POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Depth Map Super-Resolution Fusing Color Information

Tesi di Laurea Magistrale in Computer Science Engineering - Ingegneria Informatica

Author: **Davide Palesano**

Student ID: 945369
Advisor: Prof. Marco Marcon
Co-advisors: Marco Paracchini
Academic Year: 2022-23

Abstract

Nowadays depth information plays a pivotal role in many computer vision applications. For instance, self driving cars exploits depth information to understand the surrounding environment so they can drive safely and avoid accidents. This kind of applications usually relies on technologies, such as LiDAR, which using active illumination methods measures the time it takes for each laser pulse to bounce back from an obstacle to crate a 3D model of the environment. On the other hand, high resolution depth cameras are very expensive and not suitable for many applications. The solution proposed in this work is to use low resolution depth cameras and then enhance the depth estimation by performing upsample operations. Image upsample is a procedure which is used to reconstruct an High Resolution image starting from a Low Resolution image. In our scenario we deal with depthmap images, where each pixel has a distance information rather than a color one. There already exist algorithms performing upsample task, such as bilinear and bicubic interpolation. However in real time scenarios like self driving cars and autonomous mobile robots, where machine needs depth information for avoiding obstacles, recognizing and tracking objects, the reconstruction quality of the mentioned algorithms is negatively affected by the presence of artifacts and noise. The purpose of this research is to develop methods to reduce the artifacts and enhance the quality of high resolution depthmap, exploiting Deep Learning techniques. We rely on Convolution Neural Network to develop a model able to perform Super-Resolution task. Whereas standard depth map upsample algorithms rely only on low resolution depthmap, our model fuse high resolution information coming from a RGB camera with low resolution depthmap in order to create a high resolution depthmap. In this work two different scaling factor networks are proposed: 8x and 4x. The proposed method was trained and tested on two different dataset: one synthetic, composed of videogame frames that simulate outdoor scene, and another real, whose images of indoor scenes are taken by a Kinect device. After the training process, we measured the effectiveness in terms of quality enhancement using RMSE, SSIM and PSNR metrics. Our architecture shows better performance if compared to classical algorithms and also with respect to other state of the art methods.

Keywords: Depthmap, Upsample, Convolution Neural Network, Super-Resolution

Abstract in lingua italiana

Al giorno d'oggi, l'informazione di profondità gioca un ruolo importante in molte applicazioni relative all'analisi e al processamento di immagini. Un esempio sono le macchine a guida autonoma, che sfruttano questa informazione per mappare l'ambiente circostante. In questo ambiente si fa uso di tecnologie quali il LiDAR, il quale emana degli impulsi ed è in grado di misurarne il tempo di ritorno, dopo aver colpito un oggetto. Le camere di profondità ad alta risoluzione però sono molto costose e non si ha la disponibilità per usarle in ogni contesto. La soluzione qui proposta è quella di usare camere di prfondità a bassa risoluzione e aumentare quest'ultima tramite un processo di sovracampionamento. Nel nostro caso adoperiamo questo processo, ovvero la trasfomrazione di immagini da bassa ad alta risoluzione, sulle mappe di profondità, le quali non sono altro che delle fotografie che contengono informazioni sulla profondità degli oggetti in scena piuttosto che informazioni sul colore. Sono giá presenti dei metodi di sovracampionamento tra i quali l'interpolazione bilineare e bicubica. Tuttavia, se consideriamo scenari come macchine a guida autonoma in cui l'informazione sulla profondità è necessaria per evitare collisioni con oggetti circostanzi o ancora più importante con pedoni, la qualità del risultato dei metodi appena citati risulta insufficiente. Lo scopo di questo lavoro è di sviluppare un metodo per aumentare la risoluzione delle mappe di profondità migliorandone la qualità e riducendone possibili artefatti. Per fare ciò sfruttiamo tecniche di Deep Learning ed in particolare delle Reti Neurali Convoluzionali per sviluppare un modello in grado di eseguire la Super-Resolution. I classici algoritmi di sovracampionamento delle mappe diprofondità si basano solo sulla mappa di profodnità a bassa risoluzione, il nostro modello utilizza anche le informazioni provenienti dalla fotografia RGB ad alta risoluzione della stessa scena. Per il nostro progetto abbiamo testato due differenti rapporti d'incremento, uno di 4 e l'altro di 8. Abbiamo inoltre allenato la nostre rete con due differenti dataset, uno contenenti delle scene di un videogioco,che simula immagini di un ambiente all'aperto, e l'altro che contiene immagini reali di scene di interni, realizzate con un dispositivo Kinect. Dopo aver allenato la rete, abbiamo misurato la qualità del nostro risultato utilizzando i tre indici RMSE, SSIM e PSNR. Il nostro lavoro mostra migliori risultati se confrontato con i classici algoritmi di sovracampionamento e anche rispetto ad altri lavori precedenti.

Parole Chiave: Mappa di Profonditá, Sovracampionamento, Reti Neurali Convoluzionali, Super-Resolution

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 State of the Art **5**

1.1 Depthmap Acquisition 5

1.1.1 Monocular vision 5

1.1.2 Binocular vision 6

1.1.3 Depthmap Acquisition 7

1.1.4 Convolution Neural Network 15

1.1.5 State of The Art works 24

2 Methods **29**

2.0.1 Architecture 29

2.0.2 Dataset 42

2.0.3 Training 48

2.0.4 Parameters 58

2.0.5 Testing 61

3 Results **63**

3.0.1 Quantitative Evaluation 64

3.0.2 Qualitative Evaluation 67

4 Conclusions **73**

4.0.1 Further developments 74

Bibliography **75**

List of Figures **81**

List of Tables **83**

Acknowledgements **85**

Introduction

In the last decades the importance of depth information has grown and nowadays it has a pivotal role in many different applications of computer vision. Self driving cars need depth information to generate 3D models of their environment so they can avoid obstacles and support the recognition of specific objects and the ability of tracking them. By understanding the environment around them, cars can drive safely and thanks to their lower reaction time than a human being, they can avoid accidents a man could be responsible of. Another important field in which depth information is useful is industry 4.0 and autonomous factories. Autonomous Mobile Robots (AMRs) are used to pick, transport, and sort items within manufacturing spaces, warehouses, retail stores, and distribution facilities without being overseen directly by an operator. Regardless of their usage, pick and place robot, patrol robot, or agricultural robot, almost all AMRs need to measure depth. Depth information has been thought also as quality control assistant when it comes to products packaging for shipment. Recently also editing photo tools take advantage of depth information by letting users edit the focus point of their photo after it was taken, applying a shallow depth of field effect that helps their subject pop up; this effect keeps the subject in focus while blurring things around them. To apply this bokeh effect, we need two things: an estimation of the depth, or relative distance from the camera, of every pixel in the image, and a depth at which to focus. There are to ways digital devices deal with depth information: depth map and 3D point cloud. The former is a two dimensional matrix with only one channel of information related to the distance of an interest object or a scene from a viewpoint; it stores a distance value for each pixel in the image. Depth maps cannot be displayed directly as they are encoded on 32 bits. To display the depth map, a monochrome (grayscale) 8-bit representation is necessary with values between [0, 255], where 255 represents the closest possible depth value and 0 the most distant possible depth value. Point cloud models a collection of individual points plotted in 3D space. Each point contains several measurements, including its coordinates along the X, Y, and Z-axes, and sometimes additional data such as a color value, which is stored in RGB format, and luminance value, which determines how bright the point is. The oldest method of 3D depth sensing is the passive stereo cameras which operate

by calculating the disparity of pixels from two sensors working in sync. However, passive technology had the disadvantage that these cameras cannot be used in the dark. Also, the depth quality depends on the texture of the objects in the scene. To overcome this, active stereo vision technology is used. This technology uses an IR pattern projector to illuminate the scene, which improves the performance in low light and works well on scenes with objects of less complex texture. However, stereo cameras usually are not able to offer large depth ranges (in the range of 10 meters). Also, the data from stereo cameras has to be processed further to calculate depth. Further more, stereo cameras often do not meet the required accuracy levels, especially when the texture of the target object is uneven. These problems could be overcame using different technologies, such as LIDAR (Light Detection and Ranging). LiDAR uses the light detection technique to calculate depth. It measures the time it takes for each laser pulse to bounce back from an obstacle. This pulsed laser measurement is used to create 3D models (also known as a point cloud) and maps of objects and environments. LiDAR is also called 3D laser scanning, which works similarly to RADAR, but instead of radio waves, it emits rapid laser signals up to 160,000 pulses per second towards the object. Autonomous equipment mentioned above depend on 3D depth-sensing cameras for obstacle detection, localization, navigation, and picking or placing objects. Obstacle detection requires low latency to react fast and, in many cases, a wider FoV. For vehicles moving at high speed, a Lidar sensor is commonly used. It has a rotating 360-degree laser beam function which in turn means they have a precise and accurate view to avoid obstacles and collision. Since LiDAR generates millions of data points in real-time, it easily creates a precise map of its ever-changing surroundings for the safe navigation of autonomous vehicles. Also, the distance accuracy of LiDAR allows the vehicle's system to identify objects in a wide variety of weather and lighting conditions. Many high end depth estimation systems, such the ones described above could be very expensive and depending of the application their usage could not be available. The main contribution of this thesis is to investigate the feasibility of substituting more sophisticated depth sensors with a combination of cheaper low resolution depth sensors, high resolution traditional cameras and Machine Learning algorithms. Since Convolution Neural Networks (CNN) are widely used for upsampling task, which goes under the name of Super Resolution, they could be used to enhance the low resolution depth map obtained with cheaper depth sensors. Moreover we propose to use as an additional input of the network, the high resolution RGB image of the same scene in order to deal with the high frequency details of the scene and recognize more easily different objects in that scene. In this work we are going to show how we built and used a Multiple Input Super Resolution (MISR) CNN in order to upsample depth images with the help of high resolution RGB images. The remaining part of the thesis is organized as follow:

Chapter 1: State of the Art

This chapter aims to describe the evolution of high resolution depth map estimation, results the artificial neural network field has reached so far, the most promising works in progress, and the different ways how convolution neural networks can be exploited for super-resolution purposes.

Chapter 2: Methods

This chapter is related to the description of the architecture we developed for the neural network, the parameter, the synthetic and the non artificial dataset we used, all the training procedures we adopted to achieve the best results in the depth map super resolution task.

Chapter 3: Results

This chapter is dedicated to the discussion of the results we obtained through the training of the network described in chapter 2, its comparison with respect to the other work we discussed in chapter 1 by means of either qualitative and quantitative results after the introduction of some metrics needed for the evaluation.

Chapter 5: Conclusions

In this final chapter we will give a brief summary of the work done and examine the possible developments that could be done in future

1 | State of the Art

This chapter is aimed to give an introduction to the research area that this work belongs to. We are going to start from the process behind the human being vision and how the electronic devices mimic it in order to take depthmap images, passing through the evolution of upsample algorithms in order to obtain high resolution images from the corresponding low resolution one and finish with some state of the arts of convolution neural network for super resolution task.

1.1. Depthmap Acquisition

1.1.1. Monocular vision

According to geometric optics, plane images are captured in the retinas and transmitted to the brain where they are projected outside to generate a mental representation of the object in space, or perceptual image. In monocular vision, the retinal image provides the brain with an exact representation of the object shape in two dimensions. As to object distance, the brain lacks geometric information enough to obtain telemetric data. Despite that, different types of pictorial cues, such as perspective, lights and shades, and logic judgments about size of familiar objects, allow the brain to make inferences concerning distance. In the absence of those cues, it becomes impossible for the brain to choose a specific location of object in space, as shown in Figure 1.1

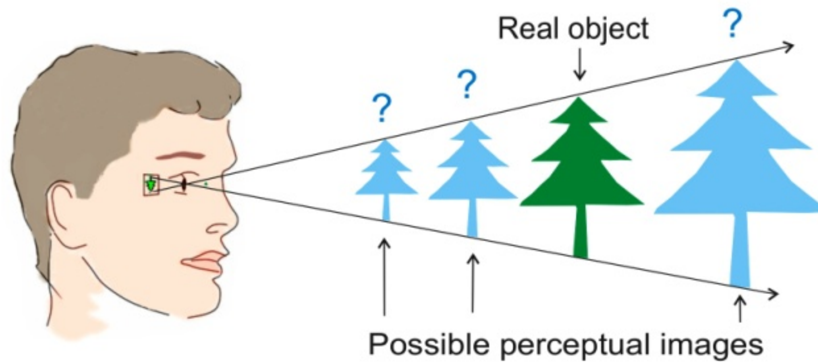


Figure 1.1: Monocular Vision

1.1.2. Binocular vision

What we see is the result of signals sent from the eyes to the brain. Usually, the brain receives signals from both (bi) eyes (ocular) at the same time. The information contained in the signal from each eye is slightly different and the brain is able to use these differences to judge distances and coordinate eye movements. Under most circumstances, we use information from both eyes to create a single visual image. This ability to converge information from both eyes is called binocular vision. Human being's eyes are both set on the front of the face, permitting binocular vision. This forward-facing orientation of the eye means each eye has a rather similar view of an image. Each eye sees slightly different spatial information and transmits these differences to the brain. The field of view, which is the area that you can see when you close one eye, overlaps significantly between each eye also. The center of the eyes' fields of view overlap with one another the most. This portion of the visual field provides the most detailed information to the brain. Visual information in the periphery or center of the visual field can, of course, be detected by one eye, but the combined visual information is what is required for binocular processing. This information is transmitted throughout the brain. Visual information is processed very early. Light is focused by the eye onto the delicate neural structure of the retina. The eye uses a series of focusing mechanisms to place a clear image on the retina. The front window of the eye called the cornea, and the focusing mechanism called the crystalline lens, focus light onto the retina. The retina organizes the information (light) that lands on it and changes the information to electrical impulses. These impulses (also called signals) exit the eye through the optic nerve and begin the pathway through the brain. Shortly after the optic nerve exits the eye, some fibers of each optic nerve become intertwined. Visual information from different parts of the visual field are combined and the neurons continue to make their way through the upper and lower lobes of the brain (called the

parietal and temporal lobes, respectively). The final stop of the neuron is the occipital lobe, an area in the back of the brain, which synthesizes and processes visual information. The brain then uses the discrepancies between the two eyes to judge distance and depth allowing us to perceive depth and relationships between objects. Depth perception is technically called stereopsis or stereoscopic vision.

1.1.3. Depthmap Acquisition

There are variety of depthmap acquisition digital system, each with its pros and cons. In this section we are going to discuss about passive stereo vision, active stereo vision such as structured light(SL) and time of light(ToF) technologies. Stereo vision is a machine vision technique that can provide full field of view 3D measurements using two or more machine vision cameras. The foundation of stereo vision is similar to 3D perception in human vision and is based on triangulation of rays from multiple viewpoints, Figure 1.2

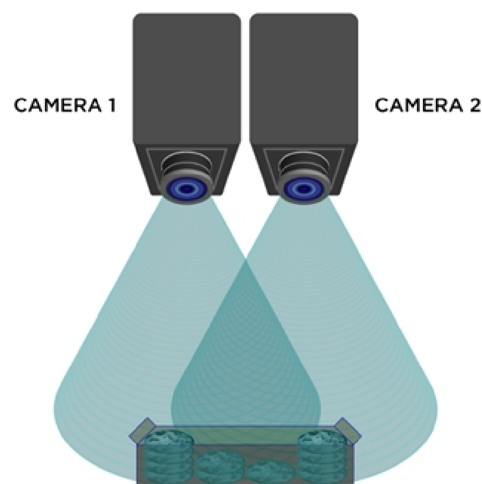


Figure 1.2: Stereo Vision Camera

A stereo camera closely copies how our eyes work to give us accurate, real-time depth perception. It achieves this by using two sensors a set distance apart to triangulate similar pixels from both 2D planes. Each pixel in a digital camera image collects light that reaches the camera along a 3D ray. If a feature in the world can be identified as a pixel location in an image, we know that this feature lies on the 3D ray associated with that pixel. If we use multiple cameras, we can obtain multiple rays. Finding where these rays intersect tells us the 3D location of an object and its features.

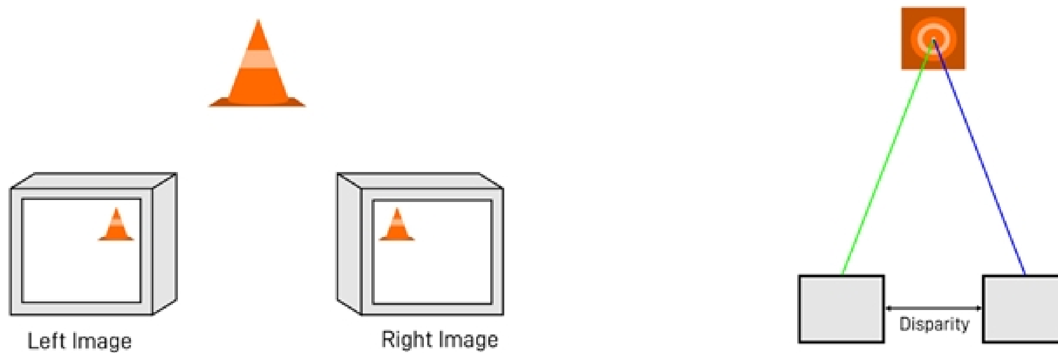


Figure 1.3: 3D Rays Intersection

Through basic triangulation of pixels and ray intersections, we can determine the 3D location of the traffic cone. The greater the disparity, the greater the angular offset from the object, and therefore the greater the 3D depth information. This will reduce problems such as optical occlusion, but will require careful calibration to succeed. Stereo vision in machine vision is considered a passive technology, as it does not require any artificial illumination to work. A stereo camera can simply be plugged in, calibrated, and deployed. Some stereo vision applications will, however, benefit from artificial illumination or a structured light source to aid visibility; in fact, some applications may rely on it to work. This is known as active stereo and has its pros and cons just as passive stereo does. Stereo vision can be CPU intensive when not hardware accelerated with FPGAs or GPUs for example. This is due to algorithms like Semi-global matching (SGM), which performs stereo matching using 2 cameras and lens distortions compensation, that need to take place for stereo vision to work accurately and consistently over time. Passive stereo camera systems can be deployed without the need for lasers or LEDs and can generally perform effectively in most ambient lighting conditions but if the system is operating in low light, or scanning non-textured scenes or objects with textureless surfaces, then stereo vision tends to underperform as a 3D technology. With no lasers or expensive lighting required, passive stereo vision can be much more affordable compared with 3D machine vision technologies. Furthermore, due to the lack of constraints on the range of motion on the target object like in 3D profiling technology, stereo vision can cope well with long distances and moving objects, something that other 3D imaging technologies tend to fall short on. Once calibrated, a stereo vision camera system can go on to detect depth in real time, and when combined with the right software to display the 3D image, users can benefit from colour depth mapping for added visibility.

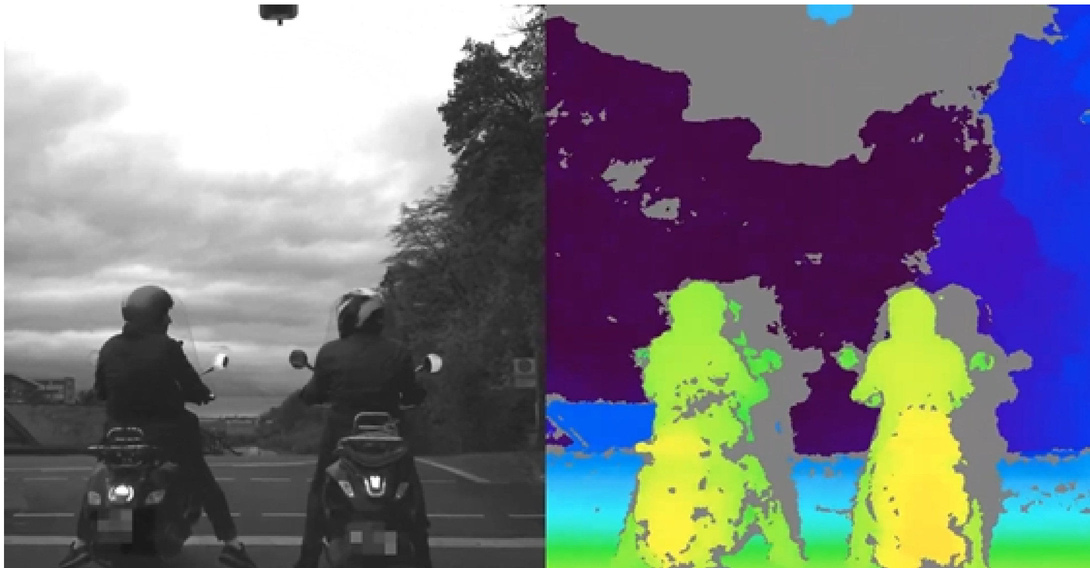


Figure 1.4: Colour mapping helps to visually quantify distances on-screen

Thanks to the work of [32] we can show an example of SL and TOF technologies making some comparison between them. We take as example Microsoft different generations of Kinect camera. The structured light approach is an active stereo-vision technique. A sequence of known patterns is sequentially projected onto an object, which gets deformed by geometric shape of the object. The object is then observed from a camera from a different direction. By analyzing the distortion of the observed pattern, the disparity from the original projected pattern, depth information can be extracted, Figure 1.4

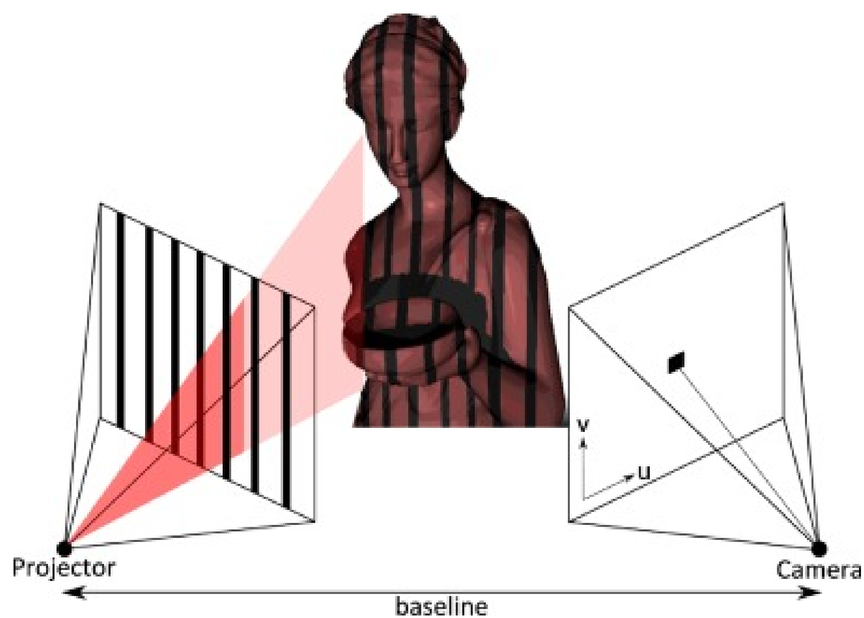


Figure 1.5: Principle of structured light based system

Knowing the intrinsic parameters of the camera like the focal length f and additionally the baseline b between the observing camera and the projector, the depth of pixel (x, y) can be computed using the disparity value m(x, y) for this pixel as $d = \frac{b \cdot f}{m(x,y)}$. As the disparity m(x,y) is usually given in pixel units, the focal length is also converted to pixel units. In most cases, the camera and the projector are only horizontally displaced, thus the disparity values are all given as horizontal distances. The depth range and the depth accuracy relate to the baseline, so longer baselines allow for robust depth measurements at long distances. Zhang et al. [40] investigate the benefit of projection patterns composed of alternative color stripes creating color transitions that are matched with observed edges. Their matching algorithm is faster and eliminates the global smoothness assumptions from the standard SL matching algorithm. Similarly, Fechteler et al. [7] use this color pattern to reconstruct at high-resolution human face using only two sequential patterns, which leads to a reduced computational complexity. SL cameras, such as the Kinect^{SL}, use a low number of patterns, maybe only one, to obtain a depth estimation of the scenery. Typically, it is composed of an near infra-red(NIR) laser projector combined with a monochrome CMOS camera which captures depth variations of object surfaces in the scene. The Kinect^{SL} camera is based on the standard structured light principle where the device is composed of two cameras, a color RGB, a monochrome NIR camera and an NIR projector including a laser diode at 850nm wavelength. The baseline between the NIR projector and the NIR camera is 7.5 cm, as we can see from Figure 1.6.

Figure 1.6: Sensor placement within a Kinect^{SL} camera. The baseline is of approximately 7.5 cm.

The NIR projector uses a known and fixed dot pattern to illuminate the scenery. Simple triangulation techniques are later on used to compute the depth information between the projected pattern seen by the NIR camera and the input pattern stored on the unit. For each pixel p_i, depth is estimated by finding the best correlation pattern patch, typically in a 9×9 pixel window, on the NIR image with the corresponding projection pattern. The disparity value is given by this best match. Kinect^{SL} device performs internally an interpolation of the best match operation in order to achieve sub-pixel accuracy of $\frac{1}{8}$

pixel. A detailed description of the Kinect disparity map computation can be found at the ROS.org community website [29], where the Kinect^{SL}'s disparity map computation has been reverse engineered and a complete calibration procedure is deduced.

The ToF technology is based on measuring the time that light emitted by an illumination unit requires to travel to an object and back to the sensor array [23]. In the last decade, this principle has found realization in microelectronic devices, i.e. chips, resulting in new range-sensing devices, the so-called ToF cameras. The KinectToF utilizes the Continuous Wave (CW) Intensity Modulation approach, which is most commonly used in ToF cameras. The general idea is to actively illuminate the scene under observation using near infrared (NIR) intensity-modulated, periodic light (see Figure 1.7).

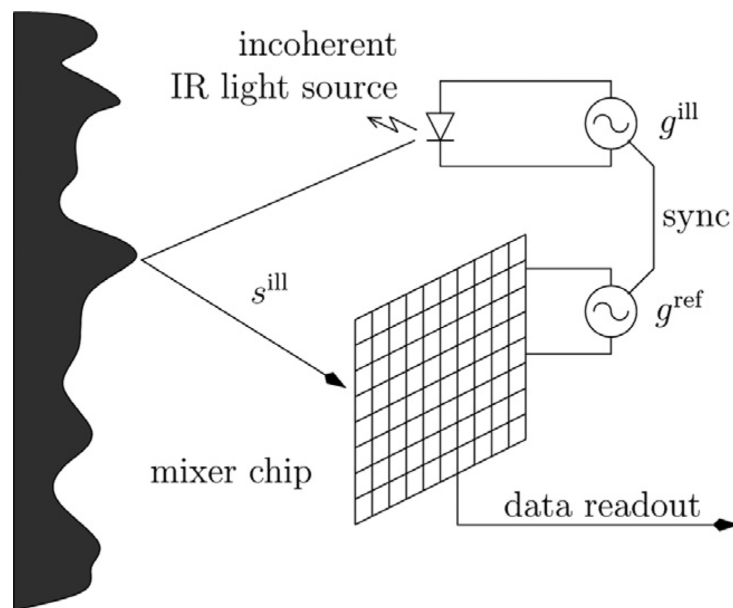


Figure 1.7: The ToF phase-measurement principle

Due to the distance between the camera and the object (sensor and illumination are assumed to be at the same location), and the finite speed of light c, a time shift ϕ[s] is caused in the optical signal which is equivalent to a phase shift in the periodic signal. This shift is detected in each sensor pixel by a so called mixing process. The time shift can be easily transformed into the sensor-object distance as the light has to travel the distance twice. From the technical perspective, the generator signal g^{ill} driving the illumination unit results in the intensity modulated signal which, after being reflected by the scene, results in an incident optical signal s^{ill} on each sensor pixel. The incident signal s^{ill} is correlated with the reference generator signal g^{ref}. This mixing approach yields the correlation function which is sampled in each pixel

$$C[g^{\text{ill}}, g^{\text{ref}}] = s \otimes g = \lim_{T \to \infty} \int_{\frac{-T}{2}}^{\frac{T}{2}} s^{\text{ill}}(t) \cdot g^{\text{ref}}(t)\, dt.$$

The phase shift is computed using several correlation measurements with varying illumination and reference signals $\mathrm{g_i}^{\text{ill}}$ and $\mathrm{g_i}^{\text{ref}}$, respectively, using some kind of demodulation function. The correlation images are acquired sequentially, however there is the theoretic option to acquire all correlation images in parallel, by having different phase shifts for neighboring pixels for example. Due to the periodicity of the reference signal, any ToF camera has a unique unambiguous measurement range. The Kinect$^{\text{ToF}}$ camera applies this CW intensity modulation approach. Kinect$^{\text{ToF}}$ acquires 10 correlation images, from which nine correlation images are used for a three-phase reconstruction approach based on phase shifts of 0°, 120° and 240° at three different frequencies. Using multiple modulation frequencies the measurement range can be exceeded [3]. Although the Kinect$^{\text{ToF}}$ camera can obtain depth values for distances longer than 9 m, the official driver masks the distances further than around 4.5 m. The illumination unit consists of a laser diode at 850 nm wavelength.

SL and ToF cameras are active imaging systems that use standard optics to focus the reflected light onto the chip area. Therefore, the typical optical effects like shifted optical centers and lateral distortion need to be corrected, which can be done using classical intrinsic camera calibration techniques. Beyond this camera specific calibration issues, SL and ToF cameras possess several specific error sources. As any other camera, ToF and SL cameras can suffer from ambient background light, as it can either lead to over saturation in case of too long exposure times in relation to the objects' distance and reflectivity, causing problems to SL systems in detecting the light pattern. Both, the Kinect ToF and the Kinect SL are utilized with a band pass filter, suppressing background light out of the range of the illumination. For ToF cameras specific circuitry has been developed, the Suppression of Background Intensity approach for PMD cameras [31] that electronically filter out the DC-part of the light. For SL systems outdoor application is usually hard to achieve. A common effect to many technical devices is the drift of the system output, the distance values in the case of Kinect cameras, during the device warm-up. The major difference between the SL and the ToF approach is that an SL camera usually does not produce as much heat as a ToF camera. This is due to the fact that the required illumination power to cover the full scene width and depth in order to get a sufficient signal to noise (SNR) for the optical signal for a ToF camera is beyond the power needed to generate the relatively sparse point based pattern applied by the Kinect$^{\text{SL}}$. As a consequence, the Kinect$^{\text{SL}}$ can be cooled passively whereas the Kinect$^{\text{ToF}}$ requires active

cooling. For the Kinect^{SL} significant temperature drift has been reported by Fiedler and Müller [8]. Early ToF-camera studies from Kahlmann et al. [18] of the Swissranger^{TM} camera exhibit the clear impact of this warm-up on the range measurement. Both Kinect cameras suffer from systematic error in their depth measurement. For the Kinect^{SL} the error is mainly due to inadequate calibration and restricted pixel resolution for estimation of the point locations in the image plane, leading to imprecise pixel coordinates of the reflected points of the light pattern [**?**]. Further range deviations for the Kinect^{SL} result from the comparably coarse quantization of the depth values which increase for further distances from the camera. For Kinect^{TOF}, on the other hand, the distance calculation based on the mixing of different optical signals s with reference signals g^{ref} requires either an approximation to the assumed, like a sinusoidal signal shape or an approximation to the phase demodulation function G. Both approximations lead to a systematic error in the depth measurement. In case of an approximated sinusoidal shape this effect is also called "wiggling" (see Figure 1.8 , top left).

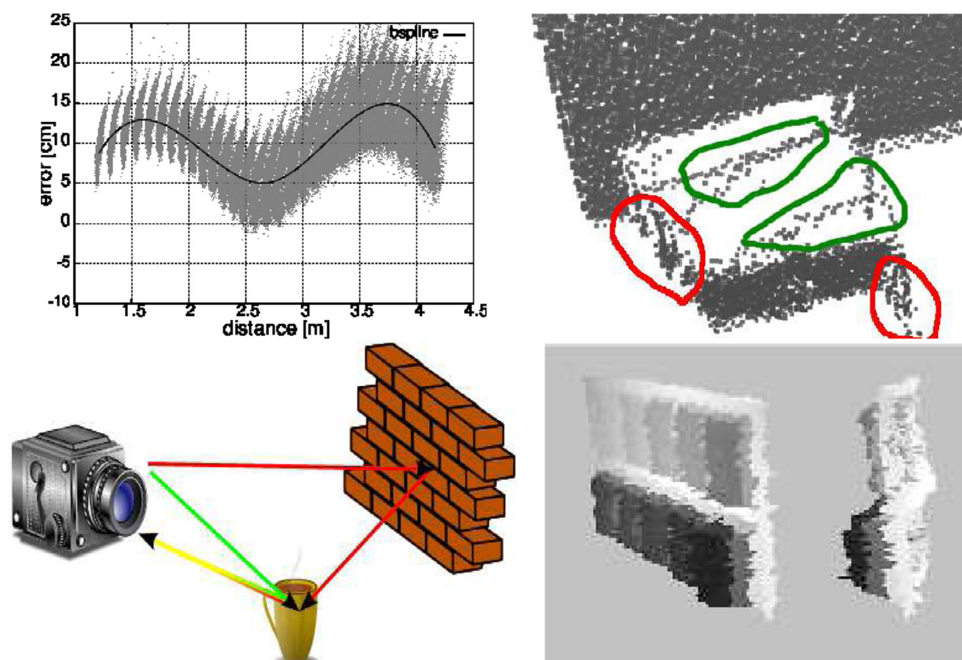


Figure 1.8: Error sources of ToF cameras. Top left: Systematic (wiggling) error for all pixels (gray) and fitted mean deviation (black). Top right: Motion artifacts (red) and flying pixels (green) for a horizontally moving planar object in front of a wall. Bottom left: Schematic illustration of multi-path effects due to reflections in the scene. Bottom right: Acquisition of a planar gray-scale checkerboard reveals the intensity related distance error

The systematic error may depend on other factors, such as the exposure time. Herrera et al. [1] proposed a joint calibration approach for the color and the depth camera of

the Kinect^{SL}. Correction schemes applied to reduce the systematic error of ToF cameras with sinusoidal reference signals simply model the depth deviation using a look-up-table or function fitting like using b-splines [26].

At object boundaries, a pixel may observe non homogeneous depth values. Due to the structured light principle, occlusion may happen at object boundaries where parts of the scene are not illuminated by the infra-red beam which results in a lack of depth information in those regions (invalid pixels). For ToF cameras, the mixing process results in a superimposed signal caused by light reflected from different depths, so-called mixed pixels. In the context of ToF cameras these pixels are sometimes called flying pixels. The mixed or flying signal leads to wrong distance values; (see Figure 1.8, top right). There are simple methods relying on geometric models that give good results in identifying flying pixel, for example by estimating the depth variance which is extremely high for flying pixel [31]. Denoising techniques, such as a median filter, can be used to correct some of the flying pixels.

Multi-path effects relate to an error source common to active measurement systems: The active light may not only travel the direct path from the illumination unit via the object's surface to the detector, but may additionally travel indirect paths, like being scattered by highly reflective objects in the scene or within the lens systems or the housing of the camera itself (see Figure 1.8, bottom left). In the context of computer graphics this effect is known as global illumination. For ToF cameras these multiple responses of the active light are superimposed in each pixel leading to an altered signal not resembling the directly reflected signal and thus a wrong distance. For Kinect^{SL} indirect illumination mainly causes problems for highly reflecting surfaces, as dots of the pattern may be projected at other objects in the scene. However, objects with a flat angle to the camera will lead to a complete lack of depth information. For ToF cameras several correction schemes for multi-path effects have been proposed for sinusoidal signal shapes. Falie and Buzuloiu [6] assume that the indirect effects are of rather low spatial frequency and analyze the pixel's neighborhood to detect the low frequency indirect component.

Considering a highly reflecting object and a second object with the same distance to the camera but with low reflectivity in the relevant NIR range, a reduced SNR is expected. Beyond this, it has frequently been reported that ToF cameras have a non-zero biased distance offset for objects with low NIR reflectivity (see Figure 1.8, bottom right). In general, there are at least two possible explanations for this intensity related effect. The first assumption explains this effect is a specific variant of a multi path effect, the second one puts this effect down to the non-linear pixel response for low amounts of incident intensity.

As for most active measuring devices, media that does not perfectly reflect the incident light potentially causes errors for ToF and SL cameras. In case of ToF cameras, light scattered within semitransparent media usually leads to an additional phase delay due to a reduced speed of light. The investigations done by Hansard et al. [10] give a nice overview for specular and translucent, i.e. semitransparent and scattering media for ToF cameras with sinusoidal reference signal and the Kinect^{SL}. Kadambi et al. [17] show that their coding method (originally designed for solving multi-path errors for ToF cameras) is able to recover depth of near-transparent objects using their resulting time-profile (transient imaging).

One key factor for any camera-based system is that each pixel observes a single object point during the whole acquisition process. This assumption is violated in case of moving objects or moving cameras, resulting in motion artifacts. In real scenarios, motion may alter the true depth. A moving object or camera leads to improper detection of the pattern in the affected region. ToF cameras require several correlation images per depth image. Furthermore, their correlation measurements get affected by a change of reflectivity observed by a pixel during the acquisition. Processing the acquired correlation images ignoring the motion present during acquisition leads to erroneous distance values at object boundaries (see Figure 1.8, top right). For ToF cameras several motion compensation schemes have been proposed. Schmidt and Jahne [35] detect motion artifacts using temporal gradients of the correlation images A_i, i.e. a large gradient in one of the correlation images indicates motion. This approach also performs a correction using extrapolated information from prior frames. Since motion artifacts result from in-plane motion between subsequent correction images, several approaches use optical flow methods in order to realign the individual correlation images. Lindner and Kolb [27] apply a fast optical flow algorithm [39] three times in order to align the four correlation images A_0, A_1, A_2, A_3 to the first correlation image A_0. As optical flow algorithms are computationally very expensive, these approaches significantly reduce the frame rates for real-time processing. A faster approach is motion detection and correction using block-matching techniques applied pixels where motion has been detected [13]

1.1.4. Convolution Neural Network

With the spread of the Machine Learning, and considering all the difficulties mentioned above related to the use of the high resolution depth sensor, new ways has been taken to deal with depthmap acquisition that exploit the convolution neural networks. The simplest attempt was made with the so called Single Image Depth Estimation (SIDE). As stated in [29], the depth estimation from a single image is the task of estimating a dense

depth map from a given single RGB image. More specifically, for each pixel in the given RGB image, one needs to estimate a metric depth value. An example of an input image and corresponding depth map can be seen in Figure 1.9.

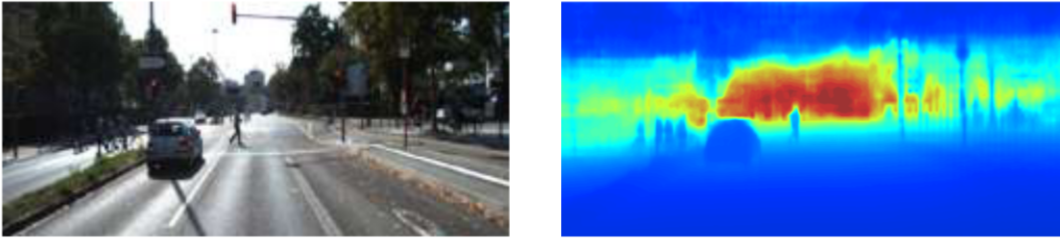


Figure 1.9: Input RGB image and the depth map estimated by the neural network

Here, the colors in the depth map correspond to the depth of that pixel: blueish means the pixel is closer to us, reddish means the pixel is further away from us. What makes the SIDE problem interesting and challenging is its inherent ambiguity. An endless number of different 3D scenes can result in the same 2D image. This suggests that there is a one to many mapping from RGB images to depth maps. But how do human beings estimate the depth from monocular images? The answer to this question lies in the cues humans use to do SIDE. There are seven such static cues that we can use to estimate the depth from a static single image. The first cue is occlusion which happens when one object partially covers another one. The partially covered object is considered to be farther away. The second cue is called perspective. We can observe this by looking at parallel lines that appear to meet in the distance. Two other cues are related to the perspective. One of them is size cue. The same object can have different sizes on the retinal image according to its distance. Therefore, the size of an object influences our depth estimates. The second cue related to the perspective is texture gradient. It happens when you look at a surface at a slant. The texture of the surface becomes denser as the distance increases. Another cue that we use to infer depth is called the atmospheric cue. It refers to the observation that objects get blurry and bluish as they move away from us. Moreover, we use patterns of light and shadows when perceiving depth. We consider things like objects casting shadows onto other objects or having shadows attached to their surfaces. The last cue that we use is the height cue. Objects closer to the horizon seem farther away. The most important cue here is the size cue. As humans, we have a rough estimate of the objects' size that we see in real world everyday. When we look at the world and observe 2D RGB images, our visual system estimates the 3D scene between an endless number of geometrically possible 3D scenes, using our prior knowledge to choose the one that fits into the world as we know it. This is also the reason why we are fooled by the images that are similar to the ones in the Figure 1.10.

Figure 1.10

Since there is no other cue that tells us otherwise, we assume the chair to have a usual size and accordingly estimate its depth closer to us. However, by looking at the relative sizes of the human and the chair in the right image, we understand that the chair is farther away than we estimated since it is bigger than that we assumed.

In 2014, Eigen et al. [5] introduce CNNs to the SIDE problem and achieve relatively good performance, when compared to earlier methods. CNN based solutions were already achieving quite satisfactory results in different vision problems at that time. Eigen et al. utilize the experience gained so far on CNNs and combine it with problem-specific knowledge to tackle the SIDE problem. They formulate the problem as a supervised regression problem and solve it with their framework, consisting of two networks, namely a Coarse and a Fine network, which are stacked on top of each other. Its operations can be summarized as follows:

$$Coarse(Input) = Depth_{\text{coarse}} \tag{1.1}$$

$$Fine(Input, Depth_{\text{coarse}}) = Depth_{\text{fine}} \tag{1.2}$$

The Coarse network consists of convolutional layers and fully connected layers at the end. Because of the fully connected layers, the network makes its decision by looking at the image as a whole. This allows it to utilize the "global context" of the image and make a coarse estimation of the depth of a scene. However, fully connected layers come with a huge computational cost. In order to be able to keep the model reasonable in terms of memory, the output resolution is decreased. The Fine network is a fully convolutional network and works by considering only the local parts of the image. It takes the original

input image and the estimation of the coarse network. It is like it refines the coarse estimation by working locally. Three reformulations of the same loss function can be seen below. $\hat{d}$ and d are the predicted and the ground truth depth, respectively. Sub indices indicate the pixels, and n is the total number of pixels in an image.

$$loss = \frac{1}{2n}\sum_{i=1}^{n}(\log \hat{d}_{\mathrm{i}} - \log d_{\mathrm{i}} + \frac{1}{n}\sum_{i}(\log d_{\mathrm{i}} - \log \hat{d}_{\mathrm{i}}))^2 \tag{1.3}$$

$$= \frac{1}{2n^2}\sum_{i,j}((\log \hat{d}_{\mathrm{i}} - \log \hat{d}_{\mathrm{j}}) - (\log d_{\mathrm{i}} - \log d_{\mathrm{j}}))^2 \tag{1.4}$$

$$= \frac{1}{n}\sum_{i}(\log \hat{d}_{\mathrm{i}} - \log d_{\mathrm{i}})^2 - \frac{1}{n^2}\sum_{i,j}((\log \hat{d}_{\mathrm{i}} - \log d_{\mathrm{i}})(\log \hat{d}_{\mathrm{j}} - \log d_{\mathrm{j}}) \tag{1.5}$$

Equation 1.3, subtracts the mean loss from the loss of each pixel in order to make up for mistakes due to scale ambiguity. We could have a different interpretation by reformulating it as Equation 1.4, which compares each pixel pair in the ground truth and the same pixel pair in the estimated depth map, in order to achieve the same distance between pixels in both the ground truth and the estimated depth maps. Similarly, the same loss function can be written as in Equation 1.5, which can be interpreted as the network being penalized for mistakes in different directions and being rewarded for mistakes in the same direction. All three above formulations are equivalent, and actually a fourth reformulation where the function can be computed in linear time, is used to train the network:

$$loss = \frac{1}{n}\sum_{i}(\log \hat{d}_{\mathrm{i}} - \log d_{\mathrm{i}})^2 - \frac{\lambda}{n^2}(\sum_{i}(\log \hat{d}_{\mathrm{i}} - \log d_{\mathrm{i}}))^2 \tag{1.6}$$

Here, λ is a hyperparameter that controls the scale invariance of the loss. $\lambda = 1$ refers to fully scale invariant, and $\lambda = 0$ refers to the normal L2 loss. Following their previous work in [5] very closely, and building on top of it, Eigen et al. [4] devise a network architecture that can successfully estimate depth, surface normals, and semantic labels. They do not optimize their system jointly for the said tasks, instead, they merely show that a single neural network architecture can solve all the tasks. Although they experiment with shared layers between depth and surface normals, it does not improve their results. As an improvement to their previous work, another scale is added to their multi-scale network architecture, which further refines the output using convolutional layers. Both [5] and [4] were able to achieve very good results at that time. Nevertheless, there were still opportunities for improvement. First of all, most real-world tasks require a real-time

vision pipeline, and Eigen et al.'s framework [5] is slow. Another of its weaknesses is having too many learnable parameters that lead to an increase in memory requirements, and the need for too many training points to train the network. Since CNNs has no prior knowledge cue such as human being, therefore estimating depth from a given single RGB image requires a high level understanding of the scene such as detecting and recognizing objects and their relations in 3D. They cannot avoid being fooled by unusual proportion of the objects present in the scene and therefore the output depth estimation will be incorrect if we pass to the network as input a scene like the one in Figure 1.10. The strange proportion of the chair will make the network assume that it is closer to the camera than it actually is. The loss from depth estimation alone may not be enough for the network model to discover those high-level relations. One way to help us solving this issue could be using the log space when talking about loss function. Working in log space the challenge of estimating the depth of close objects and distant objects is not the same. While being a few centimeters off in our estimation of depth for an object that is meters away is acceptable, it is definitely a bigger mistake to be a few centimeters off if the object is only ten centimeters away. This is as much the case for humans as it is for computer vision systems. While we can be more precise in our estimations for smaller depths, we could only provide a rough depth range for bigger depths. For this reason, the errors are usually calculated in log space since the logarithm function maps the depth values in a way that error functions become more forgiving for mistakes in bigger depths. Also using relative depth can help; the term "relative depth" refers to the ordering of the depth of pixels instead of absolute depth which refers to the metric depth values. Relative depth can be used to measure the performance of the system or can be used as an error function. This way, the system would not be penalized for mistakes due to scale ambiguity. Even spatial coordinates can come in handy; using spatial coordinates can help to exploit the structure of the scenes that the system has seen before. When working on a dataset consisting of outdoor images, the pixels in the upper rows have a high chance of being part of a sky, and statistical learning systems can exploit this kind of relations easily. On the other hand, this exploitation can create bias in the system. So far, the works we considered deal with indoor and outdoor datasets separately. However, eventually, we would like to train systems that can work on images coming from unstructured environments, i.e. systems that work in the wild. There are two important issues to consider for depth in the wild problem. The first issue a system faces that aims to work in the wild is that the depth ranges may vary quite a lot which is a problem for the learning process. While the range of absolute depth values for an image of a table may be between 10 cm and a couple of meters, the image of a touristic place such as "Eiffel Tower" contains pixels that are a hundred meters away, if they have a depth value at all (sky regions can be considered

infinitely far away). This kind of diversity makes the problem of estimating an absolute depth quite hard. Moreover, collecting datasets with absolute depth annotations for a diverse set of scenes that will allow a system to generalize unseen natural scenes is also very problematic. It requires a lot of effort in terms of time and money. It is clear that a new approach with new datasets was needed. Even having a valid dataset is a problem that has not to be underestimate. Training with synthetic data Machine learning systems need lots of data points in order to be able to learn the task at hand. For the SIDE task, datasets consist of RGB images and corresponding depth maps. Unfortunately, acquiring RGB images and corresponding depth maps is a costly job. Even though different datasets have been collected throughout the last decades the need for labeled data is considered to be a problem in general in machine learning. One of the ways to overcome this problem is the usage of synthetic data. The system still learns with labeled data, however, synthetic data can be created and labeled automatically with ease in great amounts. Moreover, similar to data augmentation, great diversity in the data can be achieved by changing the texture of the objects or the lighting of the scene while keeping the ground truth the same, which will increase the robustness of the system. For all these reasons, the usage of synthetic data is considered to be a solution. On the other hand, the usage of synthetic data, in itself, creates a problem. Synthetic and real data are considered as different domains and the system needs to adapt to the real data after it is trained on synthetic data. Additionally, while diversity for a given scene can be achieved with ease, creating diverse sets of natural scenes synthetically is an incredibly time-consuming job. Last but not least, the resolution of the output is very low. Again, from the perspective of the application, higher resolution outputs are more desirable. Instead of starting from an RGB image of the scene we could start from a low resolution depth estimation maybe acquired with the cameras we mentioned earlier and then upsample it in order to obtain a depthmap at the resolution we want. Throughout the time different algorithms for this purpose has been written. By doing this we settle for a cheaper depth sensor and let the software do the rest. There are more than one upsample algorithms out there; the simplest is the nearest neighbour, which simply replace every new added pixel with the value of the nearest pixel on the original image; the drawback is the pixelated effect in the output image because we only duplicate a bunch of pixels in the gaps and as a result there is no slight fading between pixels of different values, actually is quite the opposite. A more advance upsample algorithm could be the linear interpolation, which compute the weighted sum above one direction between the original pixel's values;the weights are the distance of the new added pixel from the two original ones. Then we have the bilinear which consist of the same as the linear but done both in vertical and horizontal directions. the evolution of all these is the cubic and its 2D version bicubic

interpolation algorithm where we retrieve the value for the new added pixels by drawing curves between the original pixel's values instead of lines. You can see the basic concept of all these algorithms in Figure 1.11.

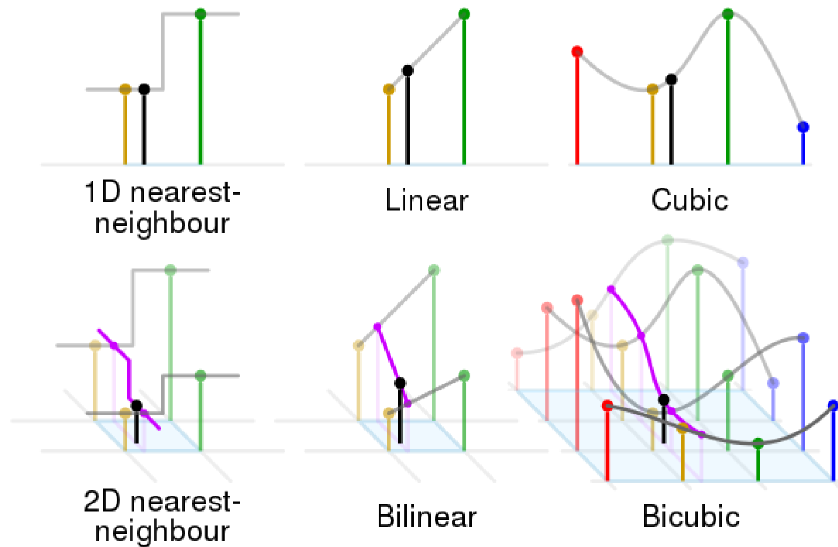


Figure 1.11: Interpolation Algorithms

Especially if the upsample factor we need is huge or if the starting picture is very little in term or resolution, these aforementioned algorithms are not the best way to do the task. Here we propose an example of an image of a bee; Figure 1.12

Figure 1.12: original image

We apply the nearest neighbor algorithm and as we can see the resulting image is very pixellated as we explained before. The result can be seen in Figure 1.13.

Figure 1.13: nearest neighbor interpolation

Then we apply the bilinear algorithm and as we can see from Figure 1.14, the pixellated effect is significantly reduced with respect to the image obtained via nearest neighbor.

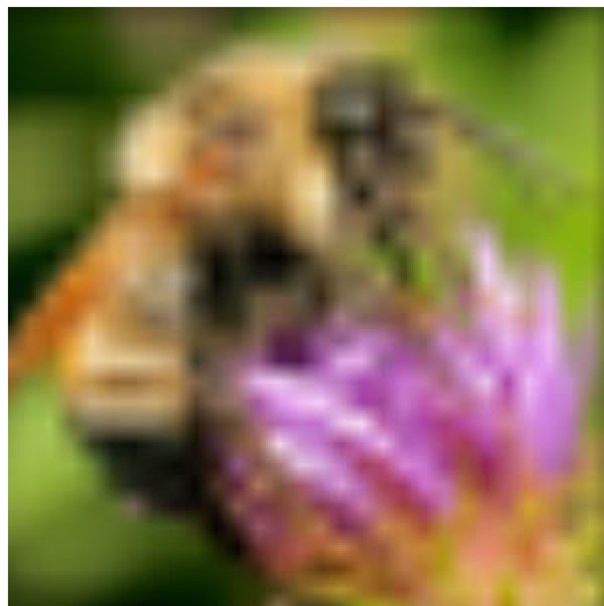


Figure 1.14: bilinear interpolation

Lastly we apply the bicubic intepolation and as we can see from Figure 1.15 the pixellated effect is no more present, the picture seems well detailed even if presents this kind of blurry effect

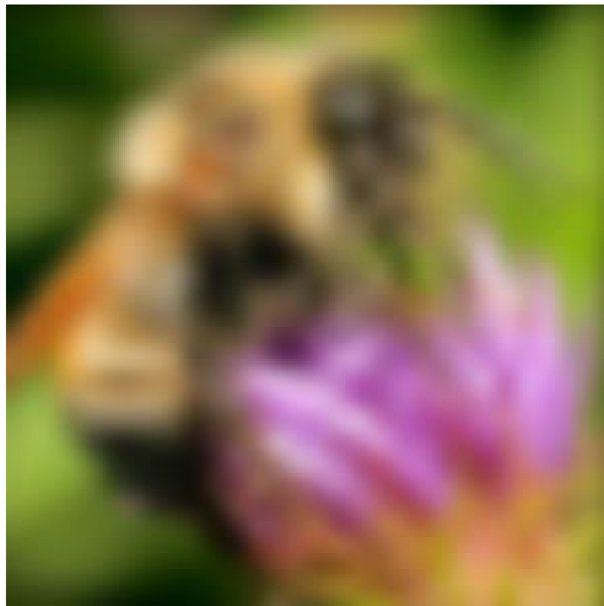


Figure 1.15: bicubic interpolation

As we can see from this example, the more sophisticated upscaling algorithms result in having smoother outputs with fewer artifacts even though the final outcome is not good enough. Upsampling doesn't add any more detail than you originally captured; it just reinterprets your image to reduce pixellation. Even in the best of cases, the image won't look as clean as a high resolution original. This example used RGB image for simplicity but the same reasoning can be done if using depthmap images.

A possible solution to achieve better result could be taking the best from both worlds, making the best out of the CNNs' features and at the same time starting our task from a low resolution depthmap in order to avoid the lack of cues the network would need if worked staring from RGB images as we showed above. This is where Super Resolution comes into play. Nowadays The CNNs are widely used today for recovering high resolution(HR) images from low resolution(LR), applied either on RGB images and depthmaps. With the terms Super Resolution we mean the upsample of depthmap using CNN. An advantage of the CNN with respect to the classical upsampling algorithms mentioned before is that if the latter do the same computation for every input we provide, the latter learns to behave differently depending on the input image. This capability should in theory be useful to overcome possible artifacts introduced in the outcome image by the traditional algorithms. Super-Resolution divides in two categories: Single Image Super-Resolution(SISR) and Multiple Image Super-Resolution (MISR), where the former has only one input image to work with,that is our image to be upsampled and the latter has more than one image as input in order to help the net improving its vulnerability.

A typical case is when we feed a CNN for Super-Resolution of depth images with its corresponding RGB high resolution image; basically we need one RGB image and one depth image of the same scene, which isn't easy because in most of the cases we need two different cameras and we need to set them properly to obtain two perfectly matching photos otherwise we could run in troubles during the training of the net. The detailed structures in the RGB image can help the net dealing with the high frequency patches of a scene and recognizing more easily different objects in the scene; can also be used to avoid blurry edges and suppress noises when up-sampling depth maps. In the following sub section we are going to show one state of the art in the filed of SR.

1.1.5. State of The Art works

The work we present is "Joint Implicit Image Function for Guided Depth Super-Resolution" by Tang et al. [38]. They revisited the MISR problem and viewed it from the perspective of implicit neural representation. The idea of implicit neural representation is to use a deep implicit function (DIF) to map continuous coordinates to signals in a certain domain. To share knowledge across different input observations, an encoder used to extract latent codes from the input to make the DIF conditional to the current observation. Thus, a scene can be represented by a set of local latent codes distributed in the coordinates of the input domain, which can be used in downstream tasks such as semantic segmentation [30] and super-resolution [2]. To make the output of DIF continuous, a weighted average of the predictions from several neighboring coordinates is usually calculated, which can be viewed as a neural implicit interpolation process. However, these weights are usually empirical, distance-based for example, in previous work since there is no other prior knowledge between the query coordinate and the neighboring coordinates. With the extra HR guide image in the guided super-resolution task, we can learn to extract this knowledge and learn the weights in a datadriven way. We hypothesize that the guide image can benefit the learning of both interpolation weights and values, and propose to learn the interpolation weights via a graph attention mechanism. Furthermore, we integrate the learning of weights and values into one unified DIF, which we call the joint implicit image function (JIIF) representation. They start from a general formulation of the image interpolation problem for image upsampling, then view it from the perspective of implicit neural representation to introduce the neural implicit image interpolation method. For each LR input image M, they want to calculate the corresponding HR target image I:

$$I(x_{\mathrm{q}}) = \sum_{i \in N_{\mathrm{q}}} w_{\mathrm{q,i}} v_{\mathrm{q,i}} \tag{1.7}$$

where x_q is the coordinate of the query pixel q in the HR domain, N_q is the set of neighbor pixels for q in the LR domain, $w_{q,i}$ is the interpolation weight between i and q, and $v_{q,i}$, is the interpolation value for i. The interpolation weights are usually normalized so that their sum is equal to one. they use a continuous image representation by scaling the image coordinates into $(-1, 1)$ to make it possible to share the coordinate between the HR and LR domain. N_q is usually chosen as the four nearest corner pixels of q in the LR domain. Different interpolation methods have different ways to calculate the interpolation weights and values. The most commonly used bilinear interpolation is implemented with:

$$w_{q,i} = \frac{S_i}{S} \tag{1.8}$$

$$v_{q,i} = M(x_i) \tag{1.9}$$

where S_i is the partial area diagonally opposite to the corner pixel i, $S = \sum_{i \in N_q} S_i$ is the total area serving as a normalization factor and $M(x_i)$ is the pixel value of the LR input image at x_i. In implicit neural representation, instead of directly using the pixel value $M(x_i)$, a DIF is applied to calculate the interpolation value $v_{q,i}$, for example,

$$v_{q,i} = f_\theta(z_i, x_q - x_i), \tag{1.10}$$

where $f_\theta(.)$ is a MLP with parameters θ that takes a local latent code z_i and a relative coordinate x_q - x_i as input. The target image is represented by a set of local latent codes distributed at the pixel coordinates of the LR domain, each storing information about its local area [2]. The latent codes map is the output feature map from an encoder network, and it is of the same resolution as the LR input image:

$$z_i = E_\phi(M)(x_i), \tag{1.11}$$

where $E_\phi(.)$ is the encoder network with parameters ϕ. The DIF models a local area centered at the coordinate of the given latent code. By querying the conditioned DIF $f_\theta(z_i, .)$ with a relative query coordinate x_q - x_i, it returns the estimated target value at the query coordinate x_q, that is the depth value in depth super resolution. The weighted average of these estimated values from the four corners is further calculated. They focus then on the problem of guided super-resolution, where an extra HR guide image G is provided with the LR input image M. They hypothesize that the information in the guide image can benefit the learning of both interpolation weights and values, and these two terms can

be learned jointly to boost the performance. Inspired by the recent neural implicit image interpolation methods, they propose to use DIFs to model both the interpolation weights and values, which they call the Joint Implicit Image Function representation. The target image is represented by a set of local latent codes, but these latent codes are extracted from both the LR input image and the HR guide image, allowing the detailed information from the guide image to help the up-sampling process. In particular, they apply two encoder networks to extract two sets of latent codes from the input image and the guide image respectively:

$$z_{\mathrm{i}} = E_{\phi}(M)(x_{\mathrm{i}}), \tag{1.12}$$

$$g_{\mathrm{j}} = E_{\psi}(G)(x_{\mathrm{j}}), \tag{1.13}$$

where E_{ψ} is another encoder network with parameters ψ. The interpolation values can be naturally calculated by querying the DIF with these two latent codes and a relative coordinate:

$$v_{\mathrm{q,i}} = f_{\theta}(z_{\mathrm{i}}, g_{\mathrm{i}}, x_{\mathrm{q}} - x_{\mathrm{i}}), \tag{1.14}$$

where i is one of the neighbors of q in the LR domain ($i \in N_q$). Due to the different resolutions of HR and LR images, we could not obtain the HR latent code at position $\mathrm{x_i}$ directly. In such cases, we conduct the bicubic interpolation operation to approximate the HR latent code at position $\mathrm{x_i}$. They propose to learn the interpolation weights at the same time. As illustrated in the neural implicit interpolation part in 1.16, the interpolation at each query pixel is viewed as a graph problem. The four corner pixels and the query pixel are the vertices, and each corner pixel is connected to the query pixel with an edge. A graph attention mechanism is used to calculate the edge weights. The guide latent code of each corner pixel $\mathrm{g_i}$ is extracted and the query pixel $\mathrm{g_i}$ in the HR domain, and apply a MLP to learn the weight:

$$a_{\mathrm{q,i}} = f_{\eta}(g_{\mathrm{i}}, g_{\mathrm{q}} - g_{\mathrm{i}}), \tag{1.15}$$

where $\mathrm{a_{q,i}}$ is the learned edge weight and f_{η} is a MLP with parameters η. Since the representation of the interpolation weights and values are of a similar form, the two functions can be integrated into a unified one:

$$a_{\mathrm{q,i}}, v_{\mathrm{q,i}} = f_{\theta}(z_{\mathrm{i}}, g_{\mathrm{i}}, g_{\mathrm{q}} - g_{\mathrm{i}}, x_{\mathrm{q}} - x_{\mathrm{i}}), \tag{1.16}$$

By integrating the learning of interpolation weights and values, they reduce the parameters

needed to model the representation, and allow interaction between these two processes, which is demonstrated to be more effective in their experiments.

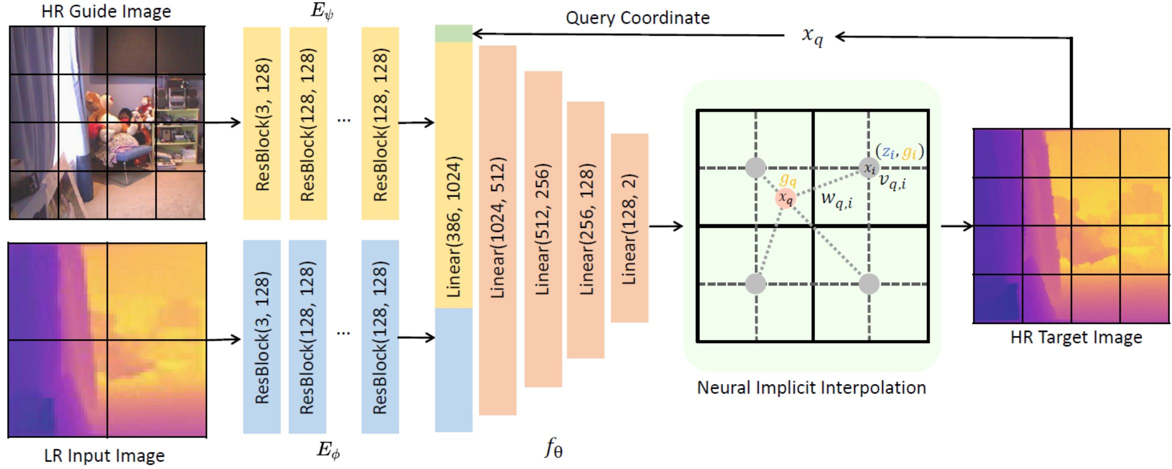


Figure 1.16: Network Architecture: the grid illustrates the relative image resolution and a $\times 2$ up-sampling as an example for simplicity. Given a HR guide image and a LR input image, two sets of latent codes are extracted via two encoders, then query the JIIF decoder with a coordinate in the HR domain to predict the pixel value at this coordinate. The prediction is a weighted average from the four nearest coordinates in the LR domain just like the standard image interpolation, but learn the interpolation weights and values via a deep implicit function.

2 | Methods

In this chapter we see the methods used to train a convolution neural network to get upsampled depthmaps with the guide of color information. The library employed for this purpose is Keras [13]. It is a high-level open-source API written in Python and designed for the implementation of neural networks. Also, it is modular, meaning that it is not tied to a single backend, but it can exploit different ones such as Tensorflow and Theano. Keras, being a high-level platform, does not handle the low-level operations by itself, but it relies on these backends which are specialized and optimized machine learning libraries [29]. Among the backend engines available for Keras we will use Tensorflow. We implemented two different netowrks, one performing 8x scale and the other performing the 4x scale operation, in order to evaluate the proposed method with two different level of difficulties.

2.0.1. Architecture

Both networks present some basic block we are going to explain from the simplest to the most complex one. The first block we present is the convolution block and is composed as follows:

```
conv_layer = Conv2D(filter_number,(kernel_size,kernel_size),
   padding ="same")(input)
activation_layer = LeakyReLU(alpha = 0.4)(conv_layer)
```

we have a two dimensions convolution operation where filter number is the number of filters we apply to the input and accordingly the number of channels the output has, the kernel size tuple specifying the height and width of the convolution window and lastly we have the padding parameter indicating if the input is filled with zeroes in order to obtain an output with the same resolution of the input, otherwise the output resolution will be reduced with respect to the input because of how convolution filter works. Kernel size is a parameter deserved to be noticed. In the current Deep Learning world, we are using the most popular choice that is used by every deep learning practitioner out there, and that is 3x3 kernel size. In 2012, when AlexNet et al. [22] introduced CNN architecture, it

used 11x11, 5x5 like larger kernel sizes that consumed two to three weeks in training. So because of extremely longer training time consumed and expensiveness, we no longer use such large kernel sizes. One of the reason to prefer small kernel sizes over fully connected network is that it reduces computational costs and weight sharing that ultimately leads to lesser weights for back-propagation. So then came VGG convolution neural networks in 2015 which replaced such large convolution layers by 3x3 convolution layers but with a lot of filters. In the later versions, the 5x5 convolutional layer of the first version of GoogleNet has been replaced by 2 stacked 3x3 convolutional layers. We can replace 5x5 or 7x7 convolution kernels with multiple 3x3 convolutions on top of one another as you can see from Figure 2.1 below.

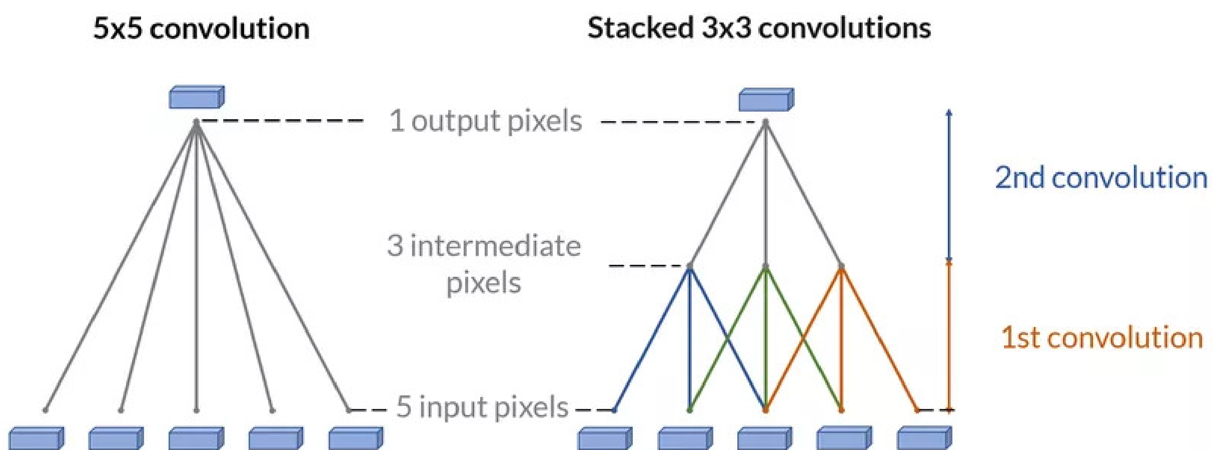


Figure 2.1: 5x5 convolution vs the equivalent stacked 3x3 convolutions

Stacking smaller convolutional layers is lighter, than having bigger ones. It also tends to improve the result, with the simple intuition that it results in more layers and deeper networks. Below is a simple example using the [21] dataset with Keras. We have 2 different Convnets. They are composed of 2 convolutions blocks and 2 dense layers. Only the construction of a block changes. In orange, the blocks are composed of 2 stacked 3x3 convolutions. In blue, the blocks are composed of a single 5x5 convolution. Notice how stacked convolutional layers yield a better result while being lighter.

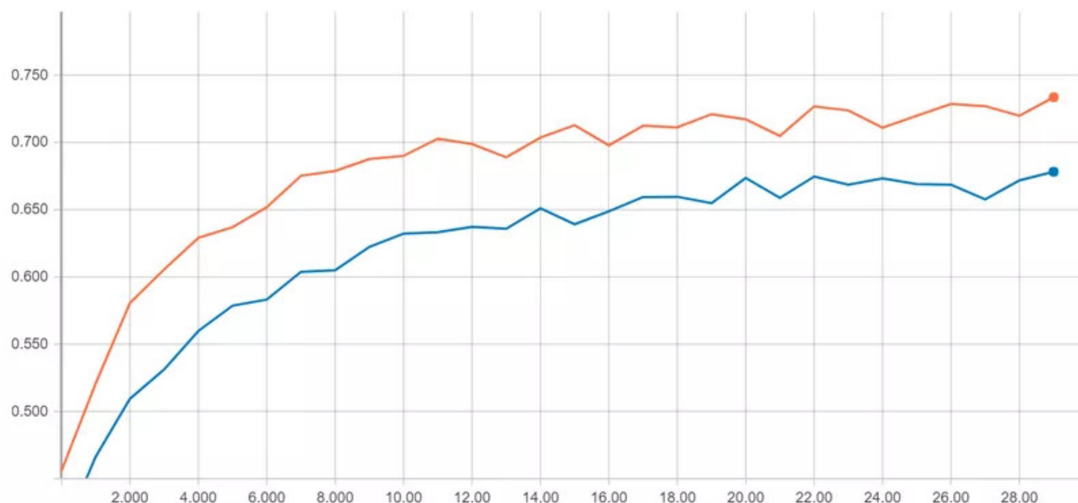


Figure 2.2: Validation Accuracy on a 3x3-based Convnet (orange) and the equivalent 5x5-based Convnet (blue)

3x3 sized kernel has became a popular choice. 2x2 and 4x4 are generally not preferred because odd-sized filters symmetrically divide the previous layer pixels around the output pixel and if this symmetry is not present, there will be distortions across the layers which happens when using an even sized kernels, that is, 2x2 and 4x4. So, this is why we don't use 2x2 and 4x4 kernel sizes.

As can be seen from Figure 2.3, after each convolution block we have the activation layer which is the Leaky Rectified Linear Unit (Leaky ReLU), a type of activation function based on a ReLU. The Rectified Linear Unit(ReLU) activation function, although it gives an impression of a linear function, it is not since the derivative isa changing for positive or negative values and allows for backpropagation while simultaneously making it computationally efficient. The main catch here is that the ReLU function does not activate all the neurons at the same time. The neurons will only be deactivated if the output of the linear transformation is less than 0. The advantages of using ReLU as an activation function are as follows: since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to other activation like the sigmoid and tanh functions; ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property. One limitations faced by this function is that the negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated. Leaky ReLU is an improved version of ReLU function as it has a it has a small slope for negative values instead of a flat slope in the negative area. The slope coefficient is determined

before training and not learnt during training. The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values. By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region. This functions introduce non linear real-world properties to artificial neural networks.

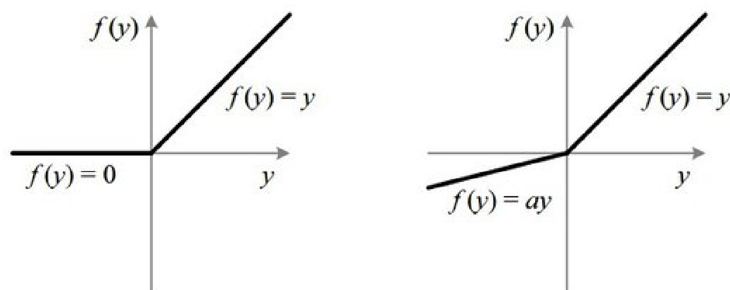


Figure 2.3: on the left the ReLU activation function and on the right the Leaky ReLU activation function

The most important among the simple blocks is the subpixel convolution block and here is the related snippet code:

```
def SubpixelConv2D(scale):
    return Lambda(lambda x: tf.depth_to_space(x, scale))
```

the function takes as input the scale parameter which is the upsample scaling factor we want to perform. The core of the subpixel convolution is the depth to space function which takes as input the scale parameter and the input tensor.

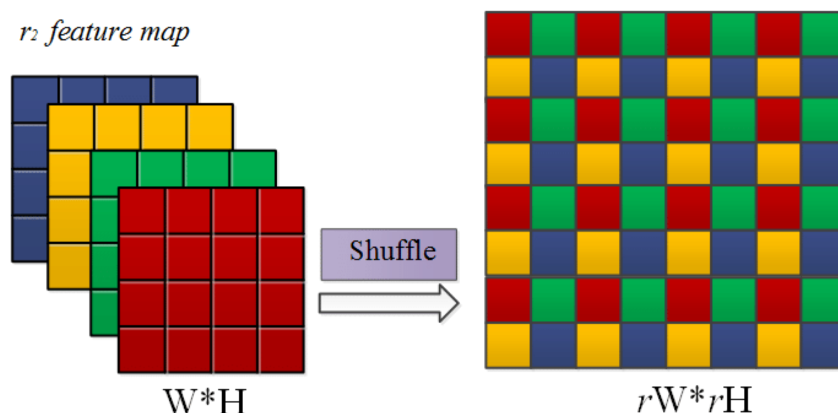


Figure 2.4: example of a subpixel convolution function: depth to space shuffle

As we can see from the Figure 2.4, the depth to space function consists of a rearrangement of the pixel. Each pixel from each one of the r^2 feature map of dimension W*H is mapped into one r^2 square area in the output image. This means that the final image will have a total dimension of r^2 x WxH so that the total number of pixels is consistent with the HR image to be obtained.

The first composed block we present you is the residual block, here is how it works:

```
def residual_block(input_, filter_number, kernel_size):
        main_branch = Conv2D(filter_number, (kernel_size,
           kernel_size), padding = "same")(input_)
        main_branch = LeakyReLU(alpha = 0.4)(main_branch)
        skip_branch = input_
        output = Add()([main_branch, skip_branch])
        return output
```

we have two branches: the main branch and the skip branch. The main branch is fed into a convolution layer and subsequently into an activation layer. The skip branch simply propagates the input without performing additional operations. Both are then summed together into the output tensor. In Figure 2.5 we show a basic example of residual network

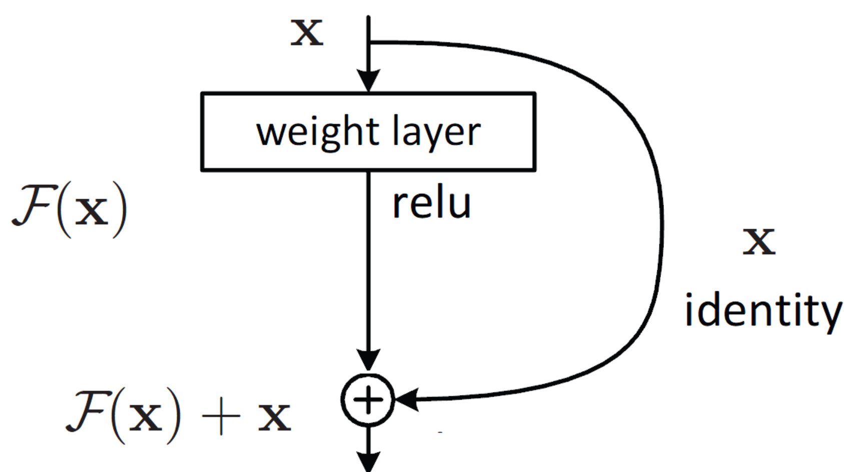


Figure 2.5: example of residual block

As stated in [11], one of the major benefits of a very deep network is that it can represent very complex functions. However, a huge barrier to training them is vanishing gradients: very deep networks often have a gradient signal that goes to zero quickly, thus making gradient descent prohibitively slow. More specifically, during gradient descent, as we backpropagate from the final layer back to the first layer, we are multiplying by the weight

matrix on each step. If the gradients are small, due to large number of multiplications, the gradient can decrease exponentially quickly to zero or, in rare cases, grow exponentially quickly and “explode” to take very large values. In order to avoid this problem in this work we use a Deep Residual learning framework. The idea is that instead of letting layers learn the underlying mapping, let the network fit the residual mapping. So, instead of say H(x), initial mapping, let the network fit, F(x) := H(x)-x which gives H(x) := F(x) + x. The approach is to add a shortcut or a skip connection that allows information to flow, well just say, more easily from one layer to the next’s next layer, i.e., you bypass data along with normal CNN flow from one layer to the next layer after the immediate next. Here are two key point of the residual block: adding either additional or new layers would not hurt the model’s performance as regularisation will skip over them if those layers were not useful and if either the additional or new layers were useful, even with the presence of regularisation, the weights or kernels of the layers will be non-zero and model performance could increase slightly. Therefore, by adding new layers, because of the skip connection, it is guaranteed that performance of the model does not decrease but it could increase slightly. By stacking these resolution network(ResNet) blocks on top of each other, you can form a very deep network.

Then we have the upsample block and as you can see below, this is the corresponding code:

```
def skip_branch_depth(depth_input, filter_number, kernel_size,
   scale_factor,):
        output = elementary_manipulation(depth_input,
           filter_number, kernel_size, "convolution_layer")
        output = elementary_manipulation(output, filter_number,
            kernel_size, "residual_block")
        if(scale_factor == 2):
                output = SubpixelConv2D(2)(output)
        if(scale_factor == 4):
                output = SubpixelConv2D(4)(output)
    if(scale_factor == 8):
                output = SubpixelConv2D(8)(output)
        output = elementary_manipulation(output, filter_number,
            kernel_size, "convolution_layer")
        return output
```

the first operation is a simple convolution layer followed by a Leaky ReLU activation,

then we perform two subsequently residual block operation; based on the scaling factor we perform the actual upsample activity and lastly we repeat the convolution block. Throughout the network we perform different kind of upsample operation we are going to explain later on when we will discuss about the whole network.

The last block is the one related to the downsample block, operation we only perform on the color branch. As you can see from the code,

```
def branch_color(color_input, filter_number, kernel_size,
   kind_of_manipulation):
        output = elementary_manipulation(color_input,
           filter_number, kernel_size, "convolution_layer")
        output = elementary_manipulation(output, filter_number,
            kernel_size, "convolution_layer")
        output = MaxPooling2D(pool_size = (2, 2), strides = 2,
           padding ='same')(output)
        output = elementary_manipulation(output, filter_number,
            kernel_size, "convolution_layer")
        return output
```

we perform two subsequent convolution layers with their related Leaky ReLU activation layers, then we have the two dimension max pooling layer which downsamples the input along its spatial dimensions (height and width) by taking the maximum value over an input window for each channel of the input. The window is shifted by strides along each dimension. These are the parameters of the max pooling layer: pool size is a tuple of 2 integers, window size over which to take the maximum; strides in an integer and specifies how far the pooling window moves for each pooling step; the padding parameter can take two values: "valid" means no padding, "same" means in padding evenly to the left,right or up,down of the input such that output has the same height and width dimension as the input.

The whole architecture of the 8x scaling factor network is reported in Figure 2.6.

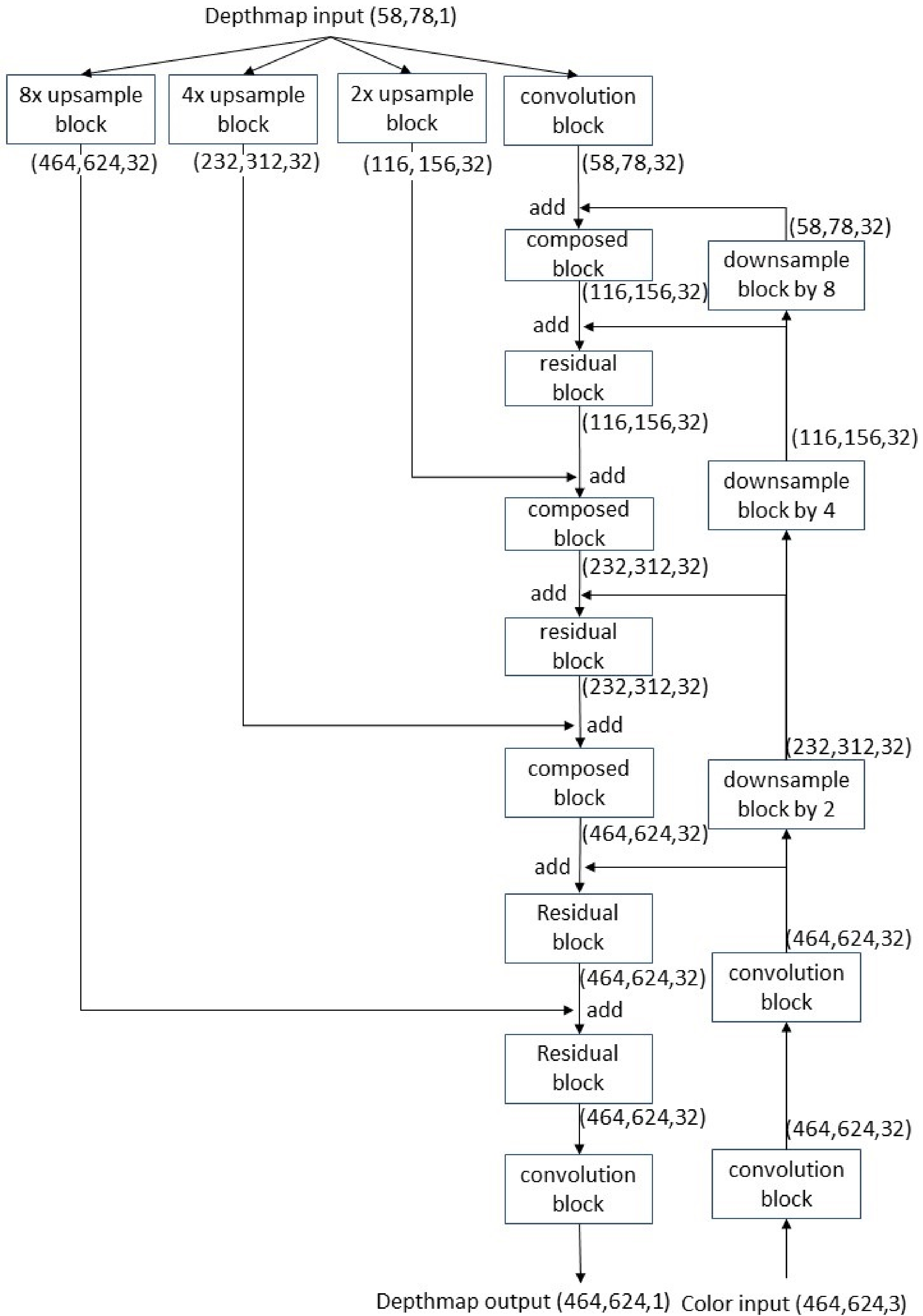


Figure 2.6: architecture of the 8x upsample network trained with NYU dataset

As can be observed from Figure 2.6 we have two branches, one primary, the depthmap branch, dealing with depthmap manipulation and the other one, the secondary, which deals with RGB images manipulation. In four different points the information belonging to the color branch are are fused together into the primary depthmap branch. We start describing the color branch, whose input is a color image of a scene we want to upsample. In this specific example the input tensor has a spatial resolution of 464x624 pixel and 3 channels. As can be observed from Figure 2.6 we start performing two subsequent convolution blocks with 32 filters each, in order to increase the number of feature extracted by the network. The tensor up to now has the dimension of 464x624 pixel and 32 channels. From this point on the color information always has a 32 channel dimension. Then we have the first max pooling block where the resolution of the color information is halved so the output tensor will be of (232,312,32) dimension. We have another max pooling block where the resolution of color information is reduced to 116x156 pixels. Again we have a max pooling block, the last one, where the spatial resolution become 58x78, matching the input depthmap resolution.

We continue the architecture presentation with the depth branch, starting from the input which has a spatial resolution of 58x78 pixel with only 1 channel. From here we have three skip branches and one main branch. The three skip branches are the upsample blocks we described before, each with a unique scaling factor. One skip branch duplicate the input resolution so the output will be of (116,156,32) dimension, since the residual net in the upsample block has a convolution with 32 filters. The second skip branch performs an upsample with a scaling factor of 4 so the output will be of (464,624,32) dimension. The last skip branch is the one operating the 8x upsample and the output will have a spatial dimension of 464x624 pixel with 32 channels. Its resolution matches the resolution of the input color used for the color branch we described above. As you can see from 2.6 the upsample and downsample network are not built symmetrically; in particular, the max pooling operation on the color information are performed subsequently, i.e. the output of the first downsample block is the input of the next one, the subpixel convolution are carried out independently one from each other. We apply different scaling factors starting from the same input. If we had a stack of upsample operation we would lose much more information because the starting depthmap would be subjected to a lot of manipulation; furthermore it is easier fuse the information belonging to these skip branch into the main one when it is needed.

Now we explain the main branch, where all the information coming from the other branches combine together. Starting from the same low resolution depthmap we need a convolution block with 32 filters in order to increase the number of parameters and the

channel when later on we need to perform the add operation with the color information. We have our first fusion operation where we add the output of the last convolution block, which has the same spatial resolution as the depthmap input, to the output of the color branch, the downsampled RGB image of the scene.

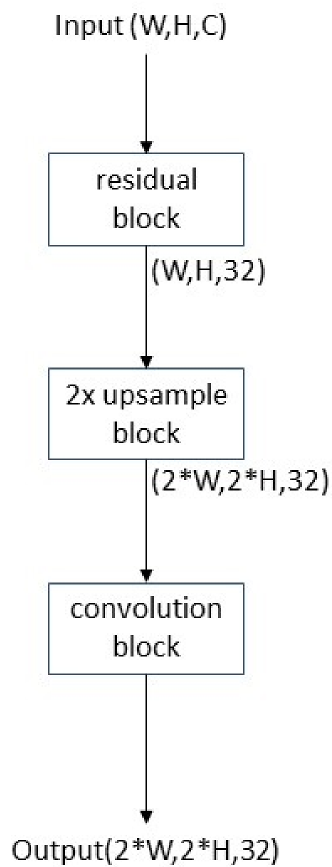


Figure 2.7: structure of the "composed block"

Then we have our first "composed block", which, as you can see from Figure 2.7, is made up of one residual block, one upsample block with a two scaling factor, which performs the depth to scale function we discussed before, and one last convolution block so the output

of the whole "composed block" has a dimension of (116,156,32). Now we have the second fusing point where we add the tensor coming from second downsample of the RGB image to the tensor we just outputted from the main branch. We use the outcome of this last adding operation as input for the next residual block and then we add information from the depth skip branch performing the upsample operation with the two scaling factor. At this point we conclude all the steps needed for the 2 scaling factor upsample. We start again with the "composed block" which gives us a tensor of (232,312,32). We have the sum of it with the tensor of the same dimension coming from the color branch and after performing a residual operation we fuse also the information coming from the clean 4x upsampled block of the depth skip branch. We have on remaining "composed block" which, after a residual block, performs the last subpixel convolution and fuse the information coming from the color and the depth skip branch, resulting in a tensor of (464,624,32) dimension. We perform the last residual network and finally we conclude with a convolution layer of only one filter, in order to have one channel in the output tensor and apply a sigmoid activation function. This function takes any real value as input and outputs values in the range of 0 to 1. The larger the input, the closer the output value will be to 1.0, whereas the smaller the input, the closer the output will be to 0.0, as shown below.

Figure 2.8: sigmoid activation function

This function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

Now we present the 4x scaling architecture, showed in Figure 2.9.

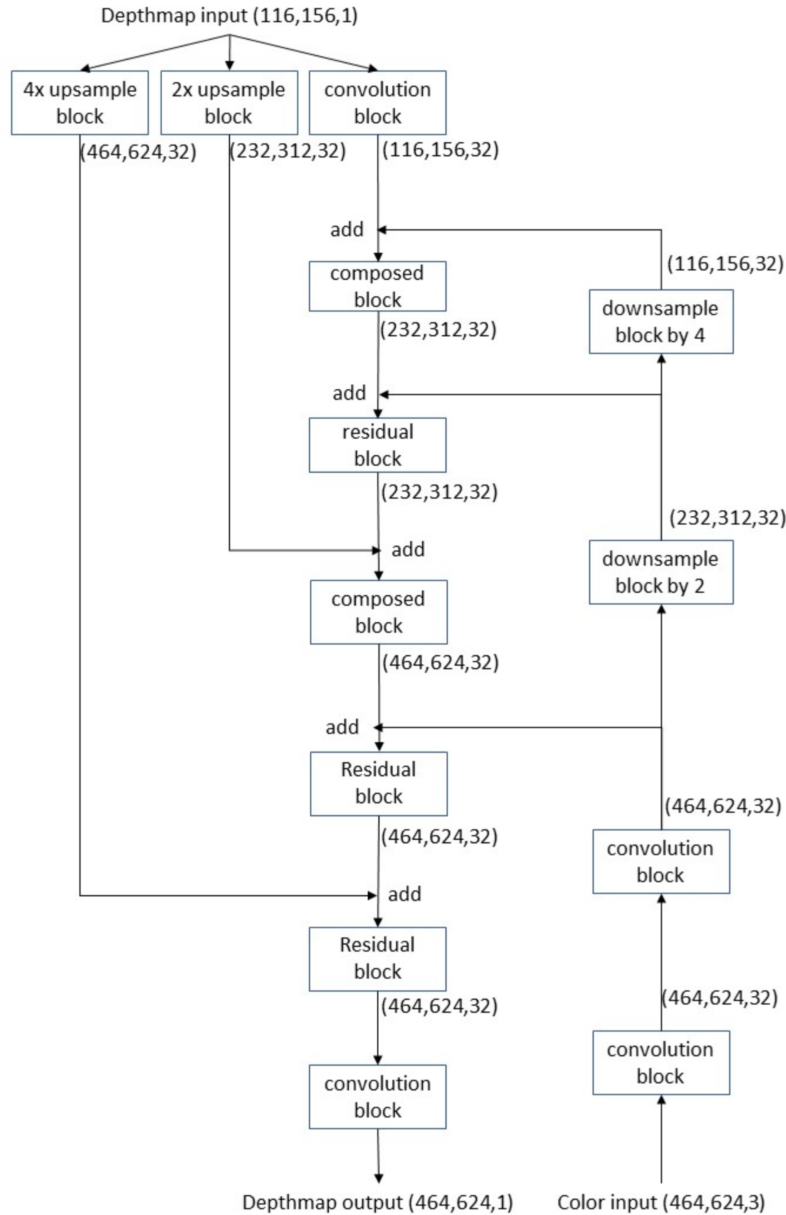


Figure 2.9: architecture of the 4x upsample network trained with NYU dataset

The main structure is similar to the one of the 8x architecture, with some differences. Since the network performs a 4x upsample, in order to later make coherent performance comparison, we decided to maintain the same resolution output and consequently we needed to carry out a downsample operation with bicubic interpolation to obtain an input for the network with resolution of 116x156. As before, we have two branches,the depthmap branch, dealing with depthmap manipulation and the other one,at the most right in Figure 2.9, dealing with RGB processing. Since this network has one layer less if compared to

the 8x, we have five points where depth and color information merges together. Starting from the color branch, its input is a tensor of (464,624,3) dimension; we perform two subsequent convolution blocks with 32 filters each, in order to increase the number of feature extracted by the network, so we have a tensor with same resolution as input but 32 channels. Then we have the first max pooling block where the resolution of the color information is halved so the output tensor will be of (232,312,32) dimension. We now have the second and last downsample block whose output has a spatial resolution of 116x156 pixel, matching the input depthmap resolution, and 32 channels. We continue starting from the depthmap input which has a spatial resolution of 116x156 pixel and 1 channel. From here we have two skip branches and one main branch. The two skip branches are the upsample blocks, carrying out upsample operation with a scaling factor of two and four and having respectively a tensor output of (232,312,32) and (464,624,32) dimension. Also in this scenario we maintained the asymmetry between the max pooling operation related to the color information and the upsample operation performed on the depth skip branches. Now we present the main branch; starting from the low resolution depthmap we perform a convolution block with 32 filters to increase the number of parameters and channel when later on we need to perform the add operation with the color information. Then we have our first fusion operation where we add the output of the last convolution block, which has the same spatial resolution as the depthmap input, to the output of the color branch, the downsampled RGB image of the scene. Then we have our first "composed block", which, performs one residual block, one upsample block with a two scaling factor, and one last convolution block so the output of the whole "composed block" has a dimension of (232,312,32). Now we have the second fusing point where we add the tensor coming from second downsample of the RGB image to the tensor we just outputted from the main branch. We use the outcome of this last adding operation as input for the next residual block and then we add information from the depth skip branch performing the 2x upsample. At this point we have just completed the x2 upsample depthmap. We carry out the last "composed block" with its belonging subpixel convolution which gives us a tensor of (464,624,32). We have the sum of it with the tensor of the same dimension coming from the last convolution block of the color branch and after performing a residual operation we fuse also the information coming from the 4x upsampled block of the depth skip branch. We perform the last residual network and finally we conclude with a convolution layer of only one filter, in order to have one channel in the output tensor and apply a sigmoid activation function. This last convolution block replaces the flatten operation we usually find at the end of a convolution nerual network.

2.0.2. Dataset

During the training process and the later testing phase of our network, we used three different datasets, one containing synthetic images and the other two containing real images. The first is MVS-Synth Dataset [15] which is a synthetic photo-realistic dataset that consists of 120 scenes with 100 frames of urban outdoor settings captured in the video game Grand Theft Auto V. Using a synthetic dataset allows us to have better accuracy in higher resolution depth maps which are hard to find since the sensors are expensive. Even the datasets claiming to have higher resolution depth maps are usually preprocessed upsampled images that have visible artifacts or noise. Compared to other synthetic datasets, MVS-Synth Dataset is more realistic in terms of context and shading, and compared to real-world datasets, MVS-Synth provides complete ground truth disparities which cover regions such as the sky, reflective surfaces, and thin structures, whose ground truths are usually missing in real-world datasets. The original resolution of the maps is 960x540, these images are first cropped to 640x480 since the numbers divisible by more numbers which are helpful when considering integer scaling factors. These images are downsampled by a factor of 4 using OpenCV's inter-area interpolation, this procedure involves taking the average of the area when producing the down sampled pixel. This allows smoother edges with Moire free pixels. A total of 1200 images are selected from 120 scenes. The images are picked at a fixed interval (every 10 frames) to provide maximum differentiation in frames. Then 8 of these scenes are picked at random to be supplied as validation set. In total 1020 depthmaps are used in the training phase and 180 depth maps are used in validation. The downsampled LR images and original HR images are then fed to the neural network for training. For the 8x scale network the original images are first resized to 512x341 then cropped to 512x256 to produce the ground truth. Then these images are resized to 64x32 for the input dataset. The RGB images are also downsampled using the same method and separated in the same manner as the depth maps. For validation we are using 180 images selected from the same dataset, containing the frames from the remaining 18 scenes that were not used in the training phase. These images are again from urban scenes, so they are similar to the training set which keeps the training error comparable to the test error. In the Figures 2.10, 2.11, 2.12 below, we show an example from the MVS-Synth dataset, one low resolution depthmap used as network input, the corresponding high resolution groundtruth and the corresponding RGB image of the scene.

Figure 2.10: low resolution image from synthetic dataset used as input

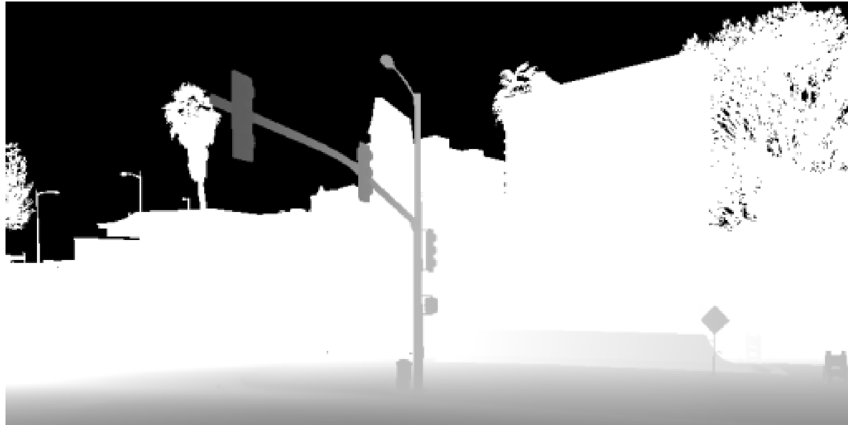


Figure 2.11: high resolution image from synthetic dataset used as groundtruth

Figure 2.12: RGB image from synthetic dataset used as guide image

There is though one important noteworthy clarification we need to do. Since we use a sigmoid as last activation layer we performed a normalization preprocess operation before feeding the network. As you can see from the image above, since we are dealing with outdoor scenes, we decided to compute a clamping operation: all the object that has an indefinite long distance, i.e. the sky in the background, has a value we fixed at 0 which is translated into a black color. This also helps creating a strong contrast to make the silhouette of long distance object emerge with respect to the background.

The second is the NYU dataset [36]. It consists of 1449 RGBD images, gathered from a wide range of commercial and residential buildings in three different US cities, comprising 464 different indoor scenes across 26 scene classes such as bathrooms, kitchens, libraries, living rooms, etc. and their corresponding depth images and semantic segmentation maps. A dense per-pixel labeling was obtained for each image using Amazon Mechanical Turk. If a scene contained multiple instances of an object class, each instance received a unique instance label, e.g. two different cups in the same image would be labeled: cup 1 and cup 2, to uniquely identify them. The dataset contains 35,064 distinct objects, spanning 894 different classes. For each of the 1449 images, support annotations were manually added. Each image's support annotations consists of a set of 3 tuples: [R_i,R_j,type] where R_i is the region ID of the supported object, R_j is the region ID of the supporting object and type indicates whether the support is from below (e.g. cup on a table) or from behind (e.g. picture on a wall). Examples of the dataset are found in Figures 2.13, 2.14, 2.15 (object category labels not shown). Additionally, it has a raw version consists of over 400k images and corresponding depth maps coming from the video recording of the same indoor environments. The original high resolution RGB images and the high resolution depthmaps used as label for the supervised training of the network show a white frame. This could be source of a problem for the network in the learning process and when computing the indexes for quantitative evaluation. Since these white pixel are at the four corner of the image, when we do upsample operation, some of the new pixel values could wrongly be influenced by them and consequently when we compute some measure indexes, their values are badly effected by these white pixel which does not reflect the real scene. In order to deal with this we decided to crop all those images from 640x480 pixel to 624x464 pixel and then we could perform the downsample operation, using bicubic algorithm, to obtain the 58x78 pixel low resolution depthmaps fed to the network as input. The low resolution depthmap and high resolution depthmap are saved as .npy files. As you might know, the "industry standard" with regard to data-files is .csv files. Now while convenient, these files are highly unoptimized when compared to the alternatives, like the .npy files. Npy files are the best choice we dealing with data science, i.e. loading data from npy is order of magnitude faster than reading the same data from a .txt file; another feature of using .npy files is the reduced storage the file occupies. In the Figures 2.13, 2.14, 2.15 below, we show an example from the NYU dataset, the first one is the low resolution depthmap , the second one is the corresponding high resolution groundtruth and the last one is the corresponding RGB image of the scene.

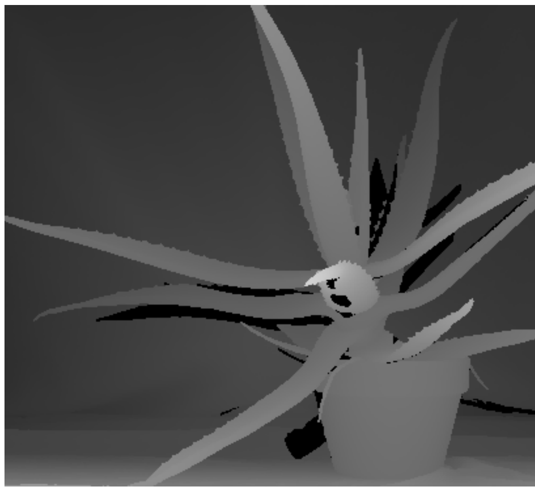
Figure 2.13: low resolution image from real dataset used as input

Figure 2.14: high resolution image from real dataset used as groundtruth

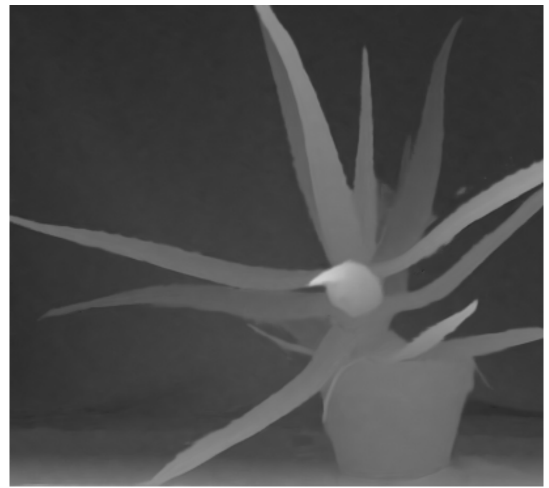
Figure 2.15: RGB image from real dataset used as guide image

If The first two datasets we presented above are used either in training and in testing phase, the third one we are about to present is related only to the testing phase of the

network trained with the NYU dataset [36]. We use a subset of 30 RGBD images coming from Middlebury dataset. The original dataset though was presented in [33], [34], [12]. D. Scharstein et al. described a method for acquiring high-complexity stereo image pairs with pixel-accurate correspondence information using structured light. Unlike preceding range-sensing approaches, their method does not require the calibration of the light sources and yields registered disparity maps between all pairs of cameras and illumination projectors. They presented new stereo data sets acquired with their method and demonstrated their suitability for stereo algorithm evaluation. As you can see from Figure 2.16a, their depthmap are not so useful for our purpose though. They present multiple black areas around the leaves and near the vase, which are not actually part of the scene and worsen the overall quality. For that reason, in our testing phase we used images provided by [28]. Depth captured by consumer RGB-D cameras were often noisy and missed values at some pixels, especially around object boundaries. Most existing methods completed the missing depth values guided by the corresponding color image. When the color image is noisy or the correlation between color and depth is weak, the depth map cannot be properly enhanced. In their paper, Lu et al. present a depth map enhancement algorithm that performs depth map completion and denoising simultaneously. Their method is based on the observation that similar RGB-D patches lie in a very low dimensional subspace. Similar patches can be assembled into a matrix and enforce the low rank subspace constraint. This constraint captures the underlying structure in the RGB-D patches and enables robust depth enhancement against the noise or weak correlation between color and depth. Their method formulates depth map enhancement as a low rank matrix completion problem. Since the rank of a matrix changes over matrices, we develop a data driven method to automatically determine the rank number for each matrix. Their experiments on both public benchmark and their own captured RGB-D images showed effectively enhanced depth maps. You can see from Figure 2.16b, their method has a huge impact on the noise present in the starting depthmap on the left.



(a) sample of a clean output



(b) filtered output sample

Figure 2.16: comparison between a clean sample and the same sample,filtered, belonging to Middlebury dataset

2.0.3. Training

Once defined the architecture and used a matlab script in order to preprocess the data, the model must be compiled using the model object obtained with the Model class. This is implemented with the following code:

```
opt = Adam(lr = learning_rate, beta_1 = 0.9, beta_2 = 0.999,
   epsilon = None, amsgrad = False)
architecture.compile(optimizer = opt, loss = custom_loss,
   metrics = ["accuracy"])
```

In the compile function the optimizer and the loss function are specified. In particular, we will use the Adam optimizer and a custom loss function we explain later on. The name Adam is derived from adaptive moment estimation. Kingma et al. in the [20] paper, presented Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and parameters. The method is also appropriate for non stationary objectives and problems with very noisy and sparse gradients. Empirical results demonstrated that Adam works well in practice and compares favorably to other stochastic optimization method. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; An image showing that Adam optimizer outperforms the other optimizer is reported in Figure 2.17, it makes faster progress in terms of both the number of iterations and wall-clock time.

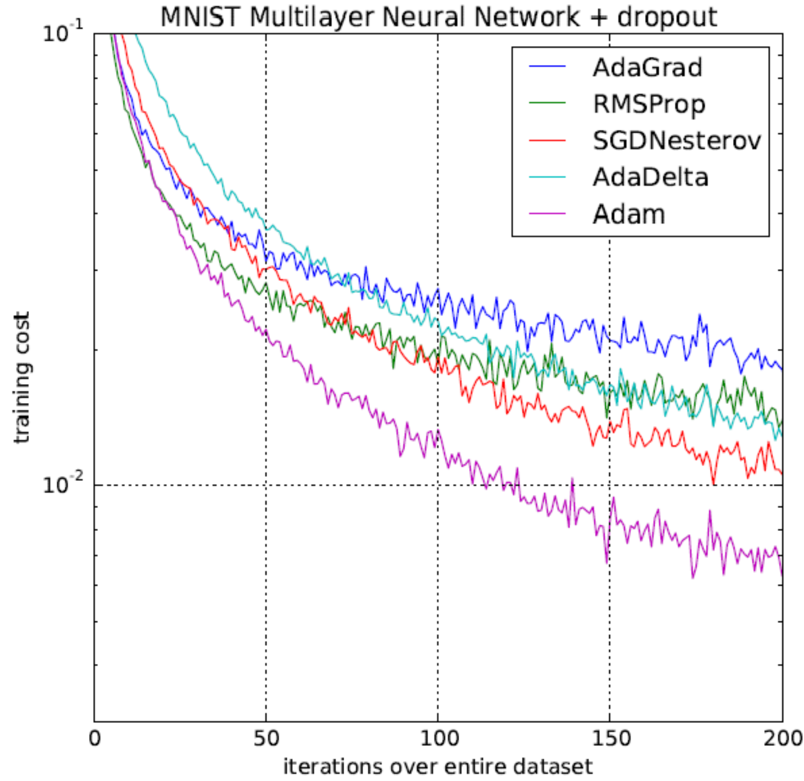


Figure 2.17: comparison showing the performance of Adam optimizer

The arguments of the adam optimizer are the followings: the learning rate or step size, which is the proportion that weights are updated, larger values results in faster initial learning before the rate is updated, smaller values slow learning right down during training; beta1 is the exponential decay rate for the first moment estimates; beta2 is the exponential decay rate for the second moment estimates, this value should be set close to 1.0 on problems with a sparse gradient; epsilon is a very small number to prevent any division by zero in the implementation; decay ; amsgrad is a boolean whether to apply AMSGrad variant of this algorithm or not. Our custom loss is showed below:

$$L1(y_{true}, y_{pred}) = \sum_{i=1}^{n} |(y_{true} - y_{pred})| \tag{2.1}$$

$$Loss(y_{true}, y_{pred}) = L1(y_{true}, y_{pred}) + alpha * DSSIM(y_{true}, y_{pred}) \tag{2.2}$$

We use the combined action of two loss functions: L1 and DSSIM. The L1 loss, also known as Absolute Error Loss, is the absolute difference between a prediction and the

actual value, calculated for each example in a dataset. The aggregation of all these loss values is called the cost function, where the cost function for L1 is commonly Mean Absolute Error (MAE). The Structural Similarity Index Measure (SSIM) is a method for predicting the perceived quality of digital television and cinematic pictures, as well as other kinds of digital images and videos. The SSIM index is a full reference metric; in other words, the measurement or prediction of image quality is based on an initial uncompressed or distortion free image as reference. SSIM is a perception-based model that considers image degradation as perceived change in structural information, while also incorporating important perceptual phenomena, including both luminance masking and contrast masking terms. The difference with other techniques is that these approaches estimate absolute errors. Structural information is the idea that the pixels have strong inter-dependencies especially when they are spatially close. These dependencies carry important information about the structure of the objects in the visual scene. Luminance masking is a phenomenon whereby image distortions (in this context) tend to be less visible in bright regions, while contrast masking is a phenomenon whereby distortions become less visible where there is significant activity or texture in the image. As you can see from the code below, we use a different version of the SSIM loss, the Data Structural Similarity Index Measure (DSSIM), which can be applied directly to the floating point data, since we normalized all input data before feeding the net on them. We did not apply this loss on the whole picture but first we extract some patches from it and then we compute the ssim. This is done at batch level and once we compute the ssim on all the patches we compute the mean between them and obtain a final loss value for that batch. We decided to use different loss function because they care about different aspects of the difference between the true value of the high resolution depthmap and the predicted value from the network

```
def CalculateDSSIM(y_true, y_pred, kernel_size = 3, k1 = 0.01,
  k2 = 0.03, max_value = 1.0):
    # DSSIM, clipped between 0.0 and 1.0. Arguments: k1:
      Parameter of the SSIM (default 0.01), k2: Parameter of
      the SSIM (default 0.03),
    # kernel_size: Size of the sliding window (default 3),
      max_value: Dinamic Range of the image (default 1.0)

    # Bring tensors from range -1.0 - +1.0 in range 0.0 - +1.0
    y_true = (y_true + 1.0) / 2
    y_pred = (y_pred + 1.0) / 2
```

```
# Define the kernel and constants
kernel = [kernel_size, kernel_size]
C1 = (k1 * max_value) ** 2
C2 = (k2 * max_value) ** 2

# Reshape the inputs to work with the sliding window
y_true = K.reshape(y_true, [-1] + list(K.int_shape(y_pred)
    [1:]))
y_pred = K.reshape(y_pred, [-1] + list(K.int_shape(y_pred)
    [1:]))

# Extract image patches for y_true and y_pred
patches_pred = extract_image_patches(y_pred, kernel, kernel
    , 'valid', K.image_data_format())
patches_true = extract_image_patches(y_true, kernel, kernel
    , 'valid', K.image_data_format())

# Reshape the extracted patches to calculate statistics on
    each patch
bs, w, h, c1, c2, c3 = K.int_shape(patches_pred)
patches_pred = K.reshape(patches_pred, [-1, w, h, c1 * c2 *
     c3])
patches_true = K.reshape(patches_true, [-1, w, h, c1 * c2 *
     c3])

# Get mean, varianche and std dev of the patches
u_true = K.mean(patches_true, axis=-1)
u_pred = K.mean(patches_pred, axis=-1)
var_true = K.var(patches_true, axis=-1)
var_pred = K.var(patches_pred, axis=-1)
covar_true_pred = K.mean(patches_true * patches_pred, axis
    =-1) - u_true * u_pred
# Compute SSIM
ssim = (2 * u_true * u_pred + C1) * (2 * covar_true_pred +
    C2)
denom = ((K.square(u_true) + K.square(u_pred) + C1) * (
    var_pred + var_true + C2))
```

```
    ssim /= denom
    return K.mean(1.0 - ssim)
```

After the definition of the model it is necessary to train it model using the fit function. This part is done with the following code:

```
architecture.fit_generator(training_generator, steps_per_epoch
    = len(training_id), validation_data = validation_generator,
    epochs = epoch, callbacks = [history, checkpoint_save])
```

Data are loaded exploiting fit generator function of Keras. By doing so we could keep high the number of learnable parameters without the drawback of limited performance, i.e you can easily deal with the memory video saturation because you don't need to preload all data you need but you can generate them at run time every time you need them. Furthermore it is useful if you need to perform some data augmentation like in our case as we will show you later on. The parameter of the generator are the followings: the training generator is obtained through the DataGenerator class you see below. The method we highlight is "data generation" where the actual generation of data is performed. Here we have the load of input data and the transformation we want to apply. In our case we have two random bit generators to perform either a random vertical and horizontal flip, a random int generator representing the pixel we cut out to perform a crop and consequently a zoom operation.

```
DataGenerator(keras.utils.Sequence):
        'Generates data for Keras'
        def __init__(self, list_IDs, batch_size=1, shuffle=True
            ):
                'Initialization'
                self.batch_size = batch_size
                self.list_IDs = list_IDs
                self.shuffle = shuffle
                self.on_epoch_end()

        def __len__(self):
                'Denotes the number of batches per epoch'
                return int(np.floor(len(self.list_IDs) / self.
                    batch_size))

        def __getitem__(self, index):
```

```
        'Generate one batch of data'
        # Generate indexes of the batch
        indexes = self.indexes[index*self.batch_size:(
           index+1)*self.batch_size]

        # Find list of IDs
        list_IDs_temp = [self.list_IDs[k] for k in
           indexes]

        # Generate data
        [depthmap_input_train, color_train],
           depthmap_label_train  = self.
           __data_generation(list_IDs_temp)

        return [depthmap_input_train, color_train],
           depthmap_label_train


def on_epoch_end(self):
        'Updates indexes after each epoch'
        self.indexes = np.arange(len(self.list_IDs))
        if self.shuffle == True:
                np.random.shuffle(self.indexes)


def __data_generation(self, list_IDs_temp):
        'Generates data containing batch_size samples'
           # X : (n_samples, *dim, n_channels)
        # Initialization
        depthmap_input_train = np.empty((self.
           batch_size, 58, 78, 1), dtype=np.float32)
        depthmap_label_train = np.empty((self.
           batch_size, 464, 624, 1), dtype=np.float32)
        color_train = np.empty((self.batch_size, 464,
           624, 3), dtype=np.float32)

        # Generate data
```

```
for i, ID in enumerate(list_IDs_temp):
        path_lr = "/media/Data1/dpalesano/
           dataset2.0/dataset_v_flipped/train/
           depth_lr/" + str(ID) + ".npy"
        path_hr = "/media/Data1/dpalesano/
           dataset2.0/dataset_v_flipped/train/
           depth_hr/" + str(ID) + ".npy"
        path_color = "/media/Data1/dpalesano/
           dataset2.0/dataset_v_flipped/train/
           color/" + str(ID) + ".jpg"

        depthmap_input_train_current = np.load(
           path_lr)
        depthmap_label_train_current = np.load(
           path_hr)
        color_train_current = cv2.imread(
           path_color)
        color_train_current = np.asarray(
           color_train_current, dtype=np.
           float32)

        #trasformazioni
        crop_size_lr = randint(0, 3)
        crop_size_hr = 8*crop_size_lr
        h_flip_flag = getrandbits(1)
        v_flip_flag = getrandbits(1)


        depthmap_input_train_current =
           horizontal_flip(
           depthmap_input_train_current,
           h_flip_flag)
        depthmap_input_train_current =
           vertical_flip(
           depthmap_input_train_current,
           v_flip_flag)
        depthmap_input_train_current = np.
```

```
                    expand_dims(
                    depthmap_input_train_current, axis =
                     3)
                depthmap_input_train[i,] =
                    depthmap_input_train_current

                depthmap_label_train_current =
                    horizontal_flip(
                    depthmap_label_train_current,
                    h_flip_flag)
                depthmap_label_train_current =
                    vertical_flip(
                    depthmap_label_train_current,
                    v_flip_flag)
                depthmap_label_train_current = np.
                    expand_dims(
                    depthmap_label_train_current, axis =
                     3)
                depthmap_label_train[i,] =
                    depthmap_label_train_current

                color_train_current =
                    color_train_current / 255
                color_train_current = horizontal_flip(
                    color_train_current, h_flip_flag)
                color_train_current = vertical_flip(
                    color_train_current, v_flip_flag)
                color_train[i,] = color_train_current

        return [depthmap_input_train, color_train],
            depthmap_label_train
```

"Steps per epoch" parameter is the total number of steps (batches of samples) to yield from generator before declaring one epoch finished and starting the next epoch. It should typically be equal to ceil of the number of samples divided by the batch size. "Validation generation" is the same as "training generator" but for the validation dataset. At last, a list of callbacks is passed to the fit function. The callbacks are a set of actions which can

be performed at different stages of the training. For instance, at the beginning or at the end of an epoch, it is possible to perform a specific action. Keras provides some useful basic callbacks such as ModelCheckpoint, EarlyStopping and CSVLogger [12]. However, Keras allows also the possibility to build custom callbacks depending on our needs. We will now see in more details the configured callbacks used:

```
class CustomCallback(Callback):
    def on_epoch_end(self, epoch, logs = None):
        history_file.write("{},{},{} \n" .format(epoch, logs.
           get("loss"), logs.get("val_loss")))
history = CustomCallback()

best_epoch = dir_name + "/checkpoint" + "/best_validation_loss:
   epoch=" + "{epoch:02d}-valoss={val_loss:.9f}" + ".hdf5"
checkpoint_save = ModelCheckpoint(best_epoch, save_best_only=
   True, monitor='val_loss', mode='min', verbose = 1)
```

"CustomCallback" is a custom class with the purpose of saving in a CSV file the epoch results. The file contains for each epoch the information related to the training such as the training and validation losses. The "ModelCheckpoint" is a callback used to save the model or weights in a checkpoint file. It provides many advantages since it saves the state of the system. So, if the training is stopped for a certain reason it can be resumed thanks to the checkpoint file. This callback in our case is defined such that it saves the model with the best performance by monitoring the validation loss. The model corresponding to the minimum validation loss is saved each time a lower validation loss is encountered during the training. We save all the models which are currently the best with its information about the epoch which the model belongs and the corresponding validation loss.

During the training phase, for each epoch, two values are reported: the training and the validation loss. The training loss, in practice, is the loss that we want to minimize to optimize the parameters of our model. The validation loss, instead, is the loss calculated on the validation set, that is used to understand if the model is overfitting the training set. If this is the case, we have to check if both the training and validation loss keep decreaseing. If we see that the losses do not decrease anymore we are probably close to the convergence and we may stop the training. Since the "ModelCheckpoint" is used, the model with the lowest validation loss is saved. The training and validation losses can be plotted into a graph using the CSV file saved thanks to the "plot drawing" function we call once the training ends. You can see the code below:

```
def plot_drawing (csv_file, saving_path):
        epoch = np.loadtxt(csv_file, dtype = int, delimiter = "
           ,", skiprows = 1, usecols = 0)
        loss = np.loadtxt(csv_file, dtype = float, delimiter =
           ",", skiprows = 1, usecols = 1)
        validation_loss = np.loadtxt(csv_file, dtype = float,
           delimiter = ",", skiprows = 1, usecols = 2)

        plt.title("Metrics", fontsize = 14)
        plt.xlabel("epoch", fontsize = 14)
        plt.ylabel("losses", fontsize = 14)
        plt.plot(epoch, loss, linewidth = 2.0, label = 'loss')
        plt.plot(epoch, validation_loss, linewidth = 2.0, label
            = "validation_loss")
        plt.legend()
        plt.savefig(saving_path + "/plot.png", dpi = 300)
```

In Figure 2.18, an example of loss function is shown. In particular, it is the one related to the training of the 8x network architecture training used the NYU dataset, using the horizontal and vertical flip and the zoom operations as data augmentation. The blue line stands for the training loss and the orange line stands for the validation loss. They are quite overlapped meaning that the network is behaving well. Even though the curve towards the 200th epoch seems straight, it is not. The loss keeps decreasing but in a very slow way, this means the network has a margin of improvement, i.e. applying a fine tuning operation modifying the learning rate.

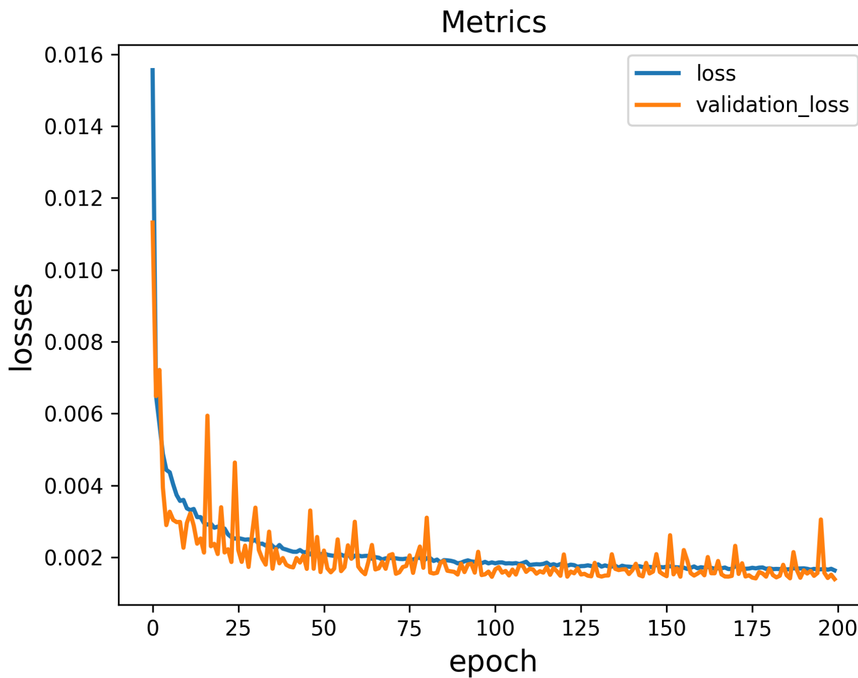


Figure 2.18: example of a plot of loss function for training and validation dataset

2.0.4. Parameters

The training phase of the network is what determine the success of the whole process. Even though the architecture we chose to model the network upon has an important role, tweaking the hyperparameter has a critical role because they can strongly modify the network performance very easily. After several trials, a filter size of 32 is selected, creating a wide network together with the branches, although it increases the training time after 70 epochs it achieves better results than the 16 filter model. The selection of learning rates like any other network is crucial here. From the training results, we can observe that the error surface has lots of error peaks in high parameter numbers. This can be the result of the network's residual nature. Since usually the low frequency details are easy to reproduce from the LR image, the residual part is usually focused on the high frequency details this might entail the network to produce tons of high frequency details that do not exist in the image resulting in high error. This means that the network needs to be trained using a lower learning rate as the network gets deeper and wider. It can be observed from the Figure 2.19 below, that a bad learning rate reverses the training stage by producing a huge error peak. The network is able to get back to the previous point of error in 5-6 epochs, which renders the training before that point ineffective. This is not easy to avoid even in smaller learning rates as you can see from the relatively small

peak on the blue line. Yet in smaller learning rates it is manageable and returns to the previous error in 1 or 2 epochs.

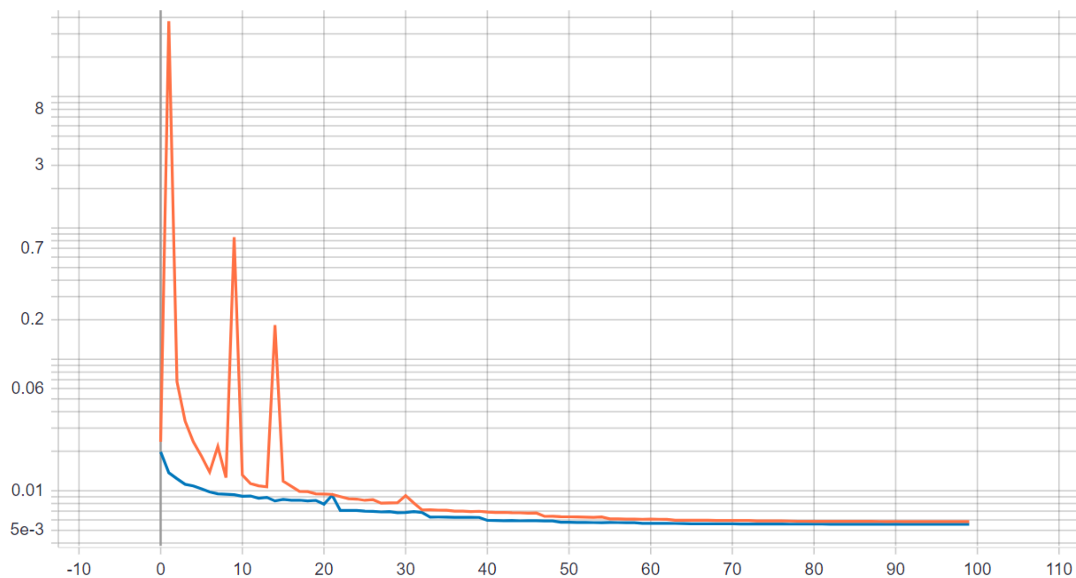


Figure 2.19: The error trend line in a logarithmic scale, epochs in x-scale, mean-absolute error in y-scale, orange line is for a larger learning rate (1×10^{-3}), while the blue line is for the smaller learning rate (2.5×10^{-4})

We found 2.5×10^{-4} to be a good value as starting learning rate value. When we need to perform a finetuning over the network we usually take small steps and keep halving the initial learning rate value until we cannot get an improvement in the qualitative index we will show you later. We anticipate that in all our test this process has not been necessary more than two times and we have obtained always a little upgrade in term of percentage, meaning that the starting value for the learning rate was actually pretty good.

Epoch is another parameter we need tweak the best way in order have a good balance between the time of training and the quality of the output. Having a huge number of epochs does not necessarily mean that training will give us back a good result. Depending also on the learning rate, training for too long could lead to missing the loss, bouncing back and forth the minimum we are looking for but never reaching it, or even exploding the value, meaning that we will keep getting away from the minimum, increasing the loss even more and never finding the minimum. Based on all the test we run, we came up with a sweet spot of 200 epochs for a regular training and additional 100 epochs when it comes to finetuning.

Also for alpha, the multiplier factor for the DSSIM loss we previously introduced, we needed to run some test in order to find the right value that suits our network. We

started from a little value: $1x10^{-2}$ and took incremental steps with a little increase each time. We use the Root Mean Square Error(RMSE) index to measure the performance of the network. We found that the maximum value until we get a worsening is $1x10^{-1}$.

Bacth size is another essential parameter to find the best value for. It has a strong link with the learning rate and the number of epoch. Practitioners often want to use a larger batch size to train their model as it allows computational speedups from the parallelism of GPUs. However, it is well known that too large of a batch size will lead to poor generalization. For functions that we are trying to optimize, there is an inherent duality between the benefits of smaller and bigger batch sizes. On the one extreme, using a batch equal to the entire dataset guarantees convergence to the global optima of the objective function. However, this is at the cost of slower, empirical convergence to that optima. On the other hand, using smaller batch sizes have been empirically shown to have faster convergence to good solutions. This is explained by the fact that smaller batch sizes allow the model to start learning before having to see all the data.

It has been empirically observed that smaller batch sizes not only has faster training dynamics but also better generalization to the test dataset versus larger batch sizes. It is generally accepted that there is some sweet spot for batch size between 1 and the entire training dataset that will provide the best generalization. This sweet spot usually depends on the dataset and the model at question. The reason for better generalization is attributed to the existence to noise in small batch size training. Because neural network systems are extremely prone overfitting, the idea is that seeing many small batch size, each batch being a noisy representation of the entire dataset, will prevent the neural network from overfitting on the training set and hence performing badly on the test set. Different experiment has been conducted to test the relationship between the batch size and the other parameters. It emerged that large batch size means the model makes very large gradient updates and very small gradient updates. The size of the update depends heavily on which particular samples are drawn from the dataset. On the other hand using small batch size means the model makes updates that are all about the same size. The size of the update only weakly depends on which particular samples that are drawn from the dataset. It has been found that the reason smaller batch sizes train more efficiently for the same number of epoch is because it takes more steps. For a given number of steps, it seems like there's an upperbound on how far the model can travel away from its original weights. Therefore, smaller batch sizes means the model can find the faraway better optima whereas large batch size means the model cannot. However, there are cases where the model travels just as far with a large batch size as a small batch size but still does worse than smaller batch size. With the other parameter fixed, we decided to maintain

the batch size equal to 1 in excahnge of a longer time of training, since it gives us the best performance.

2.0.5. Testing

Once the model is properly trained, it can be used to predict new data. In our case the model is trained so that it is able to perform the upsample of new depthmap the network has never seen before. The testing python script reported below shows the whole testing phase which is not compose of the predict instruction only.

```
for depth_file_name in sorted(glob.glob('/media/Data1/dpalesano
   /dataset2.0/dataset_v_flipped/test/color/*jpg')):
        #save low resolution input image
        input_image = depthmap_test[index]
        file_name = dir_name + "/input/depth_input_" +
           depth_file_name[63:]
        cv2.imwrite(file_name, input_image * 255)

        #save high resolution input label
        truth_image = depthmap_label_test[index]
        file_name = dir_name + "/truth/ground_truth_" +
           depth_file_name[63:]
        cv2.imwrite(file_name, truth_image * 255)

        #save input bicubic interpolation
        bicubic_interpolation = cv2.resize(depthmap_test[index
           ], (464, 624), interpolation = cv2.INTER_CUBIC)
        file_name = dir_name + "/bicubic_interpolation/bicubic_
           " + depth_file_name[63:]
        cv2.imwrite(file_name, bicubic_interpolation * 255)


        #save high resolution output
        output_image = np.squeeze(best_architecture.predict([np
           .expand_dims(depthmap_test[index], axis = 0), np.
           expand_dims(color_test[index], axis = 0)]), axis =
           0)
        file_name = dir_name + "/output/depth_output_" +
```

```
        depth_file_name[63:]
    cv2.imwrite(file_name, output_image * 255)

    file_name = dir_name + "/output/matlab/depth_output_" +
         depth_file_name[63:-4]
    scipy.io.savemat(file_name, {"data" : output_image},
        appendmat = True)

    index = index + 1;
```

The predictions are performed on a pool composed of the ten percentage of the whole dataset we have taken out initially. We start by importing the .npy files corresponding to the low resolution and the high resolution depthmpas and the .jpg file corresponding to the high resolution color information of the input. Then, considering that we previously normalized the depthmaps, in order to feed the network with coherent data, i.e. having data in the range [0,1], we need to normalize also the RGB images and this is done by simply dividing all the pixel values by 255 since the the color images are encoded using 8 bit of information, so the minimum and the maximum value they can have are respectively 0 and 255. Then we cycle through the input testing dataset. Apart from the color information, the rest of the dataset is organized in .npy files; so we exploit this cycle to save as images the depthmap input and the groundtruth in order to make a qualitative comparison between the network outcome and how the high resolution version of the depthmap actually is. We also saved an upsampled version of the input depthmap using bicubic interpolation to make a qualitative comparison between the convolution network upsample and one among the standard upsample algorithms we described in the introduction. Lastly we save the prediction of the network, the high resolution depthmap, as image for the reason we stated just before and as .mat files, so we are able to use Matlab software in order to perform the quantitative evaluation using the metrics we will present you in the next chapter.

3 | Results

In this chapter we will see the qualitative and quantitative results obtained by applying the deep learning super resolution methods seen in the previous chapter. We show the performance of either the x4 upsample network and the 8x upsample network, considering the training with the synthetic and the realistic dataset. Before proceeding directly on that, we have to firstly introduce the metrics used to compare the performance of the different models. All the considered metrics produce a value which quantify the errors between the upsample depthmap obtained using a software, either a standard one or a convolution netowrk, and a native high resolution depthmap used as reference. The metrics used are the following:

1. **Root Mean Squared Error (RMSE)**. It is the most simple and most used metric for image quality evaluation. It basically measures the root of the average squared error of the network output and reference image pixels. Let I1 and I2 be the two images and I1(m, n) and I2(m, n) the intensity value at the pixel in the row m and column n of the corresponding images. Also, let M and N be respectively the total number of rows and columns of each image. The MSE is defined as follows:

$$RMSE = \sqrt{\frac{\sum_{M,N}[I_1(m,n) - I_2(m,n)]^2}{MN}} \tag{3.1}$$

2. **Peak Signal to Noise Ratio (PSNR)**. It is a metric which measures the peak error exploiting the MSE value and tries to quantify the quality of the reconstructed image with the original image. The PSNR is expressed as follows:

$$PSNR = 10log_{10}(\frac{R^2}{MSE}) \tag{3.2}$$

   where R is the maximum fluctuation of the input intensity. This value is 1 in the case of float data type and 255 in the case of 8 unsigned integer data. The MSE in the formula is at the denominator of the ratio. As a result, the lower the MSE, the higher is the PSNR. The output value it returns is expressed in dB (decibel).

Therefore, a high PSNR is equivalent to have a high signal and a low noise. This means that the higher the PSNR, the more similar the two images.

3. **Structural Similarity Index Measure (SSIM)**. It is a metric that extracts from images 3 different information: the luminance, the contrast and the structure. For each of these information the corresponding term must be computed and their multiplicative combinations return as output the SSIM. Hence, the SSIM is expressed as follows:

$$SSIM = [l(x,y)]^{\alpha}[c(x,y)]^{\beta}[s(x,y)]^{\gamma} \tag{3.3}$$

where

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{3.4}$$

$$c(x,y) = \frac{2\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{3.5}$$

$$s(x,y) = \frac{\sigma_{\text{xy}} + C_3}{\sigma_x\sigma_y + C_3} \tag{3.6}$$

l(x,y), c(x,y), s(x,y) are respectively the luminance, contrast and structure terms. Instead, σ_x, σ_y, σ_{xy}, μ_x, μ_x are the standard deviations, the cross-covariance and the local means for the images x and y. Lastly, C_1, C_2, C_3 are constants which are included at the denominator of each of the three terms in order to avoid instability when the denominator becomes zero. If the exponents of the three terms α, β, γ are all equal to 1 and the $C_3 = C_2/2$, then the SSIM formula can be simplified as follows:

$$SSIM = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{\text{xy}} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{3.7}$$

The SSIM returns a value which ranges from 0 to 1. The closer this value gets to 1 and the better the quality of the image coming from the network, meaning that the latter one is almost identical to the reference one.

3.0.1. Quantitative Evaluation

We start presenting the result obtained using the model trained with the synthetic dataset. In the table 3.1 you can see the metrics related to the CNN using 8 and 4 as scaling factor. In this table we take into examination the nearest neighbor, the bilinear and the bicubic upsample algorithms. Since the MVS-Synth dataset provides us low resolution depthmap as .png images, we run a matlab script in order to compute the upsample process using nearest neighbor, bilinear and bicubic algorithms. Then, we also use a matlab script

to perform the index cmputation. Our netowrk outperforms the standards upsample algorithms by far in both cases. For what concerns the 8x upsample architecture we had an improvement of the 65% , 12% and 40% respectively on the RMSE, SSIM and PSNR metrics of our network compared to the bicubic method, which is the best between the three classical upsample methods. Speaking of the 4x architecture, since the resolution output is the same as the 8x scaling upsample network, but the upscaling the network will perform is different, we needed a different resolution input depthmap. We obtained it by using a bicubic downsample interpolation on the groundtruth depthmap, so we have our 128x64 pixel resolution depthmap input. In order to obtain the nearest neighbor, the bilinear and the bicubic interpolation, we did use, as before, a matlab script and then we computed the three indexes. In this case, as you can see, even the standard interpolation algorithms performs better than the x8 scaling factor case. That's because the upsample factor is smaller and consequently the information loss in the upsample process is reduced and less artifacts are produced. Also in this case our netowrk outperforms the standards upsample algorithms by far. We had an improvement of 60% , 7% and over the 35% respectively on the RMSE, SSIM and PSNR metrics of our network compared to the bicubic method. If we consider the overall percentage of improvement of the network output with respect to the network input, we notice that that x8 upsample architecture performs better than the x4 upsample architecture, demonstrating that our model is suited for this task.

MVS dataset	**8x upsample**			**4x upsample**		
	RMSE	**SSIM**	**PSNR**	**RMSE**	**SSIM**	**PSNR**
Nearest	0.0960	0.8452	20.5823	0.0674	0.9041	23.6919
Bilinear	0.0870	0.8540	21.4665	0.0614	0.9100	24.5334
Bicubic	0.0826	0.8669	21.9263	0.0573	0.9215	25.1587
Proposed Method	**0.0297**	**0.9733**	**31.0944**	**0.0210**	**0.9842**	**34.1971**

Table 3.1: Table comparing the metrics of the classical upsample algorithm and our convolution neural network for both the x8 and 4x upsample model, trained with MVS-Synth dataset

We report in table 3.2 the progress our model can perform if a finetuning operation is carried out. We exhibit each step we did to achieve the performance we showed in the 3.1 table above. We start by the simplest version of our network: 200 epochs of training, Mean Square Error(MSE) as loss function. Its performance are showed in the first row. Then we perform our first finetuning step where we introduce the combined action of two

losses, Mean Absolute Error(MAE) and the DSSIM we introduced before; we trained the network with this setting 100 times more. We gained a 5.6% on the RMSE, meaning that similiraty loss takes into account aspect that the MSE does not. We take a step further and add 100 supplementary epochs of training reducing the learning rate. The little improvement indicates that the network has reached its limit and it does not make sense to investigate even further with this configuration.

MVS dataset	RMSE	SSIM	PSNR
Single Loss	0.0316	0.9693	30.4971
Double Loss	0.0298	0.9719	31.0221
Halved Learning Rate	**0.0297**	**0.9733**	**31.0944**

Table 3.2: Table comparing the metrics of our 8x network going through each step of finetuning

Now we take into exam the model trained with the realistic indoor dataset NYU2. In the table 3.3 below, you can see the metrics related to the CNN using both 8 and 4 as scaling factor. In this table we take into examination the nearest neighbor, the biliniear and the bicubic upsample algorithms. Since the NYU V.2 dataset provides us the high resolution depthmap as .npy arrays, we run a matlab script in order to obtain the low resolution depthamp used as input for the network using a downsample bicubic algorithm. We also use a matlab script to compute the nearest, bilinear and bicubic upsamples. Starting from the 8x configuration, we perform the index computation. Also in this case our netowrk outperforms the standards upsample algorithms. We had an improvement of the 60% , 10% and 33% respectively on the RMSE, SSIM and PSNR metrics of our network compared to the bicubic method. Continuing with the 4x scenario analysis, we found a situation which is inline with the one trained with the synthetic dataset for what concerns the difference between the x4 upsample architecture and the one with x8 scaling factor. If we take the indexes measurement on their own, the one belonging to the 4x network are better then the 8x network. After the testing phase we achieved the following improvement: 61% on RMSE, 5% on SSIM and 24% on PSNR compared to the bicubic interpolation.

NYU dataset	8x upsample			4x upsample		
	RMSE	SSIM	PSNR	RMSE	SSIM	PSNR
Nearest	0.0932	0.7422	20.7804	0.0576	0.8742	25.1889
Bilinear	0.0740	0.8746	22.7875	0.0444	0.9362	27.4649
Bicubic	0.0676	0.8775	23.5404	0.0398	0.9423	28.4838
Proposed Method	**0.0284**	**0.9637**	**31.4272**	**0.0152**	**0.9847**	**35.1915**

Table 3.3: Table comparing the metrics of the classical upsample algorithm and our convolution neural network for both the x8 and 4x upsample model,trained with NYU dataset

Here in table 3.4 we show the results obtained in the testing phase,using the Middlebury dataset [28]. If for the NYU dataset, the average RMSE is measured in meters, on the other side for the Middlebury dataset the average RMSE is measured in the original scale of the provided disparity. We compared both our 4x and 8x architecture to other works previously published in this field which used [28] dataset as well in their testing phase, including recent learning based methods such as DJFR [24] and DKN [19]. Table shows that our method outperforms the existing one.

RMSE on Middlebury dataset	4x upsample	8x upsample
Bicubic	2.32	3.99
DSMG[16]	1.88	3.45
DG[9]	1.97	4.16
DJF[24]	1.68	3.24
DJFR[25]	1.32	3.19
PAC[37]	1.32	2.62
DKN[19]	1.23	2.12
Proposed Method	**1.19**	**1.97**

Table 3.4: Table comparing the RMSE index of our network trained with Middlebury dataset to some of the state of the art methods,both for the x8 and 4x upsample model

3.0.2. Qualitative Evaluation

In this section qualitative results are reported in order to validate what quantitative evaluation showed us. We start from results obtained in the 8x upsampling scenario,

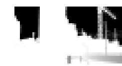
presented in Figure 3.1. In particular, in Figure 3.3a the HR color image is shown, while in Figure 3.1b the LR depth map is depicted. The result obtained with bicubic interpolation is reported in Figure 3.3b, while the groundtruth HR depth map is shown in Figure 3.2e. The output of the network is reported in Figure 3.1d. We can notice that deep learning based solutions show improvement over the bicubic interpolation. This can be seen by looking at the sharp edges of the central building silhouette. Also for what concerns the crane, its structure is as detailed as the groundtruth, even the wholes among its main core through which you can see the sky are sharp and does not appear obfuscated like in the bicubic interpolation output. Moreover, the result obtained exploiting the color information is much more accurate in high frequency details such as the crane and cables. This is due to the fact that the objects edges could be easily estimated by the network by analysing the color image input. Groundtruth depthmap presents a better representation of the bridge on the left,especially of the handrail. The RGB image of the scene does not help us that much since we have different colors without a huge contrast in the empty spaces and the network is not able to distinguish properly the handrail from the objects behind it. zone vicine con poco constrasto

We continue presenting you a comparison between the 8x and the 4x upsample architecture. As came out from the quantitative results showed above, the pure values of the indexes are better in the x4 scenario. The same outcome can be proved also by Figure 3.2. We have in Figure 3.3a the RGB image of the scene; Figure 3.3b shows the bicubic interpolation of the input depthmap. Then we have in Figure3.2c and Figure 3.2d respectively the output coming from x4 and x8 architectures and finally in Figure 3.2e we have the groundtruth depthmap. It is clearly visible that both the outputs, x4 and x8 upsample's, are better than the bicubic interpolation. The latter presents a blurry effect spread through the whole image, the edges of the objects, i.e. the buildings, the car and the stop light, lack of sharpness. In this case, though, we want to highlight that the output coming from the x4 architecture is better looking than the output coming from the x8 architecture. We notice that the silhouette of the pedestrian crossing the street is better defined, indeed, it appears like a poor defined shape in the 8x output. This happens also for the road signs on the car right and left, in the 8x netowrk their shape are not well recognisable, they don't have sharpe edges, they even merges with each others becoming indistinguishable, especially the ones on the left. In the x4 output they are well defined, even the one on the left, far from the camera, are distinct one from each others. Also the stop light on the left, in the 8x output presents some blurring effect in the zone near the coloured lights, where there is a space through which the building behind is visible.

In Figure 3.3 a comparison between our model 3.3e, the groundtruth 3.3f, bicubic in-

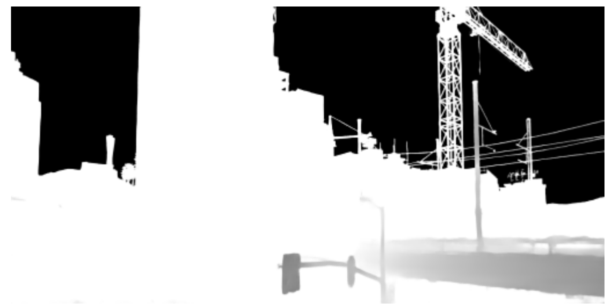
terpolation 3.3b and other previous works DKN 3.3c and DJFR 3.3d is conducted. We used a plasma colormap instead of the usual grayscale to better emphasize the contrasts in the depthmaps, since the low range of measurement in the indoor scenes. We used the model trained with the NYU dataset. We want to call the attention to the details we highlighted within the red and green rectangles. These are related to the sides of a night table near the sofa, which present empty gaps and so a sharp change in the depth information. As can be seen, going from the bicubic interpolation to our method, a huge improvement takes place; the blurry effect reduces and the edges become sharper. This happens because in the other works a mean between the depth values belonging to the night table and the ones belonging to the floor behind is carried out. In our work the color information allows the network to assign each pixel a more precise value.

(a) Input RGB (512 x 256)

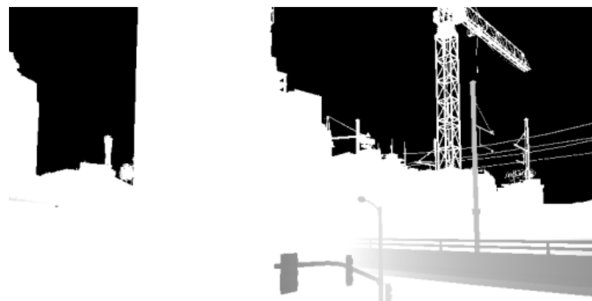


(b) Input depthmap (64 x 32)

(c) Bicubic interpolation (512x256)



(d) Output depthmap (512x256)



(e) Groundtruth depthmap (512x256)

Figure 3.1: 8x upsampling results obtained on one sample belonging to the test set

(a) Input RGB (512 x 256)

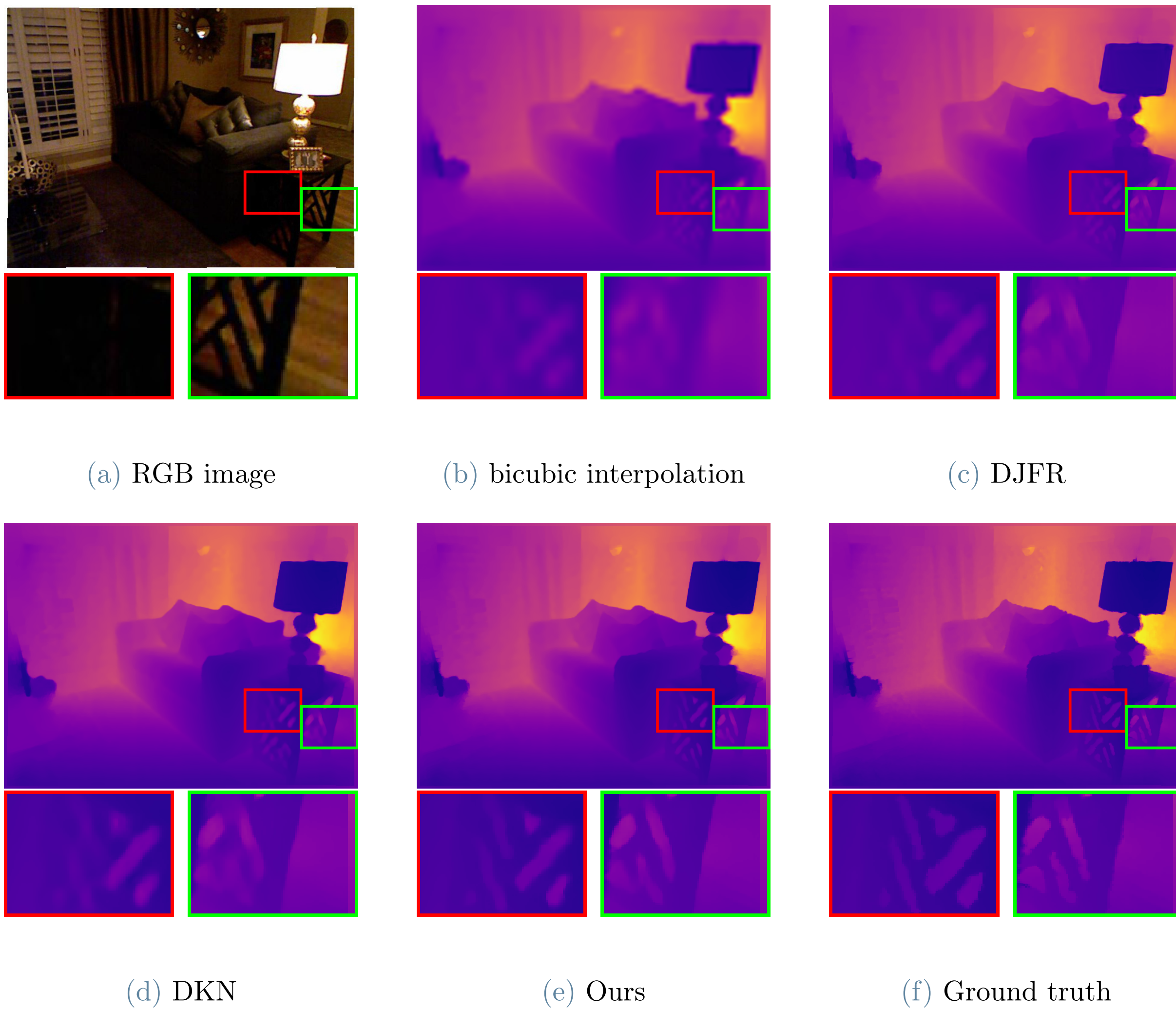
(b) Bicubic interpolation (512x256)

(c) Output depthmap 4x network(512x256)

(d) Output depthmap 8x network (512x256)

(e) Groundtruth depthmap (512x256)

Figure 3.2: comparison between the x4 and x8 upsample architectures trained with syntethic dataset



(a) RGB image (b) bicubic interpolation (c) DJFR

(d) DKN (e) Ours (f) Ground truth

Figure 3.3: Qualitative comparisons of $\times 8$ guided depth map super-resolution on the NYU v2 dataset

4 | Conclusions

The purpose of this research is to demonstrate the importance and the advantages of using deep learning Super-Resolution techniques in upsampling operation for depthmap images. Nowadays the trade off between cost reduction and result accuracy playsa pivotal role in many application. As consequence, if only cheaper system are used, the result quality is badly affected by artifacts and noise in the final image reconstruction. Therefore, in this work, a combination between low resolution depthmap sensor and convolution neural network is presented to eliminate noise, reduce artifacts and at the same time enhance the quality of details in the reconstructed high resolution depthmap. Different model are trained to fulfil this purpose on different dataset. In particular we used a synthetic photo-realistic dataset containing scenes of urban outdoor settings captured in the video game Grand Theft Auto V and a real dataset containing images, gathered from a wide range of commercial and residential buildings, comprising indoor scenes. For each of these dataset we trained two networks, one performing upsample with a scaling factor of 4 and one with a scaling factor of 8. These models are compared using MSE, SSIM, PSNR, as far as it concerns the quantitative results. In order to see the improvement of each model, the performances are compared against a reference high resolution depthmap. For what concerns the networks trained with the synthetic dataset we observe a huge improvement with respect to nearest neighbor, bilinear and bicubic interpolation algorithms either in the 4x and the 8x upsample architecture. If we compare the indexes percentage presented in the Results section of output of our network to the groundtruth, we find that the 8x architecture performs slightly better than the 4x architecture. Also it does not present the overtransfering issue related to the texture from RGB images. This issue shows up when, for example, in the RGB image we have a billboard in the front wall of a building with different colors in it and in the output of the network we have the same wall of the building with different shades of gray because the high frequencies of the billboard make the network think there are different objects whereas they belong to the same object and so they share the same depth. Also for what concerns the model trained with the real dataset we gained a huge upgrade in the upsampling operation with a slight better improvement related to the 8x scenario. Also, the introduction of the DSSIM loss we

applied in the first finetuning step brings us some further but little improvement in the training operation , meaning that the collaboration of multiple losses can enhance the overall performance of the network. We see that our network, if trained with the proper training set, performs well either when dealing with outdoor and indoor scenario.

4.0.1. Further developments

Although the proposed model offers good results in retaining sharpness in the resulting images. There are still many things that can be researched to advance this model. Comparing the percentage of the indexes, especially the ones belonging to the 8x upsample architecture, between the outdoor and the indoor dataset trained models, we notice that the quantitative results of the former are better than the latter ones. This happens because the network architecture cannot totally compensate the low quality of the real images dataset provided us by the Kinect device, which is a crucial aspect needed to be further investigated. Another aspect to examine in depth is the well generalization of the network: a network trained with indoor dataset and tested on outdoor dataset or the other way around. This is not an easy property to obtain from a network since the so different nature of the two datasets. Creating a new dataset could help a lot since many of the current dataset either lack in number of images or in quality of high resolution depth maps. An alternative suggestion could be training for the slight variations in RGB image such as different perspective rotations, warping to differentiate the depth of field. In other words, training the network for the cases where the depth map and RGB image do not perfectly align. Another idea can be using transfer learning to overcome dataset deficiencies, since depth map super-resolution can be considered a special case of image super-resolution. A trained single image super resolution network can be trained for depthmaps. Although the training times for individual networks are not high this will help for the retention of sharpness. Additionally, this network can be reutilized with depth-wise separable convolution [14], where the convolutional layer's operation is broken down into two main operations: depth-wise and pointwise convolution. This approach drastically reduces the number of operations and performs much faster than a traditional convolutional network. Since the operation is additive in terms of depth rather than multiplicative. This can be applied in order to have real-time upsampling with a stream of depth data. Generative Adversarial Networks are recently being used in the image super-resolution networks with great success although the depthmaps lack any textures it can help resolve the small blurry details in the image such as leaves and branches.

Bibliography

[1] D. H. C., J. Kannala, and J. Heikkilä. Joint depth and color camera calibration with distortion correction. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(10): 2058–2064, 2012. doi: 10.1109/TPAMI.2012.125. URL `https://doi.org/10.1109/TPAMI.2012.125`.

[2] Y. Chen, S. Liu, and X. Wang. Learning continuous image representation with local implicit image function. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 8628–8638. Computer Vision Foundation / IEEE, 2021. doi: 10.1109/CVPR46437.2021.00852. URL `https://openaccess.thecvf.com/content/CVPR2021/html/Chen_Learning_Continuous_Image_Representation_With_Local_Implicit_Image_Function_CVPR_2021_paper.html`.

[3] D. Droeschel, D. Holz, and S. Behnke. Multi-frequency phase unwrapping for time-of-flight cameras. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan*, pages 1463–1469. IEEE, 2010. doi: 10.1109/IROS.2010.5649488. URL `https://doi.org/10.1109/IROS.2010.5649488`.

[4] D. Eigen and R. Fergus. Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 2650–2658. IEEE Computer Society, 2015. doi: 10.1109/ICCV.2015.304. URL `https://doi.org/10.1109/ICCV.2015.304`.

[5] D. Eigen, C. Puhrsch, and R. Fergus. Depth map prediction from a single image using a multi-scale deep network. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2366–2374, 2014. URL `https://proceedings.neurips.cc/paper/2014/hash/7bccfde7714a1ebadf06c5f4cea752c1-Abstract.html`.

[6] D. Falie and V. Buzuloiu. Distance errors correction for the time of flight (tof) cameras. In *2008 4th European Conference on Circuits and Systems for Communications*, pages 193–196, 2008. doi: 10.1109/ECCSC.2008.4611675.

[7] P. Fechteler, P. Eisert, and J. Rurainsky. Fast and high resolution 3d face scanning. In *Proceedings of the International Conference on Image Processing, ICIP 2007, September 16-19, 2007, San Antonio, Texas, USA*, pages 81–84. IEEE, 2007. doi: 10.1109/ICIP.2007.4379251. URL https://doi.org/10.1109/ICIP.2007.4379251.

[8] D. Fiedler and H. Müller. Impact of thermal and environmental conditions on the kinect sensor. In X. Jiang, O. R. P. Bellon, D. B. Goldgof, and T. Oishi, editors, *Advances in Depth Image Analysis and Applications - International Workshop, WDIA 2012, Tsukuba, Japan, November 11, 2012, Revised Selected and Invited Papers*, volume 7854 of *Lecture Notes in Computer Science*, pages 21–31. Springer, 2012. doi: 10.1007/978-3-642-40303-3_3. URL https://doi.org/10.1007/978-3-642-40303-3_3.

[9] S. Gu, W. Zuo, S. Guo, Y. Chen, C. Chen, and L. Zhang. Learning dynamic guidance for depth image enhancement. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 712–721. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.83. URL https://doi.org/10.1109/CVPR.2017.83.

[10] M. E. Hansard, S. Lee, O. Choi, and R. Horaud. *Time-of-Flight Cameras - Principles, Methods and Applications*. Springer Briefs in Computer Science. Springer, 2013. ISBN 978-1-4471-4657-5. doi: 10.1007/978-1-4471-4658-2. URL https://doi.org/10.1007/978-1-4471-4658-2.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[12] H. Hirschmüller and D. Scharstein. Evaluation of cost functions for stereo matching. In *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007. doi: 10.1109/CVPR.2007.383248. URL https://doi.org/10.1109/CVPR.2007.383248.

[13] T. Hoegg, D. Lefloch, and A. Kolb. Real-time motion artifact compensation for pmd-tof images. In M. Grzegorzek, C. Theobalt, R. Koch, and A. Kolb, editors, *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications - Dagstuhl 2012 Seminar on Time-of-Flight Imaging and GCPR 2013 Workshop on Imaging*

New Modalities, volume 8200 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2013. doi: 10.1007/978-3-642-44964-2_13. URL https://doi.org/10.1007/978-3-642-44964-2_13.

[14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL http://arxiv.org/abs/1704.04861.

[15] P.-H. Huang, K. Matzen, J. Kopf, N. Ahuja, and J.-B. Huang. Deepmvs: Learning multi-view stereopsis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[16] T. Hui, C. C. Loy, and X. Tang. Depth map super-resolution by deep multi-scale guidance. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III*, volume 9907 of *Lecture Notes in Computer Science*, pages 353–369. Springer, 2016. doi: 10.1007/978-3-319-46487-9_22. URL https://doi.org/10.1007/978-3-319-46487-9_22.

[17] A. Kadambi, R. Whyte, A. Bhandari, L. V. Streeter, C. Barsi, A. A. Dorrington, and R. Raskar. Coded time of flight cameras: sparse deconvolution to address multipath interference and recover time profiles. *ACM Trans. Graph.*, 32(6):167:1–167:10, 2013. doi: 10.1145/2508363.2508428. URL https://doi.org/10.1145/2508363.2508428.

[18] T. Kahlmann, F. Remondino, and H. Ingensand. Calibration for increased accuracy of the range imaging camera swissranger. 2006.

[19] B. Kim, J. Ponce, and B. Ham. Deformable kernel networks for joint image filtering. *Int. J. Comput. Vis.*, 129(2):579–600, 2021. doi: 10.1007/s11263-020-01386-z. URL https://doi.org/10.1007/s11263-020-01386-z.

[20] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[21] A. Krizhevsky. Learning multiple layers of features from tiny images. pages 32–33, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012. URL `https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html`.

[23] D. Lefloch, R. Nair, F. Lenzen, H. Schäfer, L. V. Streeter, M. J. Cree, R. Koch, and A. Kolb. Technical foundation and calibration methods for time-of-flight cameras. In M. Grzegorzek, C. Theobalt, R. Koch, and A. Kolb, editors, *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications - Dagstuhl 2012 Seminar on Time-of-Flight Imaging and GCPR 2013 Workshop on Imaging New Modalities*, volume 8200 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2013. doi: 10.1007/978-3-642-44964-2_1. URL `https://doi.org/10.1007/978-3-642-44964-2_1`.

[24] Y. Li, J. Huang, N. Ahuja, and M. Yang. Deep joint image filtering. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2016. doi: 10.1007/978-3-319-46493-0_10. URL `https://doi.org/10.1007/978-3-319-46493-0_10`.

[25] Y. Li, J. Huang, N. Ahuja, and M. Yang. Joint image filtering with deep convolutional networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(8):1909–1923, 2019. doi: 10.1109/TPAMI.2018.2890623. URL `https://doi.org/10.1109/TPAMI.2018.2890623`.

[26] M. Lindner and A. Kolb. Lateral and depth calibration of pmd-distance sensors. In G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, A. V. Nefian, M. Gopi, V. Pascucci, J. Zara, J. Molineros, H. Theisel, and T. Malzbender, editors, *Advances in Visual Computing, Second International Symposium, ISVC 2006 Lake Tahoe, NV, USA, November 6-8, 2006. Proceedings, Part II*, volume 4292 of *Lecture Notes in Computer Science*, pages 524–533. Springer, 2006. doi: 10.1007/11919629_53. URL `https://doi.org/10.1007/11919629_53`.

[27] M. Lindner and A. Kolb. Compensation of motion artifacts for time-of-flight cameras. In A. Kolb and R. Koch, editors, *Dynamic 3D Imaging, DAGM 2009 Workshop, Dyn3D 2009, Jena, Germany, September 9, 2009. Proceedings*, volume 5742

of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2009. doi: 10.1007/978-3-642-03778-8_2. URL https://doi.org/10.1007/978-3-642-03778-8_2.

[28] S. Lu, X. Ren, and F. Liu. Depth enhancement via low-rank matrix completion. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 3390–3397. IEEE Computer Society, 2014. doi: 10.1109/CVPR.2014.433. URL https://doi.org/10.1109/CVPR.2014.433.

[29] A. Mertan, D. J. Duff, and G. Unal. Single image depth estimation: An overview. *CoRR*, abs/2104.06456, 2021. URL https://arxiv.org/abs/2104.06456.

[30] C. B. Rist, D. Schmidt, M. Enzweiler, and D. M. Gavrila. Scssnet: Learning spatially-conditioned scene segmentation on lidar point clouds. In *IEEE Intelligent Vehicles Symposium, IV 2020, Las Vegas, NV, USA, October 19 - November 13, 2020*, pages 1086–1093. IEEE, 2020. doi: 10.1109/IV47402.2020.9304824. URL https://doi.org/10.1109/IV47402.2020.9304824.

[31] A. Sabov and J. Krüger. Identification and correction of flying pixels in range camera data. In K. Myszkowski, editor, *Spring Conference on Computer Graphics, SCCG 2008, Budmerice Castle, Slovakia, April 21-23, 2008*, pages 135–142. ACM, 2008. doi: 10.1145/1921264.1921293. URL https://doi.org/10.1145/1921264.1921293.

[32] H. Sarbolandi, D. Lefloch, and A. Kolb. Kinect range sensing: Structured-light versus time-of-flight kinect. *CoRR*, abs/1505.05459, 2015. URL http://arxiv.org/abs/1505.05459.

[33] D. Scharstein and C. Pal. Learning conditional random fields for stereo. In *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007. doi: 10.1109/CVPR.2007.383191. URL https://doi.org/10.1109/CVPR.2007.383191.

[34] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003), 16-22 June 2003, Madison, WI, USA*, pages 195–202. IEEE Computer Society, 2003. doi: 10.1109/CVPR.2003.1211354. URL https://doi.org/10.1109/CVPR.2003.1211354.

[35] M. Schmidt and B. Jähne. Efficient and robust reduction of motion artifacts for 3d time-of-flight cameras. In *International Conference on 3D Imaging, IC3D 2011,*

Liège, Belgium, December 7-8, 2011, pages 1–8. IEEE, 2011. doi: 10.1109/IC3D.2011.6584391. URL https://doi.org/10.1109/IC3D.2011.6584391.

[36] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus. Indoor segmentation and support inference from RGBD images. In A. W. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, editors, *Computer Vision - ECCV 2012 - 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part V*, volume 7576 of *Lecture Notes in Computer Science*, pages 746–760. Springer, 2012. doi: 10.1007/978-3-642-33715-4_54. URL https://doi.org/10.1007/978-3-642-33715-4_54.

[37] H. Su, V. Jampani, D. Sun, O. Gallo, E. G. Learned-Miller, and J. Kautz. Pixel-adaptive convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 11166–11175. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.01142. URL http://openaccess.thecvf.com/content_CVPR_2019/html/Su_Pixel-Adaptive_Convolutional_Neural_Networks_CVPR_2019_paper.html.

[38] J. Tang, X. Chen, and G. Zeng. Joint implicit image function for guided depth super-resolution. In H. T. Shen, Y. Zhuang, J. R. Smith, Y. Yang, P. César, F. Metze, and B. Prabhakaran, editors, *MM '21: ACM Multimedia Conference, Virtual Event, China, October 20 - 24, 2021*, pages 4390–4399. ACM, 2021. doi: 10.1145/3474085.3475584. URL https://doi.org/10.1145/3474085.3475584.

[39] C. Zach, T. Pock, and H. Bischof. A duality based approach for realtime tv-L^1 optical flow. In F. A. Hamprecht, C. Schnörr, and B. Jähne, editors, *Pattern Recognition, 29th DAGM Symposium, Heidelberg, Germany, September 12-14, 2007, Proceedings*, volume 4713 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2007. doi: 10.1007/978-3-540-74936-3_22. URL https://doi.org/10.1007/978-3-540-74936-3_22.

[40] L. Zhang, B. Curless, and S. M. Seitz. Rapid shape acquisition using color structured light and multi-pass dynamic programming. In *1st International Symposium on 3D Data Processing Visualization and Transmission (3DPVT 2002), 19-21 June 2002, Padova, Italy*, pages 24–37. IEEE Computer Society, 2002. doi: 10.1109/TDPVT.2002.1024035. URL https://doi.org/10.1109/TDPVT.2002.1024035.

List of Figures

1.1 Monocular Vision 6
1.2 Stereo Vision Camera 7
1.3 3D Rays Intersection 8
1.4 Colour mapping helps to visually quantify distances on-screen 9
1.5 Principle of structured light based system 9
1.6 Sensor placement within a Kinect^{SL} camera. The baseline is of approximately 7.5 cm. 10
1.7 The ToF phase-measurement principle 11
1.8 Error sources of ToF cameras. Top left: Systematic (wiggling) error for all pixels (gray) and fitted mean deviation (black). Top right: Motion artifacts (red) and flying pixels (green) for a horizontally moving planar object in front of a wall. Bottom left: Schematic illustration of multi-path effects due to reflections in the scene. Bottom right: Acquisition of a planar gray-scale checkerboard reveals the intensity related distance error 13
1.9 Shorter caption 16
1.10 17
1.11 Interpolation Algorithms 21
1.12 original image 21
1.13 nearest neighbor interpolation 22
1.14 bilinear interpolation 22
1.15 bicubic interpolation 23
1.16 Network Architecture: the grid illustrates the relative image resolution and a $\times 2$ up-sampling as an example for simplicity. Given a HR guide image and a LR input image, two sets of latent codes are extracted via two encoders, then query the JIIF decoder with a coordinate in the HR domain to predict the pixel value at this coordinate. The prediction is a weighted average from the four nearest coordinates in the LR domain just like the standard image interpolation, but learn the interpolation weights and values via a deep implicit function. 27

2.1 5x5 convolution vs the equivalent stacked 3x3 convolutions 30
2.2 Validation Accuracy on a 3x3-based Convnet (orange) and the equivalent 5x5-based Convnet (blue) 31
2.3 on the left the ReLU activation function and on the right the Leaky ReLU activation function 32
2.4 example of a subpixel convolution function: depth to space shuffle 32
2.5 example of residual block 33
2.6 architecture of the 8x upsample network trained with NYU dataset 36
2.7 structure of the "composed block" 38
2.8 sigmoid activation function 39
2.9 architecture of the 4x upsample network trained with NYU dataset 40
2.10 low resolution image from synthetic dataset used as input 43
2.11 high resolution image from synthetic dataset used as groundtruth 43
2.12 RGB image from synthetic dataset used as guide image 43
2.13 low resolution image from real dataset used as input 45
2.14 high resolution image from real dataset used as groundtruth 45
2.15 RGB image from real dataset used as guide image 45
2.16 comparison between a clean sample and the same sample,filtered, belonging to Middlebury dataset 47
2.17 comparison showing the performance of Adam optimizer 49
2.18 example of a plot of loss function for training and validation dataset 58
2.19 The error trend line in a logarithmic scale, epochs in x-scale, mean-absolute error in y-scale, orange line is for a larger learning rate (1×10^{-3}), while the blue line is for the smaller learning rate (2.5×10^{-4}) 59

3.1 8x upsampling results obtained on one sample belonging to the test set . . 70
3.2 comparison between the x4 and x8 upsample architectures trained with syntethic dataset 71
3.3 Qualitative comparisons of $\times8$ guided depth map super-resolution on the NYU v2 dataset 72

List of Tables

3.1 Table comparing the metrics of the classical upsample algorithm and our convolution neural network for both the x8 and 4x upsample model, trained with MVS-Synth dataset 65
3.2 Table comparing the metrics of our 8x network going through each step of finetuning 66
3.3 Table comparing the metrics of the classical upsample algorithm and our convolution neural network for both the x8 and 4x upsample model,trained with NYU dataset 67
3.4 Table comparing the RMSE index of our network trained with Middlebury dataset to some of the state of the art methods,both for the x8 and 4x upsample model 67

Acknowledgements

I would like to first thank my thesis advisor Professor Marco Marcon and my Co-advisor Marco Paracchini which gave me the possibility to learn a lot in the fields of convolution nerual network. Also, I would like to thank them for their kindness, availability and their patience in clearing all my questions and doubts about the above mentioned topics and about the writing of this work. In this long path that brought me at the end of my university study, I have met many people. Most of them are the colleagues that i met at the first days at Politecnico di Milano. Some of them came later joining the group. Some of them does not live in the same city. Some of them i meet frequently, some of them i do not. Despite that we have fun every time we see each other. Recently i met a friend i have not seen for long time and despite that we join each other's company as time never passed. With some of them i spent nights talking nonsense and playing some relaxing video game. I thank all friends who encouraged me and supported me especially during the time i was writing this work and are here today sharing with me this important goal: Lorenzo Reposo, Enrico Ruggiano, David Petrucci, Marcello Zangrossi, Roberta Parmose, Matteo Muffo, Riccardo Remigio, Massimiliano Ventura, Nilo Giacomucci, Maurizio Sorrenti, Giulia di Lorenzo, Francesco Rampa. I thank my spiritual sisters: Arianna Converti, Delia Converti, Yue and my spiritual lover Luca Scarantino. I want to thank all my relatives, who always believed in me and made me feel their support. Unfortunately, lots of them live far away and could not come today but i feel very close to them in this important day. I must express my very profound gratitude to my parents for their continuous support and encouragement throughout my years of study and for the period dedicated to the writing of this work. This accomplishment would not have been possible without them, this is the result of their sacrifices. So, i would like to show again my appreciation to my parents for what they have done for me. Thank you. Finally i beat my Majinbu. Last but not least, i dedicate this moment to my grandfathers and grandmothers who have always wanted to see this day happening but unfortunately they all passed away. I'm sorry it took so long. I hope you could be proud of me.