POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Estimating Depth Images from Monocular Camera with Deep Learning for Service Robotics Applications

Supervisors

Prof. Marcello CHIABERGE

Mauro MARTINI

Candidate

Luisa SANGREGORIO

S277837

April 2022

Abstract

Estimating depth information from images is a fundamental and critical job in computer vision, as it may be utilized in a large range of applications such as simultaneous localization and mapping, navigation, object identification, and semantic segmentation. Depth extraction can be faced with different techniques: geometry-based (stereo-matching, structure from motion), sensor-based (LiDAR, structured-light, TOF), and deep learning-based.
In particular, monocular depth estimation is the challenge of predicting a depth map using just a single RGB image as input. This significantly reduces the cost and the power consumption for robotics and embedded devices. However, it is frequently described as an ill-posed problem, since an infinite number of 3D scenes might actually correspond to a single 2D view of a scene.
Recently, thanks to the fast development of deep neural networks, monocular depth estimation via Deep Learning (DL), using Convolutional Neural Networks (CNN), has garnered considerable attention, and demonstrated promising and accurate results.
This work compares two different CNN models, with the aim of enabling depth-based real-time applications. Keeping in mind the need to find a lightweight algorithm, capable to run on a mobile platform, a great effort has been also devoted to the choice of models able to reach state-of-the-art performance on widely used datasets, such as NYU-Depth and KITTI.
Since the behaviour of a neural network highly relies on the training data and the intended context for this network is indoors, two unique datasets have been gathered capturing photos from a mobile robot (TurtleBot3) to better fit the target environment.
The first one, made with RealSense D435i is composed of pairs of aligned RGB and depth images. The second dataset, made with ZED 2, is composed of left and right stereo pairs which come along depth and disparity ground truth.
Fast Depth, is devoted to obtaining a dense depth map in a supervised manner while keeping the complexity and computational effort low. The supervised approach has some weaknesses since requires quality depth data in a range of environments. Depth estimation at close distances was not reliable enough for obstacle avoidance.
Monodepth faces depth extraction as an image reconstruction problem. By taking the left frame of a stereo pair as input, it generates two disparity maps - left and right view - while minimizing a photometric reconstruction loss, which involves the right frame as well. Moreover, to improve the predicted depth map for collision-avoidance tasks, the loss function has been regulated by a weighting map that considers the information of the nearest obstacles.

To enhance the effectiveness of these frameworks, both networks have been fine tuned with the collected custom datasets. Finally, the results have been analysed qualitatively and quantitatively, using the main evaluation metrics such as RMSE, RMSE log, Abs Rel, Sq Rel, and Accuracy between the estimated depth and the ground truth.
The success of the predicted depth map has been tested by developing a simple navigation algorithm for obstacle avoidance with a TurtleBot3. The natural progression of this work may be the integration of the depth estimator with advanced autonomous navigation systems in support of indoor service robotics tasks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Estimating depth information from images is a critical task in computer vision. Many robotic applications, such as simultaneous localization and mapping, navigation, object identification, and semantic segmentation, rely on the understanding of the 3D world around us. Depth information is essential to achieving these goals. Deep neural networks using stereo or multiple cameras have recently delivered excellent results in the field of depth estimation, associated with the rapid progress of machine learning. Furthermore, because of the low cost and widespread availability of single sensors in many consumer-grade devices, such as mobile phones, supervised dense depth prediction from a single image has received a lot of interest. However, the most effective techniques rely on a massive amount of accurate, ground truth data for supervision, which is not only difficult and expensive to get, but also susceptible to sensor-introduced noise.
The state of the art has presented several self-supervised image-to-depth algorithms, tackling the issue of depth estimation and offering promising results without the requirement for expensive sensors or ground truth data.

1.1 Objective of the thesis

This research examines two alternative CNN models in order to enable depth-based real-time applications. Keeping in mind the requirement to identify a lightweight approach, considerable attention has also been invested into selecting models able to achieve state-of-the-art performance on commonly used datasets such as NYU depth and KITTI. Both networks were fine-tuned and evaluated on the same testset. The unsupervised network outperformed the other, and the estimated depth map's effectiveness was tested by developing a simple obstacle avoidance algorithm with a TurtleBot3.

1.2 Organization of the thesis

An overview of the thesis is given, with a brief description of each chapter.
Chapter 1 summarizes the goal of the thesis.
Chapter 2 introduces the problem of depth estimation starting from the depth human cues and then explaining some computer vision, sensor-based and deep learning methods.
Chapter 3 addresses the topic of machine learning from its history to its foundations.
Chapter 4 is about the most common data-sets used to train depth estimation networks.
Chapters 5 and 6 deepen on the two neural networks evaluated in this work, with a special focus on Monodepth thanks to its most promising results. The evaluation metrics and comparison of the two networks are shown here.
Chapter 7 addresses the problem of how we can adapt the model to do inference with different cameras.
Chapter 8 explains how an autonomous navigation experiment exploiting the predicted depth map has been carried out.
Chapter 9 briefly summarizes the thesis and concludes with future developments and improvements.

Chapter 2

State of the art

2.1 Introduction

Depth is an important information for autonomous systems, in order to perceive environments and evaluate their own state. Estimating depth information from images is one of the basic and important tasks in computer vision, which can be widely used in simultaneous localization and mapping (SLAM), navigation, object detection, semantic segmentation, etc.
Monocular depth estimation can be framed as: given a single RGB image as input, predict a dense depth map for each pixel This is commonly defined as an ill-posed problem, it means that infinite 3D scenes observed in the world can indeed correspond to the same 2D plane (figure 2.1).

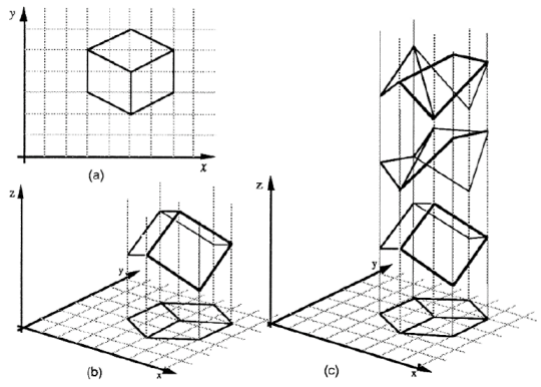


Figure 2.1: (a) A line drawing provide informations about the x,y coordinates of points lying along the object contours. (b) The human visual system is able to reconstruct an object in 3D given only a 2D projection. (c) Any planar line drawing is geometrically consistent with infinite 3D structures [1].

2.1.1 How humans estimate depth?

The human visual system derives depth and distance information from a variety of sources, this taxonomy differentiates between Primary (or Physiological) cues (which are derived from the physiological mechanisms of accommodation, convergence, and stereopsis) and Secondary (or Pictorial or Psychological).
In popular culture, the perception of three-dimensional (3-D) space is often associated with stereovision. However, stereopsis is only one of many depth cues that inform our visual system [2], in fact, other cues are available also when looking at images with only one open eye (monocular).

Primary depth cues

The Primary depth cues are *accommodation, convergence, and stereopsis (i.e., binocular disparity)* [3].
Accomodation is to shift the eye's focus from a distant target to a close target.
Convergence is the rotation of our two eyes to focus on the same target, the vergence angle of the eyes correlates to the distance from the observer to the target (figure 2.2).
Stereopsis or Binocular Parallax is the ability to perceive depth due to two dif-

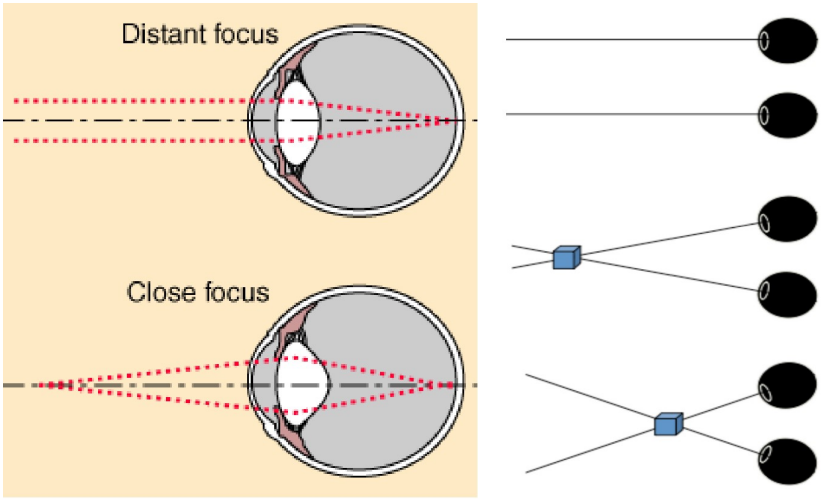


Figure 2.2: [Left] Accomodation: to shift the eye's focus from a distant target (top) to a close target (bottom), the eye muscles controlling lens thickness receive efferent signals. [Right] Convergence: the vergence angle increases as the eyes shift their gaze from a distant target (top) to progressively closer targets (middle and bottom). [2]

ferent perspectives of the world. By comparing images from the retinas in the two eyes, the brain gets depth informations. The greater the disparity, the closer things are to you.
Another one is *motion parallax*, that utilizes the image transformations resulting from observer motion. During motion, in fact, near objects move faster than the

ones farther away. The farther something appears, the slower it seems to pass away from the observer.

Secondary depth cues

The psychological depth cues are retinal image size, linear perspective, texture gradient, occlusion, aerial perspective, and shades and shadows.
Image size is a powerful pictorial cue when portraying items of known or assumed identical size but it can be ambiguous, if considered isolated (figure 2.3).

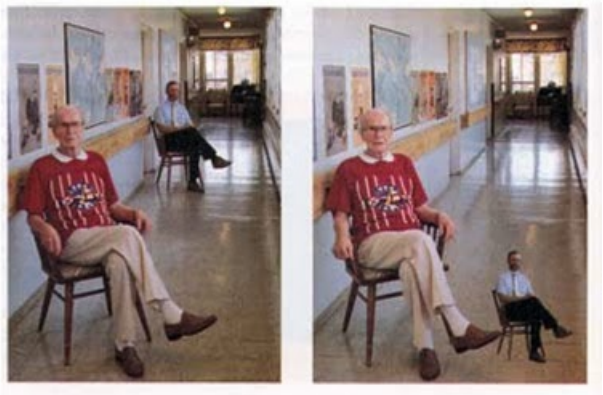


Figure 2.3: Image size is a powerful indicator to object depth, when combined with other linear perspective cues. Even with items of known or assumed identical size, image size becomes confusing when viewed in isolation.

Linear perspective, i.e., that parallel lines converge at the horizon and height in the picture plane correlates with depth. Roads and runways create trapezoids in photographs of normal environments, and the base of a nearer object is further from the horizon than the base of a distant one.
Texture gradients provides continuous depth scaling. Furthermore, because gradients are most typically seen on the ground plane, this signal is important for ground-based navigation and activities. The surface texture of an object becomes more detailed as we get closer to it. As a result, things with smooth textures are typically perceived as being further away.
Occlusion is another powerful visual indication to depth: closer items obstruct more distant things along the line of sight (figure 2.4).

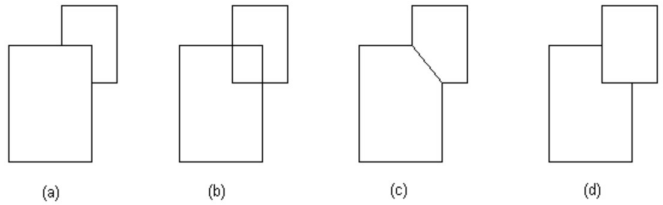


Figure 2.4: Depth perception based on occlusion: the object with more continous border line is felt to lie closer. In (a) the bigger rectangle seems nearer, in (d) the smaller. No depth information can be exploited by (b) and (c)

Shadows and shades. When we know the location of a light source and see objects casting shadows on other objects, we learn that the object shadowing the other is closer to the light source. Shading usually gives some information about depth changes within a surface rather than in relation to other regions of the image.
Aerial (or atmospheric) perspective finally, is limited to objects viewed at fairly large distances as it depends on the effect that the air, between the observer and objects, has on image quality.

2.2 Depth estimation methods

Depth estimation, often known as extraction, is a set of techniques and algorithms for obtaining a representation of a scene's spatial structure. The domain of depth estimation is a complicated object of research, with several methodologies and settings presented.
We'll refer to a *relative measure of depth* as one where we can only determine if one point is closer or farther than another, and an *absolute measure of depth* as one where we can determine what is the real distance between a pixel and the camera.

2.2.1 Geometry-based methods

Geometry-based methods can efficiently calculate the depth values of sparse points, and these methods usually rely on image pairs or image sequences.
Recovering 3D structures from a couple of images based on geometric constraints is a popular way to perceive depth and has been widely investigated in the last forty years.

Stereo vision

Stereo vision 3D information may be derived by comparing a scene from two perspective and assessing the relative positions of objects in the two views. Stereopsis is a biological mechanism that is similar to this.
In order to locate a point on the 3D space, starting from a stereo camera, we can rely on the epipolar geometry.
The **standard epipolar geometry** setup involves two cameras observing *the same 3D point P*, whose projection in each of the image planes is located at p_L and p_R

respectively (figure 2.5). The camera centers are located at C_L and C_R, and the line between them is referred to as the **baseline**. We call the plane defined by the two camera centers and P the **epipolar plane**.
The points where the baseline intersects the two images planes are known as the the **epipoles** e_L and e_R.
Finally, the lines defined by the intersection of the epipolar plane and the two image planes are known as the **epipolar lines**.
The epipolar geometry is useful to find corresponding points in stereo matching. P's projection into the second image must be located on the epipolar line of the second image (figure 2.7)

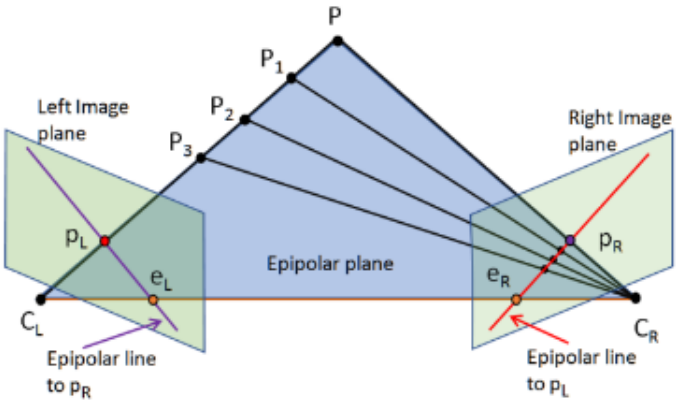


Figure 2.5: Epipolar Geometry [4], general case.

Assuming the specific case of parallel stereo cameras, we can project a point from the real world in the image planes (figure 2.6). When the image planes are parallel to each other, then the epipoles e_L and e_R will be located at infinity since the baseline joining the centers C_L, C_R is parallel to the image planes.
The following relations are derived based on the assumption that the cameras can be approximated by the pinhole camera model. We can then use similar triangles to compute the depth of the point P. So, the knowledge of x_l and x_r, the *focal length* f and the *baseline* B, gives the possibility to calculate the distance Z, through the formula:

$$Z = \frac{fB}{x_l - x_r} = \left[\frac{px * mm}{px}\right] = [mm] \tag{2.1}$$

The difference $x_l - x_r$ is called **disparity**.

$$Z \propto \frac{1}{d} \tag{2.2}$$

Disparity is the difference in image location of the same 3D point when projected under perspective to two different cameras. Depth and disparity are inversely proportional, large disparities correspond to small depths and viceversa.

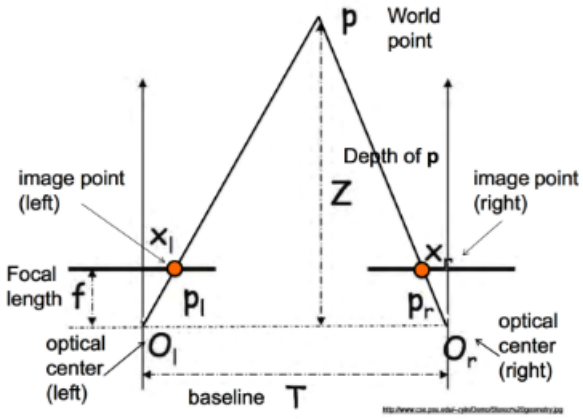


Figure 2.6: Parallel Calibrated Cameras [5]

Stereo-matching

Stereo - matching is the process of finding the pixels in the multiscopic views that correspond to the same 3D point in the scene. The output of a stereo-matching algorithm is a disparity map. There are different approaches on the stereo matching, an example is to minimize a loss function, in order to find, along the epipolar line, the patch around the point (xr,yr) on the source image that look similar to the patch around (xl,yl) on the target image.

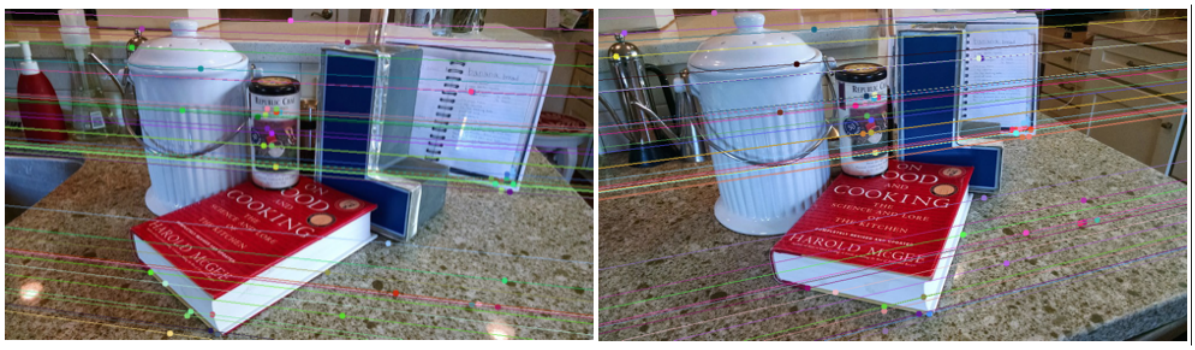


Figure 2.7: Epipolar lines and feature matching

2.2.2 Sensor-based depth

Depth sensors, such as RGB-D cameras, LID TOFAR, can directly obtain the depth information of the associated image.

- In a TOF camera, by transmitting continuous near-infrared pulses to the target scene, the light pulses reflected back from the object are received by the sensor. By measuring the phase difference between the emitted light pulse and the light pulse reflected by the object, resulting in a depth image, the transmission delay between the light pulses may be estimated and the distance of the object relative to the emitter can be derived.

- Stereo 3D Sensors are divided in passive and active. Passive sensors (e.g. Stereolabs ZED [1]) are the least cheaper and use only the stereo pair, they exploit computer vision strategies to get depth, i.e. stereo matching, and they are available in a wide range of baseline to adapt to every use case, but they struggle to perform in low-light conditions. Active stereo 3D sensors (e.g. RealSense D400 series [2]) are equipped also with an infrared (IR) pattern projector which adds more reliable operation in lower light conditions or with low texture-less area. However, the IR projectors have a restricted range of action.

- Structured-light sensors (e.g. Microsoft Kinect for Xbox 360 [3]) project a sequence of light patterns onto a 3D surface, on which it will appear distorted. This distortion will be acquired by a camera and the displacement of any single stripe will directly be converted into 3D coordinates.

- LiDAR (light imaging, detection and ranging) is a special scanning based ToF technology. Common setups of a LiDAR consist of a laser source that emits the laser pulses; a scanner that deflects the light onto the scene; and a detector that picks up the reflected light. Despite its widespread application in the unmanned driving industry for depth measurement, LiDAR can only produce a sparse 3D map.
  It is widely utilized in artificial intelligence systems of outdoor three-dimensional space perception, such as obstacle avoidance navigation of autonomous cars, three-dimensional scene reconstruction, and other applications, due to its large range and excellent measurement accuracy. However, its price and its power consumption are relatively high, and the texture information of the target is lacking.

Because of the low cost, compact size, and wide range of applications of monocular cameras, predicting the dense depth map from a single image has gained increased attention, and it has recently been thoroughly explored based on deep learning in an end-to-end manner.

[1] https://www.stereolabs.com/?ref=hackernoon.com

[2] https://www.intelrealsense.com/?ref=hackernoon.com

[3] https://developer.microsoft.com/it-it/windows/kinect/

2.3 Deep learning-based methods

Monocular depth estimation based on deep learning (DL) has lately received a lot of attention and has shown promising accuracy results, with the fast development of deep neural networks. Convolutional Neural Networks (CNNs) have been used extensively in recent techniques. These approaches are classified as supervised, semi-supervised, or self-supervised.

- *Supervised methods.* Supervised techniques accept as input an RGB image and its corresponding depth map as ground truth, it outputs directly the estimated depth map. This methods may require large, differentiated, high quality data set .

- *Semi-supervised methods.* For training, semi-supervised techniques require a small amount of labeled data and a big number of unlabeled data. Semi-supervised algorithms have the drawback of not being able to correct their own bias, necessitating the use of extra domain data such as camera focus length and sensor data.

- *Unsupervised or self-supervised methods.* Instead of using the ground truth, which is difficult to get, the geometric constraints between frames are used as a supervisory signal during the unsupervised methods' training phase. To train the networks for depth estimation, self-supervised approaches simply require a minimal number of unlabeled images. By relating diverse input modalities, these algorithms extract depth information automatically.

In this thesis I mainly focused on the following works. MonoDepth [6] represents the state-of-the-art technique, while [7] is most suitable for real time applications with low-performance devices.

2.3.1 MonoDepth: Unsupervised Monocular Depth Estimation with Left-Right Consistency

In this paper, Godard et al.[6] innovate beyond existing approaches, which treats depth prediction as a supervised regression problem, replacing the use of explicit depth data during training with easier-to-obtain binocular stereo images. The network, exploiting epipolar geometry constraints, learns how to generate a disparity map that minimizes the photometric reconstruction error. To obtain higher quality depth image, they propose a novel training loss that enforces consistency between the disparities produced relative to both the left and right images. This method produces state of the art results for monocular depth estimation on the KITTI driving dataset.

2.3.2 FastDepth: Fast Monocular Depth Estimation on Em-| bedded Systems

a large number of parameters in these works limit the applications of their network in practice, especially on embedded systems. Hence, Woft et al. [7] address this problem by designing a lightweight auto-encoder network framework. Meanwhile, the network pruning is ap- plied to reduce computational complexity and improves real- time performance. In this work, Woft et al. focus on the design of a lightweight network framework. The proposed archi- tecture is a fully convolutional encoder-decoder. The encoder extracts high-level low-resolution features from the input image. These features are then fed into the decoder, where they are upsampled and merged to form the final high-resolution output depth map. In order to develop a depth estimation network that can run in real-time, they seek low-latency designs for both the encoder and the decoder. The trainable parameters are only 1.34M and the training protocol is supervised with depth ground truth.

Chapter 3

Machine Learning & Deep Learning

3.1 Introduction to the chapter

This chapter's aim is to introduce the main concept behind machine learning and deep learning. After a brief history of machine learning, its foundations are examined, specifically to grasp what a neural net is and how it operates.

3.2 AI, Machine learning, Deep Learning

Artificial Intelligence is a term that is frequently heard in the digitalization era. However, non-experts frequently misunderstand the meanings of the phrases Artificial Intelligence (AI) and Machine Learning (ML).

For sure, the concepts of AI and ML, as well as Deep Learning (DL), are linked by the progress of the technical sector throughout time. AI initially appeared in the 1950s. A. Samuel coined the term ML in 1959 [8], whereas DL was born very lately. The mutual relation between the three of them, can be described with concentric circles, as shown in Figure 3.1 [1].
As a result, AI is a wide concept. AI may be defined as the collection of actions done by computers that enable them to execute activities normally associated with intelligent beings. Alternatively, ML may be seen of as a subfield or approach of AI, with the goal of automatically learning from data. Then, DL is a subfield of Machine Learning that evolved from ML, defined by the use of 'deep' artificial neural networks, i.e. learning models with a large number of layers and neurons.

[1] https://www.edureka.co/blog/ai-vs-machine-learning-vs-deep-learning/

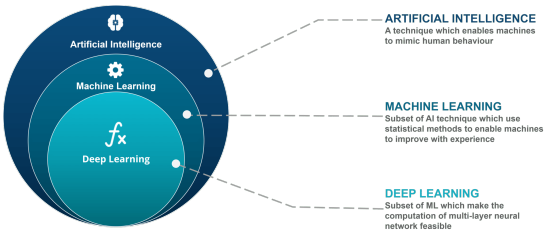


Figure 3.1: AI vs Machine Learning vs Deep Learning

3.3 Machine Learning's History

The initial phase of machine learning research yielded extremely simplistic models of neurons. Cybernetics is another term for this sequence of attempts to replicate brain function. McCulloch and Pitts are credited with creating the first model in 1943, when they used a basic linear binary classifier to simulate a neuron, conceived as simple digital processor and the whole brain as a computing machine, called "Electronic Brain". It was possible to discriminate between two groups of inputs by examining the sign of the function $f(x, w) = x_1w_1 + ... + x_nw_n$. A human was responsible for properly setting the weights.
A significant evolutionary step occurs in the 1950s with the introduction of the Rosenblatt's perceptron, which was able to modify the weights using an early concept of an iterative 'training process' that makes use of a collection of sample inputs for each class. The perceptron algorithm achieved considerable success. Alan Turing devised a test during these years to determine whether a machine could convince a person that they were conversing with another human. Another significant contribution to machine learning is Widrow and Hoff's 1960 Adaptive Linear Element (ADALINE).
The technique used to adjust the ADALINE's weights was a basic version of stochastic gradient descent, which is a primary training process in current deep learning as well. Today, linear models are still frequently used and reinterpreted, despite their limitations. For instance, it is well-known that they struggle with learning the XOR function. Minsky and Papert uncovered these unfavorable implications in 1969 in their work. This resulted in a decline in interest in learning linear models based on neurons during the subsequent period, named the 'first winter' of artificial intelligence.
A second important phase of neural network research began in the 1980s, fueled by the connectionism approach, which attempt to understand and reproduce human mental phenomena.
Connectionism envisioned fundamental contributions to deep learning. The core connectionism premise is that mental phenomena can be characterized as networks of simple and frequently uniform elements. The connections and units might vary in appearance from model to model. For instance, the network's units may be

neurons and its connections could be synapses, like in the human brain.

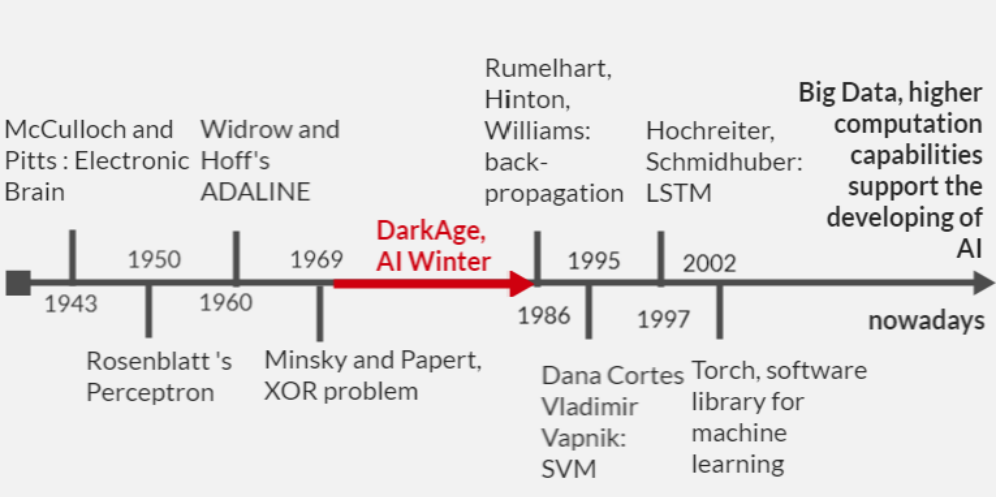


Figure 3.2: Machine Learning Timeline

This resulted in the creation of a novel learning process, back-propagation, for networks of neurone-like units by Rumelhart, Hinton, and Williams in 1986.
The process iteratively adjusts the weights of the network's connections in order to minimize a measure of the difference between the net's actual and desired output vectors. The network is constructed using hidden layers and weights that are not included in the input or output but come to represent critical characteristics of the task domain.
In 1995, Dana Cortes and Vladimir Vapnik developed the support vector machine (VSM), a system for mapping and recognizing similar data. In 1997, the long short-term memory was introduced (LSTM), Hochreiter and Schmidhuber devised a network to overcome the challenges inherent in mathematically modeling extended sequences.
However, there have always been opposing views on machine learning, which was deemed overly ambitious and impractical.
In 2009, Fei-Fei Li, an AI professor at Stanford launched ImageNet, assembled a free database of more than 14 million labeled images. Meanwhile, as research progressed, thanks to the work of o Yoshua Bengio, Yann LeCun, Geoffrey Hinton, it became clear that deep neural networks could be trained. Nowadays, deep learning methods beat AI systems based on other machine learning techniques.
One explanation is because we now have the resources to support these algorithms. Indeed, as the world becomes more digitized and networked, we are living in the age of "Big Data," as we now have access to billions of data points, which provides an enormous resource for training our deep learning algorithm. Because traditional computer hardware was not built to perform algorithms similar to those found

in the brain, many machine-learning activities demand orders of magnitude more computational power than the human brain does.
Simultaneously, there was a dramatic rise in computational resources, such as faster CPUs and general-purpose GPUs, which enabled the operation of increasingly sophisticated networks. While ANN are structurally similar to the human brain in terms of weights, neurons (functional units), topology, and learning algorithms, they are not yet capable of simulating many classes of complicated activities.
It's worth noting that our brains accumulate data for the whole of our lives, and while it's hard to capture life experience in a dataset, especially for some tasks, we don't know how far technology can go.

3.4 Machine learning basic concepts

In these section some basics of machine learning will introduced, in order to have a low-level idea of how a neural network works.

3.4.1 Threshold Logic Unit (TLU)

As previously said, neural networks are derived from the concept of the human brain. To help the reader, the following is a brief introduction to biological neurons and what they have in common with an ANN. Neurons in biology are made up of a cell nucleus that receives information from other neurons via a network of input terminals, or branches, called dendrites (figure 3.3). Through an electrical exchange

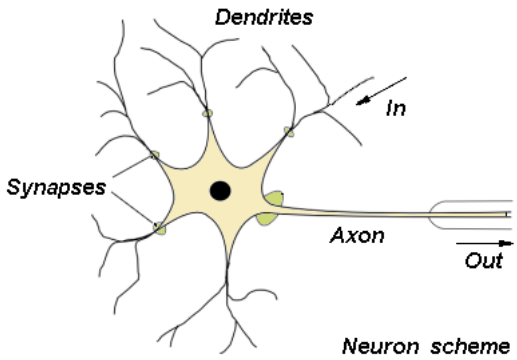


Figure 3.3: Scheme of a biological neuron

of neurotransmitters, a chemical substance, the dendritic tree receives excitatory or inhibitory messages from other neurons. The quantity of this substance, discharged in the synaptic gap, defines the synapse conductivity, which can be seen as a measure of how much the synapse attenuates or boots the axon signal. The nucleus then aggregates these electrochemical signals.

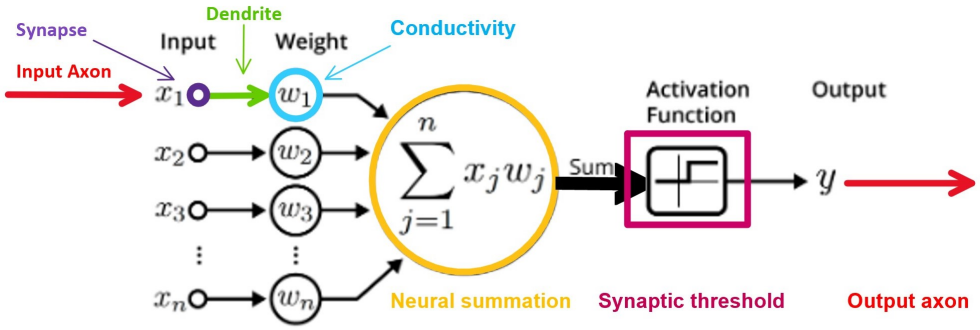
After a process called neural summation, if the aggregate exceeds a synaptic threshold, the neuron is either "activated" or "inhibited," or in other words, switched "on" or "off." The final output is subsequently sent into other neurons, and the process is repeated.

Artificial neural networks are the product of a reverse engineering process of the biological structure just described.

The Threshold Logic Unit (TLU) is the simplest type of an artificial intelligence computational unit, proposed by McCulloch Pitts in 1943. It mimics the high-level schema of biological neurons.

A typical neuron gets several inputs, each of which is paired with a weight w that corresponds to the synapses' conductivity. In a TLU, neurons have identical weights and binary outputs. If no inhibitory signals are present, then all the weighted signals are summed. If the value exceeds the threshold of a particular activation function, the signal is generated (figure 3.4). The behaviour of the model of the artificial neuron can be expressed as follows:

$$f(x) = \begin{cases} 1, & \text{if } \sum_{j=1}^{n} w_j\, x_j \geq \theta \wedge \ \textit{no inhibition} \\ 0, & \textit{otherwise} \end{cases}$$



An illustration of an artificial neuron. Source: Becoming Human.

Figure 3.4: Scheme of a TLU.

3.4.2 The Perceptron

The Perceptron, introduced by Rosenblatt, is the natural evolution of the TLU proposed by McCulloch and Pitts. It relaxes some MCP rules and achieves learning from data in a supervised manner. There are few difference with the TLU's structure:

- The neuron takes also a constant bias input term, b, i.e. weight w_0.
- The weights can be different from each other.
- The input can be either negative with an inhibitory influence.

The model can be written as:

$$y = \sigma(\sum_{j=1}^{n} w_j^T x_j + w_0) = \sigma(w^T x + b) \tag{3.1}$$

This can be translated into:

$$\sigma(w^T\, x + b) = \begin{cases} 1, & \text{if } \sum_{j=1}^{n} w_j^T\, x_j + b \geq 0 \\ 0, & otherwise \end{cases} \tag{3.2}$$

The activation function σ is the Heaviside function ($H(z) = 0\ if\ z < 0$, $H(z) = 1\ otherwise$). However, we can also consider, with equivalent results:

- The **Signium** activation function, with output between (-1,1).

$$sgn(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ -1, & otherwise \end{cases} \tag{3.3}$$

- The **Threshold Gate** activation function, that has as output 1 if the sum of the inputs is over a certain threshold.

$$thr(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & otherwise \end{cases} \tag{3.4}$$

In any case, the decision function linearly depends on the input, hence it is also called *Linear Classifier*.
The learning process, simple and relatively efficient, can perform binary classification, learning the correct synaptic weights w_i from the training examples. The convergence to the optimal weights is achieved with the iteration through all the data, and the updating of the weights whenever the prediction error is different from zero. These kind of neuron model was the pillar of the so-called **first generation** of neural networks, able to implement every binary discrete function with a single hidden layer. The limitation here is that we can classify only linearly separable features (figure 3.5). Many real-case problems can't be faced with linear functions so, in order to get better results, it is necessary to add non-linearity in the architecture.

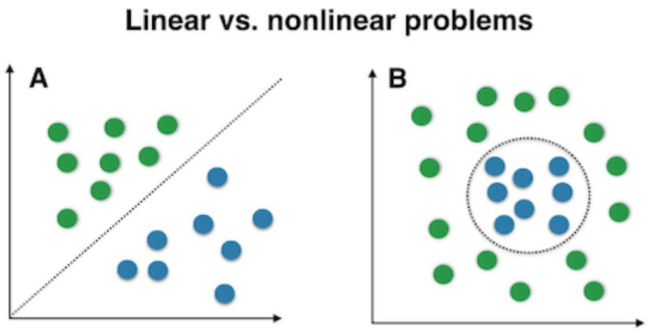


Figure 3.5: Non linear-activation functions allow to classify also non-separable data.

3.5 Artificial Neural Networks

More realistic neural models consider at least three main units: an input layer, an output layer, and one or more hidden layers.

- The input layer receives the data, raw or pre-processed, from the external world, that the system is trying to interpret, recognize and translate.
- The hidden layer is located between the input and output of the algorithm, in which the function applies weights to the inputs and directs them through an activation function to the output.
- The output layer transmits the network's final, processed response.

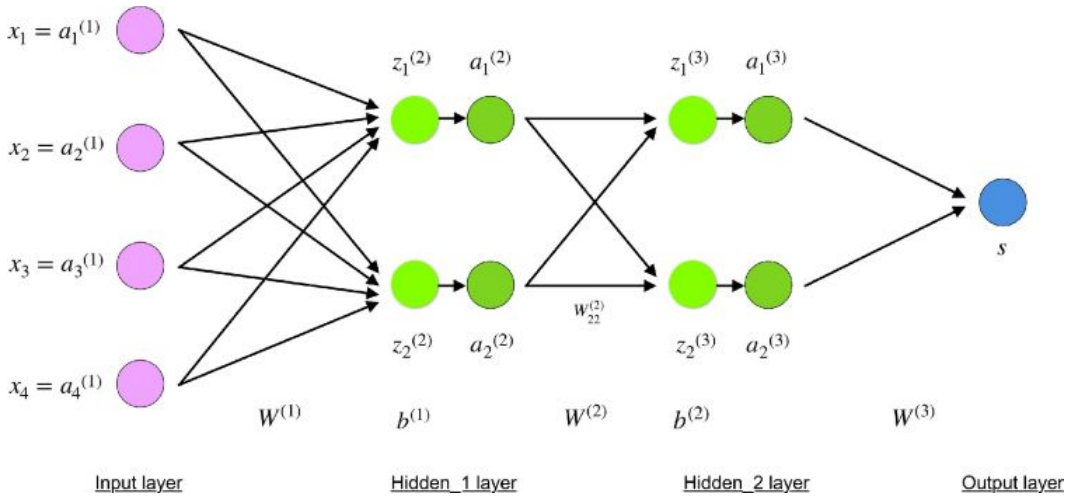


Figure 3.6: An example of a deep neural network with two hidden layers

In a fully-connected neural network, all the neurons of a layer are connected to all the neurons of the next layer and non-consecutive layers are not connected. In the

case of figure 3.6 [2], we can write in vectorized form, for the layer 2 (first hidden layer) and the consecutive activation function:

$$\begin{aligned} \mathbf{z^{(2)}} &= \mathbf{W^{(1)}x} + \mathbf{b^{(1)}} \\ \mathbf{a^{(2)}} &= \sigma(\mathbf{z^{(2)}}) \end{aligned} \tag{3.5}$$

Where x is the input array, $\mathbf{W}$ is the weights matrix with shape $[n, m]$ and b is the offset, with shape $[n,1]$, where n is the number of neurons in the current layer and m is the number of neurons in the previous layer.

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} & W_{14}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} & W_{24}^{(1)} \end{bmatrix}, \quad b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} \tag{3.6}$$

The first number of the subscript matches the index of the neuron in the successive layer and the second matches the index of the previous layer. x is the input vector with shape of $[m,1]$. The complete equation will be:

$$z^{(2)} = \begin{bmatrix} W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + W_{14}^{(1)}x_4 \\ W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + W_{24}^{(1)}x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} \tag{3.7}$$

For a single neuron j of the layer l We can write in a more general form:

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \tag{3.8}$$

Non-linear activation functions

A step forward in the evolution of the NN, is to add non-linearity to the model, in order to deal with more complex problems, through the activation functions. This is also what makes the difference between a Perceptron and an ANN. The activation function must be monotonic, continuous and differentiable everywhere.

- **Sigmoid** Initially, the most used one was the *Sigmoid function*, which squashes the input in a range between 0 and 1.

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(w^T x + b)}} \tag{3.9}$$

In particular, large negative numbers become 0 and large positive numbers become 1. Historically it was preferred since it interprets better the neural behaviour, but now it is fallen out of favor. The main drawback is that it "kills gradient": the gradient is almost zero in the tails of 0 or 1. This means, as we will see later, that the network will barely learn.

[2] https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

Tanh The *tanh* is similar to the sigmoid, it squashes the output between (-1,1) and, since it is zero-centered, it is preferred to the sigmoid nonlinearity.

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{3.10}$$

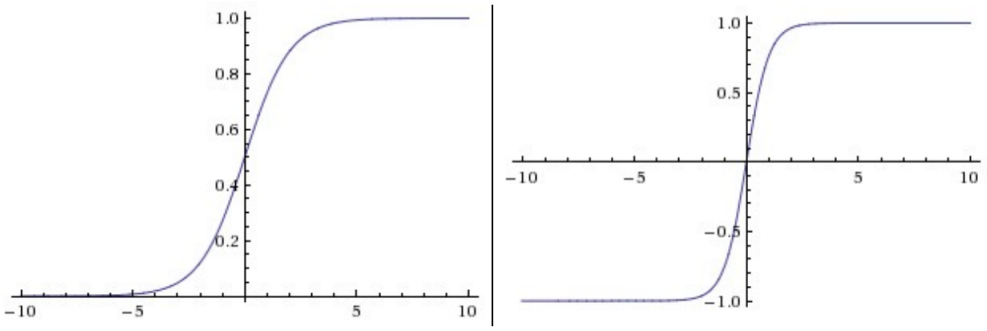


Figure 3.7: Left: Sigmoid function. Right: Tanh function

ReLu The *Rectified Linear Unit* is defined like this:

$$\sigma(z) = max(0, w^T z + b) \tag{3.11}$$

The output will be a linear ramp for positive inputs and 0 otherwise. In other words, the activation is thresholded to zero. It is preferred to the tanh or the sigmoid for many reasons. First, it accelerates the convergence of the stochastic gradient descent due to its linear, non-saturating form and it is computationally less expensive since it involves a simple mathematical operation.

Leaky ReLU The *Leaky ReLU* is a variation of the ReLU that allows a small gradient even when the input is negative.

$$LReLu(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & otherwise \end{cases} \tag{3.12}$$

α is a parameter chosen in design phase.

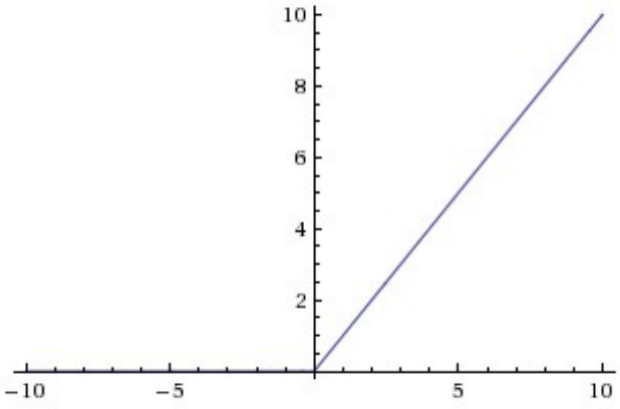


Figure 3.8: Rectified Linear Unit (ReLu)

3.5.1 Training an Artificial Neural Network

The cascade of equations that lead us to the output s is called **forward propagation**, its last step is to evaluate the predicted output s against an expected output y. In uor learning process the goal is about finding the best weights and bias with which the model will provide an accurate output, depending on the task. Starting from some initial values, that could be randomly chosen or pre-trained, weights and bias are updated during the training phase through the optimization of a *loss - or cost - function*, which can tell us how the network is improving its predictions.

$$L(y, s) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - s_i)^2 \tag{3.13}$$

The above formula represents a specific form of quadratic cost function, known as mean square error (MSE). It compares the desired output y and output of the network s , for each training input i, it tells how much the prediction departs from the ground truth. As long as s is the output of the activation function, as we have already said before, it depends on weights w and biases b. The training problem becomes equivalent to finding the best parameters that minimize the loss function. For this purpose, it is used an optimization algorithm called *gradient descent*, which takes the opposite direction of the gradient (steepest ascent) to reach the minimum of a function. Given a generic n-dimensional array v, for small variation of each component v_j, we can define the variation of the cost function as:

$$\Delta L \approx \frac{\delta L}{\delta v_1} \Delta v_1 + ... + \frac{\delta L}{\delta v_j} \Delta v_j + ... + \frac{\delta L}{\delta v_n} \Delta v_n \tag{3.14}$$

Exploiting the concept of gradient we can also write:

$$\Delta L \approx \nabla L \cdot \Delta v \tag{3.15}$$

Where Δv is the vector of the variations of v.

This process can be visualized as the descent from the top of a mountain - the cost function -, possibly through the shortest path 3.10. At each iteration we

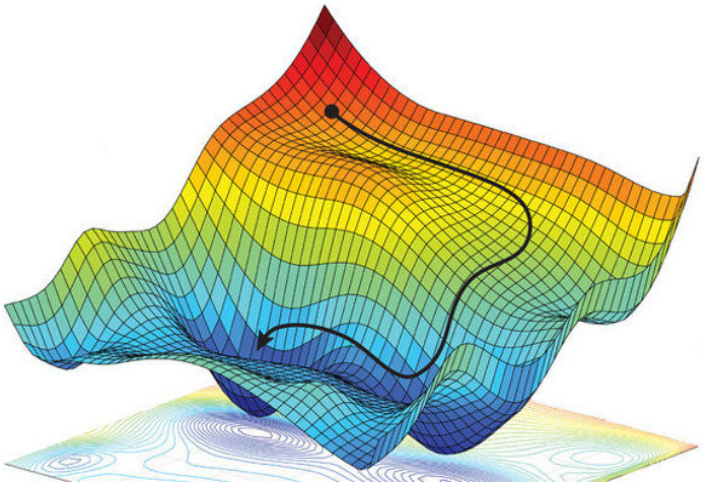


Figure 3.9: Gradient descent to find a local optimum.

calculate the negative gradient and take a step in the direction it specifies, until we reach the bottom of the graph. The size of these steps is regulated by the learning rate η :

$$\Delta v = v' - v = -\eta \nabla L \tag{3.16}$$

So, combining the last two equation we can write:

$$\Delta L \approx -\eta \nabla L \cdot \nabla L = -\eta ||\nabla L||^2 \tag{3.17}$$

The rule for updating the parameters on each iteration is the following:

$$v \to v' = v - \eta \nabla L \tag{3.18}$$

The choice of the learning rate is fundamental for a successful ending of our training. An high learning rate allows us to go faster and cover more ground at each step, but we risk to overshooting the lowest point and even completely diverge from the result. A small learning rate we move trough the optimum more carefully since we are recalculating frequently, but it can be very slow and time consuming. If we apply the above definitions to our specific case, we can find the updating rule for weights and biases:

$$w_j \to w'_j = w_j - \eta \frac{\delta L}{\delta w_j}, b_j \to b'_j = b_j - \eta \frac{\delta L}{\delta b_j} \tag{3.19}$$

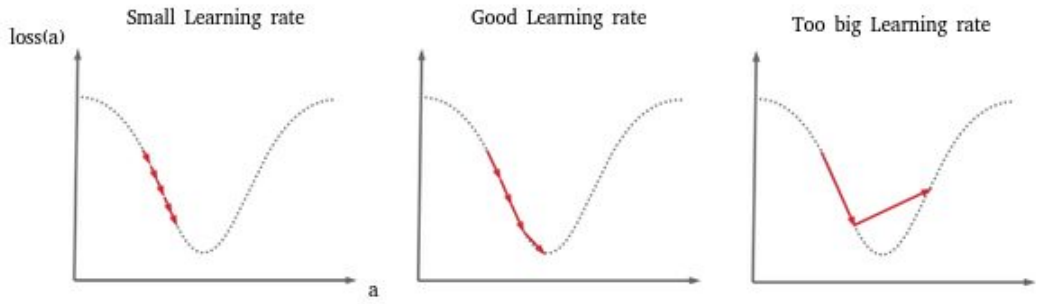


Figure 3.10: Effects of learning rate.

3.5.2 Gradient Descent Methodologies

There are different approaches on computing these weights update. In the **Batch Gradient Descent** all the training data are processed in a single step. The gradient cost is computed for each input and then we take the average. When the number of data is huge, it will result in a very slow convergence. A more efficient approach is the **Mini-batch Gradient Descent** which takes into consideration a small set of the training set at each step, a mini-batch actually, and calculates the average over them to update the weights. When all have been used once, a *training epoch* is finished and a new cycle is started. This approach speed up the computation even if the cost will fluctuate a bit. An extreme version of mini-batch GD is the **Stochastic Gradient Descent** which consider only one example at time to take the single step, so the weights are updated after each training input. It is like setting the batch size equal to one. This will lead to an highly fluctuating cost but it converges faster and it can be used for large datasets. While Batch Gradient Descent guarantees a loss decrease, the other two methodologies won't always take steps in optimal direction, that's the reason why it will fluctuate over iterations.

3.5.3 Back-propagation

The back-propagation is the process which actually acts on the loss function's gradient computation for each layer with a chain rule and on the weights' updating. The algorithm takes its name from the fact that, starting from the last layer L, **the error is propagated backwards through the network**. The central core of the algorithm is, in fact, a quantity called *error* δ_j^l which actually express the error committed by the neuron.

$$\delta_j^l = \frac{\delta L}{\delta z_j^l} \tag{3.20}$$

At the level of the output layer, the error will be composed by two terms, one specifying the influence of the last neuron's output on the cost function and the

other concerning the influence of the input z_j^l on the activation function σ.

$$\delta_j^L = \frac{\delta L}{\delta a_j^L} \sigma'(z_j^L) \tag{3.21}$$

The back-propagation algorithm starts by applying the *chain rule* to the loss function partial derivatives. The gradient for a **single weight** w_{jk}^l of the layer l is:

$$\begin{aligned} \text{Chain rule: } & \frac{\delta L}{\delta w_{jk}^l} = \frac{\delta L}{\delta z_j^l} \frac{\delta z_j^l}{\delta w_{jk}^l} \\ \text{By definition: } & z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \\ \text{By differentiation: } & \frac{\delta z_j^l}{\delta w_{jk}^l} = a_k^{l-1} \\ \text{Final value: } & \frac{\delta L}{\delta w_{jk}^l} = \frac{\delta L}{\delta z_j^l} a_k^{l-1} \end{aligned} \tag{3.22}$$

A similar set of equation can be applied to the **bias term** b_j^l:

$$\frac{\delta L}{\delta b_j^l} = \frac{\delta L}{\delta z_j^l} \frac{\delta z_j^l}{\delta b_k^l} = \frac{\delta L}{\delta z_j^l} 1 = \frac{\delta L}{\delta z_j^l} \tag{3.23}$$

The error term $\delta_j^l = \frac{\delta L}{\delta z_j^l}$ of the hidden layers, which propagates the error from a layer to the previous one, takes care of the activation function in between:

$$\delta_j^l = [(w^{l+1})^T \delta^{l+1}] * \sigma^{l\prime}(z^l) \tag{3.24}$$

Where in the notation * denotes the element-wise multiplication. The last four equation (3.21, 3.22, 3.23, 3.24) describe the essence algorithm.

The back-propagation algorithm along with the mini-batch gradient descent proceed in the following steps, assuming a suitable learning rate η, a random initialization of the parameters and a batch size m.

1. A random mini-batch is taken as input.
2. For each input x in the mini-batch:
   - x enters the input layer
   - **Feed Forward propagation:** for each layer l are computed and stored the results $z^{x,l}$, $a^{x,l}$ and the final prediction $\hat{y}$.
   - **Output Error:** the output error $\delta^{x,L}$ of the final layer L is computed according to the formula 3.21.

- **Back- propagation:** the error is being propagate through all the network $l = L, L-1, L-2, ...,2$ according to formula 3.24.

3. According to the update rule of mini-batch gradient descent for weights and biases do:

$$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} a^{x,l-1}$$

$$b_j \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l} \tag{3.25}$$

4. Another mini-batch is processed until all there are no more data input left and the whole epoch is over.

3.5.4 Momentum, RMSprop, Adam Optimizer

Optimizing the loss function consists in finding the global minimum. The quadratic cost function considered above 3.13 is convex for definition and SGD finds easily the optimum. Real world cost functions have many local minima and reaching the global one could take a very long time. An advanced version of Gradient Descent uses Momentum, which speeds up the gradient in the right direction. **Momentum** adds history to the parameter update equation, using the exponentially weighted averages of the gradients to update the weights instead of the gradients itself directly.

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$$
$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$$

Where β_1 is an hyper parameter to tune. So the updating rule will be:

$$W = W - \eta V_{dW}$$
$$b = b - \eta V_{db}$$

Root Mean Square Propagation (RMSprop) is similar to momentum, it restricts the oscillations in the vertical direction and allows a faster convergence.

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2$$
$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$$

In the notation β_2 is again an hyper parameter to tune and . If dW^2 is smaller than db^2, the algorithm will take steps horizontally, in the direction of w, since the updating on b will be more dumped:

$$W = W - \eta \frac{dW}{\sqrt{S_{dW}}}$$
$$b = b - \eta \frac{db}{\sqrt{S_{db}}}$$

The Adam Optimizer (Adaptive Momentum Estimation) is a combination of Momentum and RMSprop. It is largely used since it can handle sparse gradients on noisy problems. It calculates an exponential weighted average of the past gradients and the squares of the past gradient, with two hyper paramters β_1 and β_2 controlling the decay rates of these moving averages, and updates the parameters in a direction based on combining both informations. So combining the equations from momentum and RMSprop we obtain:

$$W \to W - \eta \frac{V_{dW}}{\sqrt{S_{dW}} + \epsilon} \tag{3.26}$$

$$b \to b - \eta \frac{V_{db}}{\sqrt{S_{db}} + \epsilon} \tag{3.27}$$

Usually for β_1 and β_2 are used the fixed values 0.9 and 0.99, while $\epsilon = 10^{-8}$.

3.5.5 Fine-Tuning

Fine-tuning is strongly related to the concept of transfer learning, which occurs when we use knowledge gained from solving one problem to a new but related problem.
Fine-tuning, in particular, is a technique that takes a model which has already been trained for one task and tunes it to accomplish a second similar task. For example, a pre-trained network on a big and diverse dataset, such as ImageNet, captures universal properties such as curves and edges in its early layers, which are relevant and useful to the majority of classification problems. When developing a model from scratch, we normally have to try many different approaches through trial and error, and we need a large dataset or a lot of time to train the network. This is why the fine-tuning method is so appealing. If we can identify a trained model that already handles one task well, and that task is at least remotely similar to ours, we can exploit whatever the model has already learned and apply it to our specific work. Fine-tuning is carried out by using the weight of a trained NN as initialization for a new model to be trained on data from the same domain (often e.g. images). Another approach is to "freeze" some of the pre-trained weights, usually belonging to the first few layers, which usually capture general properties, which are also relevant to our new challenge. Finally, it is important to choose a lower learning rate. We don't want to distort the pre-trained weights too rapidly or too much because we expect them to be fairly good already when compared to random initialization.

3.6 Data Overfitting

Overfitting is a prevalent issue in neural networks. It happens when a model fits the training data very well but fails against extra and unknown data, defeating its main

objective. An ANN that cannot handle new data is completely useless because it cannot fulfil the classification or prediction task for which it was designed. When a model trains for too long or becomes too complex, it learns the dataset's intrinsic noise or irrelevant information. When a model is sensitive to small variations - outliers - in the training set, it is said to have an *high variance*. Overfitting is evidenced by low error rates in the training phase compared to high rates in the test set. A simple strategy may be the *early stopping*, pausing the training earlier or remove some less relevant inputs, but this may lead to the opposite problem. The contrary to overfitting is, in fact, *underfitting*, which occurs when the model has not being trained for enough time or when the input variables where not significant enough. A trial and error procedure In this section, we will go over the main techniques for preventing this effect.

Splitting data

A good practice in machine learning is to split our dataset into three parts. The *training set* to actually train the network, the *validation set* to evaluate the performance of the network at the end of each epoch, the entity of the gap between the results of training and test set is an index of overfitting. The *test set*is used to finally test the network.

Data augmentation

Data augmentation is a very common practice, it allows to increase the number of data by adding some modified copies of already existing ones. In the specific case of images, there are many transformation that could be applied such as random rotations, flipping, cropping and color modifications.

Regularization

Regularization is another technique which consists on adding a term, or penalty, to the cost function to dump the excessively fluctuation on the function. Thre are three very popular and efficient regularization techniques: $L1$, $L2$ and *dropout*.

The L2 is the most common type, also known as weight decay or ride regression. Let J be the cost function to optimize:

$$J(w^1, b^1, ..., w^L, b^L) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}_i, y_i) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||w^l||^2 \tag{3.28}$$

The regularization term is composed by the sum of the squared weights with a multiplicative factor depending on λ, the *regularization rate*. Increasing this parameter will give more importance to the second term, with the effect of favor smaller weights.

$$w \to w' = w\left(1 - \frac{\eta\lambda}{m}\right) - \eta\frac{\delta L}{\delta w}$$

The regularization introduces an additional subtraction from the current weights.

The L1 loss is similar but instead of the squared sum we have the sum of the absolute values of the weights.

$$J(w^1, b^1, ..., w^L, b^L) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}_i, y_i) + \frac{\lambda}{2m}\sum_{l=1}^{L} |w^l| \tag{3.29}$$

The related updating rule will be:

$$w \to w' = w\frac{\eta\lambda}{m} sgn(w) - \eta\frac{\delta L}{\delta w}$$

L1 regularization produces sparse matrices, i.e. with several values close to zero. In L2 instead the regularization is proportional to the weight itself, so it will keep the weights low without setting them to zero. Smaller weights, intuitively speaking, lessen the influence of the hidden neurons. In that instance, the hidden neurons become negligible, reducing the overall complexity of the neural network. Less complex will avoid overfitting, but increasing too much λ may lead to underfitting. This is why the regularization term is an hyper parameter to be tuned.

Dropout

Dropout is another regularization method that doesn't act on the cost function. During training, with some probability P a neuron of the ANN gets turned off.

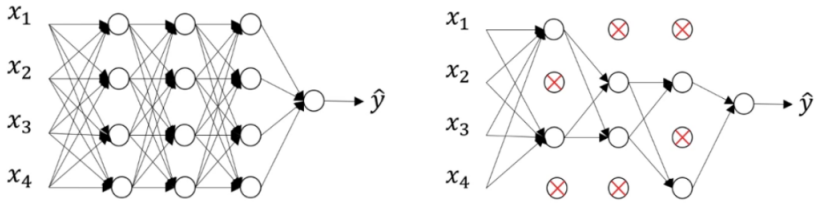


Figure 3.11: Neural Network with dropout, probability P=0.5, (right) and without (left). Source: Journal of Machine Learning Research 15 (2014)

For these neurons weights and biases will not be updated. The result is a simpler network with less connections, that can reduce overfitting.

3.7 Convolutional Neural Networks

The architecture that has been taken in consideration until now is a fully-connected, neuron inside a hidden layer are connected to all the neurons of the previous and the

next layer. This won't be very effective when we are talking about with images as input. First, the spatial structure and temporal dependecies would be lost. Second, the number of parameter would be too high. A fully connected layer for a (small) image of size 100 by 100, for example, has 10,000 weights for *each* neuron in the second layer. *Convolutional Neural Networks* are most suitable for computer vision purposes. There are three main types of layers:

- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

The **convolutional layer** is the core buliding block of a CNN. Considering an input RGB image so with 3 channels - height, width, depth - we have a feature detector, also known as *kernel or filter*, which will check if a feature is present in the image moving across its *receptive field*. The receptive field is in fact the restricted area of the previous layer, to which the neuron is associated (figure 3.12)[3]. In fully connected neural networks instead each neuron receives inputs from every location of the previous layer. The convolution performs the dot product between the kernel and the receptive field, the result is fed into the output matrix. Then, the filter shift by a stride until the kernel hasn't covered the whole image. It is important to highlight that the weights (and the bias) in the kernel remain fixed while it moves across the image, so they are a shared parameter. The final output is known as *feature map* and it will represent the neurons in the next hidden layer, with dimensions n_hxn_h:

$$n_h = \frac{W - F - 2P}{S} + 1$$

Where W is the input image width, F is the kernel size, S is the stride and P is the zero-padding. The **padding** is used to produce larger or equally sized output:

- Valid padding, actually is no padding, the input pixels fallen outside that don't fit the dimensions are dropped.
- Same padding, the output layer is equal to the input one.

Another parameter to choose is the **number of filters**, which affect the depth of the output: four distinct filters would yield to four different feature maps.

The **pooling layer**, performs dimensions reduction using a filter, without weights, applied to small input regions that simplify the contained information. There are two type of pooling:

[3] `https://www.ibm.com/cloud/learn/convolutional-neural-networks`

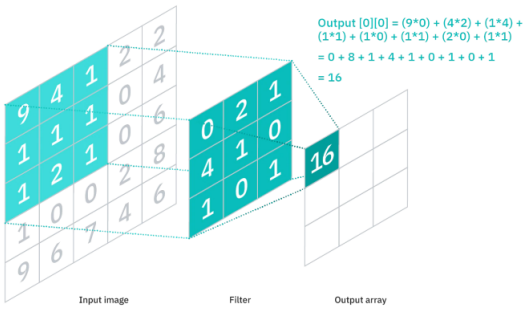


Figure 3.12: Example of convolution: each output value in the feature map is connected only to the receptive field where the filter is applied and not to each pixel of the previous layer

.

- Max pooling: the filter select the pixel with maximum value to populate the output array.
- Average pooling: the filter calculates the average of the input region where it is applied and send it to the output array.

The **fully connected layer** has each neuron of the output layer connected to the previous layer. In this stage the classification task based on the extracted feature is performed, usually it is used a softmax activation function, so that the output ranges between 0 and 1.

In a CNN, the training algorithms previously described, such as back propagation with gradient descent or ADAM optimizer, work in the same way.

Chapter 4

Datasets

Datasets for depth estimation are numerous, with differents depth ranges and indoor or outdoor scenes. In this work, the main used datasets were **KITTI** and **NYUDepth**, along with a **Custom Data Set** in our specific indoor environment.

4.1 KITTI

The KITTI dataset [9] [1] serves as a valid baseline for autonomous driving tasks such as depth prediction, stereo matching, object identification and other similar tasks. A normal station wagon was equipped with two gray scale and two RGB cameras with a 0.54 meter baseline.
Accurate ground truth is provided by a Velodyne laser scanner and a GPS localization system.

Figure 4.1: Station wagon set up.

Its images are outdoor and include scenes of a mid-size city, rural areas and highways. In its raw form, the dataset contains 42,382 rectified stereo pairs from 61 scenes, with a typical image size of 1242×375 pixels. A common spli is the Eigen one, which contains 23,488 images from 32 scenes for training and 697 images from 29 scenes for testing. The **KITTI 2015** [10] split test is made by 200 high quality training scenes and 200 test scenes, taken by the raw KITTI data collection.

[1] `http://www.cvlibs.net/datasets/kitti`

These images contains improved depth ground truth thanks to some post-process techniques. Moreover, moving cars were substitute with 3D cad models to improve accuracy.

Figure 4.2: Sample left-right pair from KITTI.

4.2 NYU-Depth V2

The NYU-Depth V2 data set is an indoor collection of video sequences recorded by both the RGB and Depth cameras from the Microsoft Kinect. It includes 464 scenes: 249 for training and 215 ones for testing. The RGB images have a resolution of 640x480 pixels but it is common to downsaple them during experiments.
There is no one-to-one connection between depth maps and RGB images due to the different frame rates between the two cameras. To align the depth maps and RGB frame, each depth map is first paired with the RGB image that is temporally nearest to it and then they are aligned using the geometrical relationship supplied by the data set. Because the projection is discrete, not all pixels have a depth value, hence the pixels with no depth value are masked off.

Figure 4.3: NYU Depth sample RGB and depth frame.

4.3 Custom Datasets

The necessity to improve the depth prediction accuracy brought me to collect new data to fine tune the models. Supervised strategies learn how to predict depth thanks to the comparison with the ground truth but they became really dependant by the used images. MonoDepth [6] learns in an unsupervised manner, exploiting the problem as an image reconstruction one, but it is highly influenced by camera parameter's. In both cases a fine tuning protocol was essential. Two dataset with two different cameras were collected in an indoor environment, in particular inside university classrooms and offices.
The main characteristic is that the photos were taken from the *robot perspective*, instead of the human one. In particular, pictures were shot from a moving a TurtleBot and the camera was positioned on the top. In order to introduce variety to the dataset, in the second one some images were shot with different camera heights.

First Dataset

The first dataset was collected using a RealSense D435i depth camera [2] (figure 4.4).

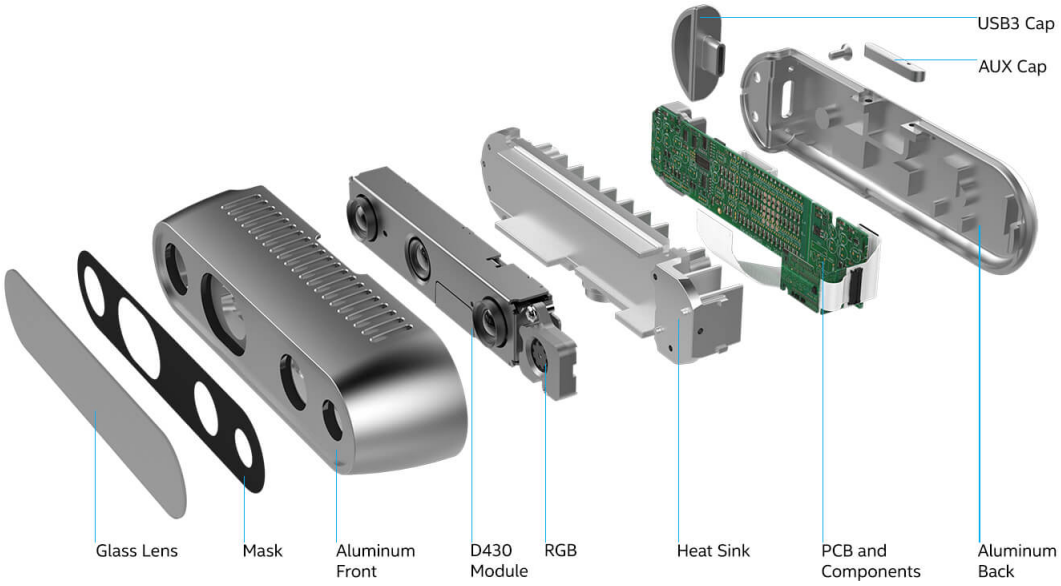


Figure 4.4: Intel RealSense d435i.

The RealSense D435i is a stereo depth camera which consist on a left imager, right imager and a infrared projector. The infrared projector projects non visibile

[2] `https://www.intelrealsense.com/depth-camera-d435i/`

static infrared (IR) light to improve depth accuracy in scenes with low texture or light. The left and right imagers capture scenes and the depth imaging processor calculates the depth values exploiting the epipolar geometry, since the baseline and the focal length are know. Moreover an RGB color sensor provide the RGB picture and, thanks to an apposite API, it is possible to align depth and color frame. The minimum depth that can be computed is $0.28m$, the maximum is $10m$.

Depth Cameras				
Camera	Baseline [mm]	Focal length [px]	Min Depth [m]	Max Depth [m]
RealSense D435i	50	322.1	0.28	10
ZED2	120	528.7	0.3	20

Table 4.1: Cameras used for datasets collection.

It is made by 6,000 RGB frames paired with aligned depth maps and 640x480 resolution. In the following depth maps we can see some "holes" that represent depth data that did not meet the confidence metric, and instead of providing a wrong value, the camera provides a value of zero at that point. These situations include:

- Occlusions- the left and the right images do not see the same object due to shadowing.
- Lack of texture – stereo matching relies on matching texture in the left and right images, so for texture-less surfaces like a flat white wall, the depth estimate can be challenging
- Multiple matches, different depth matches equally good.
- No signal – this happens if the images are under-exposed or over-exposed.
- Below minZ – if the object is very close.

There are some post-process functionalities to fill these holes but the resulting values are approximations,which can lead to inaccuracy. For this reason these maps comes with these holes without being post-processed.

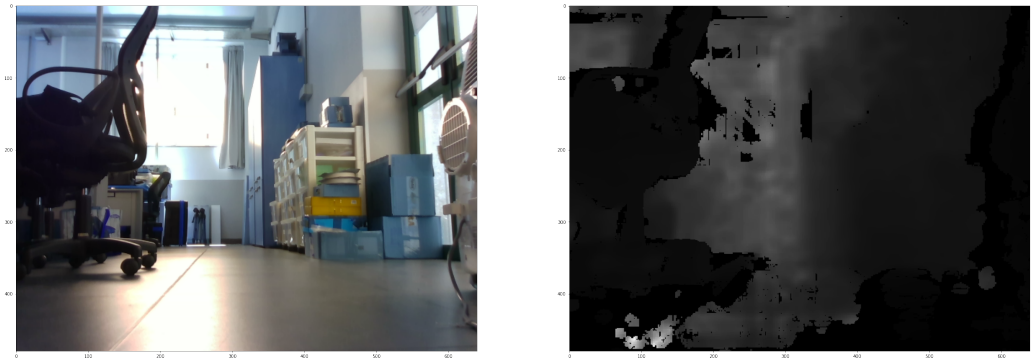


Figure 4.5: Sample of RGB and depth frame from RealSense D435i: backlight condition.

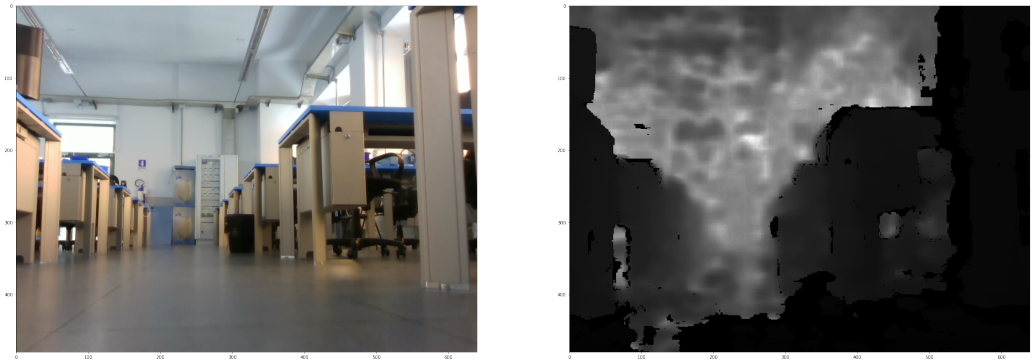


Figure 4.6: Sample of RGB and depth frame from RealSense D435i: side light condition.

Second Dataset

The second dataset was collected using a ZED2 [3] camera which is a passive stereovision based depth camera, this means that it doesn't emit any laser or IR light like active sensors.

Figure 4.7: ZED2 camera.

[3]https://www.stereolabs.com/zed-2/

The ZED camera outputs a high resolution side-by-side color video on USB 3.0. The video contains two synchronized left and right video streams. This color video is used by the ZED software on the host machine to create a depth map of the scene, by comparing the displacement of pixels between the left and right images. The distance is expressed in metric units (meters for example) and calculated from the back of the left eye of the camera to the scene object.

The dataset was collected using HD resolution 1280x720. It is made by 13,000 left and right RGB pair, along with a left aligned depth map. It has a wider range with respect to the RealSense, in fact the minimum and maximum depth values are *0.2 m* and *20 m*. Since the camera reproduces the human view, there are some trouble on estimating depth in situation of poor light condition but, in the same time, they don't suffer from sunlight interferences as IR.
Furthermore, because the number of available rooms was limited, the camera height was varied to add some variety to the dataset, from floor: 15 cm, 38 cm and 60 cm. In the following examples the left-aligned ground truth depth appears with blank spaces associated with an enum:

Figure 4.8: ZED2 camera on TurtleBot, 60 cm height.

- NAN if the depth of the pixel cannot be estimated as it is occluded or an outlier.

- Infinity or -Inifnity if the pixels is respectively too far or too close frome the camera.

As in the first dataset, these holes are left as they are, since filtering stages can alter the actual distance of objects in the scene.

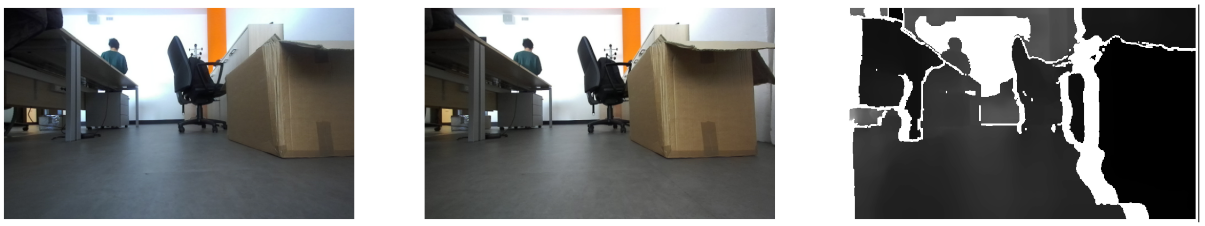


Figure 4.9: Left,right and depth frame from **15 cm** height.

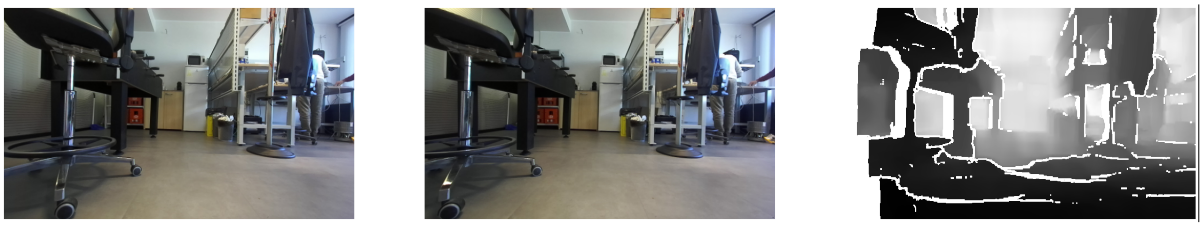


Figure 4.10: Left,right and depth frame from **38 cm** height.

Figure 4.11: Left,right and depth frame from **60 cm** height.

Chapter 5

Fast Depth

Fast Depth is a convolutional Neural network characterized by low-latency network design, it is fully supervised and lightweight (onluy1.34M trainable parameters). In this work, this network was studied, reproduced and fine tuned to achieve better results.

5.1 Network Structure

FastDepth has a fully convolutional architecture with an encoder-decoder structure (see Figure 5.2). From the input image, the encoder extracts high-level low-resolution features. These features are then passed into the encoder, where they are gradually improved and upsampled and then combined to create the final high-resolution output depth map.

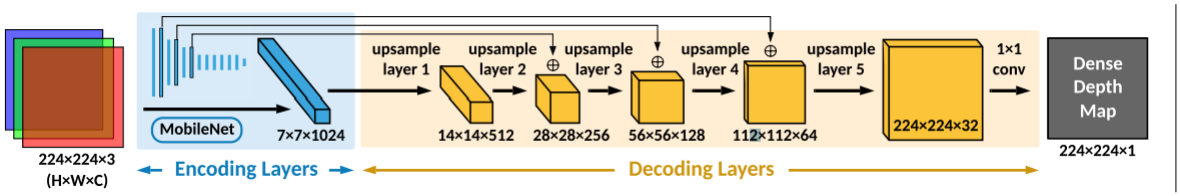


Figure 5.1: Encoder-Decoder Structure

Encoder

The chosen encoder is MobileNet [11] which, using depthwise decomposition, significantly reduces the number of parameters compared with normal convolution with same number of filters. A depthwise separable convolution factorizes a $RxSxCxF$ into a depthwise convolution with C filters $RxSx1$ plus a pointwise convolution with F filters $1x1xC$. In the depthwise layer, each filter convolves only with a single

input channel while in a normal convolution each filter convolves with *all* input channels. Moreover,the pointwise filter has a $1x1$ kernel so that its complexity is RxS times lower. This translates in a reduction of latency.

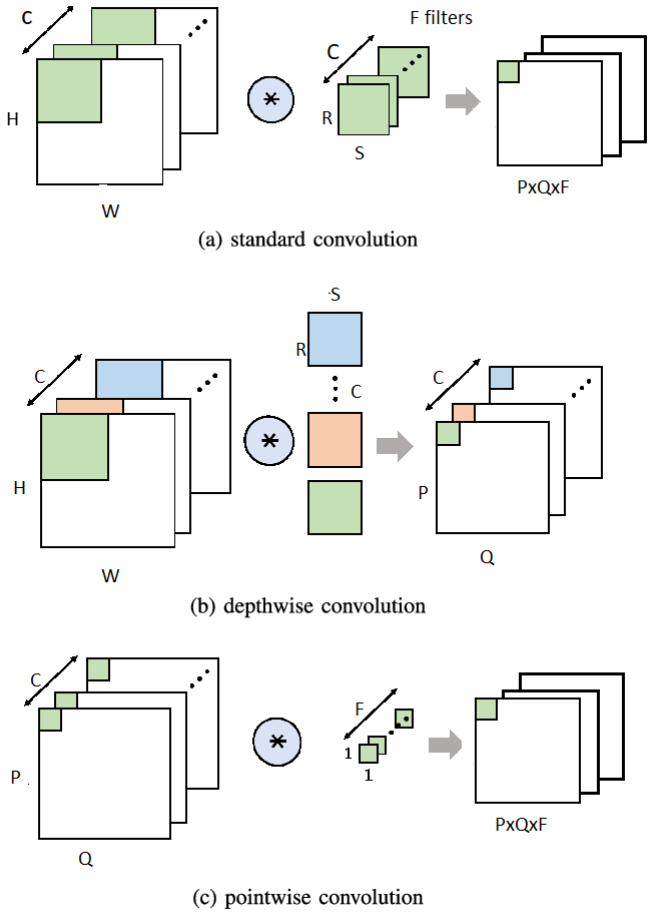


Figure 5.2: Standard convolution Vs Depthwise decomposition

The encoder takes as input a square image with resolution 224x224 pixels.

Decoder

The role of the decoder is to merge and upsample the output of the encoder to obtain a dense prediction. In FastDepth the used decoder is NNConv5 which is made by 5 upsample layers and a single pointwise layer at the end. Each upsample layer compute a 5x5 convolution that halves the number of channels. The encoder's output into the decoder becomes a set of low-resolution features, which might result in the loss of numerous visual details, making it more difficult for the decoder to

recover pixel-wise (dense) data. Image details from the encoder's high-resolution feature maps can be integrated in the decoder using **skip connections**, allowing the decoding layers to reconstruct a more detailed dense output. To prevent increasing the number of feature map channels, they are added together rather than concatenated. In order to reduce latency even further, Wofk pruned the network using NetAdapt [12] which removes redundant channels form the feature maps.

5.2 Implementation Details

FastDepth has a fully supervised approach, which means that it needs ground truth to train. The network was trained and evaluated on the NYU Depth V2 dataset, with the official train/test data split, The following parameters were left untouched, with the exception of the learning rate that was changed in the fine tuning phase:

- **Batch size**: 16.
- **Epochs**: 20
- **Learning rate**: 0.01, reduced by a factor of 2 every 5 epochs.
- **Loss Function**: L1 (mean absolute error).
- **Optimizer**: SGD with momentum of 0.9 and weight decay of 0.0001.
- **Encoder pretrained** on ImageNet.

Pre Process and Data Augmentation

The original frames of size 640×480 are first downsampled to half resolution and then center-cropped, producing a final size of 304×228, then they additionally resize the frames to 224×224 to match the input size to MobileNet encoder.
The **data augmentation** is performed in an online manner, in both RGB and depth frames, with random transformations such as scale, flips and rotations. Moreover, RGB images are augmented in terms of color jitter (brightness, constrat and saturation) and color normalization.

5.3 Experiments

The network performance has been first tested with its best weights given by the official model, then it has been fine tuned to improve the depth estimation.

In order to evaluate the prediction Wofk et al. focused on RMSE, the root mean squared error and $\delta 1$, the percentage of predicted pixels where the relative error is within 25%. Lower RMSE and higher $\delta 1$ values indicate better predictions.
For completeness, more metrics are given to be comparable to other networks. Let

D and $\hat{D}$ be the ground truth and the prediction in meters, after being center-cropped to 304×228 and resized to 224×224. If N is the set of finite points in ground truth and n is any pixel in N:

- **Absolute Error**
$$\text{Abs} = \frac{1}{N}\sum_{n \in N}\left|D[n] - \hat{D[n]}\right|$$
- **Relative Absolute Error**
$$\text{Abs Rel} = \frac{1}{N}\sum_{n \in N}\frac{\left|D[n] - \hat{D[n]}\right|}{D[n]}$$
- **Squared Relative Error**
$$\text{Sqr rel} = \frac{1}{N}\sum_{n \in N}\frac{\|D[n] - \hat{D[n]}\|^2}{D[n]}$$
- **Root Mean Squared Error (linear)**
$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{n \in N}\|D[n] - \hat{D[n]}\|^2}$$
- **Root Mean Squared Error (logaritmic)**
$$\text{log RMSE} = \sqrt{\frac{1}{N}\sum_{n \in N}\|log(D[n]) - log(\hat{D[n]})\|^2}$$
- **Accuracy metric:**
$$\delta = max\left(\frac{D[n]}{\hat{D[n]}}, \frac{\hat{D[n]}}{D[n]}\right) \leq \text{Threshold}$$
With Threshold = 1.25^k k=1,2,3

.

5.3.1 Fine Tuning

The official checkpoint has been tested on a custom 100 photos test set that reproduces the target environment. As we can see from table 5.2 the results are worst in the custom photos with respect to the test set of NYU Depth-V2, so a fine-tuning was necessary.

		Lower is better					Higher is better		
Method	Test Dataset	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
FastDepth	NYUD	0.429	0.165	0.520	0.604	0.073	0.771	0.936	0.980
FastDepth	C	0.885	0.606	1.140	1.009	0.203	0.147	0.485	0.839

Table 5.1: Evaluation metrics of the original model on the official test set NYUDepth-V2 (NYUD) and on custom dataset (C).

The fine tuning was made for 15 epochs, with different initial learning rate values, starting from 0.005 it was decreased looking for the optimum. The other parameters were left untouched.
The dataset used for this purpose was made by 7,600 aligned RGB/depth pair. The photos of the custom dataset were taken by the TurtleBot to fix a new horizon line and with objects at short distance, always under 20 cm according to RealSense limitations, to train the network to recognize nearby obstacles. The evaluation is made with by 100 photos, the same will be used for Monodepth to compare both performances equally.

	Lower is better					Higher is better		
Learning Rate	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
$5.00e^{-3}$	0.887	0.454	1.289	0.982	0.306	0.080	0.185	0.534
$1.00e^{-3}$	0.860	0.435	1.241	0.947	0.257	0.090	0.237	0.623
$5.00e^{-4}$	0.814	0.424	1.117	0.917	0.250	0.098	0.287	0.670
$1.00e^{-4}$	0.804	0.414	1.104	0.900	0.220	0.191	0.290	0.597
$5.00e^{-5}$	0.765	0.356	1.090	0.856	0.194	0.235	0.436	0.799
$1.00e^{-5}$	0.621	0.285	1.018	0.788	0.145	0.420	0.714	0.880
$5.00e^{-6}$	0.594	0.284	0.979	0.774	0.134	0.471	0.763	0.886
$1.00e^{-6}$	0.577	0.118	0.957	0.630	0.104	0.551	0.799	0.894
$5.00e^{-7}$	0.604	0.198	1.011	0.701	0.155	0.458	0.634	0.794

Table 5.2: Evaluation metrics of the original model on the official test set NYUDepth-V2 (NYUD) and on my custom dataset (C)

The line highlighted in green represents the best result obtained with fine-tuning, which is also close to the official results of the paper. However, the real time tests showed insufficient results, with poorly defined contours and errors too large for obstacle avoidance.

Chapter 6

Monodepth

Godard et al. [6] developed a model for predicting depth from a single image. Their goal is to figure out a function f that can predict the depth d of each pixel in an image I, $d = f(I)$. Most extant learning-based old approaches tackle this as a supervised learning issue, with an RGB image as input and an output that is compared to the real depth map. This forces the network to make predictions that are globally consistent. Obtaining solid depth data, on the other hand, is both costly and time consuming. As an example, for moving and translucent objects, the ground truth depth maps from LiDAR sensors offered in KITTI are unreliable. As a result, unsupervised learning becomes a fundamental tool for generating feasible depth maps. Sacrificing some computational lightness, Monodepth is a network that can achieve good results and achieve real-time speeds on the Jetson Xavier AGX, and it is largely used for robotic tasks purposes [13], [14], [15].

6.1 Method

At training time, the important notion is to pose depth estimation as an image reconstruction issue. Once we develop a function that can rebuild the left picture from the right or vice versa using a calibrated stereo camera, we have a small glimpse of the 3D geometry of the scene the camera is pointing to. The network requires I_l and I_r during training, which correspond to the left and right color images from a calibrated stereo pair recorded at the same time and with a known baseline.

The network *learns to generate per-pixel disparity* that produces the right target image by *shifting pixels* from the left input image, using an image sampler. As the sampler uses backward mapping, the resulting disparity map is aligned to the right image, we will call it "right-to-left" disparity map, since it allows to obtain the left image starting from the right one (Naive in figure 6.1).

However, we want the output disparity map to match the input left picture,

which necessitates the network sampling from the right image. Instead, we may train the network to generate the left view by sampling from the right image, resulting in a left view aligned disparity map (No LR in figure 6.1). This method

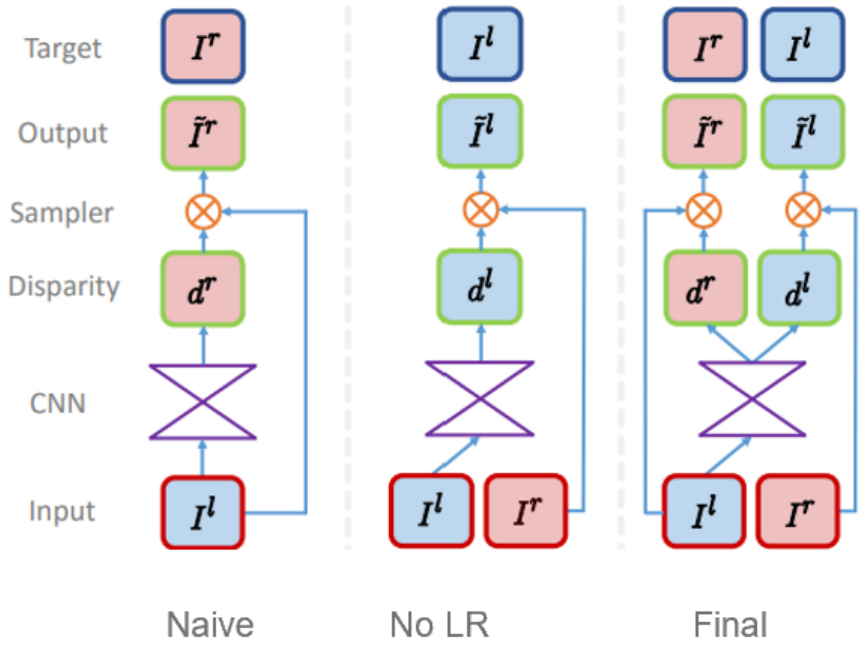


Figure 6.1: Sampling strategies

works reasonably well, but it produces 'texture-copy' artifacts and error at depth discontinuities. As a result, Godard and colleagues solved the problem by training the network to infer two disparity maps at the same time, left-to-right and right-to-left from only one input image. Moreover, a left-right loss enforce the maps to be mutually consistent. Given the baseline distance b between cameras and the focal length f, disparity is a scalar value per pixel that allows the depth D to be recovered, according to 6.1.

$$Depth = \frac{focal\ length * baseline}{disparity} \tag{6.1}$$

6.1.1 Network Structure

The network requires only the left image as input and exploits the right one for the losses during training. The novel approach consists in enforcing consistency between both disparity maps, leading to more accurate results. For training, they employed VggNet [16] and a ResNet50 [17] version, resulting in a convolutional neural network with encoder and decoder. In order to determine higher resolution

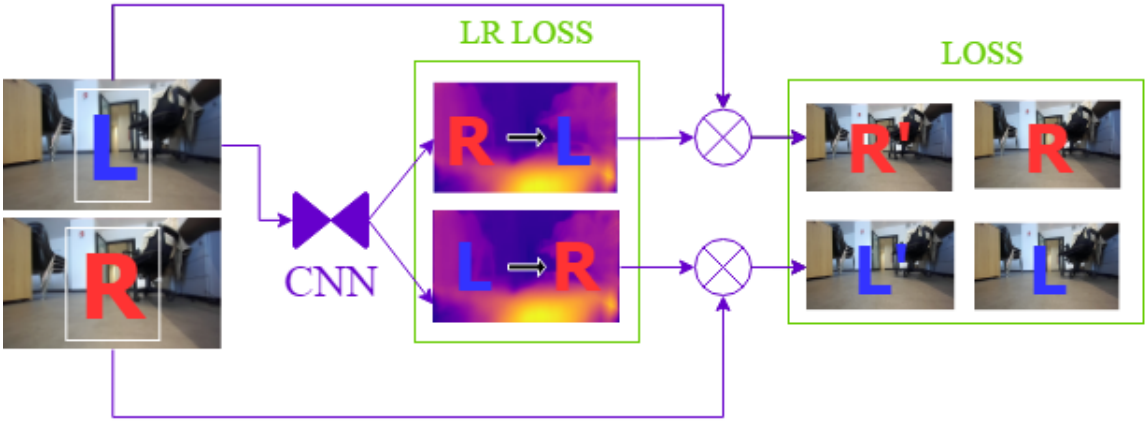


Figure 6.2: The whole strategy, which employs the left image to generate disparities in both images, boosting quality and ensuring consistency.

information, the decoder employs skip connections from the encoder's activations block. The network generates two disparity maps (left-to-right and right-to-left, respectively) at four distinct scales.

Image Reconstruction

The reconstruction of the image is performed finding the the dense correspondence field d_r (or d_l), i.e. the network's output, that, when applied to the left (or right) image, would enable us to reconstruct the right (or left) image. The network generates the predicted image with **backward mapping using a bilinear sampler**, resulting in a fully differentiable image formation model. Forward and backward mapping can be used to construct any geometric transformation.

The forward mapping iterates through the input image, computing new coordinates for each pixel and copying its value to the brand new position. However, the new coordinates may not be integers and may not be inside the boundaries of the output image. The main issue is that each output pixel may be addressed multiple times or not at all, which leads to "holes" where no value is assigned to a pixel in the output image.

The backward mapping solves this problem, since it iterates over each pixel of the output image and apply the inverse transformation to find the position in the input image from which the value must be sampled. This explains why the network learns to produce the right view disparity from the left image and viceversa: the right-to-left map is the inverse transformation from output (right) to input (left).

Algorithm 1 Backward Mapping

for int($m = 0; m <$width$; m{+}{+}$) **do**
  for (int($n = 0; n <$height$; n{+}{+}$) **do**
    Float $x = f_x^{-1}(m, n)$
    Float $y = f_y^{-1}(m, n)$
    target(m,n)=source(x,y)
  end for
end for

The only issue left is that the obtained coordinates, from the inverse transformation, in the source image, are floating points (i.e. "x" in Figure 6.3, is out of the "integer grid"). They can be simply approximated to the nearest integer, but

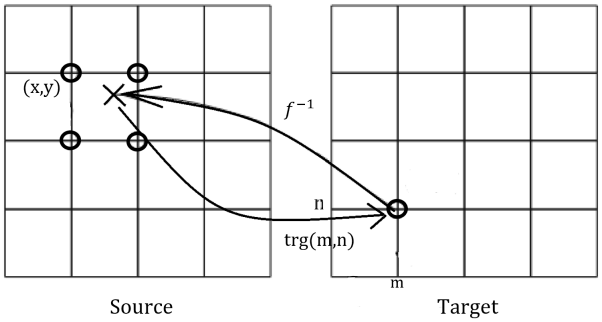


Figure 6.3: Backward mapping

this will lead to aliasing problems. Godard uses **bilinear interpolation**, which interpolates the value of the four closest pixels.
The bilinear interpolant is not linear; but it is linear (i.e. affine) along lines parallel to either the x or the y direction, if x or y is keep constant. The solution can be seen as a weighted mean of each point (figure 6.11), where the weight of each pixel is proportional to its distance from the sampling point 'x'. The bilinear function is also fully differentiable so it allows a simple loss function optimization.

6.1.2 Losses

Monodepth network infers well defined disparity maps according to three main losses. Each scale s has a loss C_s, thus the total loss is the sum $C = \sum_{s=1}^{4} C_s$, with C_s equal to:

$$C_s = \alpha_{ap}(C_{ap}^L + C_{ap}^R) + \alpha_{ds}(C_{ds}^L + C_{ds}^R) + \alpha_{lr}(C_{lr}^L + C_{lr}^R)$$

Each of the three terms is composed by the left and the right component,but only the left image is fed to the CNN.

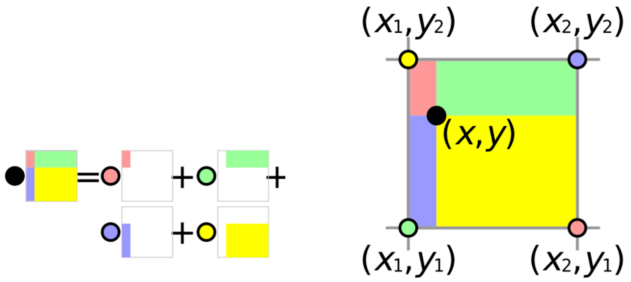


Figure 6.4: A geometric visualisation of bilinear interpolation. The product of the value at the desired point (black) and the entire area is equal to the sum of the products of the value at each corner and the partial area diagonally opposite the corner (corresponding colours). Source: Wikipedia [18].

An **enhanced loss function** with a weighting map has been implemented to increase the depth map quality and it will discussed in the successive subsection.

Appearance Matching or Reprojection Loss

This loss enforces the reconstructed image to appear similar to the corresponding training input. It's a sort of photometric loss composed by a combination of L1 and single scale SSIM which compares the input image I_{ij}^L and its reconstruction $\tilde{I_{ij}^L}$ obtained by bilinear sampling.

$$C_{ap}^L = \frac{1}{N}\sum_{ij} \alpha \frac{1 - SSIM(I_{ij}^L, I_{ij}^{\tilde{L}})}{2} + (1-\alpha)||I_{ij}^L - \tilde{I_{ij}^L}||$$

where N is the number of pixels.

The The Structural Similarity Index (SSIM) [19] is a metric that extracts and compares 3 key features from an image: Luminance, Contrast, Structure.
For image quality assessment, it is useful to apply the SSIM index locally rather than globally. The main reason is that image statistical features and distortions are usually highly spatially non-stationary. So, local comparison would achieve higher quality images comparisons. At each step, the local SSIM index is calculated within the local window and the mean over all the windows is taken.
The choice of Godard was to use a 3x3 block filter, with $\alpha = 0.85$.

Disparity Smoothness Loss

This loss enforces smooth disparities exploiting an L1 penalty on the disparity gradients δd. To preserve edges on disparity maps, there is an *edge aware* weight term, which exploits the image gradients δI. When there is a large change in pixels

intensity, the gradient is high and the exponential brings the loss to zero, since there could be possible edge pixels.
Otherwise, the loss term penalizes drastic depth changes in 'flat' regions.

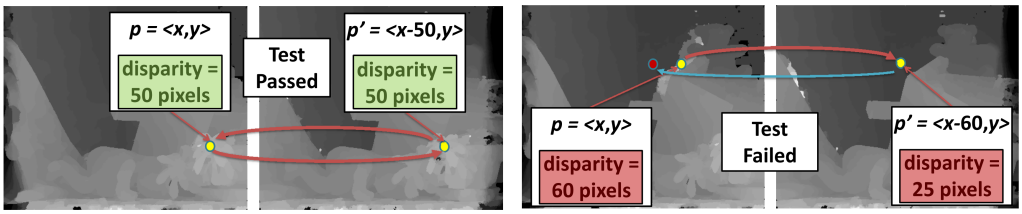
$$C_{ds}^{L} = \frac{1}{N} \sum_{ij} |\delta_x d_{ij}^{L}| e^{-||\delta_x I_{ij}^{L}||} + |\delta_y d_{ij}^{L}| e^{-||\delta_y I_{ij}^{L}||}$$

Left-Right Consistency Loss

In order to improve disparity map quality, the network produces both left and right disparities, To ensure coherence, they introduce an L1 left-right disparity consistency penalty, that tries to make the left-view disparity map be equal to the projected right-view disparity map.

$$C_{lr}^{L} = \frac{1}{N} \sum_{ij} |d_{ij}^{L} - d_{ij+dij^{L}}^{R}|$$

Given a map of the two disparities, for each pixel p_l in the left view, using its disparity d_l, we find the corresponding pixel p' in the right view by horizontally shifting its coordinates by an amount equal to the disparity itself. For the pixel p', we lookup its matching point q_l in the left view using the right disparity map. The loss minimize the difference between p_l and q_l, which should be the same.



(a) Passed consistency check: $q_l = p_l$ (b) Failed consistency check: $q_l \neq p_l$

Figure 6.5: Example of left-right consistency passed check

6.1.3 Modified Loss

The loss function was modified following the idea of Chen [14]. The modified loss catches novel obstacle information that can subsequently be used exploited to support collision avoidance.
Consequently, the accuracy of disparity estimation results is improved and objects appear sharper, while the original model's effectiveness is maintained. So, with the pursuit of obstacle avoidance, large disparities (smaller depths) are more critical so, this modified loss will give more importance to them, through a weighting map. Exploiting the usage of the stereo pair, a **Semi-Global Matching [20]** algorithm outputs a reference disparity map $D^l(u,v)$ which is used to obtain a **weighting map** $M^l(u,v)$ with values comprised in the interval $[p,q]$. Nearest pixels, below a certain distance, are assigned to have higher weight, q, and, on the contrary, further pixels are assigned to weight p:

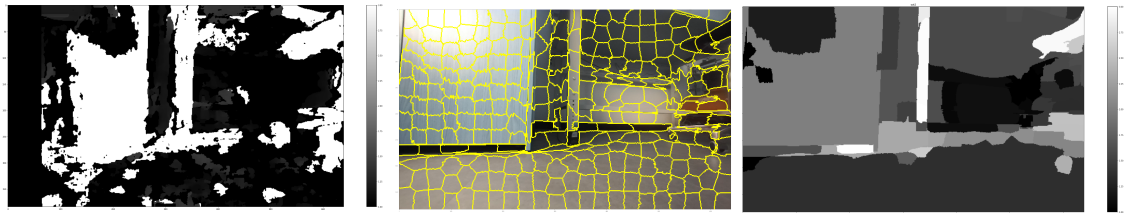
$$M^l(u,v) = \begin{cases} q, & \text{if } \ d^l(u,v) \geq \frac{fb}{z^n} \\ p, & \text{if } \ d^l(u,v) \leq \frac{fb}{z^f} \\ \frac{(q-p)(d^l(u,v)-\frac{fb}{z^f})}{fb(\frac{1}{z^n}-\frac{1}{z^f})} + p, & otherwise \end{cases}$$

Where z^n and z^f are arbitrary the nearest and the furthest distances to define the weight matrix.
In particular the optimal choices were:

- $p = 1,\ q = 3$
- $z^n = 0.6\ m,\ z^f = 3\ m$

Meanwhile, the **SLIC [21] approach** segments the left-view image into an image full of Superpixels. The concept of superpixel helps to perceive in a more semantically way the image, since it is defined, instead by a grid pixels (given by $width$x$height$), by clusters of pixels. In fact a single pixel, standing alone by itself, is not a natural representation of an image and it has not any semantic meaning.
Superpixels has more perceptual meaningfulness, since its pixels that share some sort of commonality, such as similar color or texture distribution.
In figure 6.6 it is showed the whole process. From left (a) and right (b) image, the left view disparity map $D^l(u,v)$ (c) is obtained through SGBM. Then, $M^l(u,v)$ is computed (d). The segmentation obtained from the left image (e) is applied to the weight map $M^l(u,v)$. Through the RAG technique, regions characterized by a similar mean color are grouped and combined.
The final result (f) is a smoother weight map, improved on its accuracy, since it takes into consideration the semantic meaning of the scene. Lighter region are characterized by an higher weight.

(a) Left image (b) Right Image (c) SGBM: $D^l(u, v)$



(d) Weighting map $M^l(u, v)$ (e) SLIC: Superpixels (f) Final weighting map

Figure 6.6

The same procedure is performed also with respect to the right view disparity map $D^r(u, v)$, to obtain the right component of the weighting map $M^r(u, v)$.

Finally, the losses are modified as follow (for convenience it is showed only the $left$ component of the los but the same is applied to the right one):

$$C_{ap}^{L\,\prime} = \frac{1}{N}\sum_{ij} \alpha M_{ij}^{l} \frac{1 - SSIM(I_{ij}^{L}, \tilde{I_{ij}^{L}})}{2} + (1-\alpha) M_{ij}^{l} ||I_{ij}^{L} - \tilde{I_{ij}^{L}}||$$

$$C_{ds}^{L\,\prime} = \frac{1}{N}\sum_{ij} M_{ij}^{l} |\delta_x d_{ij}^{L}| e^{-||\delta_x I_{ij}^{L}||} + M_{ij}^{l} |\delta_y d_{ij}^{L}| e^{-||\delta_y I_{ij}^{L}||}$$

$$C_{lr}^{L\prime} = \frac{1}{N}\sum_{ij} M_{ij}^{l} |d_{ij}^{L} - d_{ij+dij^{L}}^{R}|$$

The weighting map is applied to the loss function for each resolution of the outputted disparity (fig 6.7):
The figure below shows the improvement in the disparity output, the nearest obstacles appear more defined and the map more clear.

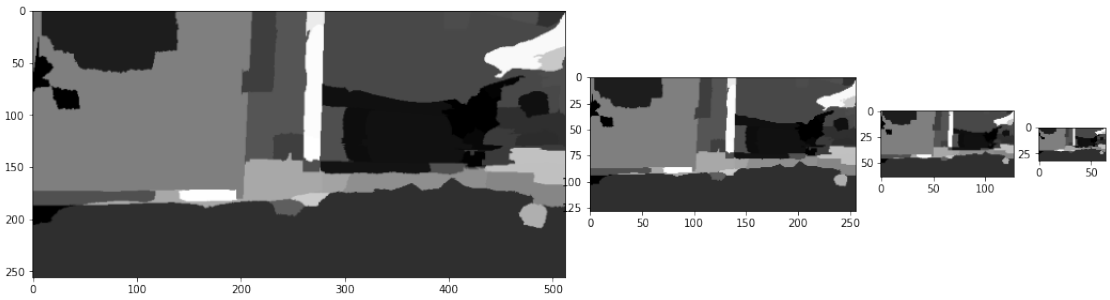
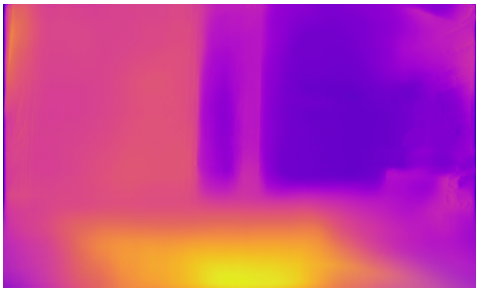
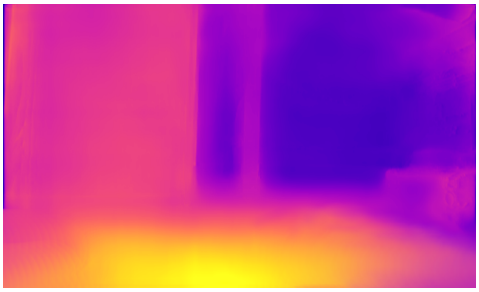


Figure 6.7: $M^l(u, v)$ scaled for s=1,2,3,4.



(a) Disparity from the model fine tuned without $M^l(u, v)$

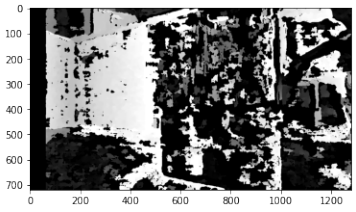


(b) Disparity from the model fine tuned with $M^l(u, v)$

Figure 6.8: Obstacles appear more defined.

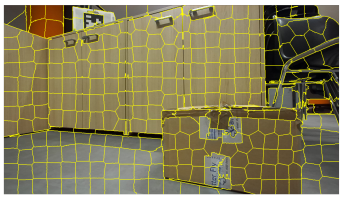
(a) Left image

(b) Right Image



(c) SGBM: $D^l(u, v)$

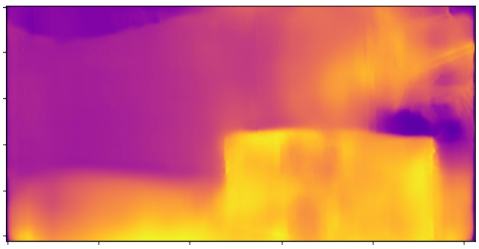
(d) Weighting map $M^l(u, v)$



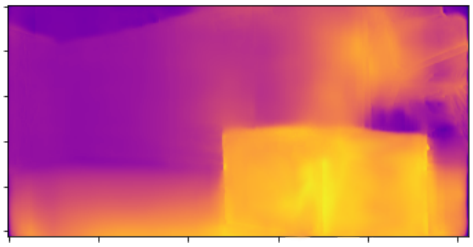
(e) SLIC: Superpixels

(f) Final weighting map

Figure 6.9: Another example of the weighting map computation.



(a) Disparity from the model fine tuned without $M^l(u, v)$



(b) Disparity from the model fine tuned with $M^l(u, v)$

6.2 Implementation details

Godard's implementation has been modified to suit the thesis purposes. Although the model is the same, training and test times are different since training has been conducted with a GeForce GTX 1180 GPU instead of Titan X GPU and testing has been performed on Jetson Xavier AGX.
On Titan X GPU it takes 25 hours on a dataset of 30 thousand images for 50 epochs, on Geforce GTX 1180 GPU on a dataset of 14 thousand images for 50 epochs it takes 7 hours. Inference on their GPU takes less than 35 ms, or more than 28 frames per second, for a 512×256 image, while on Jetson Xavier AGX is a bit slower but still good since it takes about 0.09 s, with 10 frames per second.

The network has been originally pre-trained with Cityscapes dataset, containing 22,973 images and trained on KITTI dataset, with 30k images. Main monodepth's settings:

- 50 epochs
- 8 as batch size
- Exponential linear units [22] instead of the commonly used rectified liner units (ReLU), for the non-linearities in the network.
- Adam Optimizer with $\beta 1 = 0.9$, $\beta 2 = 0.999$ and $\epsilon = 10^{-8}$
- Learning rate of $\lambda = 10^{-}4$ for the first 30 epochs, then it's been haven every 10 epochs.
- Data augmentation: horizontal flip of the input and color augmentation, both with with a 50% chance; random gamma, brightness and color shifts.
- Loss weights: $\alpha_{ap} = 1, \alpha_{lr} = 1, \alpha_{ds} = 0.1/r$ where r is the downscaling factor of the corresponding layer with respect to the resolution of the input image that is passed into the network.

- The possible output disparities are scaled by the original image width and constrained to be between 0 and dmax, using a scaled sigmoid non-linearity.

$$dmax = 0.3 * \textit{the image width at a given output scale}$$

The number 0.3 has been found by Godard considering that the maximum disparity in the KITTI images is around the 15% of the width, so he took twice this value as the maximum value the disparity could reach. A study was conducted in the evaluation chapter to find out how this parameter could affect the predicion and which could be the most suitable one for the custom dataset.

6.2.1 Post-Processing

Godard et al. design a post process algorithm in order to obtain higher quality disparity maps and reduce the effect of stereo disocclusion, which creates disparity ramps on both the left side of the image and of the occluders. It takes as input the two disparities maps outputted by the network and a per-pixel weight map. At test time, for an input image I they compute d_l and d'_l for I', the horizontally flipped input image. By flipping back d'_l, they obtain d''_l, which aligns to d_l but with disparity ramps located on the right of occluders and on the right side of the image. The weight map combines both disparities maps: the first 5% on the left is taken from d''_l and the last 5% from d_l. The central part of the final disparity map is the average.

$$w^l(i,j) = \begin{cases} 1, & \text{if } j \leq 0.1 \\ 0.5, & \text{if } j < 0.2 \\ 5(0.2 - i) + 0.5. & otherwise \end{cases}$$

where i,j are normalized pixel coordinates. The final output will be:

$$d = d_l w^l + d''_l w^{l'}$$

where $w^{l'}$ is w^l horizontally flipped.

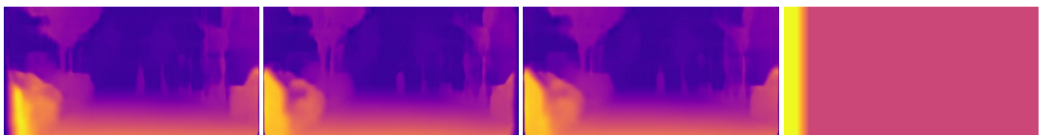


Figure 6.11: Example of a post-processed disparity map. From left to right: the disparities d_l, d''_l, d and the weight map w^l.

6.2.2 Experiments

In order to a obtain depth map it is necessary to convert disparity into depth, using the focal lenght and the baseline of the current camera. A limitation on this approach is that its performance is highly dependant by the camera used for training. As Godard said, there is no obvious way to convert the disparity of an image with focal and aspect ratio different from the last dataset the network was trained with. Computing depth, using the untouched monodepth model by Godard, with the test images from the custom dataset, gives high errors compared with the paper result. In this section I evaluate the official model in a custom environment using the common metrics. Subsequently I will introduce the fine tuning protocol and its consequently results improvement.

6.2.3 Evaluation Metrics

Evaluation is based on 7 metrics for test sets with ground truth depth maps. Since the ground truth is obtained from ZED 2 framework, it contains some non-valid pixels values (NaN or Inf), looking as "holes" in the depth map, which are not considered in the errors computation. Every metric is evaluate on post-processed disparity maps resized to the ground truth shape and converted to depth according to equation 6.1. In particular, for ZED2, we have $f = 528.675$ px and $b = 0.12$ m.

Given D and $\hat{D}$ as the ground truth and the resized-prediction in meters, N as the set of finite points in ground truth and n indicates any pixel in N:

- **Absolute Error**

$$\text{Abs} = \frac{1}{N} \sum_{n \in N} \left| D[n] - \hat{D[n]} \right|$$

- **Relative Absolute Error**

$$\text{Abs Rel} = \frac{1}{N} \sum_{n \in N} \frac{\left| D[n] - \hat{D[n]} \right|}{D[n]}$$

- **Squared Relative Error**

$$\text{Sqr rel} = \frac{1}{N} \sum_{n \in N} \frac{\| D[n] - \hat{D[n]} \|^2}{D[n]}$$

- **Root Mean Squared Error (linear)**

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{n \in N} \| D[n] - \hat{D[n]} \|^2}$$

- **Root Mean Squared Error (logaritmic)**

$$\log \text{RMSE} = \sqrt{\frac{1}{N}\sum_{n\in N} \|log(D[n]) - log(D[\hat{n}])\|^2}$$

The second group contains accuracy-based metrics with three different threshold:

$$\delta = max\left(\frac{D[n]}{D\hat{[}n]}, \frac{D\hat{[}n]}{D[n]}\right) \leq \text{Threshold}$$

With Threshold = 1.25^k k=1,2,3.

6.3 Fine tuning motivation and protocol

The official Monodepth model, pre-trained with CityScapes and then trained with KITTI 2015, highly degrades its performance when tested on different data-sets. This could be due to the different environments characterizing the two datasets, outdoor and indoor respectively. Moreover, another reason is the difference in the specific camera parameters, baseline and focal length.

In particular, it is worthy saying that the conversion from disparity to depth need the parameters of the camera o the test photos, and not with the ones of the training set. The following results in table 6.1 motivate the necessity of a fine tuning protocol, with a some application specific ground truth data.

		Lower is better					Higher is better		
Method	Test Dataset	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Monodepth	K	2.212	0.0996	0.9319	5.144	0.178	0.878	0.961	0.986
Monodepth	C	2.031	0.8920	2.9618	3.387	0.833	0.150	0.334	0.589

Table 6.1: Evaluation metrics of the official model on Kitti 2015 dataset (K) and on my custom dataset (C)

Other goals of fine tuning were to collect photos with objects at short distances, to teach the network to recognize small distances, since KITTI 2015 dataset has much wider ranges. Also, taking the photo from the robot's perspective introduces a new horizon line, which is one of the determining factors for a good prediction, as we will see in the section dedicated to the limitations of the method.

Fine Tuning protocol

The fine tuning protocol was conducted taking as initial weights the official ones - pretrained with CityScapes and trained with KITTI - and by varying the learning rate in different conditions, keeping fixed some parameters (table 6.2):

Fine tuning enviroments		
Encoder	dmax	Loss
VGG	0.3	Standard
VGG	0.2	Standard
VGG	0.2	Modified
ResNet50	0.3	Standard
ResNet50	0.2	Standard

Table 6.2: Fine tuning: keeping fixed these parameters, the learning rate was changed.

The favorite encoder is VGG, since it can run on Jetson Xavier AGX, but also the fine tuning study was conducted also with the ResNet50 encoder to verify the improvement in the results. Training the network from scratch with randomly initialized weights brings to poor results, since the dataset is not big enough to train well this deep CNN (table 6.3).

		Lower is better					Higher is better		
Method	Dataset	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Monodepth	C	1.278	0.679	3.689	2.640	0.664	0.235	0.621	0.826

Table 6.3: Evaluation metrics of Monodepth trained from scratch.

In the following, the fixed parameters will be explained and the results showed.

Fixed Parameters: Encoder

Godard proposes two encoders for its work, VGG with 31M trainable parameters and ResNet50 with 48M parameters. ResNet50 shows better results bu, due to its higher complexity, it is also more computationally heavy and the inference on Jetson Xavier AGX is too much slower.

Fixed Parameters: dmax

As already said, the number 0.3 has been found by Godard considering that the maximum disparity - on average - in the KITTI images is around the 15% of the width, so twice this value has been taken as the maximum value the disparity could reach. So we can write : $dmax = \frac{2*max\_disp}{width}$. The maximum value limits the minimum depth that could be computed and, following the same procedure by Godard, in the custom dataset the maximum value of the disparity is the 10% of the width, which leads to a disparity limiting factor of 0.2.

DMAX		
Dataset	KITTI	Custom
Width [px]	1242	1280
Max disparity [px]	186.3	122.0
Dmax	0.3	0.2

Table 6.4: Dmax parameter

Fixed Parameters: Modified Loss function

As explained before, a modified loss function was implemented. In order to test its effectiveness, fine tuning training were conducted both with the original and the modified one.

6.4 Results

The performance of the model has been evaluated, without any change, on the custom dataset test set, as we have already seen from table 6.1 the results are worst with respect to the ones obtained with KITTI 2015.

The following tables shows the results achieved on the same test set with fine tuning. While varying the learning rate as indicated, the following parameters were kept constant: encoder= VGG, input size= 256x512, number of epochs= 50. In evey situation, the best results, highlighted in green, are obtained with learning rate lr=$5.00e^{-6}$, since the performance degrades while decreasing it more:

1. Table 6.5: **dmax= 0.3**;

	Lower is better					Higher is better		
Learning rate	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
$5.0e^{-5}$	0.9101	0.2818	0.8599	2.292	0.615	0.598	0.710	0.849
$2.5e^{-5}$	0.8678	0.2788	0.7663	2.124	0.618	0.617	0.744	0.856
$1.0e^{-5}$	0.8664	0.2712	0.7150	2.010	0.634	0.593	0.779	0.852
$7.0e^{-6}$	0.8700	0.2667	0.7144	2.047	0.648	0.593	0.778	0.852
$5.0e^{-6}$	0.8686	0.2651	0.6960	2.039	0.648	0.584	0.777	0.856
$4.0e^{-6}$	0.8754	0.2662	0.6989	2.066	0.668	0.589	0.783	0.855
$1.0e^{-6}$	0.8907	0.2786	0.6954	2.058	0.642	0.570	0.771	0.853

Table 6.5: dmax=0.3, VGG, 256x512.

2. Table 6.6: **dmax=0.2**;

	Lower is better					Higher is better		
Learning rate	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
$5.00e^{-5}$	0.9209	0.3315	1.4195	2.214	0.617	0.606	0.787	0.848
$1.00e^{-5}$	0.8595	0.2755	0.7302	2.017	0.637	0.569	0.786	0.861
$5.00e^{-6}$	0.8541	0.2668	0.6852	2.022	0.630	0.584	0.792	0.872
$3.00e^{-6}$	0.88407	0.2759	0.6961	2.047	0.658	0.558	0.769	0.850

Table 6.6: dmax=0.2, VGG, 256x512.

3. Table 6.7: fine tuning using the **modified loss function** with weighting map **M** (section 6.1.3). The best model, highlighted in green, is identified by the lowest absolute error.

		Lower is better					Higher is better		
Learning rate	dmax	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
$5.00e^{-5}$	0.3	0.8665	0.3103	1.2470	2.100	0.606	0.608	0.789	0.852
$5.00e^{-6}$	**0.2**	0.8494	0.2600	0.5992	1.997	0.637	0.587	0.787	0.860
$5.00e^{-6}$	0.3	0.8602	0.2542	0.5963	2.006	0.651	0.596	0.782	0.852
$1.00e^{-6}$	0.3	0.8936	0.2724	0.6156	2.035	0.651	0.564	0.767	0.848

Table 6.7: Loss with weighting map **M**, VGG, 256x512.

4. Table 6.8: for the sake of completeness some fine tuning training were conducted also using **ResNet50** encoder, which has 48M parameters. The best result, highlighted in green, is achieved with an higher learning rate with respect to the model with VGG as encoder. Due to its higher complexity, it achieves *slightly better* results, but its perfomances on the target hardware were too much slow - as we will see in the next section - so, no further analysis were conducted.

		Lower is better					Higher is better		
Learning rate	dmax	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
$7e^{-5}$	0.3	0.8458	0.2875	0.7345	2.546	0.632	0.588	0.679	0.847
$9e^{-6}$	0.2	0.8301	0.2502	0.6432	1.9873	0.678	0.643	0.798	0.847
$9e^{-6}$	0.3	0.8368	0.2625	0.6692	2.014	0.659	0.609	0.779	0.847
$5e^{-6}$	0.3	0.8545	0.2655	0.6986	2.050	0.666	0.603	0.779	0.849
$4e^{-6}$	0.3	0.8522	0.2708	0.6988	2.042	0.662	0.595	0.774	0.848

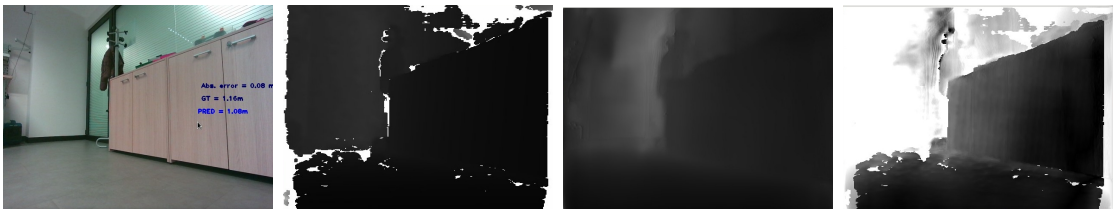
Table 6.8: ResNet50, 256x512.

6.4.1 Final Model

The model highlighted in green from table 6.7 was chosen as the best one. The network, after the fine tuning, learned to recognise nearest depth values from a lower horizon line in an indoor environment. In the figures below we can see some qualitative results of the model, these frames are taken from a real-time inference test. From left to right: the input image, the ground truth, the depth map and the absolute error map. In particular, this last image tells us which are the most reliable areas of our depth map (darker zones) and the most incorrect ones (lighter zones), where distance is quite always underestimated. For near object we have satisfactory and reliable results, making the depth map suitable for robotic tasks. In the RGB image there are shown predicted and actual depth values and absolute error, relative to where the arrow is located.

(a) (b) (c) (d)

Figure 6.12: (a) RGB Image, (b) Depth GT, (c) Depth Prediction, (d) Absolute Error Map. GT=2.08 m, Pred=1.89 m, AbsErr=0.15 m.



(a) (b) (c) (d)

Figure 6.13: (a) RGB Image, (b) Depth GT, (c) Depth Prediction, (d) Absolute Error Map. GT=1.16 m, Pred=1.08 m, AbsErr=0.08 m.
We can see the gradient of the absolute error in the cabinet, increasing along with distance.

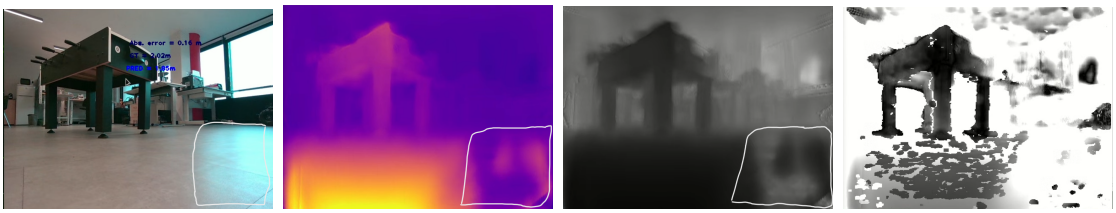
Limitations

The prediction, in general, has an error between 0.01 and 0.3 under 3 meters with exceptions depending on the texture and the light conditions. There are indeed

(a) (b) (c) (d)

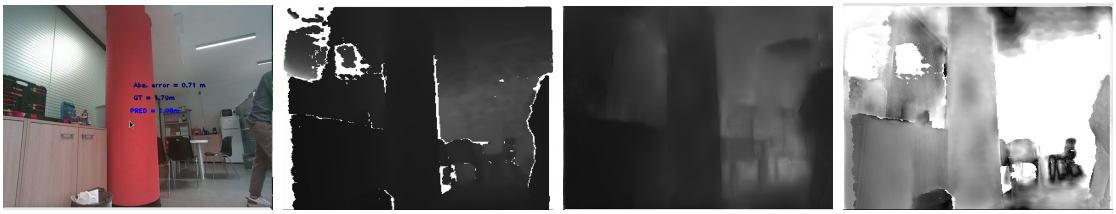
Figure 6.14: (a) RGB Image, (b) Depth GT, (c) Depth Prediction, (d) Absolute Error Map. GT=1.76 m, Pred=1.68 m, AbsErr=0.09 m.
Camera height 70 cm, zero roll and pitch angles.

some limitations due to the fact that the network relies on the RGB image and on the photometric reconstruction term, which means that non lambertian surfaces may lead to errors. In *back light conditions* there could be some artifacts, also *texture-less regions* like walls are difficult to estimate or *transparent surfaces* causes some inaccuracies.
According to [23] Monodepth relies on the vertical coordinate of the obstacles to estimate the depth. The vertical position of obstacles in an image depends on the relative pose of the camera. In fact, also in this work it has been noticed that by varying the camera *pitch and roll angles* and its height above the ground, the horizon line changes and a noticeable drop in prediction is observed. *Edges of an object and its contrast with the environment* seem to matter a lot for correct distance estimation, [23] demonstrates that the neural network 'fills in' an object, if its bottom and side edges are left in the image.



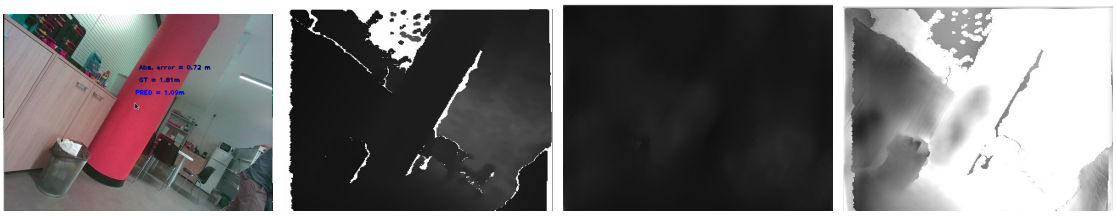
(a) (b) (c) (d)

Figure 6.15: (a) RGB Image, (b) Disparity Predicion, (c) Depth Prediction, (d) Absolute Error Map. GT=2.02 m, Pred=1.85 m, AbsErr=0.16 m.
Backlighting creates a reflection on the floor that cannot be handle by the network.



(a) (b) (c) (d)

Figure 6.16: (a) RGB Image, (b) Depth GT, (c) Depth Prediction, (d) Absolute Error Map. GT=1.79 m, Pred=1.08 m, AbsErr=0.71 m.
Pitch angle different from zero, horizon line is lower than training and test photos.



(a) (b) (c) (d)

Figure 6.17: (a) RGB Image, (b) Depth GT, (c) Depth Prediction, (d) Absolute Error Map. GT=1.79 m, Pred=1.08 m, AbsErr=0.71 m.
Roll angle different from zero, horizon line is inclined.

6.4.2 Monodepth Vs FastDepth

Looking at the metrics of both network, FastDepth seems to be better when considering the whole depth map. However, during the inference, Monodepth produces more accurate results for obastacle avoidance, in order to demonstrate it is sufficient to consider the same test set for depth values below 3 m, in this case Monodepth far outperforms FastDepth, which produces unreliable results for short distances (table 6.9).

		Lower is better					Higher is better		
Method	Dataset	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Monodepth	C	0.347	0.222	0.159	0.508	0.378	0.623	0.820	0.898
FastDepth	C	0.467	0.361	0.399	0.510	0.403	0.465	0.789	0.805

Table 6.9: Comparison between fine-tuned models of Monodepth and FastDepth, considering depth values below 3 meters.

Chapter 7

Mapping function from disparity to depth

The main limitation of Monodepth approach is that its result highly depends on the camera used for the training set. Monodepth needs to have test images with a focal length identical to the dataset it was trained with. Even if it is true that pre-training with another dataset - captured with a different camera - improves the prediction, the network will learn to recognize the last focal length it has seen. It is straightforward that The training in fact must be done on single datasets at a time.

In the case we want to use a different camera for inference, there isn't an obvious way to convert the disparity. Some experiments were conducted testing the model with different cameras, that differ from ZED2 in baseline and focal length. The depth, in this case, cannot be directly calculated by a formula. Hence we need to find a relation from disparity to true depth. This experiment involves the depth camera RealSense D455 to grab RGB frames with depth ground truth. For each frame going into the model, a disparity image is predicted. he result is an improvement in the error metrics, compared to the usage of the formula.

RealSense D455

The RealSense D455 differs slightly from the D435i stated in the dataset chapter, but the operating principle is the same because they both belong to RealSense's D400 series and combine stereo matching with an infrared projector. This camera, in particular, has a wider baseline and a different focal length. For this experiment, the only parameter we need to convert the disparity is the focal length; since we are just utilising the RGB frame to test the model, the baseline is superfluous, and we will keep the ZED2 one, which was used for training.

Depth Cameras				
Camera	Baseline [mm]	Focal length [px]	Min Depth [m]	Max Depth [m]
RealSense D435i	50	322	0.2	10
RealSense D455	95	383.51	0.4	10

Table 7.1: Differences between RealSense cameras.

The first step is to align the depth ground truth and the disparity map predicted by the model. These data are well known to be inversely proportional, so the main goal is to find the conversion factor according to:

$$z \propto \frac{1}{d} = \frac{\alpha}{d} \tag{7.1}$$

The disparity-to-depth mapping can be simply fitted as $z = \frac{32.61}{d}$ (figure 7.1), with a score $R^2 = 0.7898$ calculated as the mean of all the residuals on the test set.

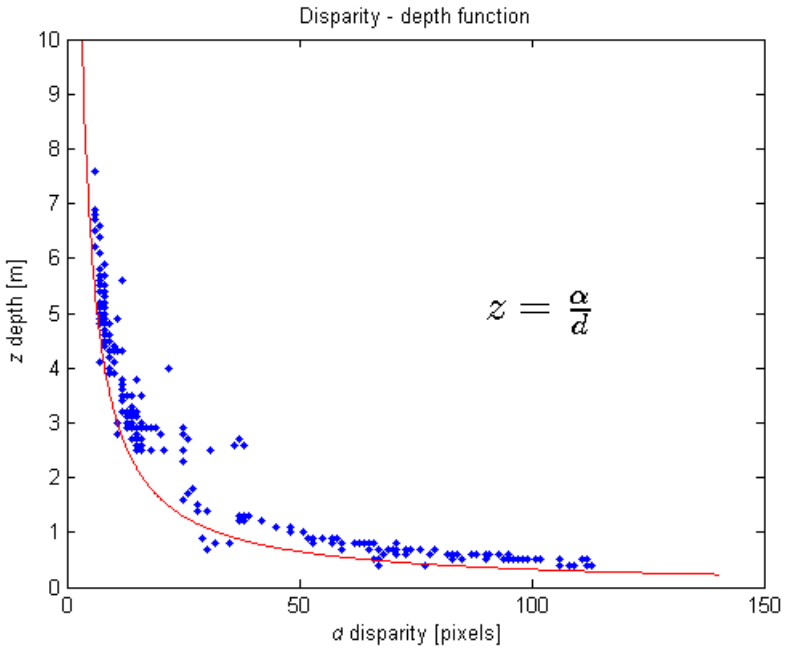


Figure 7.1: The mapping function from disparity to depth prediction when using the RealSense D455 webcam.

Thus, experimental results show that the proposed technique is applicable to cameras with different intrinsic parameters if the process of RGB-depth ground truth calibration is performed first. The error metrics using the conversion factor are better than the ones using the conventional formula, as we can see in table 7.2.

	Lower is better					Higher is better		
Method	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Conventional Formula	1.7544	0.3524	1.1771	2.493	0.436	0.376	0.810	0.953
Conversion Factor	1.3087	0.2253	1.1134	2.613	0.396	0.568	0.919	0.954

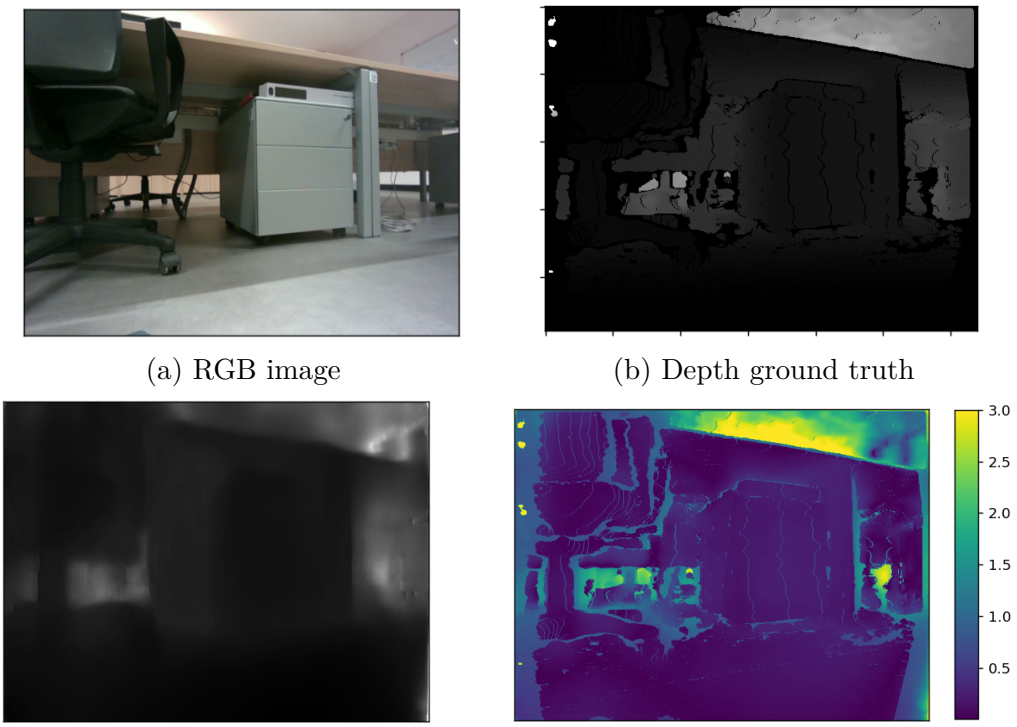
Table 7.2: Evaluation metrics of the disparity-of-depth conversion using the two proposed methods.

Even if the absolute error is high, the predicted maps can be used for obstacle avoidance since the error get higher increasing distance. Metrics in table 7.3 were calculated considering only the points were the ground truth was less than 3 meters.

	Lower is better					Higher is better		
Method	Abs	Abs Rel	Sqr rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Conventional Formula	0.4505	0.4204	0.1913	0.488	0.366	0.184	0.787	0.990
Conversion Factor	0.2333	0.2348	0.0605	0.260	0.233	0.516	0.982	0.999

Table 7.3: Evaluation metrics of the disparity-to-depth conversion using the two proposed methods, considering only GT depth less than 3 meters.

Moreover, figures below show how the absolute error is lower for nearest object, while the prediction becomes worst for further distances. In the absolute error map, saturated at maximum level of 3, lighter borders many times coincide with NaN values in the ground truth, mainly due to occlusion. In particular, windows in figure 7.3 are tricky to be predicted also for the depth camera.



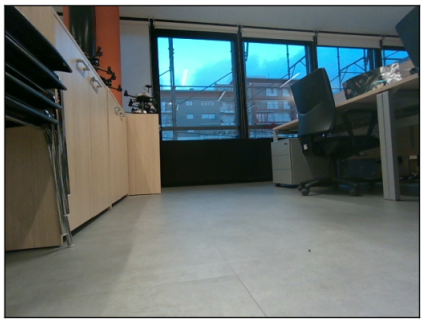
(a) RGB image

(b) Depth ground truth

(c) Predicted depth

(d) Absolute error map

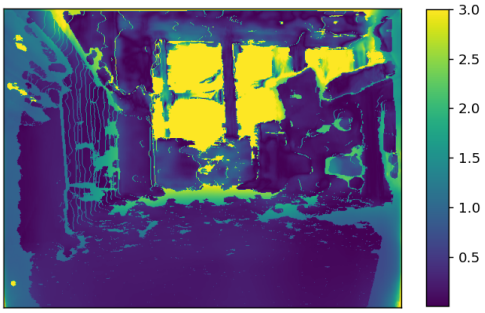
Figure 7.2: Qualitative example.



(a) RGB image

(b) Depth ground truth

(c) Predicted depth



(d) Absolute error map

Figure 7.3: Qualitative example.

Chapter 8

Navigation Demo

Autonomous navigation can be supported by a depth map, from which obstacles and free space can be detected, exploiting different approaches. In this experiment, an odometry-based navigation algorithm guides a TurtleBot3 to a fixed goal while a simple and heuristic control strategy exploits the depth map to detect free space region. The combination of these two algorithms allows to reach the goal while avoiding obstacles.

8.1 Jetson Xavier AGX

Jetson Xavier AGX is a is an embedded system-on-module (SoM) from the NVIDIA AGX Systems family [1] and it includes an integrated Volta GPU with 512 CUDA cores and 64 Tensor Cores, octal-core ARMv8 CPU, 32GB flash storage and 16GB RAM.
Useful for deploying computer vision and deep learning to the edge since it provides provides 32 TeraOPS (Tera Operations per Seconds [2]) making it ideal for high-performance compute and AI in embedded and edge systems.
Monodepth inference on Jetson Xavier AGX takes about 0.09 s to make a prediction, which allows a quite smooth visualization of the depth map and a good response to moving objects.
A study to compare both encoders' time performance and power consumption was conducted and demonstrated that ResNet50 improvements in depth prediction didn't for the compensate the slow down in inference and the higher power consumptions. During inference frames are grabbed at size 640x480, a depth map

[1] https://www.nvidia.com/en-us/deep-learning-ai/products/agx-systems/

[2] It is a measure of the maximum achievable throughput, TOPS = (number of MAC units) x (frequency of MAC operations) x 2.

is published every 0.1 s, the framerate is about 7 to 9 frames per second (FPS) which is acceptable to guide a mobile robot in general applications.

JETSON XAVIER AGX		
Encoder	Inference Time [s]	Power [W]
VGG	0.09	24
ResNet50	0.15	27

Table 8.1: Encoders comparison on Jetson.

8.2 TurtleBot3

TurtleBot3 by ROBOTIS [3] can be regarded as a standard for robotic platforms.It is a differential wheeled mobile robot which runs the open source Robot Operating System (ROS) and includes a Single Board Computer (SBC), an embedded controller OpenCR, and Dynamixel actuators.
In this case, the robot, a TurtleBot3 Waffle (figure 8.1), was equipped with Jetson Xavier AGX as processor and RealSense D455 as the camera to get the RGB input image. In order to retrieve depth, the mapping function from disparity to depth has been exploited.

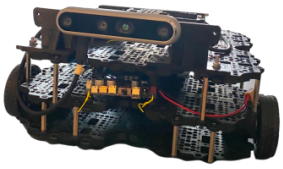


Figure 8.1: TurtleBot3 Waffle equipped with RealSense D455

8.3 ROS2 Nodes

The experiment was supported by two main algorithms, the odometry-based navigation system and the controller that continuously examines the depth.
The **navigation** was based on dead reckoning. At bring up, two reference frames are initialized, a world-fixed frame (e.g. x=0,y=0,z=0) called "odom", and a coordinate frame called "base link" rigidly attached to the mobile robot base. The robot's position and orientation are estimated over time using data from sensors, i.e. wheels encoders. The robot orientation is adjusted to reach the established goal and the motion is stopped when the distance, calculated at each timestep, is below a certain threshold. The **depth controller** is a simple algorithm that exploits the depth map to find free-space areas in the *visual field of the camera*.

The depth map is analyzed at each prediction with this steps:

[3] `https://emanual.robotis.com/docs/en/platform/turtlebot3`

1. Assuming a fixed the camera height, pixels always belonging to the ground, are cropped out.

2. Each pixel less than 0.5 m is penalized with negative weight.

3. The map is divided into 7 columns, depth pixels are summed up obtaining a row vector of nine components, one for each column's weight: the larger the weight, the more free area the column has and and vice versa, the lower the weight, the greater the chance of an obstacle.

4. If one of the columns exceeds a specific - empirically determined - threshold, it indicates that there is an obstruction within - more or less, depending on the dimension of the obstacle - 50 centimetres.

5. If there are at least two columns satisfying this requirement, the depth controller is triggered and sends an angular velocity command to the TurtleBot. The waypoint is identified in the column corresponding to the biggest weight. The entity of the rotation is proportional to the distance from the waypoint to the image's centre, which is assumed to be the robot's centre (figure 8.2):

$$yaw\ rate = K_p * (centre - waypoint)^4 \tag{8.1}$$

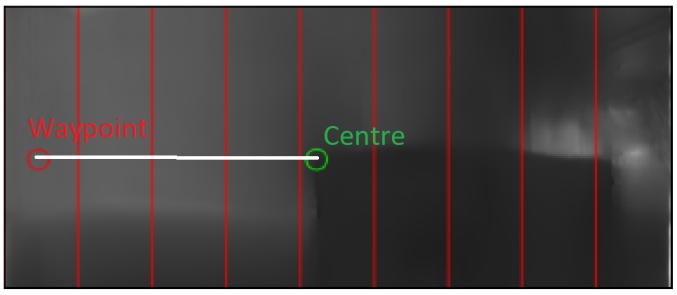


Figure 8.2: Chosen waypoint and its distance from the image's centre.

6. The command is kept constant for an appropriate amount of time, enough to overcome the obstacle.

7. Finally, if the field of view becomes unobstructed again, the navigation obstacle returns to action in order to approach the destination.

[4]Positive rotations are counterclockwise.

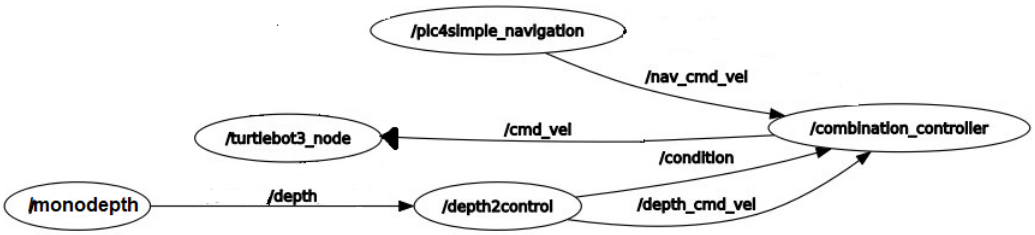
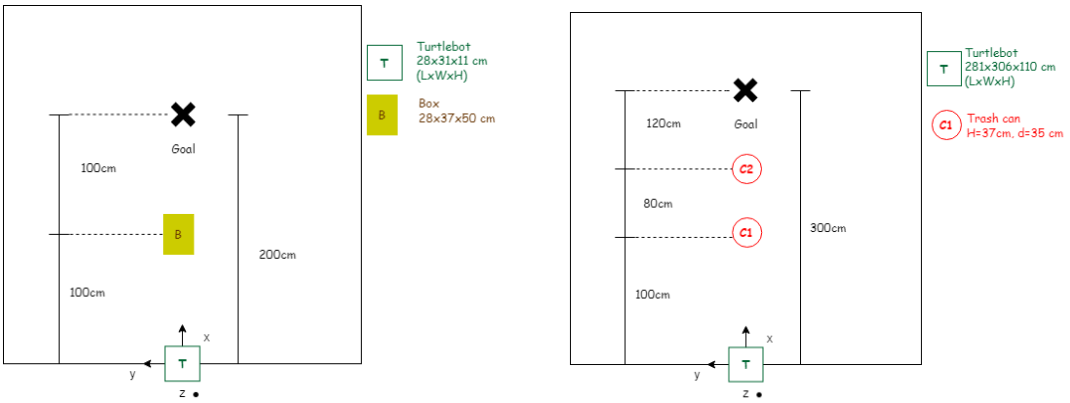


Figure 8.3: A simplified scheme of the ROS2 nodes: the depth map is published every 0.1 s and the presence of obstacles in the surroundings is evaluated, meanwhile the navigation algorithm guides the robot to the goal. If the *condition* is verified, the depth algorithm takes action.

8.4 Experiments

The experiments were conducted in a spacious room with controlled, artificial light. The set up was always the same: the robot would start one meter from the obstacle and the goal was one meter later. In one experiment, two obstacles were also used in sequence as shown in the figure 8.4.
Experiments were performed with three different types of obstacle, a black basket, a yellow box, and a beige box. The robot is able to detect the approaching obstacle and activate the depth controller, but what sometimes happen is that the obstacle, even if it has not been surpassed completely, goes out of the camera's field of view so that the navigation algorithm takes control again, bringing the robot back in the previous direction and resulting in a collision with the object.
Nevertheless, the goal is to demonstrate that the depth map can be effectively used for obstacle avoidance since it provides a reliable prediction for nearby objects, even if the inference is made with a camera different from the one used for training.



(a) One obstacle set up

(b) Two obstacles set up

Figure 8.4: Experiments set up.

The first restriction of this strategy is that the ground should be segmented out by the picture since it could be perceived as a close "object" even though it is not an obstacle. UV disparity is a possible method for detecting obstacles and ground boundaries, as suggested by various states of the art methods, such as [24], [14], [25]. Secondly, if the dimensions of the obstacles were estimated, more appropriate times and maneuvers could be chosen each time. In fact, by using fixed times, sometimes it is not possible to overcome the obstacle or sometimes the robot turns too much compared to the real need, with small obstacles. However, a sophisticated navigation algorithm is beyond the intent of this thesis but it would be a starting point for future works.

Chapter 9

Conclusions

The goal of this thesis was to investigate depth estimation from a single image with Deep Learning. In particular, the work aims at studying a cost-effective deep neural network that provides sufficiently accurate depth predictions to enable its usage in robotic tasks such as autonomous navigation.

Therefore, a comparison of state-of-the-art solutions for monocular depth estimation with Deep Neural Networks has been firstly carried out to identify the model which offers the best combination of accurate predictions and computational cost. Among the selected models, FastDepth was mainly chosen for its low computational cost being also able to run on a CPU, at the cost of much lower accuracy. Differently, Monodepth achieves satisfying results at real-time speed on the Jetson Xavier AGX. The output of this algorithm is the disparity map, which can be converted into a depth map knowing the baseline and focal length of the specific camera, applying the triangulation formula.
Consequently, this second neural network has been selected to conduct a refinement process and further investigations on depth estimation in indoor environments.
A fine-tuning training process has been used to let the network adapt to indoor environments scenes, from the low-height perspective of a mobile robot.
In addition, the accuracy of depth estimation results for obstacles is improved through a modified loss optimized to consider the information of nearby objects.
At this point, the network predictions were quantitatively evaluated on a test set of photos, with depth ground truth, using common error metrics such as RMSE, log RMSE, Abs Rel, Abs Ass, Sq Rel, and Accuracy.
Overall, the re-trained Monodepth is able to estimate the depth map with an absolute error between 0.01 and 0.3 m, evaluated over a maximum distance of 3 meters. The most common error conditions are due to poorly textured surfaces and backlighting.
The main limitation of Monodepth is that depth prediction is not reliable when inference is conducted with cameras different from the one used to train the network;

in our case, it was ZED2 from StereoLab.
In order to overcome this restriction, an experiment was conducted to find a numerical conversion method. By aligning the network's disparity prediction and depth ground truth grabbed with RealSense D455 depth camera, as expected an inverse proportionality relationship was found and, through polynomial regression, a conversion factor was estimated.
The experiment resulted in improved evaluation metrics and showed that generalization to other cameras is possible if this calibration process is performed first.

The actual effectiveness of the predicted depth map has been demonstrated thanks to a simple navigation demo with a mobile robot. The basic idea was to use a dead reckoning navigation algorithm to reach a fixed goal while avoiding objects in the path.
The obstacle avoidance system exploits the estimated depth map to identify nearby "surfaces" against which the robot could collide and therefore turns towards free-space regions. Nonetheless, the scope of the demo is to validate the quality of the predicted depth images, hence the basic control algorithm proposed only provides a naïve obstacle avoidance without precisely planning a trajectory towards the goal. This is the reason why sometimes the robot moves with inaccurate control actions turning too much or touching the obstacle. However, the distance from obstacles is successfully detected by the deep neural network, reaching a sufficiently high precision to act accordingly.

Future work could be devoted to the following improvements and developments:

- Lighten the network e.g. with filter pruning and perform optimization for embedded devices.
- Improve depth map prediction, maybe adding supervisory elements in the training such as sparse ground truth.
- Create an obstacle avoidance algorithm that segments out the floor and computes the manoeuvre to avoid the obstacle based on the size and distance of the obstacle.

Bibliography

[1] Christopher Town. *Computer Vision* (cit. on p. 3).

[2] Barbara Sweet and Mary Kaiser. "Depth perception, cueing, and control". In: *AIAA Modeling and Simulation Technologies Conference*. 2011, p. 6424 (cit. on p. 4).

[3] Marko Teittinen. *Depth Cues in the Human Visual System* (cit. on p. 4).

[4] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004 (cit. on p. 7).

[5] Sanja Fidler. *Intro to Image Understanding* (cit. on p. 8).

[6] Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. "Unsupervised monocular depth estimation with left-right consistency". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 270–279 (cit. on pp. 10, 33, 43).

[7] Diana Wofk, Fangchang Ma, Tien-Ju Yang, Sertac Karaman, and Vivienne Sze. "Fastdepth: Fast monocular depth estimation on embedded systems". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 6101–6108 (cit. on pp. 10, 11).

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on p. 12).

[9] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for autonomous driving? the kitti vision benchmark suite". In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, pp. 3354–3361 (cit. on p. 31).

[10] Moritz Menze and Andreas Geiger. "Object scene flow for autonomous vehicles". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3061–3070 (cit. on p. 31).

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520 (cit. on p. 38).

[12] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. "Netadapt: Platform-aware neural network adaptation for mobile applications". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 285–300 (cit. on p. 40).

[13] Róbert-Adrian Rill and Kinga Bettina Faragó. "Collision Avoidance Using Deep Learning-Based Monocular Vision". In: *SN Computer Science* 2.5 (2021), pp. 1–10 (cit. on p. 43).

[14] Hsiang-Chieh Chen. "Monocular vision-based obstacle detection and avoidance for a multicopter". In: *IEEE Access* 7 (2019), pp. 167869–167883 (cit. on pp. 43, 49, 71).

[15] Tom van Dijk. "Self-Supervised Learning for Visual Obstacle Avoidance". In: (2020) (cit. on p. 43).

[16] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 44).

[17] 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*. 2016, pp. 770–778 (cit. on p. 44).

[18] *Bilinear Interpolation*. `https://en.wikipedia.org/wiki/Bilinear_interpolation` (cit. on p. 47).

[19] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612 (cit. on p. 47).

[20] Heiko Hirschmuller. "Stereo processing by semiglobal matching and mutual information". In: *IEEE Transactions on pattern analysis and machine intelligence* 30.2 (2007), pp. 328–341 (cit. on p. 49).

[21] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. "SLIC superpixels compared to state-of-the-art superpixel methods". In: *IEEE transactions on pattern analysis and machine intelligence* 34.11 (2012), pp. 2274–2282 (cit. on p. 49).

[22] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015) (cit. on p. 52).

[23] Tom van Dijk and Guido de Croon. "How do neural networks see depth in single images?" In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 2183–2191 (cit. on p. 60).

[24] Hsieh-Chang Huang, Ching-Tang Hsieh, and Cheng-Hsiang Yeh. "An indoor obstacle detection system using depth information and region growth". In: *Sensors* 15.10 (2015), pp. 27116–27141 (cit. on p. 71).

[25] Jiahao Lin, Hai Zhu, and Javier Alonso-Mora. "Robust vision-based obstacle avoidance for micro aerial vehicles in dynamic environments". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 2682–2688 (cit. on p. 71).