

The idea of applying machine learning to path planning comes from the desire to improve the performance and adaptability of traditional path planning algorithms. These traditional algorithms, such as A*, rely on predefined maps and require significant computational resources to find optimal paths in complex environments.

Machine learning approaches to path planning, on the other hand, have the potential to learn from experience and adapt to changing environments, making them more efficient and effective in many applications. Machine learning models can also handle high-dimensional state spaces, noisy sensor data, and uncertain environments, which can be challenging for traditional path planning algorithms.

One of the earliest examples of a learning-based approach to path planning was in the field of robotics. In the 1990s [24] researchers began using reinforcement learning to train robots to navigate complex environments. It is recalled from Chapter1 that reinforcement learning is a type of machine learning that involves an agent learning from interactions with an environment through what could be (improperly) referred to as a trial and error approach.

More recently, deep learning approaches, such as neural networks, have been used to learn mappings between sensory inputs and control outputs in robotics. For example, convolutional neural networks have been used to learn obstacle avoidance behaviors in robots [25], while recurrent neural networks have been used to learn navigation policies [26].

In addition to robotics, machine learning approaches to path planning have also been applied in other fields, such as autonomous vehicles, video games, and virtual reality. The potential for learning-based approaches to improve path planning, in such diverse applications, has led to significant research in this area, and the development of new algorithms and techniques for applying machine learning to path planning is an ever-growing area of research.

In [2], researchers present an innovative method for learning heuristics for the A* algorithm using neural networks. As a matter of fact, the quality of the heuristic function plays a critical role in the performance of the A* algorithm, but designing a good heuristic function can be a difficult and time-consuming task. In the context of the Neural A* algorithm, learning an optimal heuristic means that the algorithm is trained to approximate the actual cost of reaching the goal from any given state as accurately as possible, using a neural network. The learned heuristic is then used by the A* algorithm to guide its search and improve its efficiency.

When the Neural A* algorithm is trained on a certain dataset, the heuristic function is optimized to fit the data and to generalize to new unseen data with a similar composition. Once the training is complete, the learned heuristic function is fixed[4] and can be used to solve new instances of the same problem; this fixed heuristic is represented by the learned weights and biases of the neural network, typically stored in a matrix or tensor, so during the testing

[4] Here, the “fixed” term actually refers to the weights and biases of the learned neural network model, that obviously do not change during the inference phase.

phase the input data are passed through the network in order to produce the heuristic value for each state, in order to then use it in as it's done in a basic A* search.

By implementing this procedure, the Neural A* algorithm is able to obtain better search efficiency than traditional A* algorithms, which use handcrafted heuristics. The learned heuristics can be generalizable to new data and can improve the performance of the A* algorithm on unseen data, as long as the new data is similar to the ones on which the heuristic was trained. Otherwise, the learned heuristic may not perform as well and may need to be retrained or adjusted based on the new dataset.

In the Neural A* paper, the authors utilized a 2D grid map as the environmental representation. The grid map serves as a discretized representation of the environment, with each cell in the map representing a specific location and labeled either as traversable or non-traversable. This representation is a commonly used and intuitive method of representing the environment for path planning problems.

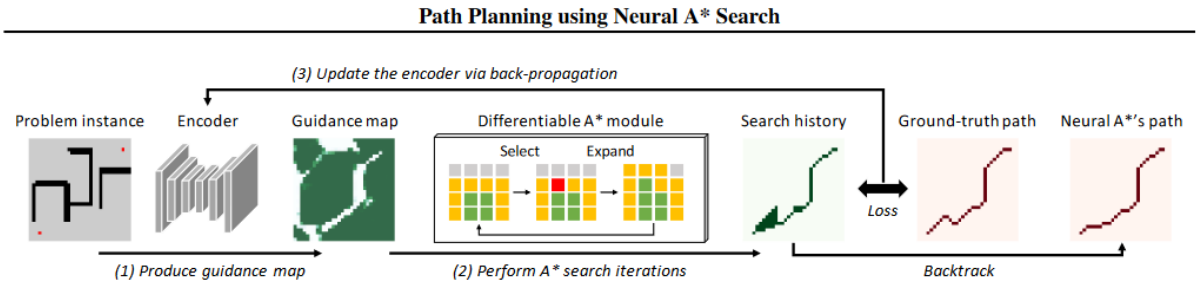

Figure 5: Neural A* Approach

The basic functioning of this algorithm can be seen in Figure 5, which mainly consists of an encoder that produces the "guidance maps" and a differentiable A* module. The input is a path planning problem instance which is fed to the encoder, that furtherly produces a 2D heatmap, also called guidance map, which represents the environment and the goal location. This guidance map is a grid of values where each of them represents the predicted cost-to-go from that grid location to the goal location; this can be interpreted as a measure of how

"desirable" each location is, in terms of proximity to the goal. The 2D heatmap is then used as a heuristic function in the A* algorithm to guide the search processes, as it is recalled that the heuristic function is indeed an estimate of the cost to traverse from a given node to the goal, which can be used to prioritize the nodes to be explored by the A* algorithm. The differentiable A* module performs a point-to-point shortest path search using the guidance map and outputs a search history and a resulting path; the search history is a sequence of guidance maps that are produced during the initial search process. The resulting path, instead, is just a sequence of states that represent the path from the start to the goal location. To train the encoder, a mean squared error (MSE) loss is computed between the search history and the ground-truth path, i.e. the path obtained with the standard A* searching procedure, which is back-propagated through the differentiable A* module to update the encoder parameters.

The authors use a generic Convolutional Neural Network (CNN) to learn the heuristic function in their neural network implementation. The CNN design comprises multiple convolutional layers, succeeded by max-pooling layers, and subsequently a few fully connected layers. The architecture takes inspiration from the U-Net design.

The researchers used three different datasets to train and evaluate their algorithm: the MP dataset, the tiled MP dataset and the CSM dataset. The first refers to a set of 46 real-world maps taken from Google Maps and OpenStreetMap, covering a range of urban, suburban, and rural areas; the tiled MP dataset is a version of the MP dataset in which the maps are divided into tiles of fixed size, to enable efficient processing of large maps; the CSM dataset, instead, is composed by simulated maps, designed to test the algorithm's performance under varying degrees of noise, as well as its ability to generalize to new environment.

The RMSProp optimizer has been adopted for the optimization procedure; it’s a variation of the standard stochastic gradient descent (SGD) algorithm that adjusts the learning rate for each weight in the network, based on the root mean square of the gradients, which helps to control the step size during training and converge faster. This algorithm also uses a moving average of the squared gradients to adjust the learning rate over time, making it more adaptable to different types of problems and reducing the likelihood of oscillations during training.

The metrics that have been employed for the evaluation of the Neural A* algorithm performance with respect to the standard A* algorithm are:

- Path optimality ratio (Opt), which represents the percentage of predicted paths that match the shortest path, for each given environmental map;

- Reduction ratio of node explorations (Exp), which measures, in percentage, the amount of search steps reduced by the model, compared to the standard A* path search;

- Harmonic mean (Hmean) between Opt and Exp, which highlights to what degree their trade-off has been improved by the model.

Table 1 shows a quantitative comparison between the results obtained through Neural A* and those derived by the best-first search (BF), weighted A* (WA*), SAIL, SAIL-SL and Blackbox-Differentiation A* (BB-A*) algorithms, which also have the guarantee of planning success by design, on three different publicly available motion planning datasets, specifically Motion Planning (MP) Dataset [27], Tiled MP Dataset [2] (which is the authors' extension of the MP dataset with more complex and diverse obstacle structures), and City/Street Map (CSM) Dataset [28].

In summary, the results show that Neural A* algorithm performed better than baseline methods for both path optimality ratio (Opt) and harmonic mean (Hmean), and improved the trade-off between path optimality and search efficiency. Although SAIL and SAIL-SL sometimes performed better, they came with lower optimality ratios, especially for larger and more complex maps. BB-A* had higher Hmean scores than SAIL/SAIL-SL but obtained far worse results compared to Neural A*. The comparison between Neural A* and BB-A* also highlights the effectiveness of the differentiable A* module, which provides richer information of internal steps necessary to effectively analyze causal relationships between individual node selections and results. This information is, in fact, black-boxed in BB-A*. It is also important to notice that classical heuristic planners, such as BF and WA*, often performed better than other learning-based algorithms.

These results finally show that the effectiveness of this approach over state-of-the-art learning based planners is undeniable, and also classify this method as the most interesting for our path planning purpose.

MP Dataset			
	Opt	Exp	Hmean
BF	65.8 (63.8, 68.0)	44.1 (42.8, 45.5)	44.8 (43.4, 46.3)
WA *	68.4 (66.5, 70.4)	35.8 (34.5, 37.1)	40.4 (39.0, 41.8)
SAIL	34.6 (32.1, 37.0)	**48.6 (47.2, 50.2)**	26.3 (24.6, 28.0)
SAIL-SL	37.2 (34.8, 39.5)	46.3 (44.8, 47.8)	28.3 (26.6, 29.9)
BB-A*	62.7 (60.6, 64.9)	42.0 (40.6, 43.4)	42.1 (40.5, 43.6)
Neural BF	75.5 (73.8, 77.1)	45.9 (44.6, 47.2)	**52.0 (50.7, 53.4)**
Neural A*	**87.7 (86.6, 88.9)**	40.1 (38.9, 41.3)	**52.0 (50.7, 53.3)**
Tiled MP Dataset			
	Opt	Exp	Hmean
BF	32.3 (30.0, 34.6)	58.9 (57.1, 60.8)	34.0 (32.1, 36.0)
WA*	35.3 (32.9, 37.7)	52.6 (50.8, 54.5)	34.3 (32.5, 36.1)
SAIL	5.3 (4.3, 6.1)	58.4 (56.6, 60.3)	7.5 (6.3, 8.6)
SAIL-SL	6.6 (5.6, 7.6)	54.6 (52.7, 56.5)	9.1 (7.9, 10.3)
BB-A*	31.2 (28.8, 33.5)	52.0 (50.2, 53.9)	31.1 (29.2, 33.0)
Neural BF	43.7 (41.4, 46.1)	**61.5 (59.7, 63.3)**	44.4 (42.5, 46.2)
Neural A*	**63.0 (60.7, 65.2)**	55.8 (54.1, 57.5)	**54.2 (52.6, 55.8)**
CSM Dataset			
	Opt	Exp	Hmean
BF	54.4 (51.8, 57.0)	39.9 (37.6, 42.2)	35.7 (33.9, 37.6)
WA*	55.7 (53.1, 58.3)	37.1 (34.8, 39.3)	34.4 (32.6, 36.3)
SAIL	20.6 (18.6, 22.6)	41.0 (38.8, 43.3)	18.3 (16.7, 19.9)
SAIL-SL	21.4 (19.4, 23.3)	39.3 (37.1, 41.6)	17.6 (16.1, 19.1)
BB-A*	54.4 (51.8, 57.1)	40.0 (37.7, 42.3)	35.6 (33.8, 37.4)
Neural BF	60.9 (58.5, 63.3)	**42.1 (39.8, 44.3)**	40.6 (38.7, 42.6)
Neural A*	**73.5 (71.5, 75.5)**	37.6 (35.5, 39.7)	**43.6 (41.7, 45.5)**

Table 1: Neural A* quantitative results[5]

[5] The best results are annotated in bold; Opt scores for Neural A* on all the three different datasets are, on average, 16.5% better than those obtained by other learning based path planning algorithms.

3. Methodologies

This Chapter is committed to a detailed description and explanation of the workflow that has been established, and the algorithms that have been employed, for the quests of traversability analysis and path planning within the context of the SINAV project. It should once again be noted that the aim of this study was not the implementation of a new machine learning model capable of extracting optimal paths, nor even traversability maps, starting from a single monocular picture of the martian terrain, but rather to explore and compare the advantages that could derive from the application of different machine learning algorithms for extracting the data that are necessary to perform the aforementioned tasks, along with presenting an improvement attempt of the chosen neural network architectures and a new pipeline for the tasks that have been addressed in this work.

Paragraph 3.1 contains the details of the overall workflow, that, as far as the writer's aware, represents the first approach to traversability mapping and path planning in Martian rovers that employs computer vision algorithms and learning-based path searching models, having the potential to encourage novel studies in this research area; Paragraph 3.2 deals with traversability analysis and mapping, analyzing the data required for such a mission, and explaining the motivations behind the different approaches that have been tested; Paragraph 3.3 presents the methodology used to integrate the chosen path planning algorithm inside the system, and how the tests regarding this part have been conducted.

3.1 Workflow Description

As described in section 2.1, the research regarding traversability analysis and mapping within the SINAV project aims at developing an end-to-end algorithm that is capable of taking, as input, the pictures of the environment taken by the drone at different lighting and height conditions, along with the segmentation masks produced by a previous processing step, and outputting the corresponding binary traversability map; the latter is a boolean matrix, of the same size of the input image, where to each pixel is assigned a value of 1 if the corresponding part of the terrain is safe for the rover to travel, and 0 if it is not. It's worth noting that the

segmentation masks are assumed to be already available for the analysis conducted in this Thesis, as the reference operational scenario doesn't require this traversability mapping and path planning system to be also capable of performing such analysis, therefore this objective does not fall within the scope of this research.

The produced traversability map has then to be fed to the path planning algorithm, that leveraging a state-of-art learning-based approach is able to estimate the best path between the emulated starting point and goal point in a more efficient way with respect to standard path searching techniques.

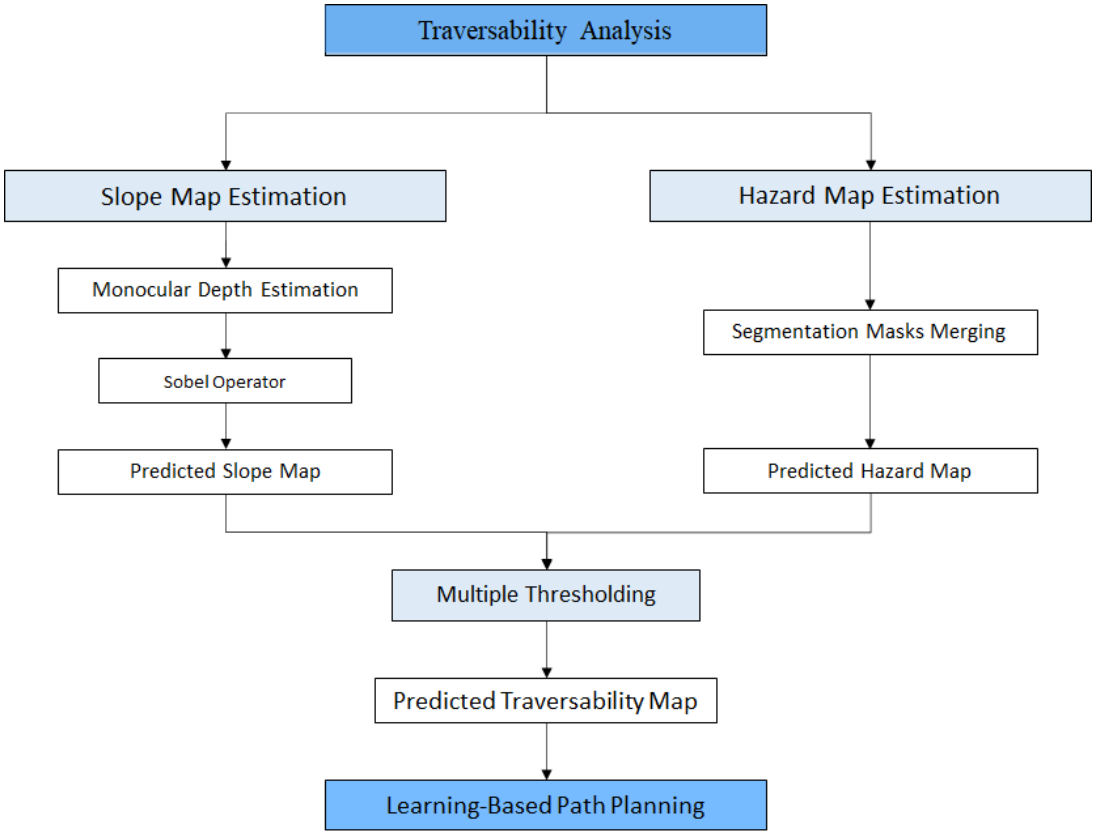

Figure 6: Traversability Analysis and Path Planning proposed Methodology

In Figure 6, a visual representation of the workflow that has been followed for the traversability map estimation, and its application to the planning subsystem, is proposed. The next paragraphs will now delve into the details of the processing pipeline.

3.2 Traversability Analysis Algorithms

In order to take into account both the terrain inclination and the presence of hazards, the proposed method first derives the slope map and the terrain hazard map from the two main sources of information that are available, and then merges them using a multi-channel threshold.

A slope map is a topographic representation of an area that shows the degree of steepness or slope of the terrain. In the rest of this work, it will be intended as a two dimensional map that assigns a numerical value to each point or pixel in the map, representing the angle (either in degrees or radians) or percentage of the slope at that location. The terrain-based hazard map, instead, in this context is defined as a binary matrix with the same size of the input image of the terrain, where each pixel has value 0 if the corresponding area does contain any type of terrain that is not traversable regardless of the slope value (i.e. the segmentation mask of that pixel assigns it to the “big rock” or to the “sand” category) and value 1 otherwise.

It’s worth mentioning that the slope of each object or area of a single scene is not actually *computed* by the method established in this study, as this term implies that it exists a mathematically accurate procedure for deriving the slope of all the objects and locations within an area starting from just one picture of it, and to the best of our knowledge this is not the case; it’s fair to say, instead, that the slope map is *estimated* through a sequence of approximation steps, where each of them inevitably implies a certain loss in terms of accuracy. An entire field of study is dedicated to edge detection, which is the challenging task of identifying the boundaries between objects or regions of an image, whose techniques typically involve the use of filters or operators that highlight the areas of the image with the greatest change in intensity or gradient, or either utilize supervised learning models to classify the pixels of unseen images with respect to the presence of edges.

Taking into consideration the fact that there were no available “ground truth” data regarding the presence of edges inside the terrain pictures, along with the fact that there is an enormous difference between knowing that in a certain section of the image an edge is present and being able to assess the degree of such edge (as it’s a whole other issue in itself), an approximating workflow has been followed in this Thesis work in order to estimate the actual

degree of steepness of the scene, starting from a single RGB picture of the terrain with 1000x1000 resolution, and will hereby be described in its completeness.

The first, and most critical, step that is needed to reach this objective is monocular depth estimation. This task has been already introduced in Chapter 2.1, along with two particular state-of-art methods (the official Keras.io Monocular Depth Estimation model, hereby referred to as Keras MDE, and the MiDaS model), whose applications rely on two completely different approaches: Keras MDE is a lightweight model, so a decision has been made to train the model from scratch and to finetune it on our dataset; the MiDaS model, instead, has been tested directly on our dataset using the two best-performing pretrained backbones (DPT-Swin2-L384 and DPT-Beit-L512) without any finetuning, complying with the zero-shot cross dataset transfer approach, as expressed by the researchers in their paper. The assumption made in the first stages of this work was that, even if the latter approach has yielded astonishing results in literature studies, our input data have been simulated by fellow students with limited expertise in hyper-realistic 3D rendering, and they present many differences (mainly in terms of simulated camera position, ground truth depth accuracy, and diversity of the depicted environments) with respect to real-world data that are present inside of most of the datasets used for monocular depth estimation, thus presenting a big limitation for the final accuracy that could be obtained by these models. At the same time, also when fine-tuning a classic monocular depth estimation model based on a U-Net architecture such as the Keras MDE, there are different issues that should be taken into account, such as the lower amount of training data and the computational resources available for the fine-tuning process. The results shown in Chapter 4 explore and finally verify such speculations.

For the application of the Keras MDE algorithm on our data collection, different adjustments have been done on the model, that regard both the way the input data were fed to the architecture and the structure of the architecture itself. The data pipeline from the original implementation was built taking the DIODE evaluation dataset and preparing a Panda's dataframe on it, with batches of size 32 that contain the images and their respective depth maps after resizing them to 256x256; the initial resizing, along with other hyperparameters such as the number and dimension of the encoder and decoder layers, led us to think that the performance of the original model could not be satisfying after training it on our data, that have a almost four times bigger resolution and a fairly less accurate ground truth depth

precision. The results obtained after the first training and validation experiment, conducted with a random split of 2:1 training to validation ratio of the SINAV drone dataset (which has been kept consistent for all the experiments that have been executed), will be presented in the next Chapter and furtherly confirm the aforementioned hypothesis.

The most important original hyperparameters are shown in Table 2, while Table 3 contains a basic schema of the original neural network architecture.

Parameter	Value
Resized Input Dimensions	Height=256, Width=256
Learning Rate	0.0002
Epochs	30
Batch Size	32

Table 2: Keras MDE Original Hyperparameters

Layer Type	Input Size	Connected To
Input	(32, 256, 256, 3)	DownscaleBlock_0
DownscaleBlock_0	(32, 256, 256, 3)	DownscaleBlock_1
DownscaleBlock_1	(32, 128, 128, 16)	DownscaleBlock_2
DownscaleBlock_2	(32, 64, 64, 32)	DownscaleBlock_3
DownscaleBlock_3	(32, 32, 32, 64)	BottleNeckBlock
BottleNeckBlock	(32, 16, 16, 128)	UpscaleBlock_0
UpscaleBlock_0	(32, 16, 16, 256)	UpscaleBlock_1
UpscaleBlock_1	(32, 32, 32, 128)	UpscaleBlock_2
UpscaleBlock_2	(32, 64, 64, 64)	UpscaleBlock_3
UpscaleBlock_3	(32, 128, 128, 32)	Conv_Layer
Conv_Layer	(32, 256, 256, 16)	Output
Output	(32, 256, 256, 1)	–

Table 3: Keras MDE Original Network Structure

For the reasons previously mentioned, several trials have been carried out on this first monocular depth estimation algorithm, which at first focused on varying the initial resizing to 512x512 using bicubic interpolation, to avoid depth accuracy loss, and then on changing the architecture by adding two blocks on different positions of the encoder and decoder layers, as the loss behavior when training the model using the original architecture showed fair signs of possible improvement; this has been done in order to complicate the network structure, hoping to obtain a more accurate hierarchical feature extraction and an increased representational power.

The MiDaS algorithm, instead, has been directly applied to our data in order to test its generalization abilities and overall accuracy on new (i.e. never before used in the training stage by the algorithm) datasets.

It's important to notice that the steps explained hereafter have been conducted, considering them as starting points, on the outputs produced by the inference of the MiDaS model (with both the backbones that have been evaluated) and the Keras MDE algorithm (using the best-performing model after the fine-tuning procedure); the output of the inference of these models have indeed been stored with the intention of later evaluating their performance on a rather large amount of examples.

In testing these two approaches and conducting the relative evaluation experiments, six different metrics, that were introduced for the validation of monocular depth estimation models in the study [29], have been taken into consideration:

- Delta threshold accuracy (a1, a2, a3), that measures the percentage of predicted depth values that are within a certain threshold of the ground truth depth values; three different thresholds are typically used in this context: $\delta_1 = 1.25$, $\delta_2 = 1.25^2$ and $\delta_3 = 1.25^3$;

$$Threshold: \%\, of\, y_i\, s.t.\ \max\left(\frac{y_i}{y_i^*}, \frac{y_i^*}{y_i}\right) = \delta < thr$$

Equation 8: Delta Threshold Accuracy

- Absolute relative difference (Abs_Rel), which computes the absolute difference between the predicted and ground truth depth values, normalized by the ground truth depth value, giving an overall measure of how well the algorithm is able to estimate the depth of the scene, without considering the sign of the difference;

$$Abs\ Rel\ difference = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y_i^*|}{y_i^*}$$

Equation 9: Absolute Relative Difference

- Squared relative difference (Sq_Rel), whereas instead of taking the absolute value of the difference between predicted depth and the ground truth values as in the absolute relative difference, it squares it, basically giving more weight to larger errors in order to penalize models that make occasional large mistakes;

$$Abs\ Squared\ difference = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y_i^*|^2}{y_i^*}$$

Equation 10: Squared Relative Difference

- Root mean squared error (RMSE), which takes the root mean squared difference between the depth values in order to measure the average error that can be expected from the model;

$$RMSE = \sqrt{\left(\sum_{i=1}^{n}\frac{(y_i - y_i^*)^2}{n}\right)}$$

Equation 11: Root Mean Squared Error

- Root mean squared logarithmic error (RMSLE), that is similar to the previous metric, but takes the logarithm of the depth values before the computation; this helps to

account for the fact that depth values are typically distributed logarithmically, rather than linearly;

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\log\left(1+y_i\right)-\log\left(1+y_i^*\right)\right)^2}$$

Equation 12: Root Mean Squared Logarithmic Error

- Structural similarity index (SSIM), which is a measure of the similarity between the predicted and ground truth depth maps; it provides a measure of how well the algorithm is able to preserve the structural information in the scene.

$$SSIM = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{\left(\mu_x^2 + \mu_y^2 + c_1\right)(\sigma_x + \sigma_y + c_2)}$$

Equation 2: Structural Similarity Index

Besides the application and comparison of these monocular depth estimation algorithms, the research regarding traversability estimation and mapping required to use the predicted depth maps to extract the information about the slope of the terrain. This has been done following the standard terrain analysis theory by [30], which states that, taking as input an elevation matrix, the slope information can be obtained by applying a first-order discrete differentiation operator, such as the Sobel-Feldman operator (hereby referred to simply as Sobel operator or Sobel filter) [31], and then using the arctangent of the gradient magnitude of the two axial components.

The OpenCV-Python library offers a built-in Sobel filter implementation that, for each point in the original image, returns an approximation of the gradient vector components of the image intensity function. It shall be noted that such image intensity function can be derived, on a digital image, only by assuming that there is an underlying differentiable intensity function which is continuous, and the actual digital image represents the sampled version,

thus its derivatives can be virtually approximated in every point of the figure at a sufficient degree of accuracy.

It's important to take into consideration the fact that spatial filters (such as Sobel, Scharr, Laplacian and Canny filters) are often used to approximate the image gradients starting from the original picture in order to perform edge detection, but in this case they are actually unable to give insights with respect to the actual steepness of the terrain present inside the image; using them in this context basically gives indication on the extent and direction of the change in intensity across neighboring pixels. On the other hand, assuming that each pixel contains information about the (real or predicted) distance between the objects present in the scene and the camera that captured the scene, the gradients of the intensity of each pixel represent the change in depthness, thus giving indication of the terrain slope.

The Sobel filter employs intensity values in a 3x3 region around each point to perform such gradient approximation, using the formulation shown in Equation 13 and 14, and we further use the resulting components to derive the slope approximation for each pixel of the depth map as in Equation 15. A visual example of the output of these processing steps can be seen in Figure 7.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

Equation 13: Sobel X-axis component of the gradient approximation[6]

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

Equation 14: Sobel Y-axis component of the gradient approximation

[6] Here, "I" represents the input digital image, which in our case corresponds to the terrain depth map, either estimated from the monocular depth estimation model or the ground truth one.

$$Slope = \arctan\left(\frac{1}{2}|G_x| + \frac{1}{2}|G_y|\right) * \frac{180}{\pi}$$

Equation 15: Slope derivation from depth map gradient approximation

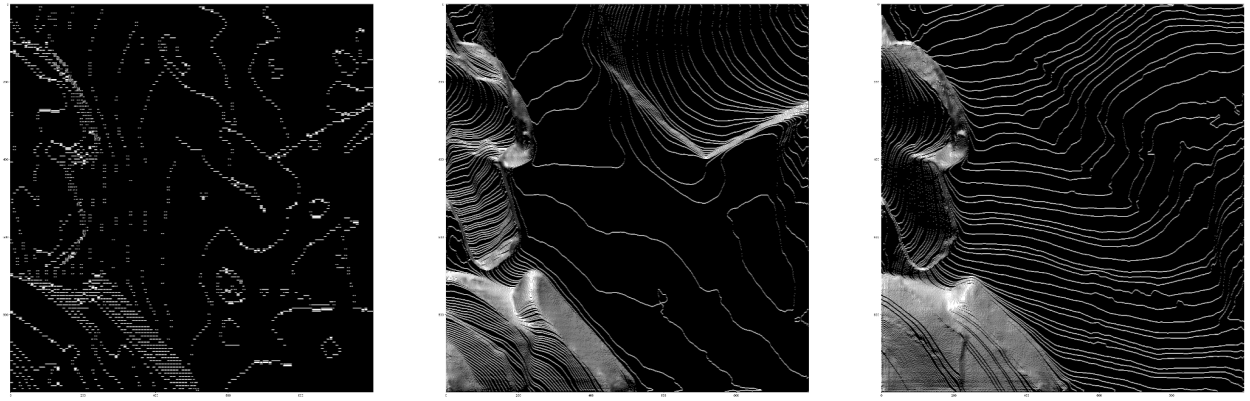

Figure 7: Example of the slope approximation algorithm output[7]

After having extensively tested both the MiDaS and the Keras models for monocular depth estimation, and once applied the slope derivation algorithm described above on the estimated depth maps, a comparison between the approximated slope maps and those derived using the ground truth depth maps has been conducted in order to validate the previous results; this is not only crucial for the sake of correctness and rigor, but it is also of significant importance in the context of this research, as it is possible for a model to perform worse in predicting the depth of a scene but perform better in predicting the slope, even if the slope is computed from the estimated depth map. This could be the case if the model is better at preserving relative depth differences between different regions of the image, rather than absolute depth values. For example, if a scene presents a flat surface and a sharp incline, a model that is better at preserving relative depth differences will be able to predict the sharp incline accurately, even if it underestimates the absolute depth; this is because the slope is a function of the gradient of the depth map, which is more sensitive to relative depth differences than absolute depth values.

[7] Respectively, the output is produced using the ground truth depth map, the MiDaS with dpt_swin2_L384 predicted depth map, and the MiDaS with dpt_beit_L512 predicted depth map

The comparison between the approximated slope maps obtained using the ground truth depth maps and the estimated depth maps, for all the variants of the MiDaS and Keras models, mainly looks at a portion of the metrics already employed for the evaluation of the monocular depth estimation task; in particular, the metrics taken into consideration are once again: delta threshold accuracy (a1, a2, a3), root mean squared error (RMSE), and root mean squared logarithmic error (RMSLE).

The absolute relative difference and squared relative difference metrics could not be applied in this case, as the terms that are in the denominator of both Equation 9 and 10 represent the values of the ground truth slope map, which very often turn out to be zero, and so the resulting metrics tend to huge values even when trying to mask those zeros by adding a small constant, therefore are considered as not significant in our considerations.

Once derived, the slope maps have been processed in order to set the pixels with a steepness that surpasses the threshold of 20° as non-traversable. This is once again an approximation, because the slopes that a Martian rover can or cannot traverse in a real-case scenario are defined only after a comprehensive analysis of the mission objectives, the rover's wheel design and suspension system, its weight, its center of gravity and the specific characteristics of the soil that is analyzed. These are all out of the scope of this research, as here the objective was giving a first hint for selecting the possible areas that respect the minimum requirements for traversability, and will be further inspected using other techniques throughout the remaining steps of the SINAV project.

The final step of the traversability analysis algorithm was the usage of terrain segmentation masks in order to obtain the final binary traversability map. This step basically consists in merging the two pieces of information that are available at this point (i.e. the thresholded slope map and the map composed of terrain segmentation masks), that are both encoded as binary matrices of the same size. The segmentation masks contain only four possible values, as it will be shown hereafter in Chapter 4, that represent the only four macro-categories of terrain types that have been segmented in the previous steps: big rock, bedrock, sand, soil. For the safety of the rover, given that the data that are available for our analysis are far from being comprehensive of the wide range of possible environmental conditions, the areas labeled as "big rock" and "sand" have been set as not traversable by default. Then, a

straightforward “logical-and” operation has been applied to these two resulting binary matrices, thus obtaining the final traversability maps. This has been applied to the slope maps obtained using the ground truth depth maps, the ones obtained through MiDaS backbones DPT-swin2-L384 and DPT-beit-L512, and the ones obtained using the best performing variant of the Keras MDE model.

The final validation step that has been performed regards the differences between these final traversability maps, setting the ones obtained starting by the ground truth depth maps as benchmark.

The quantitative analysis of these methods can be found in the next Chapter.

3.3 Neural A* Algorithm

In order to fulfill the need for an end-to-end procedure that is capable of taking the aforementioned input data, performing the traversability analysis, obtaining a 2D binary grid representation of the environment based on the approximated rover navigation capabilities, and finally predicting an optimal path from the starting point and the end goal, the Neural A* has been applied to the output of the traversability mapping algorithms; this was also done for the purpose of employing once again Machine Learning algorithms inside the overall pipeline of the study, comparing the results of this method to those of a classic path searching algorithm to evaluate the potential of learning-based approaches in another passage of the workflow.

The first step that has been done in this direction is the evaluation of the pre-trained Neural A* algorithm on the SINAV drone dataset; the best performing model from literature has been trained on a collection of 3200 grid-world environments with distinctive obstacle shapes, as reported in Chapter 2.2.1; these input images (also referred to as “mazes”) have been resized to 64x64 pixels in order to complete the whole experiment in a reasonable time. The dataset they used, Tiled MP Dataset, has been obtained using data augmentation techniques (for example, random tiling of four original maps) on the Motion Planning Dataset (MP Dataset); three examples of the latter can be found in Figure 8. At first sight, it’s easy to determine that such shapes have been useful in order to teach the basic functioning of A*

path searching to the model in simple scenarios, but this does not necessarily imply that the behavior of the model when evaluated on a very different dataset, such as the SINAV drone dataset, could be able to reach the performance of the standard A* algorithm, let alone surpassing it.

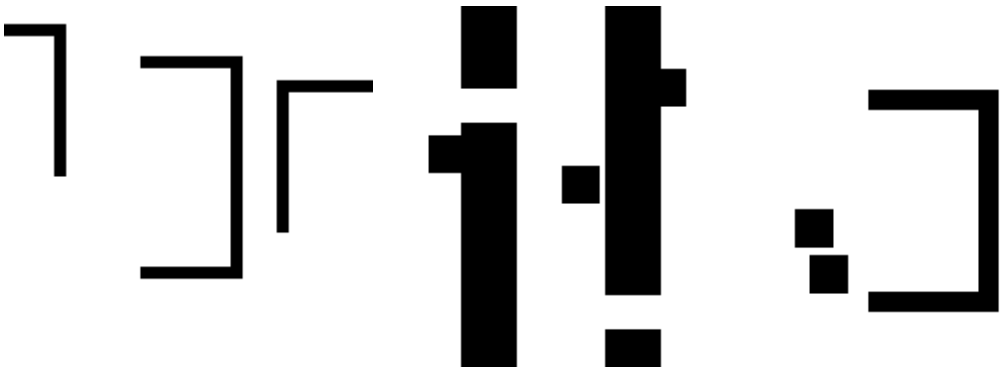

Figure 8: Three examples from the Motion Planning Dataset

The results of the first batch of experiments conducted with the original pre-trained model, employing the RMSprop optimizer with a learning rate of 0.001 for 40 epochs on the Tiled MP Dataset and tested on the traversability maps obtained from the SINAV drone dataset, were fairly disappointing and will be reported in the next Chapter. Once again, this was expected, as even though for a matter of execution times, and to avoid “out of memory” exceptions, the input traversability maps have been resized from 1000x1000 to 256x256 using bicubic interpolation, the difference between the training mazes’ size (64x64) and our resized traversability maps, along with their varying complexities, had already raised concerns towards this zero-shot approach.

For this reason, another attempt has been taken in order to train and fine-tune the algorithm on a bigger, more complex and more overall fitting to our study dataset, which is the HIRISE (High Resolution Imaging Science Experiment) dataset. As anticipated in Chapter 2, this is a collection of high-resolution images of the Martian surface, taken by Mars Reconnaissance Orbiter's HIRISE camera. Each image is captured at a resolution of up to 25 centimeters per pixel, which is much higher than any other camera currently in orbit around Mars, covering

an area of up to 6 square kilometers. HIRISE is considered to be the largest annotated dataset of Martian areas available in literature, with version 3.2 containing 64,947 landmark images.

Unfortunately, no literature studies ever focused on the generation of traversability maps starting from this huge collection of Martian images, which we believe could be a great further application of the methodology presented in this paper in order to benefit the scientific community. Consequently, the binary Otsu [33] thresholding method has been applied to a subset of this dataset (as this was the same technique used to preprocess the images that were fed to the Neural A* algorithm in the training stage), which basically separates the input image into two clusters based on pixel intensities; given that the pictures are always taken with the orbiter's camera facing down towards the terrain, the pixel values in the image represent the brightness or intensity of light reflected from the terrain in the direction of the camera. This implies that when applying the binary Otsu threshold, whose objective is finding the threshold that maximizes the variance between the two regions such that the separation is optimal, the method returns a binary map of the original image that segments the areas with highest variance in terms of light reflection, which is an indicator of many terrain properties, such as texture, composition and geometry. Four examples of this processing can be seen in Figure 9[8].

It's crucial to consider that this is not an accurate way to perform traversability mapping on such a dataset, and it is in general a very faulty way to do the same on any dataset that is not composed of binary images, as the information that can be acquired employing this method clearly does not represent, nor even approximate, the actual capability of a rover system to navigate the terrains present inside the images. However, this has been done in this step of the research only in order to provide training mazes to the Neural A* algorithm that are far more chaotic with respect to the datasets utilized in the original paper implementation, and that also present an image size which is the same as the target one for the path planning experiment.

[8] These are of size 64x64, but the actual images that have been used for the training are of size 256x256.

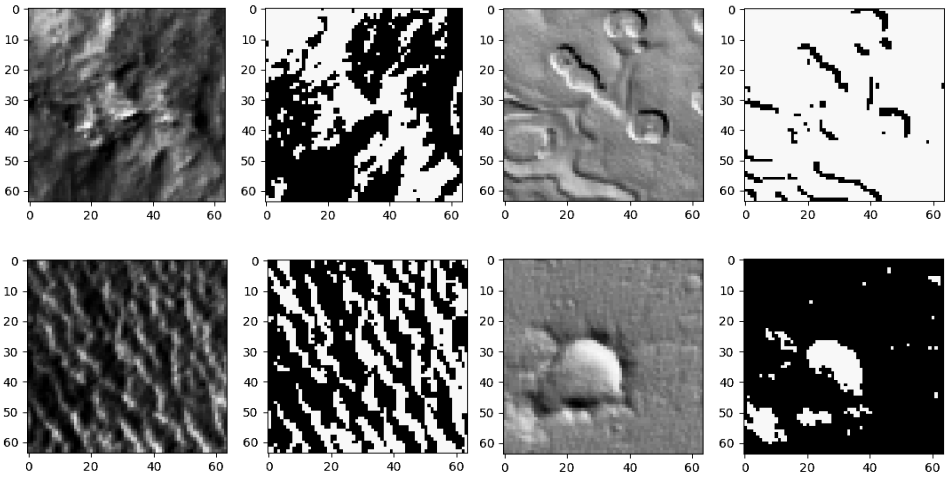

Figure 9: Four examples of the Otsu-segmented HIRISE images

After training the path planning algorithm on 2,400 HIRISE images that have been preprocessed as mentioned before, with the RMSprop optimizer and a learning rate of 0.001 for 40 epochs, the results when evaluating the model were encouraging, but still left room for improvement. For this reason, another trial was conducted employing the Adam optimizer, again with a learning rate of 0.001 for 40 epochs, gaining a small improvement. Both experiments, and the further tests that have been executed, relied on a basic CNN encoder architecture composed of 2D convolutional layers with kernel size=3, stride=1, padding=1, followed by batch normalization and ReLU activation function.

Finally, the Neural A* algorithm that was trained on the HIRISE dataset was tested on the traversability maps that have been derived from the SINAV drone dataset, i.e. on the binary navigation maps obtained from the ground truth depth maps, from the two MiDaS backbones and from the best-performing Keras MDE model. The quantitative results will be presented in the following Chapter.

4. Results

In this Chapter, a complete overview of the tests that have been used to compare the different algorithms employed both for traversability mapping and path planning, along with a visual and quantitative comparison of their results, is presented. A brief description of the SINAV drone dataset is now brought to the attention of the readers, which has been employed for all of the tests conducted in this work.

4.1 SINAV Drone Dataset Description

For any supervised machine learning algorithm, regardless of the objective task, the need for large, diverse and accurate datasets is essential. These shall provide many examples for the model to learn from, in order to help reducing the effects of biases and outliers, and prevent overfitting. The lack of training data represented a challenging issue that had to be dealt with in the first stages of the SINAV project, as no labeled dataset was available for our means at all. For the tests and evaluations that will be presented in this Chapter, a wide number of pictures of martian terrains, taken by the drone that is expected to support rover navigation within a range of 100-300 m as shown in the reference operational scenario, had to be collected and labeled (i.e. accurate depth values shall be available for each terrain image), in order to first estimate the traversability maps, and then finally perform path planning.

This has been addressed by the fellow researchers that took part in the project, and were responsible for data generation and formatting, using the Unity® Engine [34]. This is a leading platform for 3D scene simulation, that provides a rendering pipeline which allows to create realistic and high-quality visuals, with features such as real-time global illumination, dynamic shadows, and post-processing effects. Such an instrument has been fundamental for the aim of the project, as fellow developers were able to use it to produce a dataset of 1008 terrain images, along with their corresponding ground truth depth maps and segmentation masks.

The dataset, that from now on will be referred to as “SINAV drone dataset” for a matter of clarity and simplicity, contains the following:

- RGB_Images, a folder containing 1008 RGB pictures of simulated terrains, saved as .jpg files;
- Depth_Maps, which contains the respective 1008 ground truth values of depthness[9] for each terrain image, ranging from 0 to 255 and saved as .jpg files in grayscale; this information can be obtained directly from the Unity® software.
- Segmentation_Masks, which contains the respective 1008 ground truth segmentation masks for each terrain image. These are also saved as .jpg files and can be interpreted by following the schema reported in Table 4.

Terrain Type	Color	RGB Value	Description
Soil	Red	[254, 0, 0]	Soil that is similar to gravel, used as a base on which to add other types of classes.
Big Rocks	Yellow	[241,226,171]	Rocks that are too big, dangerous, or in general an obstacle for the rover to cross.
Bedrock	Pink	[255, 177, 173]	Type of terrain consisting of rock that is slightly exposed from the ground, a rover can generally pass over it or cross it.
Sand	Brown	[152, 62, 0]	Sandy deposits that are hazardous for the rover to cross due to wheel slippage.

Table 4: Segmentation Masks Description

Additionally, each file has some metadata associated, regarding the ID of the scene, the altitude of the drone at the time of the shooting, the lighting condition and others, that are crucial for the analyses conducted in this research, and can be accessed by using the filename which is formatted as follows:

[9] In the drone dataset, the direct depth representation has been adopted, i.e. the values of depth maps directly represent the distance from the camera.

Example: N_005_000071_999_1675353701_0000_1_20.jpg

(1) (2) (3) (4) (5) (6) (7) (8)

(1) Image type:

- N —> RGB image
- D —> Depth Map
- U —> Segmentation Mask

(2) Dataset version (in this study, the dataset number 6 is used);

(3) File number in the sequence;

(4*[10]) Indication of exposure, brightness, etc.;

(5*) Time-stamp (date-time);

(6) Scene ID (sequential number);

(7*) Lighting condition: 0 –> Zenith - 1 –> Low West - 2 –> Low East - 3 –> High East;

(8) Height in Unity units (to convert to meters, it needs to be multiplied by a 0.25 factor).

The pictures of martian terrains present inside this dataset were captured from a simulated camera positioned at different heights, ranging from 5 to 15m above the ground, and pointing directly downward, resulting in a perpendicular perspective. This is a notable aspect of the data that are available for this study, as the main difference with respect to the datasets regularly used for monocular depth estimation tasks lays in the fact that the latter are usually populated by pictures taken from a frontal perspective [20], given that one of the most important use case scenario for monocular depth estimation models is autonomous driving vehicles. This difference induced the author of this research to speculate that the pretrained models from literature would not be able to infer the depth predictions on our dataset in an accurate way. This assumption will be further investigated in this Chapter.

[10] The elements marked with an asterisk were needed for other analyses that fall out of the scope of this research, and thus have not been employed for the means of our work.

4.2 Monocular Depth Estimation Results

The first task that has been carried out for traversability analysis was monocular depth estimation, in order to extract the depth information from the original RGB images. For this objective, the first model that was tested is referred to as Keras MDE, which has been initially trained and evaluated on the SINAV drone dataset using the original parameters that are reported in Table 2. An example of the output of this algorithm is shown in Figure 10, where the first image is the original RGB input, the second one is the ground truth depth map (objects in blue are nearer to the camera, in dark red are farther[11]), the third one is the predicted depth map[12], and the resulting training and validation losses are shown in Figure 11.

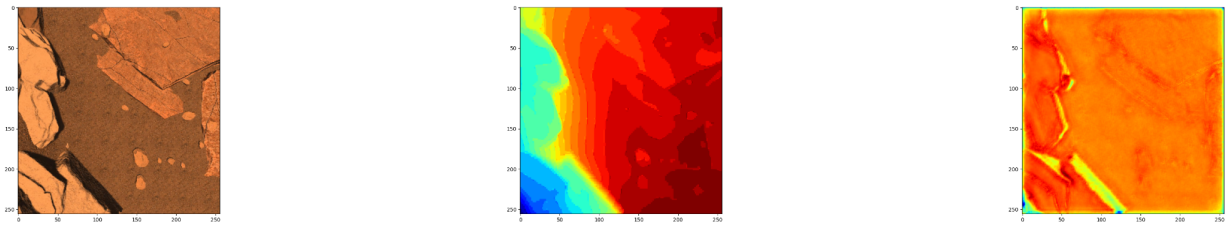

Figure 10: Example of the output from Keras MDE trained with the original hyperparameters

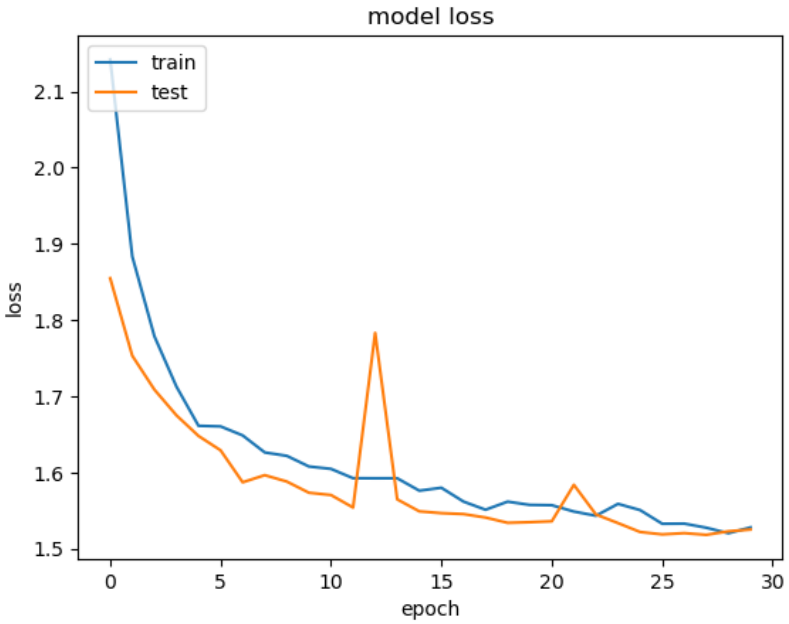

Figure 11: Learning Curve of Keras MDE trained with the original hyperparameters

[11] The depth values have been visualized using a "jet" colormap, but they are actually stored in grayscale.
[12] It's important to notice that this order has been kept consistent when showing the resulting depth maps, for both Keras MDE and MiDaS models: the first image always shows the RGB picture of the terrain, the second one shows the ground truth depth map and the third one is the estimated depth map.

Seeing the behavior of the training loss and the resulting output, it was clear that some fine-tuning was required for improving the performance of the algorithm. A step in this direction implied, in the first place, changing the initial resizing that is applied to the input RGB images and ground truth depth maps, as a resolution of 256x256 was deemed as too lossy in terms of detail and accuracy; given that the U-Net model relies on encoder-decoder architecture to capture both low-level and high-level features in the input image, if the input image has been excessively downsized, the encoder may not be able to capture all of the necessary features. Unfortunately, changing this parameter led to “Out of Memory” exceptions when repeating the experiment, given that a 2x increase of the input RGB images and depth maps size causes a quadratic increase in memory usage and computational time needed for the training, so it was decided to respectively adjust the batch size to 8, from the original value of 32, to avoid excessive memory consumption; simultaneously, the learning rate has been increased from 0.0002 to 0.0008, in order to maintain the same overall update step-size in the weight space, which is important for ensuring that the training process remains stable and efficient. An example of the output, and of the overall training loss trend, which already represent an improvement from the first experiment, are shown in Figure 12 and 13.

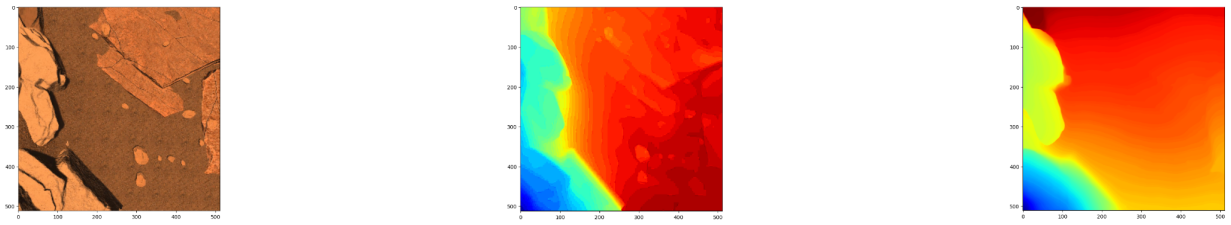

Figure 12: Example of the output from Keras MDE trained with 512x512 input, batch size=8, learning rate=0.0008

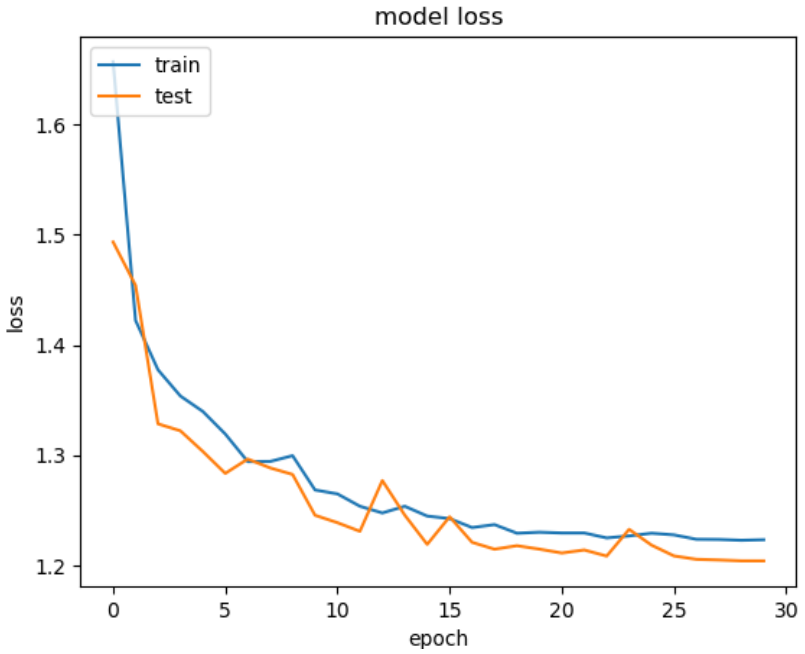

Figure 13: Learning Curve of Keras MDE trained with 512x512 input, batch size=8, learning rate=0.0008

After this experiment gained significant improvements in terms of prediction accuracy, the last trial that has been conducted consisted in the insertion of a DownscaleBlock and an UpscaleBlock in position 3 of the corresponding encoder/decoder structure, passing from 4 downsampling and upsampling stages to 5; this was done hoping that the increased model capacity would then be reflected in the results, and it was actually proven by the overall trend of the training. Figure 14 contains an example of the output predictions, and Figure 15 shows the training and evaluation losses.

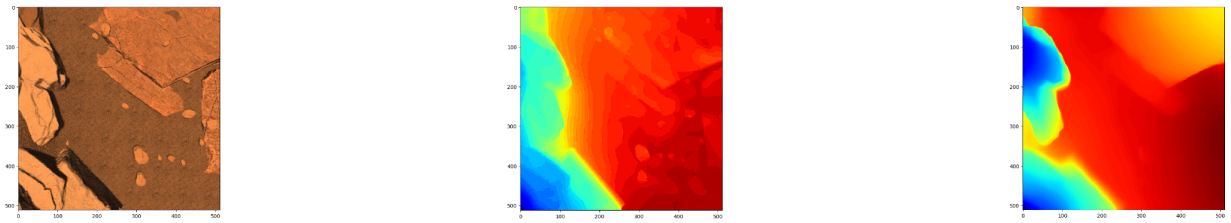

Figure 14: Example of the output from Keras MDE trained with 512x512 input, batch size=8, learning rate=0.0008, 5 encoding/decoding stages

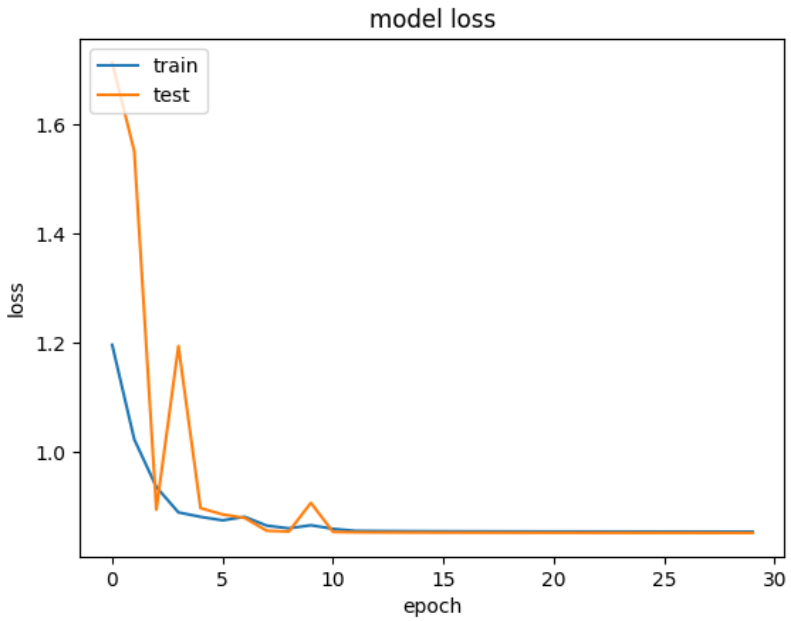

Figure 15: Learning Curve of Keras MDE trained with 512x512 input, batch size=8, learning rate=0.0008, 5 encoding/decoding stages

The results obtained with this model were promising, but for the task of monocular depth estimation another interesting approach that has been investigated is the one proposed in the MiDaS paper, where researchers state that their method outperformed other state-of-the-art algorithms that required fine-tuning on each dataset separately, showing that their model is able to generalize the features learnt during the comprehensive training stage to unseen scenes without the need to modify the algorithm.

Figures 16 and 17 show some examples of the inference of this model, with the two different backbones that produced the best results on the other benchmark datasets that the researchers used for their tests, on the SINAV drone dataset. These still follow the same convention as Figures 10, 12 and 14, whereas the first column shows the original pictures of the terrain (this time using a “winter” colormap), the second column contains the ground truth depth maps and the third one shows the predicted depth maps.

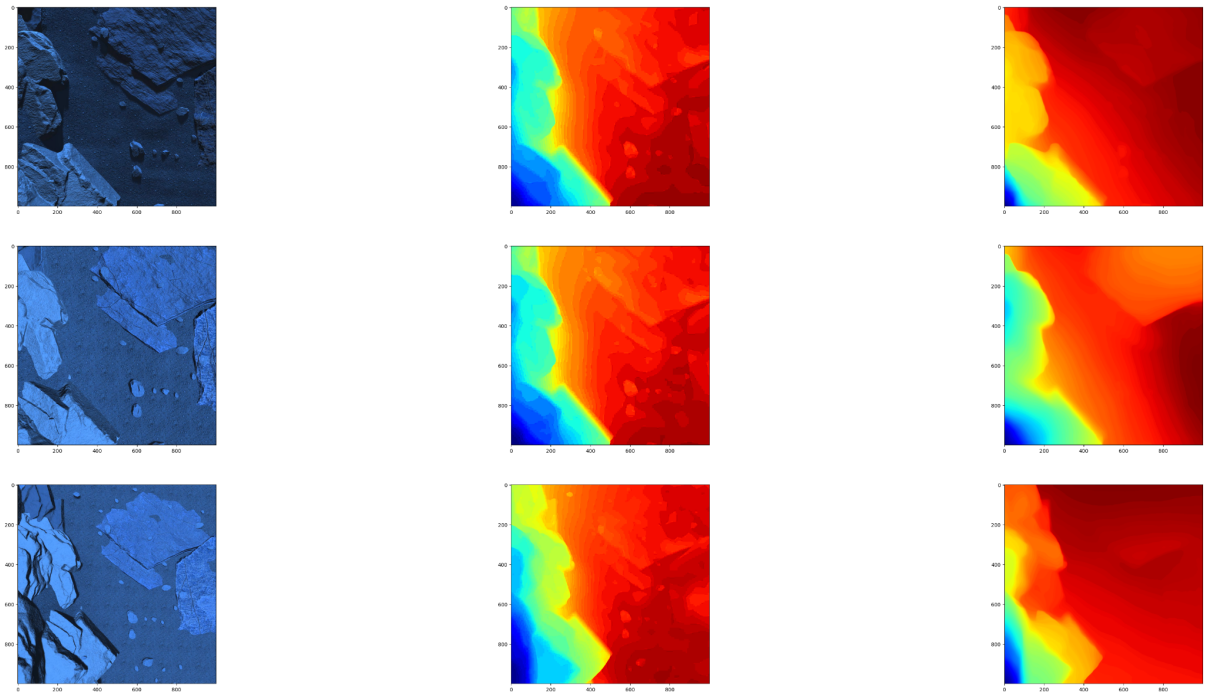

Figure 16: Example of the output from MiDaS with DPT-beit-L512 backbone

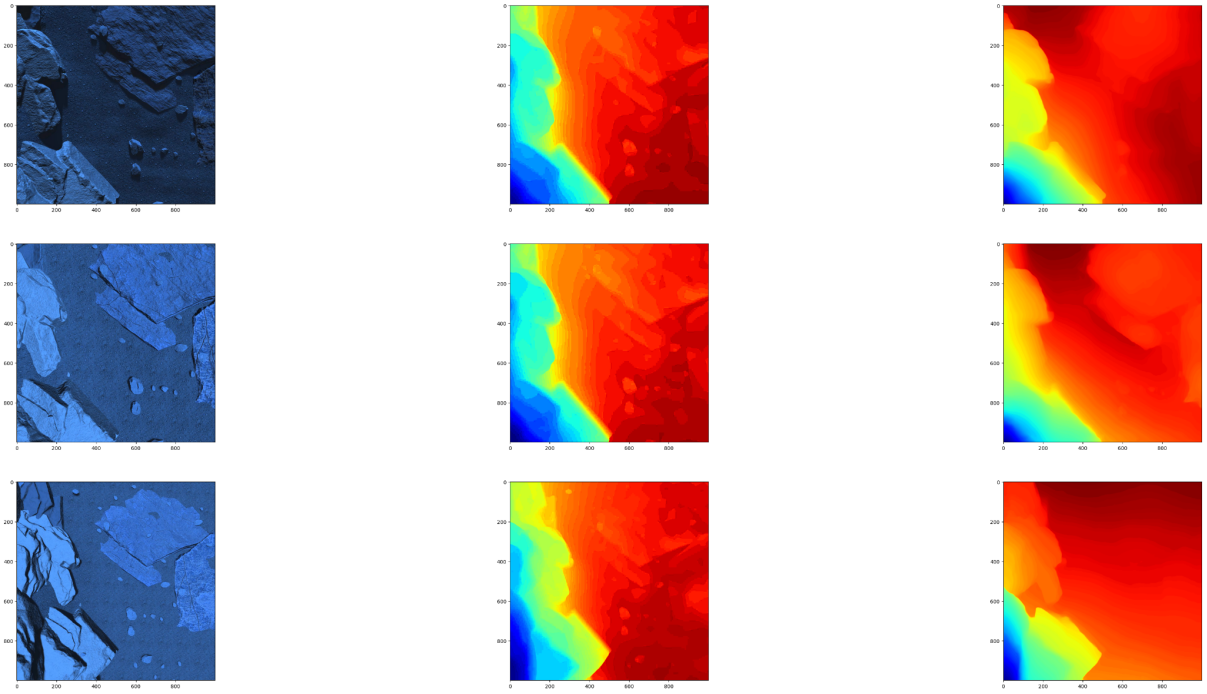

Figure 17: Example of the output from MiDaS with DPT-swin2-L384 backbone

Even if at first sight these results could seem ordinary and overall not a great advancement with respect to those obtained with the Keras MDE algorithm, it's crucial to keep into consideration that these are obtained from just a single inference of a model that was trained on a multitude of images and depth maps which are very different with respect to our digital dataset, mainly in terms of depth accuracy, camera angle, environment complexity and lighting conditions, given also the fact that most of the datasets that the algorithm has been pre-trained on are from real-world scenarios.

Many of the datasets that have been used for the original training were also focused on indoor scenes and street-view environments, as it's shown in Figure 18 and 19 that contain some examples of the NYU Depth V2 [35] and Kitti [36] datasets, that are two of the most important benchmark datasets for monocular depth estimation.

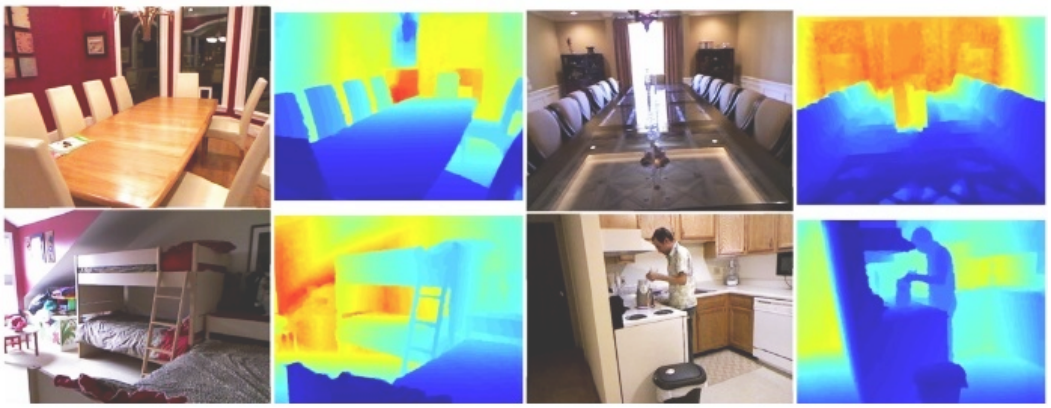

Figure 18: Examples from the NYU Depth V2 Dataset

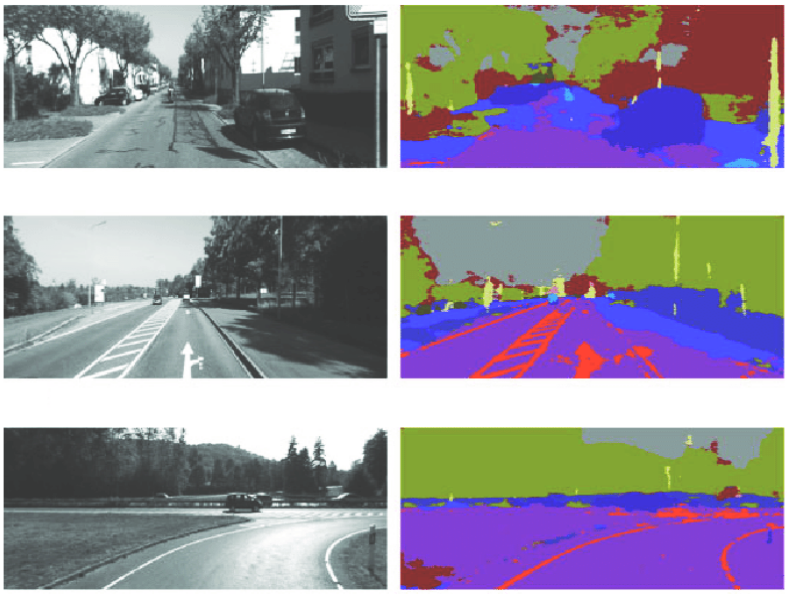

Figure 19: Examples from the Kitti Dataset

In order to finally verify which of the proposed methods yielded the best results in terms of similarity to the ground truth depth maps, the metrics introduced in the previous Chapter (Equations 2, 8-12) have been calculated on the inference of each model on the entire SINAV drone dataset.

The results, shown in Table 5, compare the estimated depth maps from the two MiDaS backbones and the final Keras MDE model (i.e. the one with additional downsampling and upsampling layers) with the ground truth.

Model	**δ_1↑**	**δ_2↑**	**δ_3↑**	**Abs_Rel ↓**	**Sq_Rel ↓**	**RMSE ↓**	**RMSLE ↓**	**SSIM ↑**
Keras MDE	0.212	0.498	0.713	1.767	1.169	10.248	**0.105**	**0.804**
MiDaS (Beit)	0.423	0.627	0.732	**1.585**	1.151	**10.240**	0.305	0.802
MiDaS (Swin2)	**0.434**	**0.654**	**0.748**	1.644	**1.144**	10.241	0.296	0.793

Table 5: Monocular Depth Estimation Metrics computed between the output of the final Keras MDE model, MiDaS DPT-Beit-L512, MiDaS DPT-Swin2-L384 and the ground truth depth maps

These results show that the performance of these three models on the SINAV drone dataset is overall similar, with Keras MDE performing slightly worse with respect to the other two, and the MiDaS model with DPT-Swin2-L384 backbone is the one that obtained the best scores (that are in bold) on a higher number of metrics out of them; it should be noted that the only metrics which were actually designed specifically for evaluating monocular depth estimation models are the delta thresholds metrics, and it is thus expected that the model that obtained the best results on these metrics is the one that will further outperform the other two in the next tests, that will be focused on slope estimation and the derivation of traversability maps starting from the output of the analysis step that has now been evaluated.

4.3 Slope Estimation Results

As anticipated in the previous Chapter, the following steps are indeed implemented taking the estimated depth maps produced by these three models as input, and the comparison between the three will be carried along on each of the later steps, as it should not be taken as granted that the outputs produced by a model that yielded better results for depth estimation will still derive slope maps that are more accurate with respect to those obtained starting from the output of other models.

After applying the slope derivation algorithm described in Chapter 3, the metrics listed in Table 6 have been evaluated to compare the approximated slope maps produced on the output of monocular depth estimation models with the ones derived from the ground truth depth maps.

Depth Maps	$\boldsymbol{\delta_1}\uparrow$	$\boldsymbol{\delta_2}\uparrow$	$\boldsymbol{\delta_3}\uparrow$	**RMSE** ↓	**RMSLE** ↓
Keras MDE	0.337	0.533	0.712	25.662	0.578
MiDaS (Beit)	**0.703**	**0.792**	**0.848**	**15.580**	**0.240**
MiDaS (Swin2)	0.647	0.738	0.787	16.574	0.264

Table 6: Slope Maps Evaluation Metrics computed on the output of Keras MDE, MiDaS DPT-Beit-L512, MiDaS DPT-Swin2-L384 and compared with those obtained from ground truth depth maps

In contrast with the previous results, that established MiDaS DPT-Swin2-L384 as the model that gained the best overall performance for monocular depth estimation and thus gave hints that the depth maps it produced were more alike to ground truth ones, these metrics show how the other backbone better preserved the relative depth differences between different regions of the image, which in our case study is far more important with respect to the accuracy in predicting absolute depth values. These results are in line with those present inside of the extensive experiments reported in [19], that qualify MiDaS DPT-Beit-L512 as the most powerful pre-trained model for monocular depth estimation, and further highlight

the need for new metrics that are capable of establishing the quality of the depth predictions not only in terms of absolute values but also relative differences.

Figure 20 shows an example of slope maps derived from ground truth depth maps (upper-left), from MiDaS DPT-Beit-L512 (upper-right), from MiDaS DPT-Swin2-L384 (lower-left) and Keras MDE (lower-right).

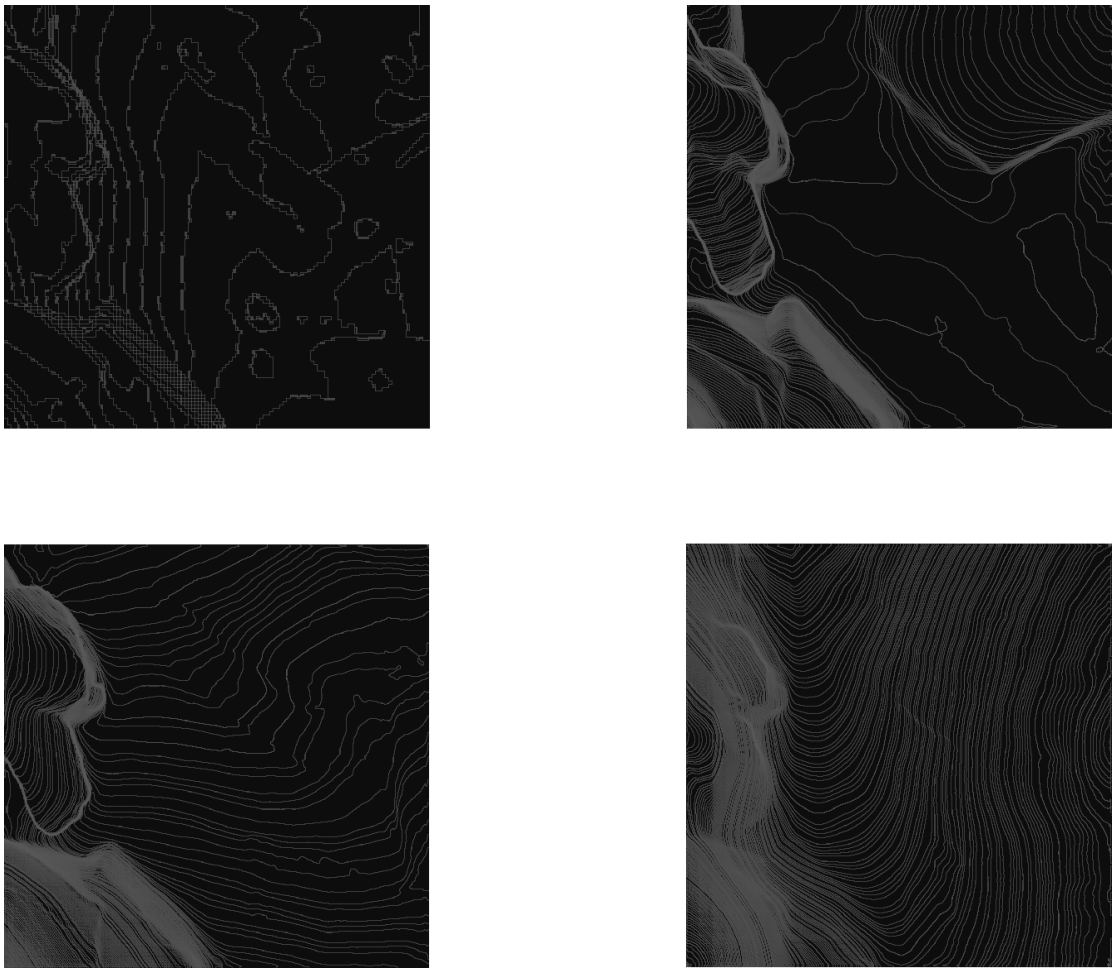

Figure 20: Example of the output from the slope derivation method

4.4 Final Traversability Mapping Results

The last evaluation step for traversability analysis and mapping was focused on the comparison of the final traversability maps, obtained with the methodology described in Chapter 3, in order to establish which monocular depth estimation model performed better in the inference of depth information and in preserving the relative depth differences within their estimation, specifically on the regions classified as “soil” or “bedrock”.

Given that traversability maps are represented in the form of binary matrices, the metrics taken into consideration for this analysis are accuracy, precision, recall and F1-score, that are commonly employed for the evaluation of classification algorithms, as it basically can be addressed as such (with pixels labeled as 1 considered positives, i.e. traversable, and those labeled as 0 negative, i.e. untraversable); these scores are contained in Table 7.

Depth Maps	**Average Accuracy (%)**	**Average Precision (%)**	**Average Recall (%)**	**Average F1-Score (%)**
Keras MDE	68.73	78.36	58.59	66.78
MiDaS (Beit)	**80.12**	**91.68**	**77.42**	**83.94**
MiDaS (Swin2)	79.43	91.57	74.66	82.25

Table 7: Traversability Maps Comparison Metrics computed on the output of Keras MDE, MiDaS DPT-Beit-L512, MiDaS DPT-Swin2-L384 and compared with those obtained from ground truth depth maps

From these final tests that have been conducted on the output of the traversability mapping algorithm, it’s fair to say that the Keras MDE model should have been further fine-tuned on the dataset in order to obtain better results, as the ones deriving from its application are by far the worst between those obtained using the other models that have been taken in consideration; even after varying the input shapes and making the overall encoder-decoder architecture deeper, it seems that for this model the improvement of monocular depth estimation metrics (that were actually pretty close to the first metrics collected for the MiDaS algorithm, as it’s shown in the first section of this Chapter) just wasn’t enough for the later

steps that have been conducted, and even changing the overall architecture of the network might have been necessary in cases where no pre-trained model was available. It is hypothesized that also providing the model with data that are of the same resolution with respect to the original RGB images would further result in better performances, but this was not a suitable solution in this case as the computational overload would have caused (as it already did when using a bigger batch size) the complete failure of the experiment.

Between the two variants of pre-trained MiDaS models, the one employing a Swin v2 architecture was the one that obtained better results in terms of delta threshold metrics for the task of monocular depth estimation, thus initially giving insights that the outputs produced by its inference were the overall best starting points for the remaining steps; instead, the tests presented above show the importance of preserving relative depth differences along the different sections of the images, especially for the objective of this study, as the pre-trained model based on the Beit architecture was able to provide estimated depth maps that served better our purpose of approximating both ground truth slope and traversability maps.

4.5 Path Planning Results

The results regarding the path planning task are now presented. In this Thesis work, at first, the focus was laid on fine-tuning the Neural A* model on the HIRISE dataset, as shown in Chapter 3, in a way that the algorithm could better fit the complex traversability maps that have been determined in the previous steps. Indeed, when trying to directly infer the pre-trained version of the Neural A* (which was trained on the Tiled-MP dataset) the algorithm was not capable to find any optimal path, resulting in a Opt score of 0 for all the tests conducted on the maps derived either from ground truth depth maps, from depth maps produced by Keras MDE and from those produced by the two variants of MiDaS; however, the fact that the Exp metric was not overall equal to 0 shows the generalization capabilities and robustness of this algorithm, as can bee seen in the example shown in Figure 24, where the Exp value was 0.291.

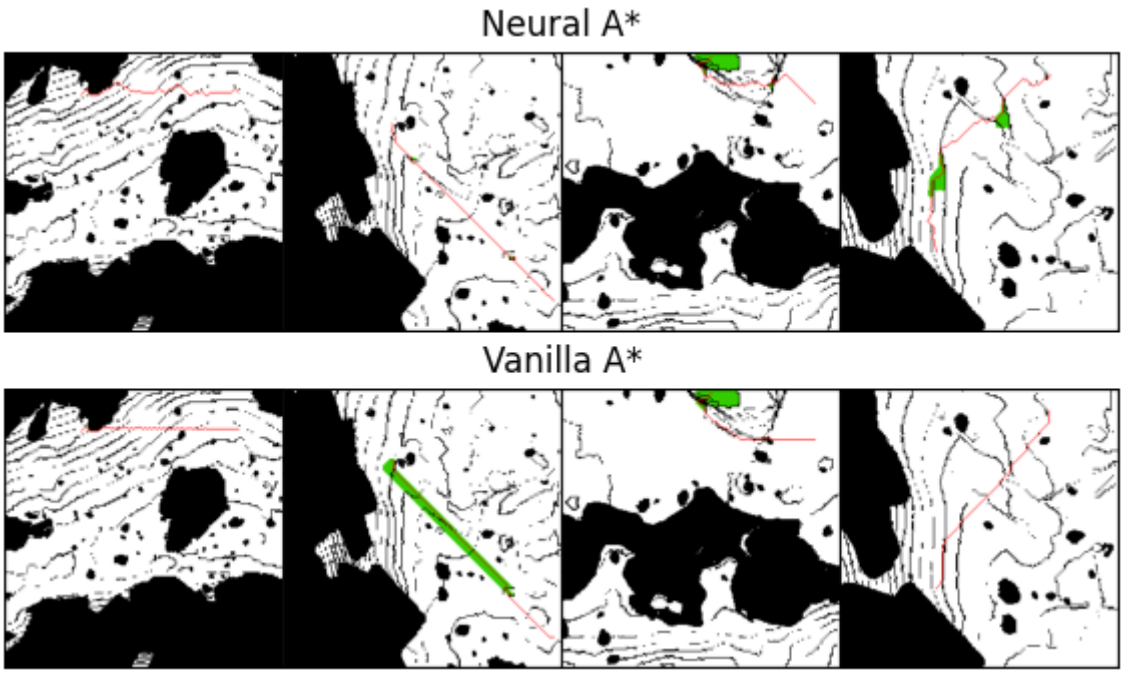

Figure 21: Example of the pre-trained Neural A* path planning output on traversability maps produced on the ground truth depth maps

The fine-tuning procedure conducted on the mock-up version of the HIRISE traversability maps, that are actually unreliable for the reasons discussed in Chapter 3 but served this research anyway as they enabled the training on a very complex environment, successfully produced better overall predicted paths. Figures 22 and 23 show the overall trend of the Opt, Exp and Hmean metrics during the training procedure, using both the RMSprop optimizer and the Adam optimizer for 40 epochs, with a learning rate of 0.001.

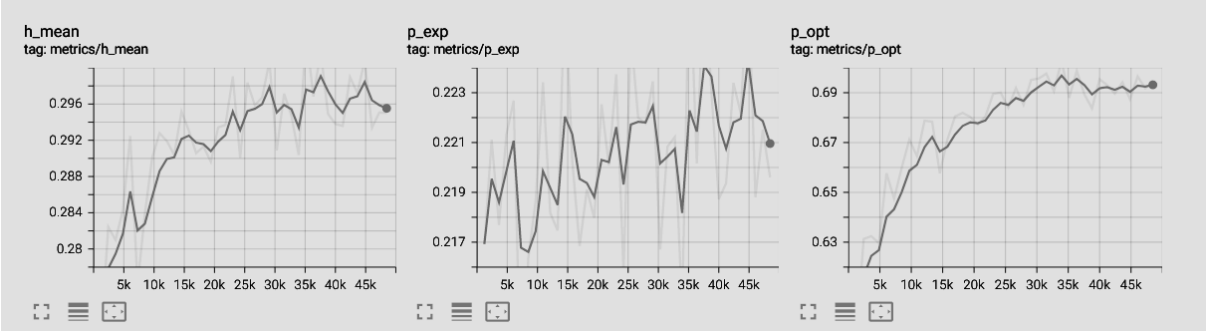

Figure 22: Neural A* training metrics trends when using the RMSprop optimizer

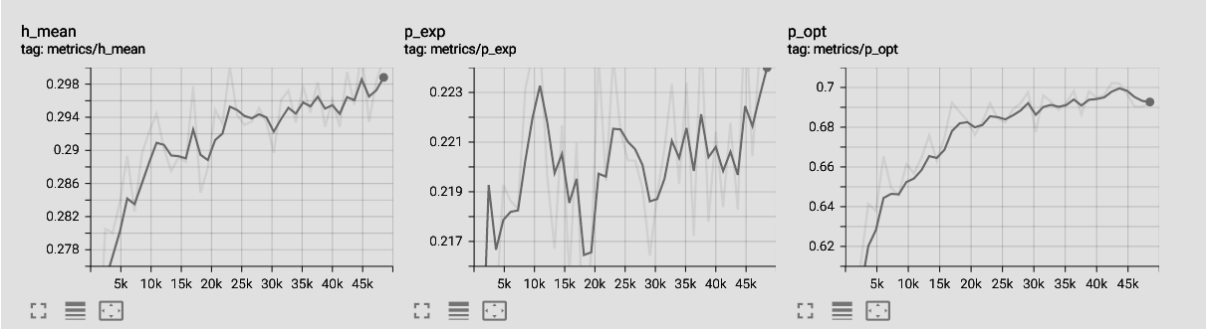

Figure 23: Neural A* training metrics trends when using the Adam optimizer

Even if the overall improvement was relatively minor, the version that was pre-trained using the Adam optimizer was further employed in order to test the algorithm on the traversability maps from the SINAV drone dataset, both ground truth and estimated.

Figures 24 and 25 respectively show some of the outputs from the path planning inference on the traversability maps obtained from ground truth depth values and those obtained using the depth predictions from the MiDaS model with DPT-Beit-L512 backbone; Table 8 contains the average values of Opt, Exp, and Hmean metrics that have been collected when testing the algorithm on a subset (258 examples for each class) of the traversability maps obtained starting from ground truth depth maps and from the other three estimated depth maps.

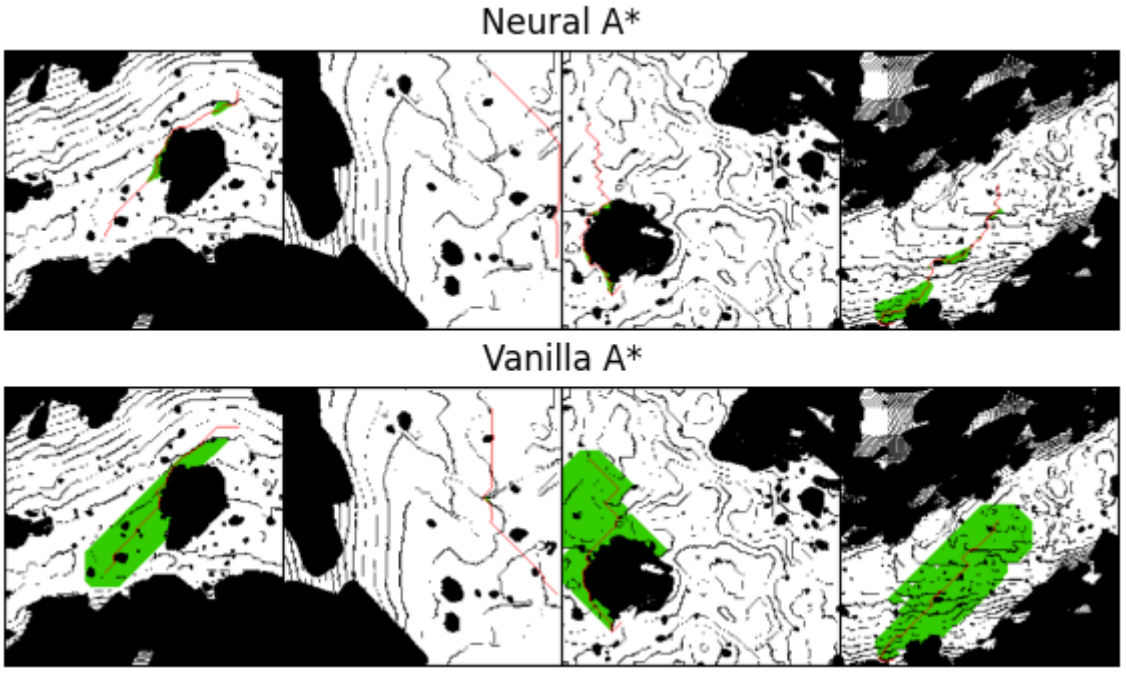

Figure 24: Example of the Neural A* output on traversability maps produced with ground truth depth maps when pre-trained on HIRISE

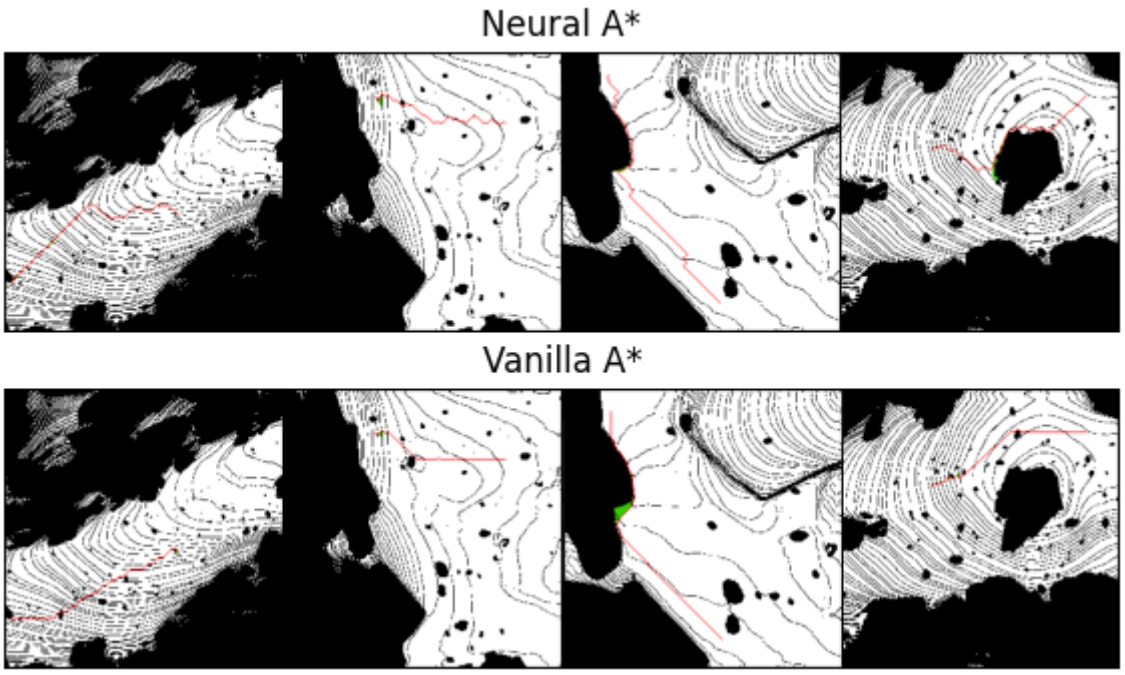

Figure 25: Example of the Neural A* output on traversability maps produced on the estimated depth maps from MiDaS DPT-Beit-L512 when pre-trained on HIRISE

Traversability Maps Based On	Average Opt ↑	Average Exp ↑	Average Hmean ↑
Ground Truth	46.7	**54.6**	50.3
Keras MDE	**68.5**	42.8	**52.7**
MiDaS DPT-Swin2-L384	23.1	39.8	29.2
MiDaS DPT-Beit-L512	29.4	50.2	37.1

Table 8: Average Path Planning Metrics for NeuralA* trained on HIRISE, tested on SINAV traversability maps

Finally, the last results highlight how the learning-based path planner was able to outperform the vanilla A* algorithm, on average, by 46.9% in terms of node exploration ratio in the traversability maps that have been tested, and it also found optimal paths in approximately

41.9% of the test cases; one should be aware that, when looking at these metrics, they're not related in any way to the accuracy of the predicted traversability maps on which the path estimation is performed, as the maps are merely taken as input, and obviously the planning is conducted in the same way regardless of how similar they are to the ground truth traversability map.

5. Conclusions

The objective of this Thesis was the definition of a methodology that incorporated data-driven Traversability Analysis and Path Planning algorithms for the purpose of enabling autonomous and efficient navigation for a Martian rover. This quest implied facing multiple constraints and difficulties, such as the scarce availability of training data, their suboptimal quality, poor computational resources and many others, that have been specified in the previous sections; nonetheless, the final results presented in Chapter 4 show how the proposed implementation of the above tasks gained satisfactory outcomes, both in terms of traversability maps estimation and path searching.

For the first part of this research, the insights gained from literature review highlighted the necessity of information regarding terrain steepness and hazard presence for the scope of defining a rover's navigation maps, and further motivated the investigation and appliance of methods that are capable of extracting the slope information from the simulated terrain pictures that were available; although ambitious, this problem has been approached by first extracting the depth values from the monocular images of the digital scenarios, through a class of Machine Learning models employed for what is referred to as "monocular depth estimation", then approximating the steepness values using a first-order discrete differentiation operator, better known as the Sobel filter, and finally using the arctangent of the gradient magnitude of the two axial components to obtain the slope estimates. This enabled to use a simple threshold for choosing which angles were or were not traversable by the rover; after this procedure, the algorithm merges the segmentation masks that have been filtered based on the presence of dangerous terrain types with the resulting thresholded slope maps from the previous step, thus producing the final predicted traversability maps.

Specifically, two completely opposite supervised-learning approaches have been tested for monocular depth estimation: the first relied on a lightweight encoder-decoder architecture, i.e. the Keras MDE algorithm based on U-Net, that has been fine-tuned on the SINAV drone dataset; the other was based on the usage of a very deep pre-trained model from literature, the MiDaS model, that was directly tested on our dataset without any fine-tuning, using the two best-performing backbones that were trained on a collection of 12 different datasets, which

additionally exploited a robust training procedure that was specifically developed in order to make the model as capable of generalization as possible.

It was difficult to make an a-priori assumption of which model between the two could obtain better results at the beginning, given the multiple complex factors involved, but the initial hypothesis was that a model fine-tuned on the target dataset would obtain better results with respect to one that is tested on such data with a zero-shot approach, given the fact that besides the proven validity and robustness of the second algorithm, the SINAV drone dataset was still very different from the datasets used for the second model pre-training procedure. The resulting metrics, that compare the output of the Keras MDE algorithm and the two MiDaS models with ground truth depth maps, show how even in cases where the first reaches an almost-optimal behavior during training it still does not outperform the second approach during the testing phase, further confirming the strength of the MiDaS model. It's worth noticing that the fine-tuned model yielded better SSIM and RMSLE scores overall, but this can be explained by taking into consideration that SSIM was also one of the three weighted components of the loss that the algorithm minimizes, and RMSLE is purely representative of large discrepancies between predicted and ground truth values, but cannot be used as a reliable indicator of the overall performance. Moreover, between the two pre-trained versions of MiDaS, the one that employed a smaller backbone (based on Swin v2 transformers) initially seemed to produce overall better results, even if the two performances were rather similar. Further research shall be conducted in order to estimate the impact of different network architectures and loss weighting on the performance of the Keras MDE algorithm, as it is believed that even after fine-tuning the model was still too weak in order to compete with the other models from MiDaS.

The metrics computed on the approximated slope maps, which are derived using the estimated depth maps from the previous step and that also represent a crucial point of this research, emphasize that for the final scope of this work it is more important for a model to preserve relative depth differences across different areas of the same image, with respect to accurately predicting absolute depth values; this is clearly due to the fact that slope is indeed a function of the gradient of the depth map. The slope maps computed using the output from the heaviest pre-trained MiDaS variant, based on Beit architecture, surprisingly and consistently outperformed the slope maps derived from the other two MDE models outputs,

in terms of similarity with slope maps computed from ground truth depth values. The model that employs the Beit architecture, indeed, showed a similar capacity with respect to the one using the Swin v2 backbone at estimating absolute depth values, but its better ability to preserve such relative depth differences is proven by these results. An interesting investigation could be performed by confronting the results of this slope derivation algorithm with ground truth slope data, that were unfortunately not available in our context.

Regarding the evaluation of the final traversability maps, the standard binary classification metrics that have been calculated further confirm that the MiDaS model with Beit backbone is the one that best fit our case-study, and that additional care has to be taken when pre-training lightweight models such as the Keras MDE algorithm for tasks that require low-grained features to be accurately predicted. It's also important to notice that the slope threshold, which was used to segment traversable and non-traversable areas with respect to the steepness of the terrain, has been chosen without the necessary validation that should derive from multiple analyses that take into consideration complex real-world constraints that fall outside of the scope of this research, as is expected in the context of planetary rovers, and clearly represents an important element whose impact on the overall research has to be further evaluated in the future.

For what concerns the path planning procedure, in this research extensive experiments have been conducted in order to first test the pre-trained version of the Neural A* model on the traversability maps produced from both the depth values that have been estimated (using the aforementioned three monocular depth estimation models that have been investigated) and the ground truth depth maps, and then evaluate its performance when the training is repeated on a simplistic approximation of HIRISE traversability maps, still testing it on the same traversability maps as in the first case. The approximated HIRISE navigability maps, although completely unreliable in real-case scenarios for the reasons shown in Chapter 3, successfully enabled the model to learn the right path-searching behavior in complex environments, as shown by the results contained in Table 8; said results are not so far from the state-of-art outcomes that the Neural A* developers obtained when training and testing the algorithm on two different splits of the Tiled MP Dataset. It is indeed important to notice that, instead, in this research the training and testing phases were conducted on two separate and quite different data collections.

Besides the fact that our test cases were not large enough to shape comprehensive final considerations about the model, and taking into account the imprecise nature of the training data, it can be noted how the traversability maps produced from the output of the Keras MDE model were those on which the planner functioned the best overall, and the hypothesis that has been made is that this can be due to the randomness of the relative estimated depth maps; the differentiable A* module that is employed in this algorithm is, indeed, very useful in complex environments, enabling the model to learn from its mistakes when the goal is difficult to find, and this could also be the reason why it also performs well when using the maps derived from ground truth depth values, that overall tend to be more tortuous with respect to those estimated by the MiDaS algorithm. This difference in depth map complexity is quite certainly caused by the total absence of training in the zero-shot approach, as the pretrained models are very deep and thus learn to generalize complex features and details in a smoother way than they are actually observed in ground truth examples. Further research shall be conducted in order to validate such hypotheses.

Although the study could have benefitted from additional tests employing the traversability maps estimated with the algorithms shown in the first part of this Thesis for both the training and evaluation stages of the Neural A* algorithm, limitations in terms of time and computational resources prevented us from conducting a more comprehensive analysis in this area, which is thus left for future research.

Ultimately, it’s dutiful to state that, to the best of the writer’s knowledge, the workflow proposed in this Thesis work represents the first attempt to incorporate computer vision and learning-based path planning models for the purposes of traversability mapping and path planning in Martian rovers, thus having the potential to inspire new research activities in this field.

Appendix

Script A: Keras MDE Data Pipeline[13]

```
class DataGenerator(tf.keras.utils.Sequence):
    def __init__(self, data, batch_size=6, dim=(1000, 1000), n_channels=3,
shuffle=True, output_filename=False):
        """
        Initialization
        """
        self.data = data
        self.indices = self.data.index.tolist()
        self.dim = dim
        self.n_channels = n_channels
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.min_depth = 0.1
        self.output_filename = output_filename
        self.on_epoch_end()

    def __len__(self):
        return int(np.ceil(len(self.data) / self.batch_size))

    def __getitem__(self, index):
        if (index + 1) * self.batch_size > len(self.indices):
            self.batch_size = len(self.indices) - index * self.batch_size
        # Generate one batch of data
        # Generate indices of the batch
        index = self.indices[index * self.batch_size : (index + 1) *
self.batch_size]
        # Find list of IDs
        batch = [self.indices[k] for k in index]
        x, y, image_names = self.data_generation(batch)
        if (not self.output_filename):
            return x, y
        else:
            return x, y, image_names
```

[13] Most of this code is taken from the official implementation presented in [16], a decision has been made to report it here as the main changes that have been made during and after the fine-tuning procedure would have not been readable in the form of simple snippets taken out of context.

```
def on_epoch_end(self):
    """
    Updates indexes after each epoch
    """
    self.index = np.arange(len(self.indices))
    if self.shuffle == True:
        np.random.shuffle(self.index)

def load(self, image_path, depth_map):
    """Load input and target image."""
    image_ = cv2.imread(image_path)

    image_ = cv2.cvtColor(image_, cv2.COLOR_BGR2RGB)
    image_ = cv2.resize(image_, self.dim)
    image_ = tf.image.convert_image_dtype(image_, tf.float32)

    depth_map = cv2.imread(depth_map, cv2.IMREAD_GRAYSCALE).squeeze()

    max_depth = min(300, np.percentile(depth_map, 99))
    depth_map = np.clip(depth_map, self.min_depth, max_depth)
    depth_map = np.log(depth_map)

    depth_map = np.clip(depth_map, 0.1, np.log(max_depth))
    depth_map = cv2.resize(depth_map, self.dim)
    depth_map = np.expand_dims(depth_map, axis=2)
    depth_map = tf.image.convert_image_dtype(depth_map, tf.float32)

    return image_, depth_map

def data_generation(self, batch):
    x = np.empty((self.batch_size, *self.dim, self.n_channels))
    y = np.empty((self.batch_size, *self.dim, 1))
    image_namelist = []
    for i, batch_id in enumerate(batch):
        x[i,], y[i,] = self.load(
            self.data["image"][batch_id],
            self.data["depth"][batch_id],
        )
        image_namelist.append(self.data["image"][batch_id])

    return x, y, image_namelist
```

Script B: Keras MDE Blocks Implementation

```
class DownscaleBlock(layers.Layer):
    def __init__(
        self, filters, kernel_size=(3, 3), padding="same", strides=1, **kwargs
    ):
        super().__init__(**kwargs)
        self.convA = layers.Conv2D(filters, kernel_size, strides, padding)
        self.convB = layers.Conv2D(filters, kernel_size, strides, padding)
        self.reluA = layers.LeakyReLU(alpha=0.2)
        self.reluB = layers.LeakyReLU(alpha=0.2)
        self.bn2a = tf.keras.layers.BatchNormalization()
        self.bn2b = tf.keras.layers.BatchNormalization()

        self.pool = layers.MaxPool2D((2, 2), (2, 2))

    def call(self, input_tensor):
        d = self.convA(input_tensor)
        x = self.bn2a(d)
        x = self.reluA(x)

        x = self.convB(x)
        x = self.bn2b(x)
        x = self.reluB(x)

        x += d
        p = self.pool(x)
        return x, p

class UpscaleBlock(layers.Layer):
    def __init__(
        self, filters, kernel_size=(3, 3), padding="same", strides=1, **kwargs
    ):
        super().__init__(**kwargs)
        self.us = layers.UpSampling2D((2, 2))
        self.convA = layers.Conv2D(filters, kernel_size, strides, padding)
        self.convB = layers.Conv2D(filters, kernel_size, strides, padding)
        self.reluA = layers.LeakyReLU(alpha=0.2)
        self.reluB = layers.LeakyReLU(alpha=0.2)
        self.bn2a = tf.keras.layers.BatchNormalization()
```

```
        self.bn2b = tf.keras.layers.BatchNormalization()
        self.conc = layers.Concatenate()

    def call(self, x, skip):
        x = self.us(x)
        concat = self.conc([x, skip])
        x = self.convA(concat)
        x = self.bn2a(x)
        x = self.reluA(x)

        x = self.convB(x)
        x = self.bn2b(x)
        x = self.reluB(x)

        return x

class BottleNeckBlock(layers.Layer):
    def __init__(
        self, filters, kernel_size=(3, 3), padding="same", strides=1, **kwargs
    ):
        super().__init__(**kwargs)
        self.convA = layers.Conv2D(filters, kernel_size, strides, padding)
        self.convB = layers.Conv2D(filters, kernel_size, strides, padding)
        self.reluA = layers.LeakyReLU(alpha=0.2)
        self.reluB = layers.LeakyReLU(alpha=0.2)

    def call(self, x):
        x = self.convA(x)
        x = self.reluA(x)
        x = self.convB(x)
        x = self.reluB(x)
        return x
```

Script C: Keras MDE Model Definition

```
import keras.backend as K

HEIGHT = 512
WIDTH = 512
LR = 0.0008
EPOCHS = 30
BATCH_SIZE = 8

class DepthEstimationModel(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.ssim_loss_weight = 0.85
        self.l1_loss_weight = 0.1
        self.edge_loss_weight = 0.9
        self.loss_metric = tf.keras.metrics.Mean(name="loss")
        self.rmse_metric =
tf.keras.metrics.RootMeanSquaredError(name='root_mean_squared_error')
        self.mae_metric = tf.keras.metrics.MeanAbsoluteError(name='mae')

        f = [16, 32, 64, 128, 128, 256]
        self.downscale_blocks = [
            DownscaleBlock(f[0]),
            DownscaleBlock(f[1]),
            DownscaleBlock(f[2]),
            DownscaleBlock(f[3]),
            DownscaleBlock(f[4])
        ]
        self.bottle_neck_block = BottleNeckBlock(f[5])
        self.upscale_blocks = [
            UpscaleBlock(f[4]),
            UpscaleBlock(f[3]),
            UpscaleBlock(f[2]),
            UpscaleBlock(f[1]),
            UpscaleBlock(f[0]),
        ]
        self.conv_layer = layers.Conv2D(1, (1, 1), padding="same",
activation="tanh")
```

```
    def calculate_mae(self, y_true, y_pred):
        mae = tf.keras.losses.mean_absolute_error(y_true, y_pred)
        return mae

    def calculate_rmse(self, target, pred):
        return K.sqrt(K.mean(K.square(target - pred)))

    def calculate_loss(self, target, pred):
        # Edges
        dy_true, dx_true = tf.image.image_gradients(target)
        dy_pred, dx_pred = tf.image.image_gradients(pred)
        weights_x = tf.exp(tf.reduce_mean(tf.abs(dx_true)))
        weights_y = tf.exp(tf.reduce_mean(tf.abs(dy_true)))

        # Depth smoothness
        smoothness_x = dx_pred * weights_x
        smoothness_y = dy_pred * weights_y

        depth_smoothness_loss = tf.reduce_mean(abs(smoothness_x)) +
tf.reduce_mean(
            abs(smoothness_y)
        )

        # Structural similarity (SSIM) index
        ssim_loss = tf.reduce_mean(
            1
            - tf.image.ssim(
                target, pred, max_val=WIDTH, filter_size=7, k1=0.01 ** 2,
k2=0.03 ** 2
            )
        )
        # Point-wise depth
        l1_loss = tf.reduce_mean(tf.abs(target - pred))

        loss = (
            (self.ssim_loss_weight * ssim_loss)
            + (self.l1_loss_weight * l1_loss)
            + (self.edge_loss_weight * depth_smoothness_loss)
        )

        return loss
```

```
    @property

    def metrics(self):
        return [self.loss_metric, self.rmse_metric, self.mae_metric]

    def train_step(self, batch_data):
        input, target = batch_data
        with tf.GradientTape() as tape:
            pred = self(input, training=True)
            loss = self.calculate_loss(target, pred)

        gradients = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(gradients,
self.trainable_variables))
        self.loss_metric.update_state(loss)

        self.rmse_metric.update_state(target, pred)
        self.mae_metric.update_state(target, pred)

        return {
            m.name: m.result() for m in self.metrics
        }

    def test_step(self, batch_data):
        input, target = batch_data

        pred = self(input, training=False)
        loss = self.calculate_loss(target, pred)

        self.loss_metric.update_state(loss)

        self.rmse_metric.update_state(target, pred)
        self.mae_metric.update_state(target, pred)

        return {
            m.name: m.result() for m in self.metrics
        }

    def call(self, x):
        c1, p1 = self.downscale_blocks[0](x)
        c2, p2 = self.downscale_blocks[1](p1)
        c3, p3 = self.downscale_blocks[2](p2)
```

```
        c4, p4 = self.downscale_blocks[3](p3)
        c5, p5 = self.downscale_blocks[4](p4)


        bn = self.bottle_neck_block(p5)

        u1 = self.upscale_blocks[0](bn, c5)
        u2 = self.upscale_blocks[1](u1, c4)
        u3 = self.upscale_blocks[2](u2, c3)
        u4 = self.upscale_blocks[3](u3, c2)
        u5 = self.upscale_blocks[4](u4, c1)

        return self.conv_layer(u5)
```

Script D: Monocular Depth Estimation Metrics Computation

```
def compute_metrics(depth_true, depth_pred, threshold=1.25):
    depth_true=depth_true.astype("float32")
    depth_pred=depth_pred.astype("float32")

    depth_true[depth_true == 0] = 1e-7
    depth_pred[depth_pred == 0] = 1e-7

    abs_rel = np.mean(np.abs(depth_true - depth_pred) / depth_true)
    sq_rel = np.mean(np.abs(depth_true - depth_pred) ** 2 / depth_true)
    rmse = np.sqrt(np.mean((depth_true - depth_pred) ** 2))
    log_rmse = np.sqrt(np.mean((np.log10(depth_true) - np.log10(depth_pred))
** 2))

    max_ratio = np.maximum(depth_true / depth_pred, depth_pred / depth_true)
    a1 = np.mean(max_ratio < threshold)
    a2 = np.mean(max_ratio < threshold ** 2)
    a3 = np.mean(max_ratio < threshold ** 3)
    ssim_val = ssim(depth_true, depth_pred, data_range=255)

    return [[a1, a2, a3], abs_rel, sq_rel, rmse, log_rmse, ssim_val]
```

Script E: Slope Derivation Method

```
def derive_slope(input_depth):
    grad_x = cv2.Sobel(input_depth, cv2.CV_64F, 1, 0, ksize=3, scale=1,
delta=0, borderType=cv2.BORDER_DEFAULT)
    grad_y = cv2.Sobel(input_depth, cv2.CV_64F, 0, 1, ksize=3, scale=1,
delta=0, borderType=cv2.BORDER_DEFAULT)
    abs_grad_x = cv2.convertScaleAbs(grad_x)
    abs_grad_y = cv2.convertScaleAbs(grad_y)
    grad = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
    slope = np.degrees(np.arctan(grad))

    return slope
```

Script F: Traversability Maps Generation

```
def mask_terrain_type(segmented_img, bigrock_mask, sand_mask):
    # bigrock_mask and sand_mask are numpy arrays that contain the RGB values
of the respective terrain type inside of the segmentation files

    filtered_bigrock = (segmented_img[:,:]!=bigrock_mask)
    masked_bigrock = filtered_bigrock.sum(axis=2).astype(bool)

    filtered_sand = (segmented_img[:,:]!=sand_mask)
    masked_sand = filtered_sand.sum(axis=2).astype(bool)

    return np.logical_and(masked_bigrock, masked_sand)

def traversability_maps_generation(slope_maps, segmentation_masks,
slope_threshold, output_folder):
    # we generate and save all the output traversability maps based on the
input slope maps, slope threshold and segmentation masks.
    # it is assumed that the slope maps files and segmentation masks files
have already been sorted accordingly

    i=0
    for slopefile, segfile in tqdm(zip(slope_maps, segmentation_masks)):
        #segmentation masks are saved as pickle files
        with open(segfile, "rb") as segmentation_m:
```

```
            seg_masks = pickle.load(segmentation_m)
            filtered_segmasks = mask_terrain_type(seg_masks, bigrock, sand)

            slope_file = cv2.imread(slopefile, cv2.IMREAD_GRAYSCALE)
            thresholded_slopefile = np.where(slope_file<=slope_threshold,
True, False)

            trav_map = np.logical_and(filtered_segmasks,
thresholded_slopefile)
            #we save traversability maps as .npz files
            output_path = output_folder+f"_{i}"
            np.savez(output_path)
            i += 1
```

6. Bibliography

[1] Otte, Michael W.. “A Survey of Machine Learning Approaches to Robotic Path-Planning.” (2009)

[2] Ryo Yonetani*, Tatsunori Taniai*, Mohammadamin Barekatain, Mai Nishimura, Asako Kanezaki, "Path Planning using Neural A* Search", ICML (2021)

[3] Rosenblatt, F. “The perceptron: A probabilistic model for information storage and organization in the brain”, Psychological Review, 65(6), 386-408 (1958)

[4] Kohonen, T, “Self-Organizing Systems of Incoherent Light Sources: Applications to the Formation of Visual Maps and Receptor Fields”, Biological Cybernetics, 43(1), 59-69 (1982)

[5] Zisserman, A., & Freeman, W. T, “Unsupervised Learning of Visual Representations using Videos”, IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2021)

[6] Zhu, X., & Goldberg, A. “Introduction to semi-supervised learning”, Synthesis Lectures on Artificial Intelligence and Machine Learning, 3(1), 1-13 (2009)

[7] Watkins, C. J. C. H., “Learning from Delayed Rewards” (Ph.D. thesis). King’s College, Cambridge, UK (1989)

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016)

[9] M. Ono, B. Rothrock, E. Almeida, A. Ansar, R. Otero, A. Huertas, M. Heverly “Data-driven surface traversability analysis for Mars 2020 landing site selection”, 2016 IEEE Aerospace Conference, (2016)

[10] Marr, D., & Poggio, T., “Cooperative computation of stereo disparity”, Science, 194(4262), 283-287, (1976)

[11] Fischler, M. A., & Bolles, R. C. “Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography”, Communications of the ACM, 24(6), 381-395, (1981)

[12] Nishihara, H. K., & Rosenfeld, A, “A pyramid approach to stereo vision. Computer Graphics and Image Processing”, 19(4), 369-395. (1982)

[13] Horn, B. K. P. “Shape from shading: A method for obtaining the shape of a smooth opaque object from one view”. Proceedings of the IEEE, 58(6), 1144-1155 (1970)

[14] Nayar, S. K. “Shape from focus”. IEEE Transactions on Pattern Analysis and Machine Intelligence, 13(8), 779-791 (1991)

[15] Barron, J. T., Fleet, D. J., & Beauchemin, S. S. “Performance of optical flow techniques”. International Journal of Computer Vision, 12(1), 43-77, (1994)

[16] Victor Basu, https://github.com/keras-team/keras-io/blob/master/examples/vision/depth_estimation.py, (2021)

[17] O Ronneberger et al., “U-Net: Convolutional Networks for Biomedical Image Segmentation”, (2015)

[18] Igor Vasiljevic, Nick Kolkin, Shanyi Zhang, Ruotian Luo, Haochen Wang, Matthew R Walter, et al. “Diode: A dense indoor and outdoor depth dataset”. arXiv preprint arXiv:1908.00463, (2019)

[19] Ranftl and Katrin Lasinger and David Hafner and Konrad Schindler and Vladlen Koltun, "Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.44 n. 3, https://github.com/isl-org/MiDaS, (2022)

[20] Geiger, A., Lenz, P., & Urtasun, R. “Are we ready for autonomous driving? The KITTI vision benchmark suite”, In Conference on Computer Vision and Pattern Recognition (CVPR), (2012)

[21] Saxena, A., Sun, M., & Ng, A. “Make3D: Learning 3D scene structure from a single still image”. IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(5), 824-840, (2009)

[22] René Ranftl, Alexey Bochkovskiy, Vladlen Koltun, “Vision Transformers for Dense Prediction”. (2021)

[23] H. Bao, L. Dong, S. Piao, F. Wei, “BEiT: BERT Pre-Training of Image Transformers”, (2022)

[24] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie, Y. Wei, J. Ning, Y. Cao, Z. Zhang, L. Dong, F. Wei, B. Guo, Swin Transformer V2: Scaling Up Capacity and Resolution, (2021)

[23] Dijkstra, E. W, “A note on two problems in connexion with graphs”, Numerische Mathematik, 1(1), 269–271, (1959)

[24] Kaelbling, L. P., Littman, M. L., & Moore, A. W. “Reinforcement learning: A survey”. Journal of artificial intelligence research, 4, 237-285, (1996)

[25] Sergey Levine et al., "End-to-End Training of Deep Visuomotor Policies", (2016)

[26] Volodymyr Mnih et al., "Learning to Navigate in Complex Environments", (2016)

[27] Bhardwaj, M., Choudhury, S., and Scherer, S., “Learning heuristic search via imitation”. In Proceedings of the Conference on Robot Learning (CoRL), (2017)

[28] Sturtevant, N. Benchmarks for grid-based pathfinding. IEEE Transactions on Computational Intelligence and AI in Game, 4(2):144 – 148, (2012)

[29] D. Eigen, C. Puhrsch, and R. Fergus, “Depth map prediction from a single image using a multi-scale deep network,” CoRR, vol.abs/1406.2283, (2014)

[30] Burrough, P.A. ”Principles of Geographic Information Systems for Land Resource Assessment.” Monographs on Soil and Resources Survey No. 12, Oxford Science Publications, New York, (1986)

[31] Irwin Sobel, “History and Definition of the Sobel Operator”, (2014)

[32] Kiri L. Wagstaff, You Lu, Alice Stanboli, Kevin Grimes, Thamme Gowda, and Jordan Padams. "Deep Mars: CNN Classification of Mars Imagery for the PDS Imaging Atlas." Proceedings of the Thirtieth Annual Conference on Innovative Applications of Artificial Intelligence, 10.5281/zenodo.1048301, (2018)

[33] Otsu, Nobuyuki. "A threshold selection method from gray-level histograms." IEEE Transactions on Systems, Man, and Cybernetics 9.1, (1979)

[34] https://unity.com/

[35] C. Couprie, C. Farabet, L. Najman, Y. LeCun, “Indoor Semantic Segmentation using depth information”, (2013)

[36] J. Hurig, N. Schneider, L. Schneider, U. Franke, T. Brox, A. Geiger, “Sparsity Invariant CNNs”, International Conference on 3D Vision (3DV), www.cvlibs.net/datasets/KITTI, (2017)

Acknowledgements

I would like to express my gratitude towards Professor Fabrizio Stesina and my Company Mentor Chiara Leuzzi for giving me the opportunity to partake in such an exciting project. Their supportive guidance and commitment to my development have been instrumental for the success of this research, as well as helping me grow both professionally and personally. To all of the Altec team, thank you for making me feel a very little part of the innovation and excellence that you daily create with your hard work; even if mostly behind a monitor, your support really has been crucial and I'm nothing but thankful for your time.

To my mother, Daniela, my father, Sandro, my aunts Enza and Annamaria, my brother Matteo and my grandmother Lucia, no acknowledgment paragraph could ever fully express the depth of my gratitude for your unwavering love, support, and encouragement. I simply want you to know that I aspire, one day, to become a person as compassionate, caring, and wise as each of you.

To my girlfriend, I am thankful for your love and patience throughout this process, you have been a vital source of motivation and inspiration every single day.

To all the friends in Lecce, and to those I met in Turin, I will never forget the time we spent together and how you made this path lighter and joyful.

And, finally, to the Polytechnic of Turin: although challenging at times, this journey provided me with education and experiences that will doubtlessly serve me in my future endeavors, and for that, I am truly thankful.