POLITECNICO DI TORINO

Master's degree course
in Mechatronic Engineering

Master's thesis

Image-Guided Joint Velocity Control of a Meca500 Manipulator through CNNs

Supervisors

Prof. Alessandro Rizzo
Prof. Rui Pedro Duarte Cortesão

Candidate

Isacco Ceri

December 2025

Abstract

Deep learning has become a powerful tool for dealing with various problems. In particular, in recent years, it has been an object of research even in the field of robotics, due to its ability to work easily with millions of parameters, exploiting the possibility of modeling complex functions. The aim of this thesis is to use one of the core architectures of DL, that is, Convolutional Neural Networks (CNNs), in order to build an open control loop to control a Meca500 robotic arm, which is intended to move towards a target.

The target object chosen for the experiment is a simple six-faced die, which is placed in the workspace; a small endoscopic camera is mounted on the end-effector, aligned with its approach axis, in such a way that the target is always visible to the camera.
The first phase is dedicated to the construction of a proper training set to be used in the successive training step. This is done by capturing the frames of the camera with a certain periodicity and pairing them with the corresponding task velocity vector, which has been normalized.

During the offline training phase, frames and velocity labels are used to train a Pytorch CNN that outputs the six predicted normalized components of the task-space velocity vector. In this way, the network is able to pair the correct velocity vector with the position of the target, as visualized by the camera. Then, online, the prediction is denormalized, mapped to joint velocities through the robot Jacobian, and executed in real time. The initial joint positions are then modified, taking care of possible singularities.
Finally, from the standpoint of practical application, an admittance-like control branch is added to the main one, and the whole open control loop is tested.

Acknowledgements

I would like to thank Professor Rizzo for giving me the opportunity to work with him and for his trust in my capabilities.
I am genuinely grateful to the University of Coimbra for hosting me in such a stimulating environment, with a special consideration to Professor Cortesão, for his guidance and support, which made this experience enjoyable.
Finally, a special thanks to my family, to my true friends, and to the great San Giorgio a Colonica.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Context

Deep learning, in particular *Convolutional Neural Networks* (CNNs), has revolutionized robotics research in the past decade, with dozens of studies exploring this subject since around 2014 [1]. *Deep neural networks* (DNNs) are powerful function approximators with millions of parameters, capable of modeling complex relationships in high-dimensional data. In robotic systems, this means a DNN can directly process raw, high-dimensional sensor inputs (such as camera images) and autonomously learn compact representations of the robot's state, without requiring manual feature engineering [2]. This offers a key advantage over traditional machine-learning or control methods, since DNNs can extract both visual and kinematic features from the data itself, avoiding reliance on predefined detectors or manual tuning. As a result, deep learning has been successfully applied to a wide variety of robotic tasks, including perception, planning, and control [1].

A particularly active domain is *vision-based robot control*, where CNNs are used to drive manipulators directly from camera images. Classical visual servoing approaches, both image-based and position-based, require selecting and tracking visual features (e.g., points or contours), and often depend on camera calibration and accurate scene geometry [3]. Deep-learning-based methods overcome these limitations by learning an end-to-end mapping from images to control commands.

For example, Saxena et al. [4] trained a CNN to predict robot motion given only the current and goal images, without needing feature tracking or known camera parameters. Bateux et al. [5] estimated the relative pose between two camera views using a CNN trained on synthetically augmented data, achieving high-precision 6-DoF positioning (down to 1 mm) even under varied lighting and occlusions. These works demonstrated that CNN-based visual servoing can outperform traditional methods in robustness and generalization. More recently, researchers have enhanced these networks with depth maps or learned feature encodings, applying them successfully to grasping, manipulation, and navigation [6]

Building on these advances, this thesis proposes a CNN-based visual control system for a 6-DoF robotic manipulator (Mecademic Meca500) using an *eye-in-hand* configuration.

The goal is to guide the end-effector toward a fixed target object using only the live video stream from a small endoscopic camera mounted along the approach axis. Unlike prior methods requiring a desired image or explicit pose estimation, the proposed CNN implicitly learns the relationship between the target's visual appearance and the correct motion required to reach it.

During offline training, a dataset of image–velocity pairs is constructed by capturing camera frames at regular intervals as the robot moves within its workspace. Each image is paired with the corresponding normalized task-space velocity vector. This supervised training process enables the network to learn the association between target appearance and the necessary velocity vector. At runtime, the CNN processes each camera frame to produce a predicted velocity, mapped through the Jacobian into joint velocities, and executed on the robot in real time. Although the system operates open-loop, the continuous camera updates induce closed-loop behavior driven by the CNN itself, with no manually defined feedback law.

The proposed method includes several novel contributions:

- **Single-image input:** Unlike Siamese or dual-input networks [4, 7], only the current image is required. The notion of a "goal" is implicitly captured by including low-velocity samples around the target during training.

- **Real world dataset:** The network is trained exclusively on images captured from the real robot in a laboratory setting, avoiding the gap that affects many synthetic approaches [5].

- **Coarse-to-fine architecture:** A two-network structure is introduced. The *coarse* CNN brings the robot close to the goal, while the *refine* CNN, trained on a denser local dataset, ensures final positioning accuracy.

- **Admittance-like smoothing:** To counteract unstable or oscillatory behavior from direct velocity execution, a filtering branch is added. It mimics an admittance controller, smoothing the predicted command for stable execution.

While Guo et al. [8] integrate CNN-based object detection (YOLOv3) with LSTM filtering and Lyapunov-stable control for an eye-in-hand manipulator, their approach depends on bounding box tracking and aims primarily at object centering. In contrast, this thesis proposes direct task-velocity prediction from raw images, requiring no bounding box detection, goal image, or geometric modeling.

In summary, this work presents a real-time, CNN-based visual control system for robotic manipulators, capable of driving the end-effector to a visually defined target using only raw camera input. Its innovations in training design, control integration, and network structure position it as a practical step forward in the development of reliable deep visuomotor policies.

1.2 Thesis Structure

The organization of this thesis follows a logical progression, guiding the reader from the foundational theoretical principles necessary for understanding the subsequent implementation and experimentation, to the final analysis and conclusions.

- **Theoretical Foundations:** This section introduces the key theoretical concepts on which the control architecture is based, covering both robotic fundamentals and convolutional neural networks.
- **Instrumentation:** This chapter presents the tools and instruments used throughout the experiments, along with a brief overview of their technical specifications and roles in the setup.
- **Implementation:** This section describes the practical implementation of the system, including dataset construction, the chosen network architecture, the training process, and hyperparameter tuning. It goes on with the mapping from task space to joint space, detailing the formulation of the geometric Jacobian, and it concludes with considerations related to potential singularities.
- **Additional Control Branch:** This section explores modifications made to the initial structure with a view toward practical applications. In particular, an admittance control branch is introduced in parallel with the main control loop.
- **Results:** With the system fully implemented, this chapter presents the outcomes of real-time testing, demonstrating the effectiveness of the proposed approach in achieving the thesis objectives.
- **Future Work:** The final chapter outlines future research directions, addressing the limitations of the current system and proposing possible enhancements for further development.

Chapter 2

Theoretical Foundations

2.1 Neural Networks

Neural networks are powerful tools for solving complex problems. Their effectiveness arises from a different perspective with respect to classical programming: instead of explicitly specifying to the computer how to solve a task, a neural network is trained on examples and learns to approximate the desired input–output mapping by itself [9].

Neural networks are loosely inspired by the structure of biological neurons in the human brain (Fig. 2.1). There, in fact, neurons form a large, highly interconnected network and communicate through electrical impulses. Analogously, an artificial neural network (ANN) is composed of artificial neurons organized in layers and connected by weighted links. Information flows from one layer to the next, and the weights are iteratively adjusted during training in order to minimize a chosen loss function.

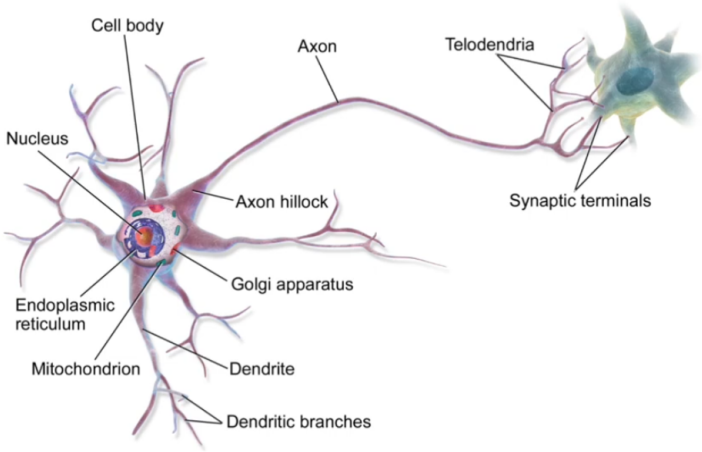

Figure 2.1. Sketch of a biological neuron.

Taking the biological neuron as a reference, composed of a cell body and dendrites that act as information channels, artificial neural networks are typically organized as shown in Fig. 2.2. The figure illustrates an example network with nine artificial neurons: three *input neurons* belonging to the *input layer*, four *hidden neurons* in the *hidden layer*, and two *output neurons* in the *output layer*.

In general, ANNs can have very different architectures, with a larger number of neurons and multiple hidden layers. The information exchange between neurons, which mimics the synapses of biological cells, is represented by the directed arrows in the diagram.

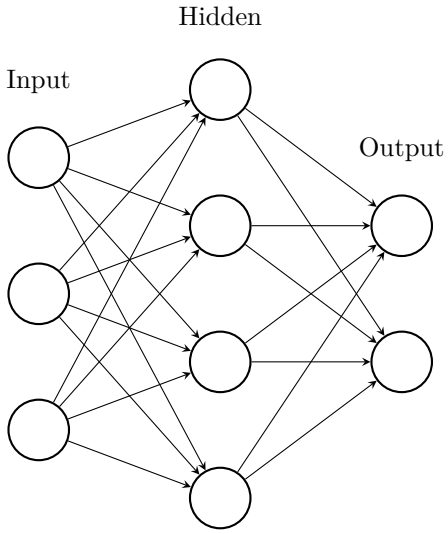

Figure 2.2. Example of a simple artificial neural network architecture.

The output of a neuron is computed by applying an *activation function* to the linear combination of its inputs, *weights*, and *bias* (the latter are introduced in the next chapter). This operation transforms a purely linear response into a nonlinear one, allowing neural networks to learn complex decision boundaries and to approximate a wide class of continuous functions.

Many different activation functions exist, and their choice determines the behaviour of the artificial neuron. In the following, two classical neuron models are considered: *perceptrons* and *sigmoid* neurons.

2.1.1 Perceptrons and Sigmoid Neurons

The basic model of an artificial neuron is the so-called *perceptron*, which takes n binary inputs and produces a single binary output, as shown in Fig. 2.3.

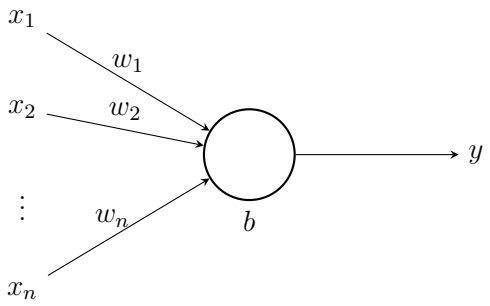

Figure 2.3. General structure of a perceptron.

The output is computed by introducing the weights $w_1, w_2, \ldots, w_n$, real-valued parameters that quantify the influence of each input on the output. A bias term b (shown inside the neuron) shifts the decision boundary. The perceptron's output $y \in \{0,1\}$ is determined by comparing the weighted sum $\sum_{i=1}^{n} w_i x_i$ with a threshold value T:

$$y = \begin{cases} 0, & \text{if } \displaystyle\sum_{i=1}^{n} w_i x_i \leq T, \\ 1, & \text{if } \displaystyle\sum_{i=1}^{n} w_i x_i > T. \end{cases} \tag{2.1}$$

One way to conceptualize the perceptron is as a decision-making device that evaluates evidence through a weighted sum of inputs, and by varying the weights and the threshold, it is possible to obtain different models of decision-making.

Going back to Fig. 2.2, the *input layer* can be viewed as presenting the raw evidence to the network, with each input neuron corresponding to one component of the input. The neurons in the *hidden layer* then make decisions based on what the previous layer has already computed, allowing them to detect more abstract patterns. By adding the *output layer*, the network can combine these intermediate representations to produce more sophisticated decisions. In this way, by stacking multiple layers, the network gradually builds a hierarchical representation of the input, enabling it to solve complex decision-making tasks.

Another way to describe the working principle of the perceptron is through the bias term b, which can be interpreted as an adjustable threshold: it measures how easily the perceptron activates to produce an output of 1. In biological terms, it is analogous to the intrinsic excitability of a neuron. By setting $b \equiv -T$ and moving the threshold term to the left-hand side, (2.1) can be rewritten as

$$y = \begin{cases} 0, & \text{if } \displaystyle\sum_{i=1}^{n} w_i x_i + b \leq 0, \\ 1, & \text{if } \displaystyle\sum_{i=1}^{n} w_i x_i + b > 0. \end{cases} \tag{2.2}$$

This form is mathematically equivalent to the original threshold formulation, but it is often more convenient in analysis and implementation.

Graphically, the activation rule in (2.2) corresponds to the step function shown in Fig. 2.4.

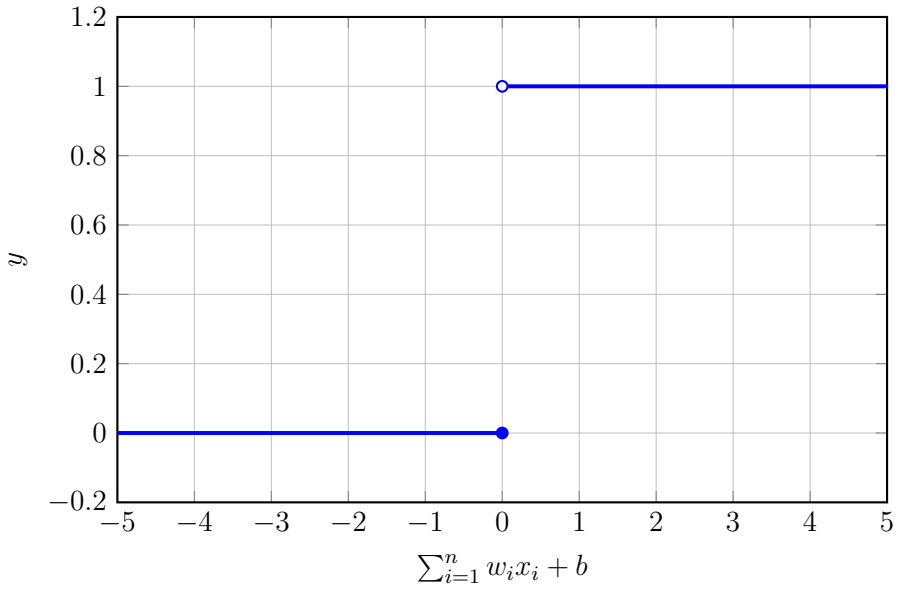

Figure 2.4. Step activation function of a perceptron.

Starting from this model, it is natural to ask whether one can design *learning algorithms* that automatically adjust the weights and biases of an ANN, without explicit intervention from a programmer. Ideally, such adjustments should be smooth: a small change in the parameters should produce only a small change in the network output. This property is crucial for optimization methods based on gradients.

However, classical perceptrons do not satisfy this requirement. Their output is strictly binary (0 or 1), and even an arbitrarily small change in the weights or bias can cause a sudden flip of the output. To overcome this limitation and enable gradient-based learning, *sigmoid neurons* are introduced in the next section.

2.1.2 Sigmoid Neurons

The structure of a *sigmoid neuron* is the same as that of the perceptron in Fig. 2.3. The key difference is that both the inputs and the output are now allowed to take real values: each input x_i is typically taken in the interval [0,1], and the output is computed through the sigmoid activation function $\sigma(z)$,

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2.3}$$

where

$$z = \sum_{i=1}^{n} w_i x_i + b. \tag{2.4}$$

Comparing the sigmoid function in Fig. 2.5 with the step function in Fig. 2.4, it is apparent that the former is a "smoothed" version of the latter. In particular, small changes in the weights and in the bias produce only small changes in the neuron output.

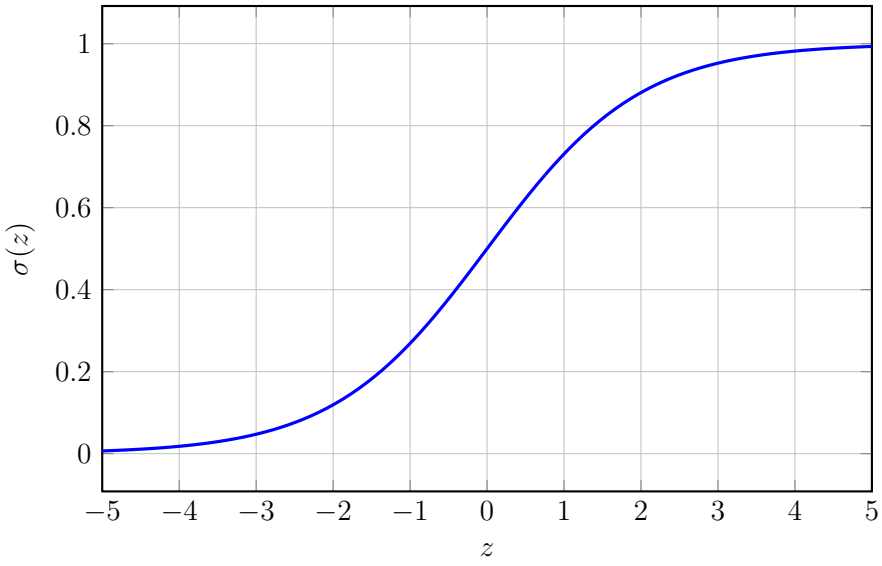

Figure 2.5. Sigmoid activation function.

Although sigmoid neurons retain much of the qualitative behaviour of perceptrons (they still "activate" more strongly when the weighted input z is large), their smoothness makes it far easier to understand and control how changes in the weights and biases affect the output [9]. This differentiable, real-valued output is what enables the use of gradient-based learning algorithms in multilayer neural networks.

2.1.3 Stochastic Gradient Descent

Once the output of the network is defined, learning consists in adjusting the parameters so that the network correctly recognizes the patterns shared by the elements of the *training set*, which contains n examples. This is achieved by minimizing a suitable *cost function*. A common choice is the quadratic (or mean squared) cost [9]:

$$C(w, b) = \frac{1}{2n} \sum_{x} \|y(x) - a(x)\|^2, \tag{2.5}$$

where the sum runs over all training inputs x, $y(x)$ denotes the desired output vector associated with x, and $a(x)$ is the output vector produced by the network when x is fed as input.

The cost function (2.5) measures the average squared difference between the target outputs and the network outputs. Minimizing $C(w, b)$ therefore means finding weights and biases such that $a(x)$ is as close as possible to $y(x)$ for all training examples. This optimization is typically performed using the *gradient descent* algorithm.

Fig. 2.6 illustrates the basic idea on a toy example: a cost function $C(t_1, t_2)$ depending on two parameters t_1 and t_2. Starting from an initial point P_0, the parameters are updated iteratively in the direction of steepest descent of the cost, i.e., in the direction opposite to the gradient ∇C. The sequence of points converges towards a local minimum, denoted by G_m in the figure. Of course, in the case of neural networks the cost function depends on thousands (or even millions) of parameters, but the underlying geometric intuition is the same [9].

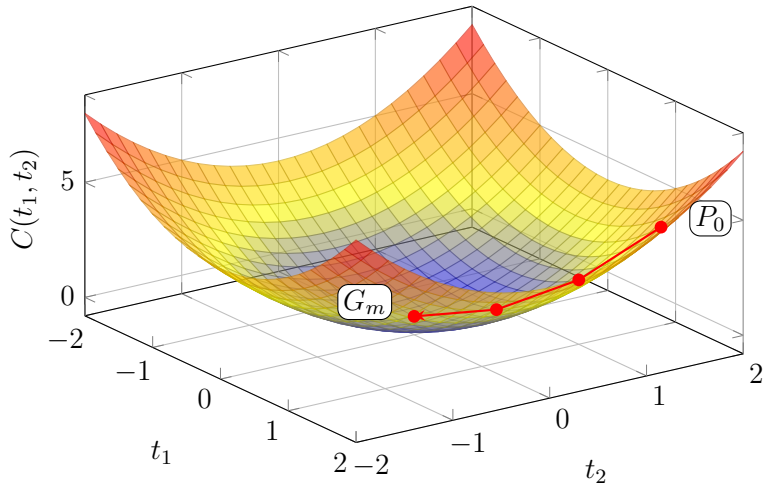

Figure 2.6. Gradient descent optimization on a quadratic cost surface.

In gradient descent, each update step requires computing the gradient of the cost $\nabla C(w, b)$ over the entire training set, which can be computationally expensive for large datasets. In practice, neural networks are usually trained with *stochastic gradient descent* (SGD), where the gradient is approximated using a single randomly chosen training example, or more commonly, a small *mini-batch* of examples. This leads to the update rule

$$(w, b) \leftarrow (w, b) - \eta \nabla C_{mini-batch}(w, b),$$

where $\eta > 0$ is the learning rate and $C_{mini-batch}$ denotes the cost computed only over the current mini-batch. Although each step is noisier, SGD dramatically reduces the computational cost per update and usually leads to efficient learning in large-scale neural networks.

In practice, the gradients required by (stochastic) gradient descent are computed by the *backpropagation* algorithm [9]. Backpropagation applies the chain rule of calculus in a systematic way: a *forward pass* first computes all weighted inputs and activations layer by layer, while a subsequent *backward pass* propagates an error signal from the output

layer back through the network. This backward recursion makes it possible to obtain the derivatives of the cost with respect to all weights and biases with a computational cost that is only a small constant factor larger than that of the forward evaluation itself.

2.2 Convolutional Neural Networks

Fully connected networks are very general function approximators, but they are often a poor fit for image data because they ignore spatial structure. All pixels are treated in the same way, regardless of whether they are adjacent or far apart, so locality and approximate translation invariance must be learned from scratch. As a consequence, dense architectures require a very large number of parameters and do not explicitly exploit the geometric structure of images.

Convolutional neural networks (CNNs), of which the general structure is sketched in Fig. 2.7, introduce architectural constraints that encode these inductive biases directly. By using *local receptive fields*, *shared weights*, and *pooling* operations, CNNs capture spatial hierarchies of features and exhibit a form of translation equivariance. This significantly reduces the number of parameters, speeds up training, and enables the design of deeper models with higher accuracy. For these reasons, most modern image recognition systems are based on CNNs or closely related architectures [9, 10].

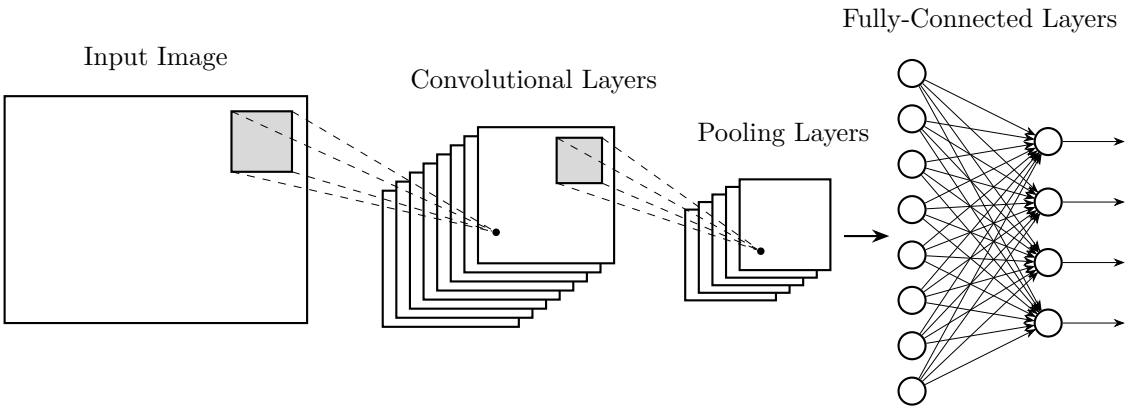

Figure 2.7. Schematic structure of a convolutional neural network.

2.2.1 Local Receptive Fields

In a convolutional neural network, the input layer is no longer represented as a simple column of neurons, as in Fig. 2.2, but as a two–dimensional grid. Each neuron in this grid corresponds to the intensity of a single pixel in the input image. Rather than connecting every neuron in the next layer to all input pixels, each neuron in the first convolutional layer is connected only to a small, localized patch of the input. This patch is called a *local receptive field* [9].

As an example, it is considered an input layer of size 20×20, shown on the left in Fig. 2.8. A 5×5 receptive field (the light gray square) is highlighted in the top–left corner of the input. The first neuron of the following convolutional layer (on the right) is connected only to the pixels inside this 5×5 patch: its weighted input is a function of those 25 values and their associated weights, plus a bias term.

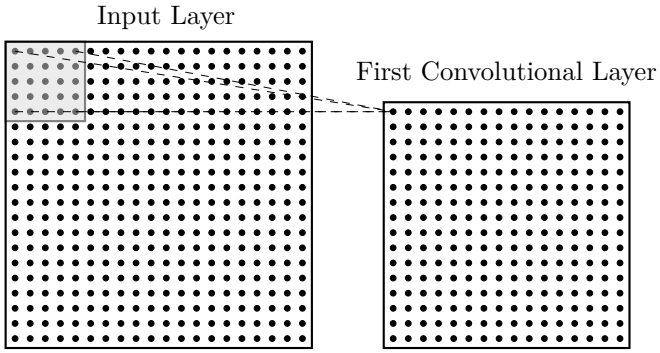

Figure 2.8. Example of an input layer with a local receptive field and the corresponding first convolutional layer.

To obtain the next neuron in the same row of the convolutional layer, the receptive field is shifted one pixel to the right in the input, as illustrated in Fig. 2.9. In this way, each neuron in the convolutional layer scans a different local region of the input, but all neurons share the same weights and bias (the convolutional kernel). For an input of size $n \times n$ and a receptive field of size $m \times m$ with step 1, the resulting feature map has size $(n - m + 1) \times (n - m + 1)$; in this example, $n = 20$ and $m = 5$, giving 16×16 neurons.

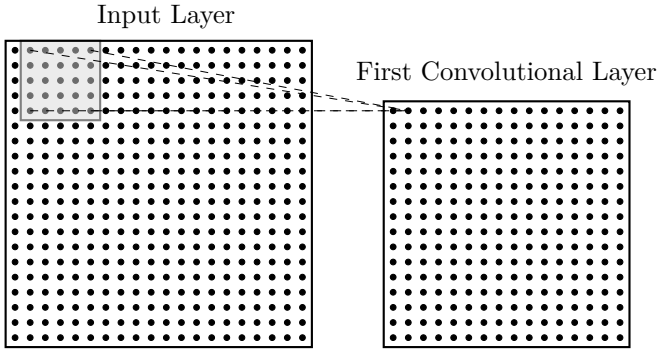

Figure 2.9. Same example with the receptive field shifted one pixel to the right, connected to the second neuron of the first convolutional layer.

Local receptive fields thus enforce *local connectivity*: each neuron in a convolutional layer "sees" only a small region of the input, focusing on simple patterns such as edges or corners. Deeper layers can then combine these local features into more complex and

abstract representations, while keeping the number of connections (and parameters) much smaller than in a fully connected network [9].

2.2.2 Shared Weights and Biases

In a convolutional layer, the output is organized into *feature maps*. Each feature map corresponds to one learned pattern (for example, a vertical edge, a corner, or a texture) that the network tries to detect across the entire input. All neurons belonging to the same feature map apply the *same* filter to different spatial locations of the previous layer.

This is achieved through *weight sharing*: neurons in the same feature map do not each have their own weights and bias. Instead, they share a single collection of weights $\{w_{l,m}\}$ and a single bias b. These shared parameters form a small filter (or kernel) that slides across the input, producing one activation value for each spatial position. For a neuron located at (j, k), the activation is

$$a'_{j,k} = \sigma(z_{j,k}), \tag{2.6}$$

where

$$z_{j,k} = b + \sum_{l=0}^{s-1} \sum_{m=0}^{s-1} w_{l,m}\, a_{j+l,k+m}. \tag{2.7}$$

Here, $a_{j+l,k+m}$ are the activations in the previous layer within the $s \times s$ local receptive field, while $w_{l,m}$ and b are the filter parameters shared across all positions (j, k) [9].

Because the same filter is applied everywhere, each feature map encodes where and how strongly its associated feature appears in the input. Weight sharing dramatically reduces the number of trainable parameters compared to fully connected layers and enforces translation equivariance: if a feature is present at a different location in the image, the same filter will still detect it.

Fig. 2.10 illustrates this idea for the same 20×20 input image considered before. On the right, a stack of eight 16×16 feature maps is shown. Each feature map corresponds to a different set of shared weights and bias, i.e., to a different learned filter. The arrows indicate that every feature map is computed by convolving the full input with its own kernel, according to (2.7).

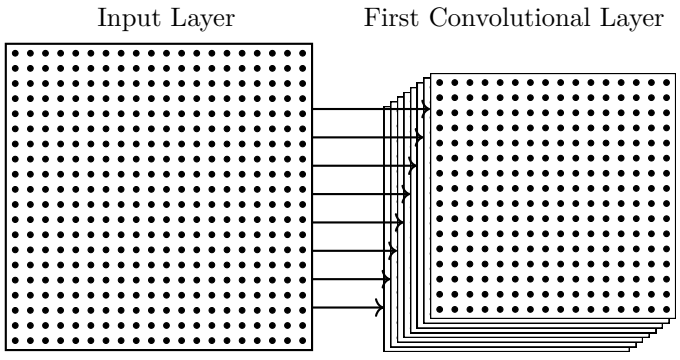

Figure 2.10. Same example with a stack of eight feature maps in the first convolutional layer.

2.2.3 Pooling

Pooling layers are typically inserted after convolutional layers and operate independently on each feature map. The idea is to aggregate the information contained in small, non-overlapping regions of a feature map into a single value, hence reducing the spatial resolution while retaining the most salient information [9].

Fig. 2.11 illustrates the same example. On the left, a 16×16 feature map (e.g., the output of the first convolutional layer) is shown. A 2×2 patch in the top–left corner is highlighted in light gray. This patch is mapped to a single neuron in the following 8×8 pooling layer on the right. In the case of *max pooling*, the output of that neuron is the maximum value within the corresponding 2×2 region; for *average pooling*, it is the average of those four values. Using a 2×2 window with step 2 splits the spatial dimensions: a 16×16 feature map becomes 8×8.

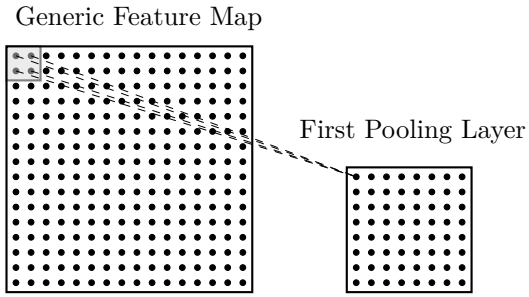

Figure 2.11. A feature map with the corresponding neuron in the pooling layer.

Pooling is applied separately to each feature map. Fig. 2.12 shows an example at the level of full stacks. Starting from the 20×20 input image on the left, a first convolutional layer produces a stack of eight 16×16 feature maps. A subsequent pooling layer then processes these feature maps to produce a stack of smaller 8×8 maps on the right. The number of feature maps is typically preserved by pooling, but the spatial dimensions are reduced.

By reducing the resolution while keeping the most responsive activations, pooling layers help to make the representation more robust to small translations and distortions of the input, decrease the number of activations, and hence the computational cost in deeper layers; moreover, they mitigate overfitting by providing a form of spatial downsampling.

For these reasons, alternating convolutional and pooling layers is a standard design pattern in many CNN architectures [9].

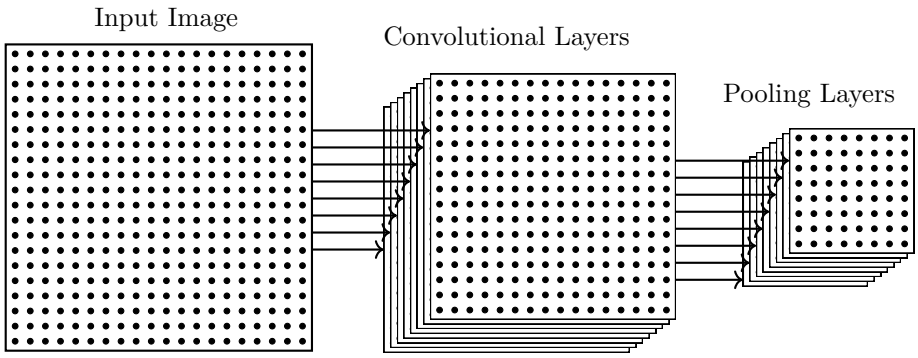

Figure 2.12. Same example containing all the elements.

2.2.4 Summary

Back to the structure in Fig. 2.7, a convolutional neural network can be viewed as a sequence of stages that gradually transform the raw image into a compact representation.

Starting from the input image, one or more *convolutional layers* apply learned filters over local receptive fields. Thanks to shared weights and biases, each filter acts as a feature detector that is scanned across the whole image, producing a corresponding feature map. Nonlinear activation functions (such as the sigmoid introduced earlier, or more commonly the rectified linear unit, ReLU, used in the thesis and discussed later) are then applied elementwise to these feature maps.

Pooling layers follow convolutional layers and operate independently on each feature map. By aggregating the information in small neighbourhoods (e.g., via max or average pooling), they reduce the spatial resolution while preserving the most salient responses. Alternating convolution and pooling stages builds a hierarchy of increasingly abstract features: lower layers capture simple patterns such as edges or corners, whereas deeper layers respond to more complex structures [9].

After several convolutional and pooling blocks, the resulting feature maps are flattened and fed into one or more *fully connected* layers, which play the role of a classical multilayer perceptron operating on the learned features rather than on raw pixels. The choice of the final activation and cost function depends on the task: for image classification, it is common to use a softmax output with a cross–entropy loss, whereas for regression problems a linear output layer with a quadratic cost is often preferred.

In all cases, the parameters of the convolutional, pooling (if any), and fully connected layers are trained by stochastic gradient descent and backpropagation, exactly as in the fully connected networks discussed in the previous section [9]. In the following, these ideas will be specialized to the robotics scenario considered in this work, where a CNN is used as a function approximator mapping camera images to suitable commands for the robot.

2.3 Robotics

Since robotics is at the core of this thesis, some basic concepts must first be introduced to fully understand the proposed approach.

A robot manipulator can be modeled as a *kinematic chain*: a sequence of rigid links ($n+1$, indexed from 0 to n) connected by joints (n, indexed from 1 to n), which are actuated by motors. Joints are typically either *revolute* (rotational) or *prismatic* (translational). One end of the chain is rigidly attached to the base, while the other end is equipped with a gripper or tool, usually referred to as the *end-effector*.

2.3.1 Direct Kinematics

The objective of *direct kinematics* is to determine the pose of the end-effector of the manipulator, i.e. its position and orientation in space. In practical terms, this corresponds to computing the homogeneous transformation matrix

$$T_e^b(q) = \begin{bmatrix} R_e^b(q) & r_e^b(q) \\ 0^{1\times 3} & 1 \end{bmatrix}, \tag{2.8}$$

which maps coordinates from the end-effector frame e to the base frame b, as shown in Fig. 2.13. Here $q = [q_1, \ldots, q_n]^T$ is the vector of joint variables, $R_e^b(q) \in R^{3\times 3}$ is the rotation matrix describing the orientation of the end-effector with respect to the base, and $r_e^b(q) \in \mathbb{R}^{3\times 1}$ is the position vector of the end-effector origin expressed in the base frame.

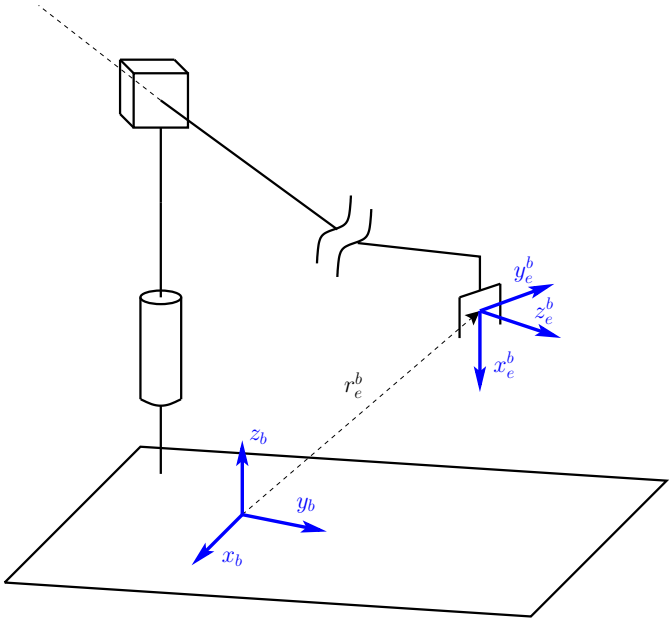

Figure 2.13. Direct Kinematics visual representation.

To simplify the computation of the direct kinematics, it is convenient to exploit the properties of homogeneous transformation matrices and express $T_e^b(q)$ as the product of the transformations between consecutive link frames. For a serial manipulator with n joints, this can be written as

$$T_e^b(q) = T_1^0\, T_2^1\, \dots\, T_n^{n-1}, \tag{2.9}$$

where T_i^{i-1} denotes the homogeneous transformation matrix from frame i to frame $i-1$.

If the reference frames are assigned according to the *Denavit–Hartenberg* (DH) convention, each matrix T_i^{i-1} depends on only four parameters and takes the form [11]:

$$T_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.10}$$

The four DH parameters have the following geometric meaning:

- a_i – *link length*: distance between the origins O_{i-1} and O_i measured along the axis x_i;
- α_i – *link twist*: angle between the axes z_{i-1} and z_i about x_i, positive when the rotation is counterclockwise (c.c.w.);
- d_i – *link offset*: coordinate of the origin O_i along the axis z_{i-1};
- θ_i – *joint angle*: angle between the axes x_{i-1} and x_i about z_{i-1}, positive when the rotation is counterclockwise (c.c.w.).

By specifying the DH parameters for each joint and link, the direct kinematics of the manipulator is completely determined through (2.9) and (2.10).

2.3.2 Differential Kinematics

Differential kinematics, instead, describes the relationship between the joint velocities of a manipulator and the corresponding linear and angular velocity of the end-effector. This relationship is encoded by the *geometric Jacobian* matrix, denoted as $J(q)$, which depends on the current joint configuration of the robot.

When the end-effector pose is expressed using a minimal parameterization in the operational space (such as Euler angles or position coordinates), an alternative formulation, known as the *analytical Jacobian*, can be derived by differentiating the direct kinematics function with respect to the joint variables. While both Jacobians describe velocity mappings, they are not equivalent in general due to the nonlinear transformation between angular velocity representations.

The Jacobian is a fundamental tool in robotics, as it plays a central role in several key areas: detection of singularities, analysis of kinematic redundancy, design of control laws, and computation of inverse kinematics solutions, as needed in this thesis. Its structure and properties greatly influence both the dexterity and stability of manipulator motion [11, 12, 13].

The Geometric Jacobian

The *Geometric Jacobian* establishes the linear relationship

$$v = \begin{bmatrix} \dot{p}_e \\ \omega_e \end{bmatrix} = J(q)\,\dot{q}, \tag{2.11}$$

between the *task-space velocity* v of the end-effector (linear velocity $\dot{p}_e$ and angular velocity ω_e) and the vector of *joint velocities* $\dot{q}$. In the case of a 6-DOF manipulator one can write

$$\begin{bmatrix} v_1 \\ \vdots \\ v_6 \end{bmatrix} = \begin{bmatrix} J_{11} & \cdots & J_{16} \\ \vdots & \ddots & \vdots \\ J_{61} & \cdots & J_{66} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_6 \end{bmatrix}, \tag{2.12}$$

where v_1, ...,v_6 denote generic components of the task-space velocity (either linear or angular) and J_{ij} are the entries of the Jacobian matrix evaluated at the current configuration q [11, 12, 13].

Each row of the Jacobian describes how all joint velocities contribute to a single component of the operational-space velocity. On the other hand, each column describes how the velocity of a single joint affects all components of the end-effector velocity. In particular, if an entire row of $J(q)$ is zero, then no motion occurs along the corresponding task-space direction, whereas if an entire column is zero, the i-th joint variable does not influence the end-effector motion at all.

The i-th column of the *geometric Jacobian* is given by

$$J_i(q) = \begin{bmatrix} J_{pi} \\ J_{oi} \end{bmatrix} = \begin{cases} \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix}, & \text{if joint } i \text{ is prismatic,} \\ \begin{bmatrix} z_{i-1} \times (p_e - P_{i-1}) \\ z_{i-1} \end{bmatrix}, & \text{if joint } i \text{ is revolute,} \end{cases} \tag{2.13}$$

where:

- z_{i-1} is the unit vector corresponding to the third column of the rotation matrix R^0_{i-1}, i.e., the joint axis z_{i-1} expressed in the base frame;
- p_e is the position of the end-effector origin, given by the first three elements of the fourth column of T^0_e;
- P_{i-1} is the position of the origin of frame $i-1$, given by the first three elements of the fourth column of T^0_{i-1}.

Thanks to the linearity of (2.11), the (local) inverse kinematics problem at the velocity level can be written as

$$\dot{q} = J(q)^{-1}\,v, \tag{2.14}$$

provided that $J(q)$ is square and nonsingular. In practice, this condition is not always satisfied: for redundant manipulators the Jacobian is rectangular, and even for square Jacobians there may exist configurations in which $\det J(q) = 0$. These *singular configurations* play a crucial role in the analysis and control of robot manipulators and require the use of generalized inverses (such as the Moore–Penrose pseudo-inverse) or alternative strategies, as discussed later in this chapter [11].

Kinematic Singularities

The Jacobian is, in general, a function of the configuration q; configurations at which $J(q)$ is rank-deficient are called *kinematic singularities*. When $J(q)$ loses rank, the mapping

$$v = J(q)\,\dot{q}$$

projects $\dot{q}$ onto a lower-dimensional subspace of the task space, so the components of v become constrained to each other.

Singularities are relevant because:

1. they represent a loss of mobility of the manipulator, i.e. it is no longer possible to impose an arbitrary motion of the end-effector;
2. they may lead to infinitely many or no solutions to the inverse kinematics problem.

In the square, nonsingular case, one can formally write

$$J(q)^{-1} = \frac{1}{\det\big(J(q)\big)}\, J^{\mathrm{adj}}, \tag{2.15}$$

where J^{adj} is the adjacency matrix. It is then immediate that, if $J(q)$ is not full rank, $\det J(q) = 0$, and the inverse is not defined. As $\det J(q)$ approaches zero, the elements of $J(q)^{-1}$ grow very large: even small task velocities v would require very high joint velocities $\dot{q}$. In reality, joint velocities saturate as the robot approaches a singular configuration [11].

Singularities may occur:

1. at the boundary of the reachable workspace (e.g. when the arm is fully stretched or fully folded);
2. inside the reachable workspace, where they are more difficult to foresee and avoid.

For manipulators with a spherical wrist, it is common to distinguish between *arm singularities* and *wrist singularities*. In practice, the numerical problems associated with singularities are often mitigated by replacing the exact inverse or pseudo-inverse of J with a *damped least-squares inverse*.

Singular Value Decomposition

Singular Value Decomposition (SVD) is a standard tool for studying the behavior of the Jacobian near singular configurations and for computing regularized inverses. Any $m \times n$ matrix J can be written as

$$J = U\,\Sigma\,V^{T}, \tag{2.16}$$

where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{m\times n}$ is of the form

$$\Sigma = \left[\begin{array}{ccc|c} \sigma_1 & & & 0 \\ & \ddots & & \vdots \\ & & \sigma_r & 0 \\ \hline 0 & \cdots & 0 & 0 \end{array}\right], \tag{2.17}$$

with $r = \operatorname{rank}(J)$ and singular values $\sigma_1 \geq \cdots \geq \sigma_r > 0$. Each singular value is related to an eigenvalue of JJ^T by

$$\sigma_i = \sqrt{\lambda_i}, \qquad \lambda_i \in \operatorname{eig}(JJ^T). \tag{2.18}$$

The (right) Moore–Penrose pseudo-inverse can be expressed as

$$J^+ = V\,\Sigma^+\,U^T, \tag{2.19}$$

where $\Sigma^+ \in \mathbb{R}^{n\times m}$ is obtained by inverting the nonzero singular values:

$$\Sigma^+ = \left[\begin{array}{ccc|c} \sigma_1^{-1} & & & 0 \\ & \ddots & & \vdots \\ & & \sigma_r^{-1} & 0 \\ \hline 0 & \cdots & 0 & 0 \end{array}\right]. \tag{2.20}$$

Near a singular configuration, one or more σ_i tend to zero, so the corresponding entries σ_i^{-1} in Σ^+ become very large and the pseudo-inverse is ill-conditioned. A common remedy is the *damped least-squares* inverse:

$$J^* = J^T\,(JJ^T + \lambda^2 I)^{-1}, \tag{2.21}$$

where $\lambda > 0$ is a damping factor and I is the identity matrix. In terms of singular values, this corresponds to replacing σ_i^{-1} with

$$\frac{\sigma_i}{\sigma_i^2 + \lambda^2},$$

which remains bounded even when $\sigma_i \to 0$. The result is a trade-off between accurate tracking of the desired task and bounded joint velocities, making the inversion numerically better conditioned in the neighbourhood of singular configurations [11].

Chapter 3

Instrumentation

The goal of this chapter is to place the manipulation task in a precise experimental context and to document the instrumentation used, so that the study can be reproduced.

Figure 3.1 depicts the whole physical setup used to design the control system, highlighting a *Meca500* manipulator, equipped with a *MEGP* end-effector, the *Rebel* endoscopic camera and a *Bota* force/torque sensor. Moreover, the picture sketches the planar workspace on which the target (a simple six-faced die) is placed.

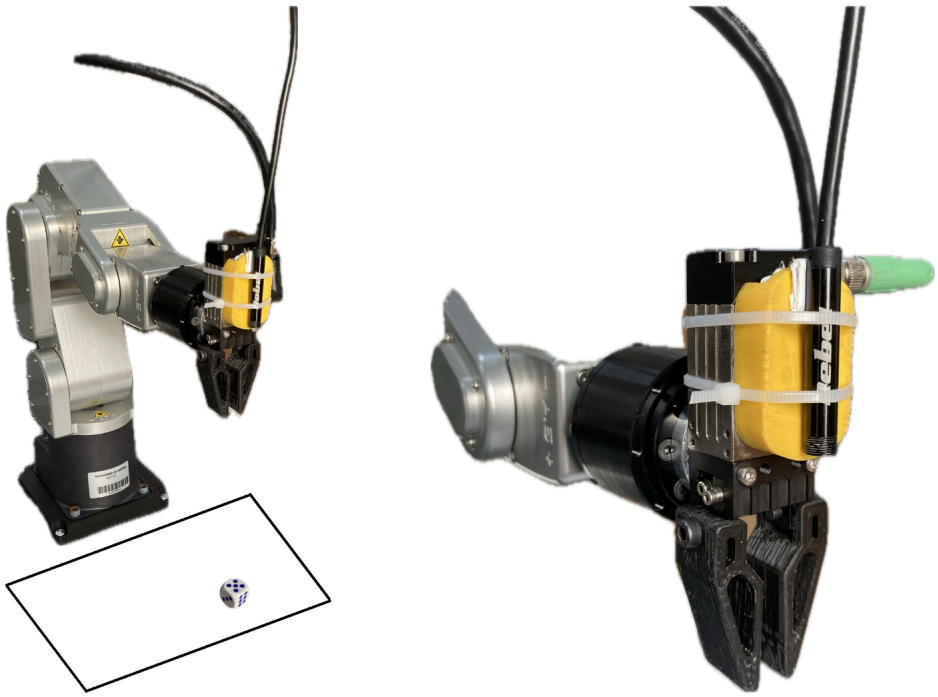

Figure 3.1. Experimental setup.

Beyond stating what hardware is used, it is essential to clarify why this configuration was chosen. The Meca500 offers precise joint-space velocity interfaces and low communication latency over TCP/IP, which are well matched to learning-based visual servoing where frequent, small velocity updates are preferred to sparse pose set-points. Mounting an endoscopic camera on the wrist keeps the target within the field of view during approach and contact, and its small form factor reduces mass and the risk of collisions.

The die is intentionally simple: strong pecularities, sharp edges, and repeated textures create realistic variations in appearance as illumination and viewpoint change. This combination yields compact yet representative characteristics for the experiments.

3.1 Meca500 Manipulator

The Meca500 (Fig. 3.2) is a compact six-axis industrial manipulator designed for high precision; the user manual specifies a repeatability of 0.005 mm, a rated payload of 0.5 kg (up to 1.0 kg under special conditions), a total arm mass of 4.6 kg, IP40 protection, 24 V DC supply with < 200 W maximum power [14].

Figure 3.2. Meca500 Industrial Robot - [14]

In the following Table 3.1 are reported speed and joint limits, taken again from the

manual user; these are notably generous, giving the arm broad workspace coverage and plenty of room to choose comfortable configurations, even when avoiding singularities or keeping the target within the camera's field of view.

The maximum joint speeds are also high for a robot of this size (fast wrists and large base/shoulder rates), enabling quick reorientation and short corrective bursts without feeling loose. In practice, this latitude translates into many feasible trajectories and IK solutions, with enough margin to absorb vision and communication delays while still maintaining smooth, responsive motion.

Joint	**Min angle** [°]	**Max angle** [°]	**Max Joint speed** [°/s]
J_1	-175	175	150
J_2	-70	90	150
J_3	-135	70	180
J_4	-170	170	300
J_5	-115	115	300
J_6	-36	36	500

Table 3.1. Meca500 joint limits and symmetric joint speed bounds (R3).

3.2 End-Effector

The robot is equipped with Mecademic's MEGP 25 electric parallel gripper (model 25E), a plug-and-play end-effector co-developed with *Schunk* for the *Meca500* (R3/R4). The gripper is natively detected by the controller, homes automatically with the robot, and is operated through simple commands [15].

Figure 3.3. MEGP 25E/25LS Electric Parallel Gripper - [15]

The custom fingertips used for this project were pre-designed with a CAD software and 3D-printed in a lightweight polymer. The design preserves the camera's field of view and ensures clearance for the wrist cable.

Using the gripper's dimensioned drawing (Fig. 3.3), the rigid offset from the robot flange to the geometric midpoint between the two fingertips was determined and absorbed into the sixth Denavit–Hartenberg link so that the forward kinematics report the TCP at the pinch center. In practice, the adapter thickness, the distance from the flange to the jaw centerline, and the fingertip protrusion are summed to obtain a tool translation $[\Delta x, \Delta y, \Delta z]$.
This translation is incorporated by updating the last-row parameters of the *DH* table (to be discussed later): $a_6 \to a_6 + \Delta x$ and $d_6 \to d_6 + \Delta z$, while Δy is assumed to be null.

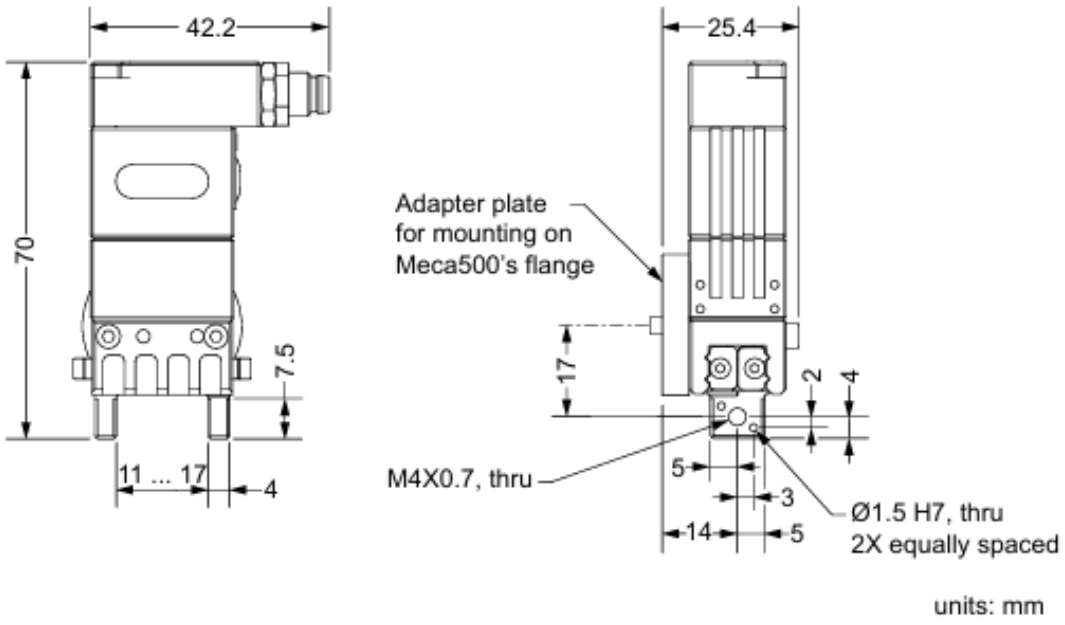

Figure 3.4. MEGP 25E gripper dimensions - [15]

3.3 Endoscopic Camera

The vision sensor used is shown in Fig. 3.5. It is a compact USB endoscope Rebel RB-1140 (480p) which delivers 640×480 images from a 1/5" sensor through a 7 mm-diameter probe with a 67° nominal field of view, powered directly from 5 V USB. The probe integrates six white LEDs around the optics to illuminate close-range scenes [16].

Figure 3.5. Endoscope Camera Rebel Tools RB-1140

3.4 Force sensor

The end-effector uses a Bota Systems Medusa 6-axis force/torque sensor (model *BFT-MEDS-SER-M8*). It is compact and light (about 48 mm in diameter and 32 mm tall, around 113 g), runs on 5 V at roughly 1 W, and operates from 0–55°C. Data are provided over USB/RS-422 through an M8 serial connector [17].

Performance is strong for a wrist-mounted device: force ranges are ±400 N on F_x, F_y and ±500 N on F_z; torque ranges are ±5 Nm on M_x, M_y and ±8 Nm on M_z. Overload limits are higher (up to 1000/1000/2000 N and 12/12/15 Nm). Typical accuracy is $< 2\%$, nonlinearity $< 0.2\%$, and at 100 Hz the noise-free resolution is about 0.18 N (x,y), 0.07 N (z), and 3.0/1.2 mNm (roll/pitch, yaw). With the serial interface, the sampling rate reaches up to 800 Hz.

The *side* cable exit helps with routing near joints J_5–J_6. The M8 connector exposes TX±, RX±, V^+, and V^-. The sensor works with common software stacks such as *ROS*, *MATLAB*, *Python*, *TwinCAT*, and LabVIEW, and is compatible with *Mecademic* robots too.

When interpreting measurements, the *combined-load* limits should be respected: the full range applies on a single axis, but allowable loads decrease when several axes are loaded at the same time. These characteristics make the Medusa well suited for contact-rich tasks such as compliant approach, contact detection, and light manipulation on a small arm.

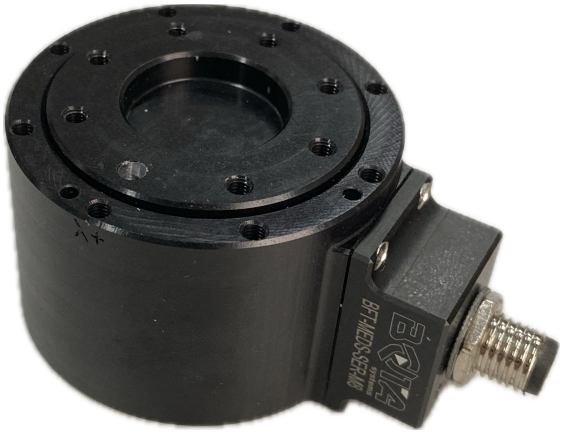

Figure 3.6. Medusa side 6-axis FT sensor with Serial interface - [17]

3.5 PyTorch

All neural network models developed in this work are implemented in PyTorch, an open-source deep learning framework that combines a Pythonic, imperative programming style with efficient CPU/GPU execution [18]. PyTorch is built around the notion of *tensors*, i.e. multi-dimensional arrays similar to NumPy arrays but with native support for automatic differentiation and hardware acceleration. This makes it possible to prototype and train complex models while keeping the implementation compact and readable.

In the context of this thesis, PyTorch is used to define and train the convolutional neural networks that map camera images to task-space velocity commands. The same framework is also employed to implement the loss functions, optimization routines, and evaluation scripts needed to analyse the learned models and integrate them into the real-time control loop of the robot.

Chapter 4

Implementation

The working principle of the proposed approach is illustrated in Fig. 4.1. The first sketch in the *Endoscopic Camera View* column represents the initial condition, before the manipulator starts moving. Once the control code is executed, the camera observes the target and the current image acquired by the endoscopic sensor is fed to the CNN, which predicts a normalized task-space velocity. This velocity command is then mapped into joint velocities through the inverse of the Jacobian, and the resulting joint rates are sent to the robot controller.

As a consequence, the manipulator moves along the red vector towards the target. At the next sampling instant a new image is acquired, the CNN is called again, and the procedure is repeated. The sequence continues until the end-effector reaches the desired pose, corresponding to the final image shown in the last sketch.

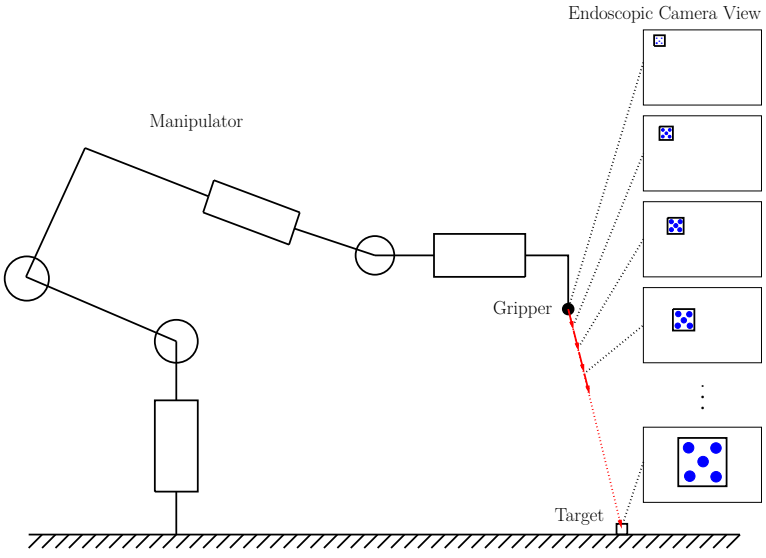

Figure 4.1. Schematic representation of the image-guided control scheme.

As a result, this procedure can be represented by the open-loop control scheme in Fig. 4.2, where v_{pred} denotes the task-space velocity predicted by the network and $\dot{q}$ is the vector of joint velocities obtained by mapping v_{pred} through the inverse of the Jacobian.

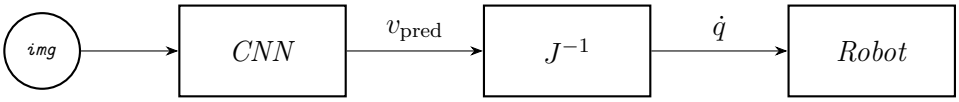

Figure 4.2. Open-loop control scheme.

If the network is trained on a sufficiently rich dataset, the gripper can reach the target for any admissible position within the workspace, even if the die is moved during the motion. Achieving this level of generalization, however, critically depends on how the dataset is constructed and on the variability of the examples it contains.

4.1 Dataset construction

The performance of the learned controller strongly depends on the quality of the training data. In this section, the procedure used to build the dataset is described in two steps. First, the acquisition process is detailed for a single target position. Then, the method is extended to generate data for multiple target locations across the workspace, enabling the network to generalize beyond a fixed configuration.

4.1.1 One target location

1. The initial pose of the end-effector, i.e. the position and orientation of the Tool Reference Frame (TRF) with respect to the Base Reference Frame (BRF), is selected so that the workspace is fully visible while the robot operates in a comfortable configuration. Figure 4.3 shows the chosen *base pose*, specified as [x_b, 0, z_b, 0, 90, 0], where the first three entries are expressed in millimetres and the last three in degrees, following the [X, Y, Z, R_X, R_Y, R_Z] convention.

   Geometrically, this pose is obtained by translating the origin O_b of the base frame by x_b millimetres along X_b and by z_b millimetres along Z_b, so that the gripper and the camera are roughly centred over the planar workspace and kept at a sufficient height for a clear field of view. The subsequent rotation of 90° about the Y_b axis aligns the tool frame in such a way that the camera optical axis points approximately towards the target area, while the manipulator remains far from joint limits and kinematic singularities. This configuration is used as the starting pose for the data acquisition procedure and for the experiments discussed in the remainder of the thesis.

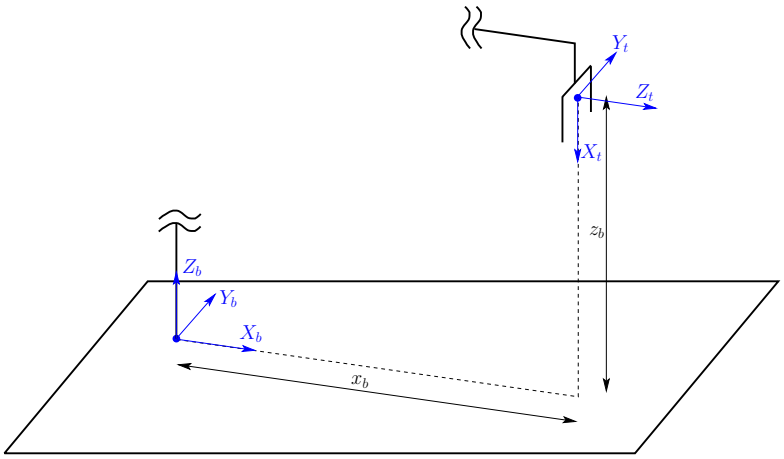

Figure 4.3. Initial relative pose between the TRF and the BRF.

2. A nominal *arrival* configuration must then be defined, corresponding to the pose in which the gripper is sufficiently close to the target. With the endoscopic camera connected to the computer and the live image displayed, the operator can move the die on the workspace and observe in real time how it appears in the camera window. A threshold value z_a is selected for the TCP z–coordinate: when the TCP reaches z_a, the control sequence is considered complete and the target is deemed reached.

   A straightforward way to determine the in–plane coordinates (x_a, y_a) would be to enforce fixed distances $\alpha, \beta, \gamma, \delta$ between the die edges and the image borders (Fig. 4.4), but this requires precise calibration of the image geometry. Instead, the arrival pose is chosen by exploiting the known geometry of the gripper fingers and of the die: the target configuration is defined so that, in the camera view, the die face and its dots appear in a repeatable alignment with the fingertips. This yields a visually well-defined and easily reproducible arrival configuration.

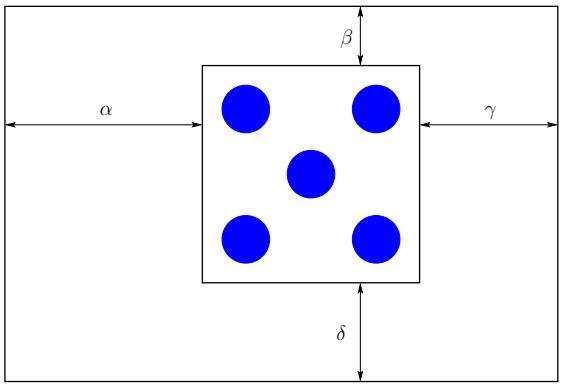

Figure 4.4. Endoscopic camera view at the nominal arrival configuration.

3. Once the initial and arrival configurations have been defined, a `MovePose` command, which "moves the TRF to a specified pose with respect to the BRF" [19], is sent to the robot in order to reach a convenient *target pose* above the workspace, illustrated Fig. 4.5. This intermediate pose is designed as follows. First, the z-coordinate of the tool is set to be exactly the same as in the nominal arrival configuration, that is z_a, ensuring that the vertical distance between the gripper and the plane of the die is fixed and known. Then, the x_a and y_a coordinates are selected arbitrarily within the reachable planar region, while the tool orientation is kept identical to that of the final arrival configuration. The robot therefore reaches a "hovering" pose at the correct height, but horizontally displaced with respect to the final grasp point.

   From this target pose, the operator manually places the die on the plane directly under the gripper. The placement is guided by the endoscopic camera view and by the geometric alignment between the fingertips and the visible face of the die, as mentioned before, so that when the robot later moves from the initial pose toward the arrival configuration, the target lies precisely along the expected approach direction and matches the visual conditions used during dataset collection.

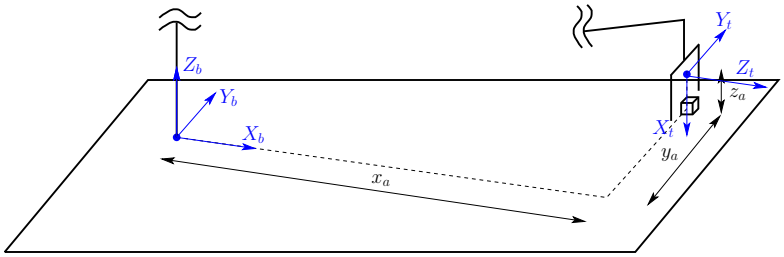

Figure 4.5. Target pose used to manually place the die under the gripper.

4. Now that the target has been placed under the gripper, a further `MovePose` command is sent to the robot to bring the manipulator back to the initial base pose $[x_b, 0, z_b, 0, 90, 0]$ of Fig. 4.3. This pose is used as the starting configuration for data collection. From here, a `MoveLin` command is issued, so that the end-effector "moves from the initial pose towards the nominal arrival pose along a straight line in Cartesian space", as sketched in Fig. 4.6 [19]. The linear speed of this motion is set via the instruction `SetCartLinVel`, which "sets the desired and maximum linear velocity of the robot TRF with respect to its WRF" [19],choosing a value that allows the camera to acquire a sufficient number of frames along the approach trajectory while keeping the duration of each run within practical limits.

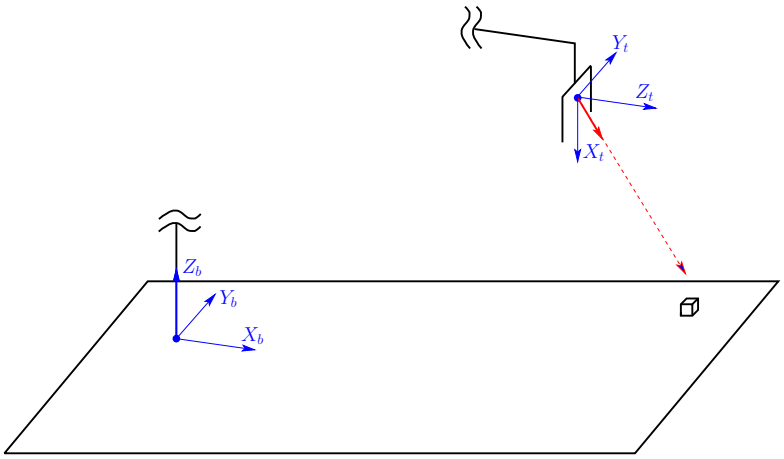

Figure 4.6. Start of the linear approach trajectory used for data collection.

5. At this stage, while the robot moves towards the target, the association between images and velocities is carried out. With a logging period T_ℓ, each camera frame is captured and, for every frame, the corresponding task-space velocity is read using the command `GetRtCartVel`. This function "returns the current Cartesian velocity $[v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]$ of the TRF with respect to the BRF, computed from the joint positions measured by the encoders" [14]. In this way, a sequence of image–velocity pairs $(\mathit{frame\_t}, v_t)$ is obtained, as illustrated in Fig. 4.7.

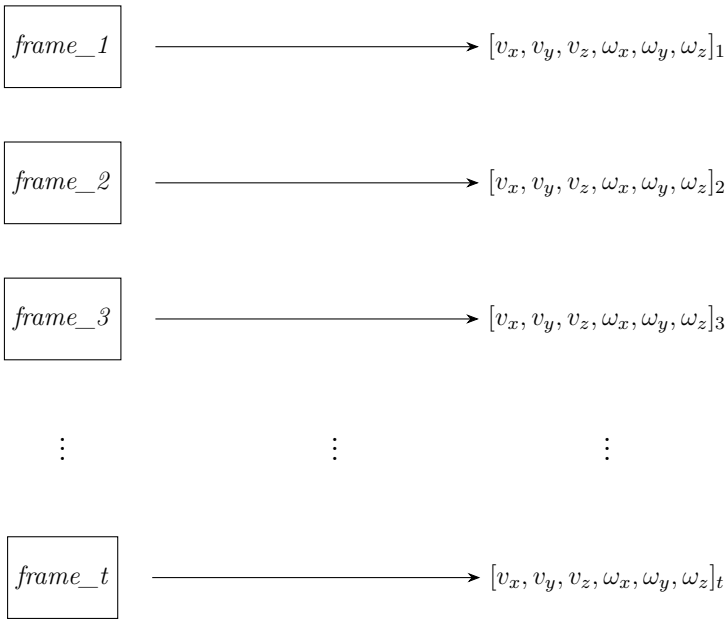

Figure 4.7. Captured frames and corresponding task-space velocity vectors.

6. Each acquired frame is saved in `.png` format, while the associated task-space velocity vector is normalized using the L_2 norm, treating the linear and angular components separately. Let

$$v = [v_x, v_y, v_z], \qquad \omega = [\omega_x, \omega_y, \omega_z], \tag{4.1}$$

The corresponding L_2 norms are

$$\|v\|_2 = \sqrt{\sum_{i=1}^{3} v_i^2}, \qquad \|\omega\|_2 = \sqrt{\sum_{i=1}^{3} \omega_i^2}. \tag{4.2}$$

To avoid numerical issues when the norm is very small, thresholds η_1 and η_2 are introduced. The normalized vectors are then computed as

$$\hat{v} = \begin{cases} \dfrac{v}{\|v\|_2}, & \text{if} \quad \|v\|_2 > \eta_1 \\ 0, & \text{otherwise} \end{cases}, \qquad \hat{\omega} = \begin{cases} \dfrac{\omega}{\|\omega\|_2} & \text{if} \quad \|\omega\|_2 > \eta_2, \\ 0 & \text{otherwise} \end{cases}.$$

In this way, non-zero velocity vectors are mapped to unit direction vectors, whose components lie in $[-1,1]$, while very small velocities are treated as zero. This representation makes the dataset independent of the absolute speed used during the data-collection runs: at deployment, the normalized task velocities predicted by the CNN can be scaled by any desired gain. Finally, each frame is paired with the six normalized components $(\hat{v}_x, \hat{v}_y, \hat{v}_z, \hat{\omega}_x, \hat{\omega}_y, \hat{\omega}_z)$ and stored in a `.csv` file, as summarized in Table 4.1.

Images	v_{norm1}	v_{norm2}	v_{norm3}	v_{norm4}	v_{norm5}	v_{norm6}
`img_1.png`						
`img_2.png`						
...	...	...	...	...	...	...
`img_t.png`						

Table 4.1. Dataset collection format.

At this stage, the dataset for a single target position is complete; the corresponding table contains t rows, one for each acquired frame. However, a network trained only on data collected from this single configuration would have limited generalization capabilities: it would learn to predict correct velocities essentially only when the target is placed at that same location. The next step is therefore to extend the procedure to multiple target positions (x_C, t_C) across the x-y plane, so as to enrich the dataset and improve the robustness of the learned controller.

4.1.2 Generalization

To enable the CNN to predict meaningful velocities for *any* admissible position of the target on the workspace, the dataset must cover a sufficiently large portion of the plane. For this purpose, the workspace is conceptually divided into a fictitious grid of $C = M \times N$ cells. The idea is to repeat the data–collection procedure described in the previous subsection for each cell, manually placing the die in the corresponding location while observing the endoscopic camera view displayed on the computer.

After completing the frame–velocity association for the first cell, the robot is translated by a fixed displacement Δy along the $-Y_b$ direction, while retaining the same coordinates x_a, z_a and the same tool orientation. This displaced hovering pose, illustrated in Fig. 4.8, preserves the same vertical geometry of the arrival configuration while shifting the horizontal projection of the gripper by a known amount.

From this new pose, the operator positions the die under the gripper exactly as before, using the endoscopic camera and the fixed arrival configuration (Fig. 4.4) as a geometric reference. Once the die is placed, the robot is returned to the original base pose and a new `MoveLin` command is executed. The acquired frames and the corresponding normalized task velocities are appended to the existing `.csv` file.

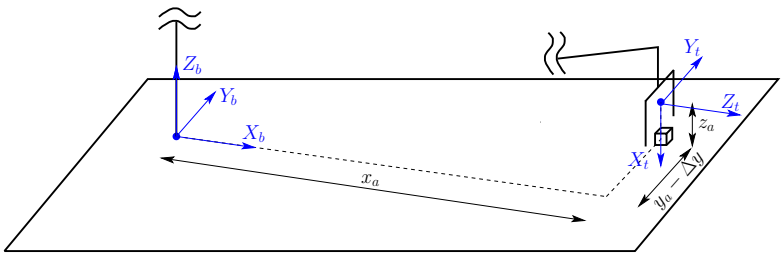

Figure 4.8. Target pose with displaced target position.

This procedure is repeated cell by cell. After covering all M cells along the first column (from row 1 to row M), the robot performs a step of size Δx along the X_b direction to reach the next column of the grid. From there, data collection continues for the second column, using the same vertical steps $\pm\Delta y$ to visit each row. The process is iterated in a scan–line fashion until every cell of the $M \times N$ grid has been visited and the corresponding set of frames and velocities has been acquired. Intuitively, a larger number of cells (i.e. smaller Δx and Δy) provides a denser sampling of the workspace and, in principle, higher prediction accuracy.

Figure 4.9 illustrates the resulting camera views. The first row schematically represents the grid on the workspace and the successive manual placements of the die in different cells, as seen by the base pose. The lower rows show the corresponding endoscopic images along the approach trajectories, from initial distant views (with the die appearing in a corner of the image) to final close-up views at arrival.

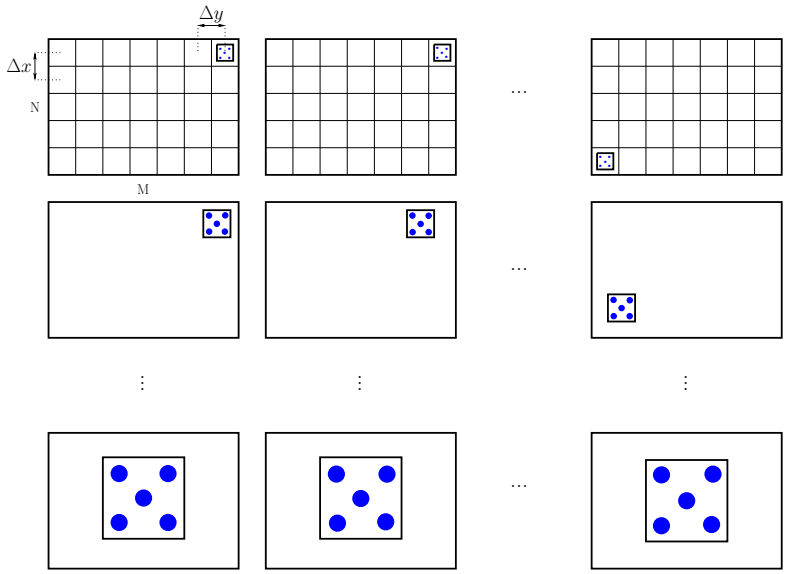

Figure 4.9. Endoscopic camera views obtained while scanning the grid of target positions.

A practical limitation of this procedure is the amount of manual work required when the grid is fine. For example, if the workspace is discretized into a 50×50 grid with steps $\Delta x = \Delta y = 1$ mm, the operator would need to reposition the die $C = 2500$ times and execute a full approach trajectory for each cell. This is time-consuming and not scalable. To mitigate this problem, a solution based on two separate neural networks is proposed in the following sections, relying on two different datasets and training stages.

4.1.3 Two-network approach

In order to reduce the amount of data required while still achieving good accuracy near the target, a *two-network* strategy is adopted (Fig. 4.10). Two distinct CNNs are trained and used in sequence:

- a first "*coarse*" network, responsible for approaching the target from a generic starting configuration;
- a second "*refinement*" network, dedicated to the final part of the motion, when the end-effector is already close to the die and higher accuracy is needed.

The switching between the two models is based on the height of the TRF with respect to the BRF. Another threshold h is defined on the z-coordinate of the TCP. As long as $z > h$, the velocity command is generated by the first CNN, which brings the end-effector from the initial pose to a neighbourhood above the target. When the TCP crosses the threshold ($z \leq h$), the control loop switches to the second CNN, which refines the motion and drives the robot towards the arrival configuration. In the picture, the arrival pose of the first CNN, z_{a1} is the same as the base pose for the second CNN, z_{b2}.

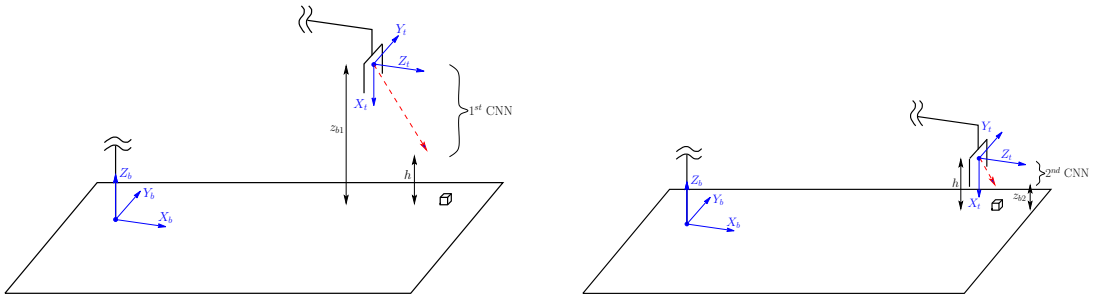

Figure 4.10. Illustration of the two-network approach, coarse and refinement phase.

The two networks are trained on datasets obtained with the same collection pipeline described in the previous subsections, but using different grid parameters, as sketched in Fig. 4.11. For the first network, which only needs to provide a rough approach, the grid on the workspace is relatively coarse: the displacements Δx_1 and Δy_1 between neighbouring cells are larger, so the total number of cells $C_1 = M_1 \times N_1$ is smaller. This reduces the number of required trajectories and manual placements of the die, while still giving the CNN a global view of the workspace.

The second network, instead, is responsible for accurate positioning in the vicinity of the target. Here the grid is finer, with smaller steps Δx_2 and Δy_2 and consequently a larger number of cells $C_2 = M_2 \times N_2$. In this way, the dataset for the refinement network densely samples the set of possible target locations, allowing the model to learn precise corrections when the die is already close to the centre of the image.

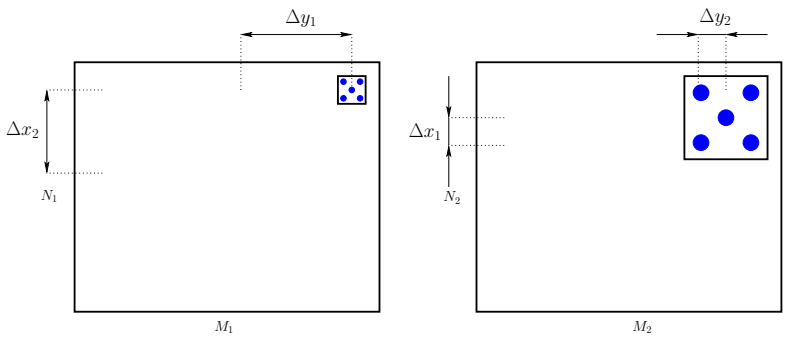

Figure 4.11. Grid parameters for the two-network approach.

This strategy requires collecting two separate datasets and training two distinct networks, which increases the overall effort and calls for careful tuning of both models. However, the structure of each dataset remains the same: at the end of the collection phase, there is a folder containing all the `.png` frames and a `.csv` file with the corresponding six normalized velocity components, stored in the format illustrated in Table 4.1.

4.2 Training phase

Once the dataset has been assembled, a dedicated script handles the training phase. The overall pipeline first partitions the data into training and validation subsets, then configures a set of hyperparameters and callbacks, and finally instantiates and optimizes the CNN model.

4.2.1 Dataset partitioning and shuffling

Before training, the dataset is split into two disjoint subsets: a *training* set, used to update the network weights, and a *validation* set, used only for evaluation and tuning of hyperparameters during development. The validation set provides an unbiased estimate of generalization, enabling detection of overfitting, model selection, and hyperparameter tuning while training is still in progress [20].

To obtain representative subsets, the dataset is randomly shuffled prior to the train-validation split. Many acquisition pipelines produce temporally or spatially ordered samples (e.g., consecutive frames along a robot trajectory or blocks of images under similar lighting). Splitting such ordered data without randomization would systematically assign the early samples to one subset and the later samples to the other, biasing the validation estimate. In implementation, a single permutation is generated and applied consistently to both the image filenames and their labels before performing, for example, an 80/20 split.

4.2.2 Training hyperparameters

Several hyperparameters must be specified before launching the optimization, which will be tuned :

- **Resolution of the input frames**
  The spatial resolution of the input frames after decoding and resizing fixes the geometric detail available to the network and determines the computational cost per batch. In this work, images are resized to 240×320 pixels (aspect ratio of the *Rebel RB-1140* camera preserved), which offers a good compromise: the model trains quickly and fits within modest hardware constraints while still retaining the main structural cues of the scene. Increasing the resolution would preserve finer details (e.g., small gripper edges or subtle texture gradients) but would also increase the number of activations and floating-point operations almost proportionally to the pixel count, slowing training and potentially requiring smaller batch sizes. Because geometric consistency matters in vision-based robotics, the aspect ratio is kept constant across the dataset.

- **Batch size**
  The batch size controls how many samples contribute to a single gradient update, trading off gradient noise against throughput and memory usage. A relatively small batch, such as 10, injects beneficial stochasticity into the gradients, which often improves generalization in regression problems with noisy labels, and ensures that

the pipeline runs even on limited GPU/CPU resources. Larger batches can exploit hardware parallelism more effectively but may require readjusting the learning rate and can sometimes lead to sharper minima that generalize worse [21]. Since the architecture does not employ batch normalization, very small batch sizes do not affect running statistics, making a batch of 10 a safe and pragmatic default.

- **Number of epochs**
  The number of epochs defines an upper bound on how many full passes over the training set are allowed. Too few epochs lead to underfitting, with both training and validation errors remaining high; too many epochs without safeguards lead to overfitting, where the training loss keeps decreasing while the validation loss increases. Here the epoch limit is combined with validation-driven callbacks (learning-rate scheduling and early stopping) so that training typically terminates automatically when validation performance stops improving. In this setting, the epoch count acts mostly as a safety cap rather than a precise stopping rule.

4.2.3 Callbacks

Three callbacks manage generalization and optimization during training: *early stopping*, *learning-rate scheduling*, and *checkpointing*.

- **Early stopping**
  As training progresses, the network tends to reduce the loss on the training set, but after a certain point the validation loss (used as a proxy for generalization) may stagnate or even worsen. Early stopping monitors a held-out metric, the validation loss in this case, and terminates training once the metric has failed to improve for a specified number of epochs, optionally restoring the best weights observed so far [22]. This prevents unnecessary overfitting and saves computational time.

- **Learning-Rate scheduling**
  When the validation loss stops improving, the current learning rate may be too large for the optimizer to settle into a better local minimum. A learning-rate scheduler addresses this by automatically reducing the learning rate when no progress is detected for several epochs. After each epoch, the monitored metric is checked; if there is no meaningful improvement for a given patience, the optimizer's learning rate is multiplied by a factor, subject to a minimum value. An optional cooldown period prevents repeated rapid reductions.

- **Checkpointing**
  Because overfitting can occur after the best validation performance has been reached, checkpointing saves the model whenever the monitored validation metric improves. With the `save_best_only=True` option, each improvement overwrites the previous best model on disk, ensuring that, at the end of training, the weights corresponding to the best validation epoch are available even if subsequent epochs degrade performance.

4.2.4 CNN model

After defining the data split, hyperparameters, and callbacks, the script instantiates the convolutional neural network that maps each camera frame to a 6-D task–space velocity vector. The overall processing pipeline is sketched in Fig. 4.12: a 320×240 RGB image is decoded and normalized, passed through a stack of convolutional and dense hidden layers, and finally mapped to the six output components $(\hat{v}_1, \ldots, \hat{v}_6)$.

To study how model capacity influences the fit, a sequence of architectures is considered, starting from a lightweight configuration (few filters/neurons) and progressively increasing the number of channels, layers, and hidden units. This incremental procedure makes it possible to observe how training and validation errors evolve and to select the smallest architecture that achieves strong validation performance without clear signs of underfitting or overfitting.

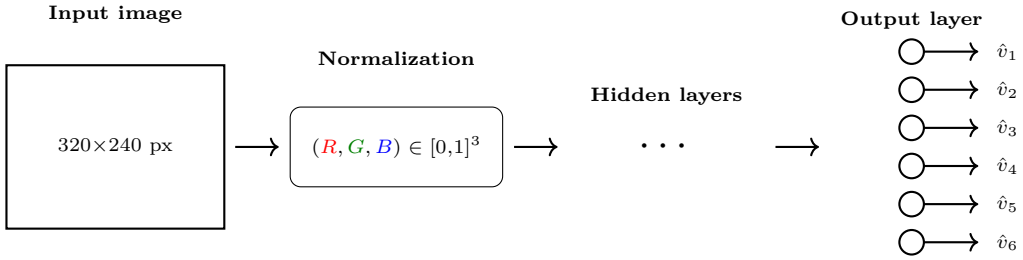

Figure 4.12. General CNN architecture implemented.

All architectures share the same input specification, `layers.Input(shape=(img_height, img_width, 3))`, where the last dimension equals 3 because each frame is encoded as an RGB image. In the preprocessing pipeline, `tf.image.decode_png(..., channels=3)` converts all images to three-channel RGB, which are then scaled to [0,1] by division by 255.0. This guarantees that every model in the capacity sweep receives inputs with the same spatial resolution, number of channels, and numerical range, so that differences in performance can be attributed to architectural choices rather than to changes in preprocessing.

ReLU Activation Function

Hidden layers use the Rectified Linear Unit ($ReLU$) activation, $\mathrm{ReLU}(z) = \max(0, z)$, applied elementwise (Fig. 4.13). ReLU leaves positive activations unchanged and sets negative activations to zero, providing a simple and computationally efficient nonlinearity that largely mitigates vanishing–gradient issues compared to sigmoid or tanh functions [23, 24]. In this vision–based regression setting, ReLU is particularly advantageous: its gradients do not saturate, which prevents the slowdown typical of sigmoidal activations; it avoids artificially compressing internal feature values into fixed intervals, preserving a wider range that is useful when the target velocities span different magnitudes; moreover, when paired with a linear output layer, it induces a flexible mapping from pixels to the

six velocity components, an appropriate inductive bias for locally linear image–motion relationships.
Even though hidden activations are non–negative, the final `Dense(6)` layer is linear, so positive and negative velocities emerge naturally from weighted combinations of ReLU features.

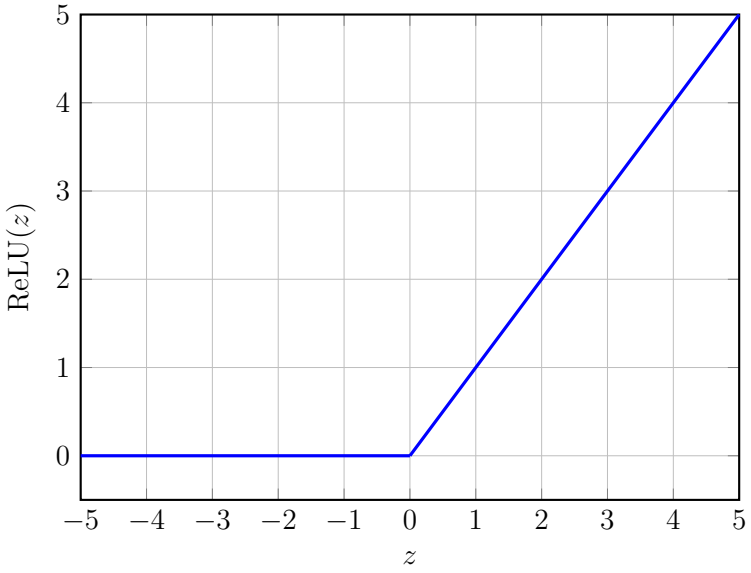

Figure 4.13. ReLU activation function.

Adam Optimizer

Optimization is performed with the Adam algorithm [25], which extends stochastic gradient descent by maintaining separate adaptive learning rates for each parameter. For a parameter vector θ and gradient $g_t = \nabla_\theta \mathcal{L}_t(\theta)$ at iteration t, Adam maintains exponentially decaying moving averages of the first and second moments of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t, \tag{4.3}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2. \tag{4.4}$$

Bias-corrected estimates are then formed as

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \tag{4.5}$$

and the parameters are updated with an elementwise adaptive step

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \tag{4.6}$$

Here $\hat{m}_t$ acts as a momentum term (a smoothed descent direction), while $\hat{v}_t$ rescales the step size coordinate-wise by an estimate of gradient magnitude, yielding per-parameter

learning rates. The default settings $\alpha = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$ are used, which are known to work well across a wide range of deep-learning problems.

Adam is computationally efficient (linear in the number of parameters), has modest memory overhead (two extra vectors of the same size as θ), and performs well on non-stationary objectives and problems with sparse gradients, which makes it a natural choice for CNN-based regression tasks such as the present one.

Loss function: Mean Squared Error

The mean squared error is used as the training criterion. For a batch of size N and six outputs per sample (three translational and three rotational task-space velocity components), the loss takes the form

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{6} \sum_{k=1}^{6} \left(y_{ik} - \hat{y}_{ik}\right)^2, \tag{4.7}$$

which corresponds to the standard `'mse'` loss in Keras. MSE can be interpreted as the negative log-likelihood under an i.i.d. Gaussian noise model on the residuals and yields smooth gradients that are proportional to the residuals: large errors receive larger corrective updates, which is desirable when the goal is to match continuous velocities.

In practice, plain MSE is used for optimization, while mean absolute error (MAE) is monitored for interpretability in native units. If different output components had markedly different scales, a standardized or weighted MSE could be adopted without changing the rest of the pipeline.

Capacity and regularization: Overfitting vs. Underfitting

To manage the bias–variance trade-off, the training strategy proceeds in two phases. First, a deliberately *larger* CNN is considered, with enough filters and hidden units to rule out underfitting: the model must be capable of driving the training loss and MAE down and of clearly learning the image-to-velocity mapping. Once the learning curves show low training error but only modest improvements on the validation set, the capacity is deemed sufficient.

In the second phase, regularization is introduced to control overfitting. Dropout [26] with a rate in the range 0.3–0.5 is applied in the regression head, randomly zeroing a fraction of activations during training to discourage co-adaptation and improve generalization (dropout is inactive at validation and test time). In addition, early stopping on the validation loss with `restore_best_weights=True` halts training near the epoch that minimizes the validation error and restores the corresponding weights. This two-step procedure— starting with ample capacity and then regularizing with dropout plus early stopping— first avoids underfitting and then reduces variance, yielding a compact model with the best validation performance observed in practice.

4.3 Jacobian Inversion

Once the CNN is properly trained, it outputs the six components of the end–effector velocity (the task–space twist). These task–space velocities could in principle be used directly for control, but in this work they are first mapped into joint velocities, which are the commands actually executed by the robot motors. This mapping is obtained via the Jacobian, and there are several advantages to working in joint space.

First, the Jacobian–based conversion makes it possible to introduce the damped least squares inverse, which regularizes the kinematic inversion near singularities and prevents the joint velocities from blowing up when the manipulator passes through poorly conditioned configurations. Second, joint space is the natural domain for enforcing per–joint limits and safety checks: after the inversion, each component of $\dot{q}$ can be saturated or filtered according to the hardware specifications of the corresponding actuator. Finally, if the manipulator is kinematically redundant, joint–velocity control provides direct access to the null space of the Jacobian, allowing posture optimization terms to be added (e.g., to keep the robot away from joint limits or from uncomfortable configurations) without affecting the primary task.

To implement this mapping, a Python module `jacobian_computation.py` is used throughout the tests. It defines two core functions: `dh_transform`, which computes the homogeneous transformation T_{i-1}^{i} from link $i-1$ to link i given the Denavit–Hartenberg parameters, and `compute_jacobian`, which chains these transforms to evaluate the forward kinematics and assemble the geometric Jacobian at the current joint configuration.

Since no official DH parameterization is provided for the Meca500, the reference frames and corresponding parameters are assigned manually, following the rules recalled in Section 2.3.1. The resulting DH frames and axes are shown in Fig. 4.14.

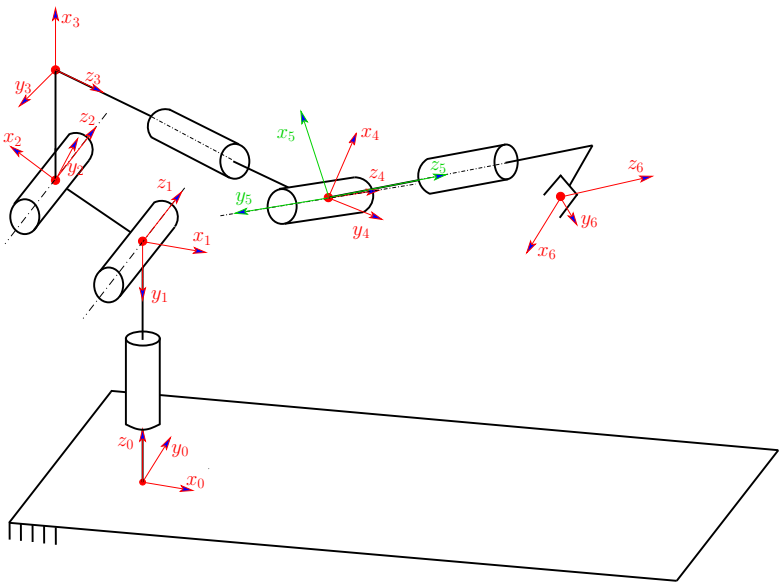

Figure 4.14. Placement of the reference frames for Denavit–Hartenberg parameterization.

Once the reference frames have been assigned, the next step is to compute the corresponding Denavit-Hartenberg parameters. Since the manipulator has six degrees of freedom and all joints are rotational, the joint variables are defined as:

$$\theta_i = q_i \qquad \text{for} \qquad i = 1, \ldots, 6. \tag{4.8}$$

The remaining DH parameters are derived based on the information provided in Fig. 4.15, which shows the link lengths (in millimeters) required to assign the parameters. The figure is sourced from the Meca500 user manual [14].

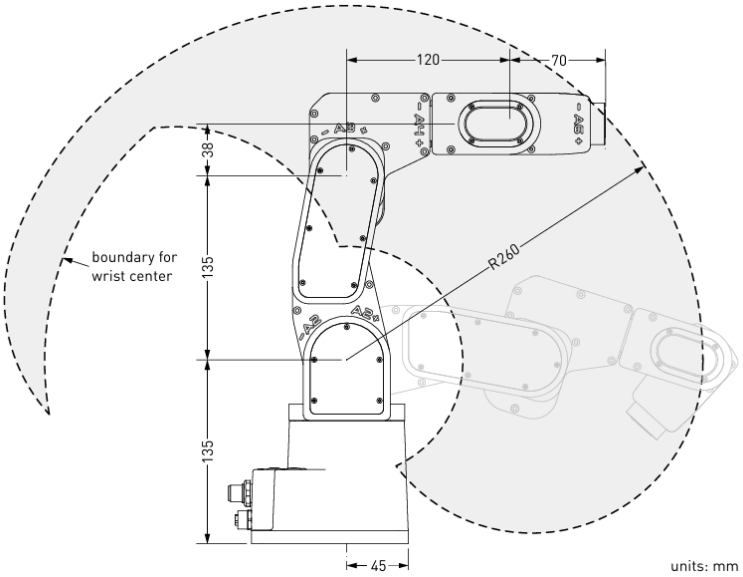

Figure 4.15. Dimensions of the Meca500.

The resulting DH parameters are summarized in Table 4.2. Note that for joints 2 and 6, an intrinsic joint offset must be accounted for. This offset is subtracted from the corresponding joint variable, as indicated in the table.

Link	$\boldsymbol{a}$[mm]	$\boldsymbol{\alpha}$[°]	$\boldsymbol{d}$[mm]	$\boldsymbol{\theta}$[°]
O_1	0	-90	135	q_1
O_2	135	0	0	$q_2 - 90$
O_3	38	-90	0	q_3
O_4	0	90	120	q_4
O_5	0	-90	0	q_5
O_6	55	0	125	$q_6 - 180$

Table 4.2. Denavit-Hartenberg parameters for the Meca500 manipulator.

So, the `dh_transform` function takes as input the defined Denavit-Hartenberg parameters and outputs the transformation matrix T_{i-1}^{i}, as shown in Equation 2.10. This matrix is essential for computing the vectors of frame origins and axes along the kinematic chain. Specifically, the function `compute_jacobian` computes the cumulative transformation T_{i-1}^{i} for each link in the chain. At each step, it calculates the frame origin $p_{i-1} = (T_{i-1}^{i})_{0:3,3}$ and the joint axis $z_{i-1} = (T_{i-1}^{i})_{0:3,2}$. By chaining the transformations T_{i-1}^{i} for all links, the end-effector position $p_e = (T_6^0)_{0:3,3}$ can also be obtained. These quantities are then used to assemble the geometric Jacobian column by column for all revolute joints, as shown in Equation 4.9:

$$J(q) = \begin{bmatrix} z_0 \times (p_e - p_0) & z_1 \times (p_e - p_1) & \cdots & z_5 \times (p_e - p_5) \\ z_0 & z_1 & \cdots & z_5 \end{bmatrix} \tag{4.9}$$

In the main script, a `renormalize` function is defined. It takes the 6-D vector from the CNN and projects it onto the unit sphere, preserving direction while discarding magnitude. The function first computes the L2 norm of the vector; if that is greater than the thresholds η_1 and η_2, it returns the normalized vector $\hat{v} = \frac{v}{\|v\|_2}$, otherwise, it returns a zero vector to avoid division by (near) zero. This normalized vector is then transformed into joint velocities using the inverse of the Jacobian matrix.

4.4 Velocity Smoothing

To attenuate high-frequency fluctuations and reduce jerk in the joint-space commands produced by the vision-based pipeline, the commanded joint velocities are filtered using a first-order exponential moving average (*EMA*), following the 'One Euro' filtering approach for real-time signals [27]. Let $q_k \in \mathbb{R}^6$ be the raw joint velocity vector at control step k (after Jacobian mapping, normalization, and saturation), and let q_k^{sm} be the smoothed command that is actually sent to the robot. The update rule for the smoothing is given by

$$q_k^{\text{sm}} = (1 - \alpha)\, q_{k-1}^{\text{sm}} + \alpha\, q_k, \tag{4.10}$$

where $\alpha \in (0,1]$ is the smoothing coefficient (`smoothing_alpha` in the code). Equation (4.10) is the standard discrete first-order low-pass filter, with a pole at $1 - \alpha$. For a loop period Δt, the filter's equivalent continuous-time time constant is

$$\tau \approx -\frac{\Delta t}{\ln(1 - \alpha)}, \tag{4.11}$$

and the -3 dB cutoff frequency is $f_c \approx \frac{1}{2\pi\tau}$. In the implementation, $\Delta t = 0.05$ s and $\alpha = 0.5$, yielding $\tau \approx 0.072$ s and $f_c \approx 2.2$ Hz. Consequently, a step change in q_k settles to within approximately 95% of its final value in about $3\tau \approx 0.22$ s (roughly five control iterations).

The filter is applied at the end of the command chain:

$$\text{CNN image} \rightarrow \; v \rightarrow \dot{q} \xrightarrow{\text{norm.} + \text{clip}} q_k \xrightarrow{\text{EMA}} q_k^{\text{sm}} \rightarrow \texttt{MoveJointsVel}.$$

Saturation is applied *before* smoothing, ensuring that both q_k and q_k^{sm} remain within the joint speed limits. The initial condition $q_0^{\mathrm{sm}} = 0$ provides a smooth ramp from rest, and the same filter naturally mitigates transients when switching between the coarse and refine networks. The selection of α follows the typical trade-off: setting $\alpha \to 1$ minimizes lag (i.e., no smoothing), whereas smaller values of α increase attenuation at the cost of responsiveness. If a target time constant $\tau^\star$ is preferred, α can be set as

$$\alpha = 1 - e^{-\Delta t/\tau^\star}. \tag{4.12}$$

The final smoothed velocity command q_k^{sm} is then sent to the robot as the joint velocity command through the `MoveJointsVel` command.

4.5 Analyzing Different Initial Conditions

It is interesting to examine the behaviour of the system when the initial conditions differ from those used during dataset collection. Such tests are useful not only for evaluating the overall robustness of the control loop, but also for revealing potential failure modes and identifying operating conditions in which the controller might behave differently than expected.

In the following, the three Cartesian coordinates of the TCP with respect to the BRF are analyzed separately, since variations along different directions may highlight distinct aspects of the system.

4.5.1 Different z-coordinate

In particular, this subsection considers the case in which the z-coordinate of the TCP with respect to the BRF is larger than in the original base pose used during data collection. In the current setup, the target is always placed on the plane generated by the vectors X_b and Y_b. As a consequence, in all training examples the die lies below the camera, and the predominant component of the learned task velocity is directed along the negative Z_b–axis. Therefore, even if the base z–coordinate is chosen to be larger, the CNN still tends to predict a motion that drives the TCP downwards toward the X_b–Y_b plane.

Figure 4.16 shows a different initial condition, in which the TCP starts at a higher altitude $z_+ > z_b$. Once the control loop is started and the first image is fed to the network, the CNN outputs a velocity whose vertical component points towards $-Z_b$, and the robot begins to descend, independently of the exact camera resolution or of moderate variations in target visibility. As the TCP approaches the plane and the viewpoint becomes closer to those seen during training, the frames enter the distribution on which the CNN was trained, and the subsequent motion follows the same qualitative behaviour as in the nominal case.

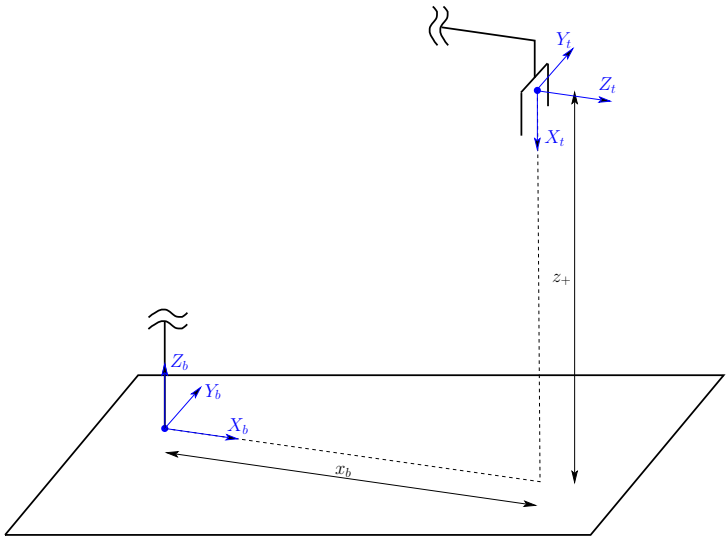

Figure 4.16. Base pose with a larger z-coordinate.

Along the motion shown in Fig. 4.16, there exists a configuration in which the joint variable $q_5 \approx 0°$. This situation corresponds to the robot being in, or very close to, a *wrist singularity*. In such a configuration, the axes of joints 4, 5, and 6 become nearly collinear, causing a loss of rank in the Jacobian associated with the spherical wrist and making certain end-effector angular velocities either uncontrollable or solvable only through unbounded joint rates. This configuration is illustrated in Fig. 4.17.

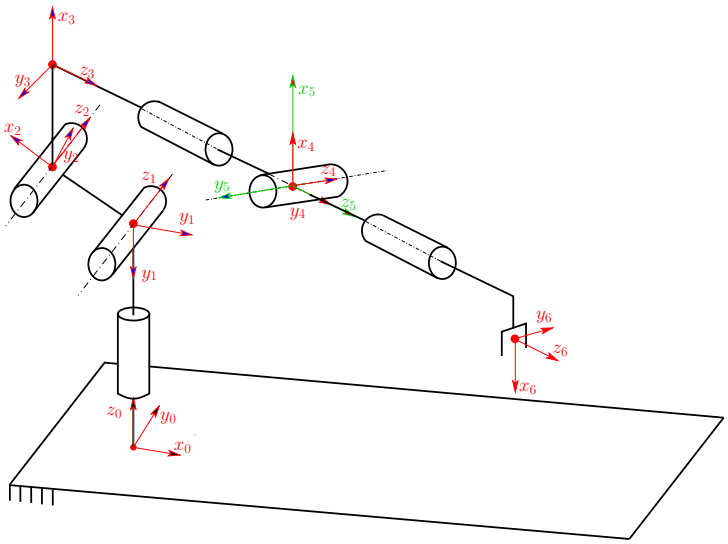

Figure 4.17. Wrist singularity configuration of the robot.

To robustly handle this situation, the control loop employs the *Damped Least Squares* (DLS) inverse kinematics method. The Jacobian $J(q)$ is computed as described in Section 4.3, but its inversion is carried out using the regularized map

$$\dot{q} = J^T \left(JJ^T + \lambda^2 I\right)^{-1} v_{\text{pred}},$$

where the damping factor λ introduces a positive-definite term that prevents numerical instability in the vicinity of singularities by compensating for small singular values of JJ^T. This regularization keeps the inverse kinematics problem well-conditioned even when the wrist axes align, ensuring that the commanded joint velocities remain bounded and physically executable.

In practice, the use of DLS avoids the characteristic joint–velocity explosions that an inverse or pure pseudo-inverse would produce when $q_5 \to 0°$. As a consequence, the robot can traverse regions of poor manipulability smoothly and continue its motion toward the target without oscillations or instability.

4.5.2 Different x and y coordinates

Instead of modifying the z-coordinate, this subsection examines variations of the TCP along the X_b and Y_b axes. From the camera's perspective, the only quantity that matters is the relative position of the target within the image plane. Consequently, the same camera frame can be obtained in two distinct ways: either by translating the die on the workspace, or by translating the TCP in the opposite direction by the same amount ΔS, as illustrated in Fig. 4.18.

During dataset collection, the target was placed manually at different positions across the plane, exposing the CNN to a wide variety of image configurations. However, the network receives only the image as input and has no knowledge of the robot's absolute starting pose. Therefore, it cannot distinguish whether a given image results from moving the die while keeping the base pose fixed, or from modifying the robot's initial pose while keeping the die fixed. If the target occupies the same pixel region, the CNN interprets the two situations as identical, producing the same predicted task–space twist. In this sense, the CNN implicitly treats the two operations as equivalent, because the image alone does not encode how the relative geometry between robot and die was produced.

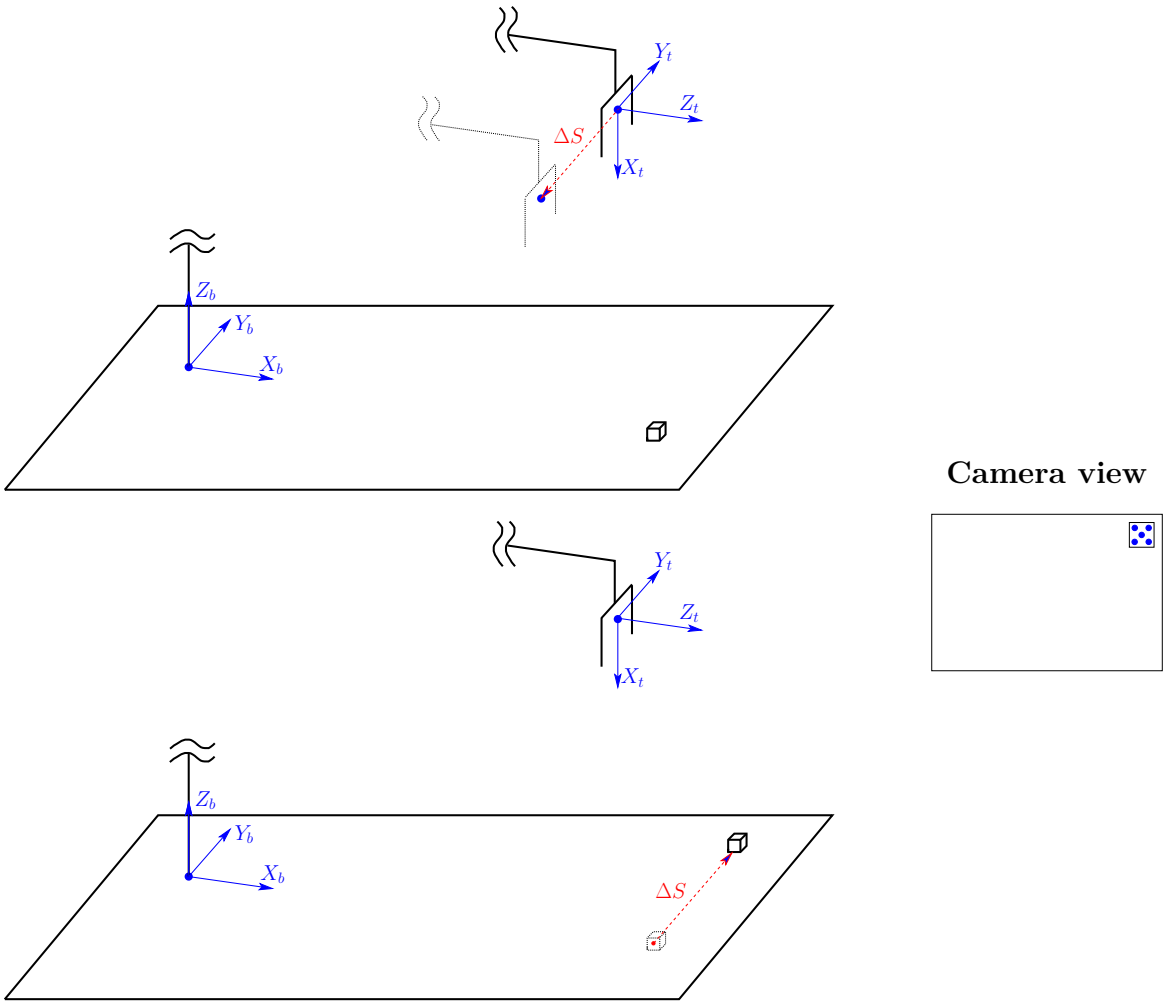

Figure 4.18. Different base–pose analysis: moving the TCP vs. moving the target.

All the implementations presented in this chapter must operate consistently across such situations. The two–network structure (coarse approach followed by refinement) and the DLS Jacobian inversion must remain reliable regardless of the initial base pose. In particular, when different starting poses produce identical camera views, the pipeline is expected to compute essentially the same joint–space commands and to guide the manipulator toward the same final configuration, independently of how the relative position between robot and target was originally achieved.

Chapter 5

Additional Control Branch

Motivated by industrial scenarios in which the end–effector may need to be gently repositioned while the task is ongoing, an *admittance–like* branch is added to the baseline CNN–driven architecture. As sketched in Fig. 5.1 (where the symbol "\" denotes a switch block), the CNN maps the camera image to the task–space v_{pred}, whereas the measured external wrench F drives a virtual mass–damper model $A\,\ddot{x} + D\,\dot{x} = F$ to produce a compliant velocity v_{adm}. The switch routes either v_{pred} or v_{adm} to the kinematic inverse $\dot{q} = J^{-1}(q)\,v$: in free space it passes the vision command; when interaction is detected ($\|F\| > F_{\text{th}}$) or a manual "move" request is issued, control authority is handed to the admittance branch to ensure safe, dissipative motion.

The design aims are to preserve fast vision–driven behavior away from contact, to introduce mechanical compliance during interaction without ad–hoc heuristics, and to keep the implementation real–time friendly by using diagonal, positive A and D (no stiffness term, avoiding bias in pure velocity control). The remainder of the chapter formalizes the switching logic, provides tuning guidelines for A and D, and evaluates the impact on tracking accuracy and interaction safety.

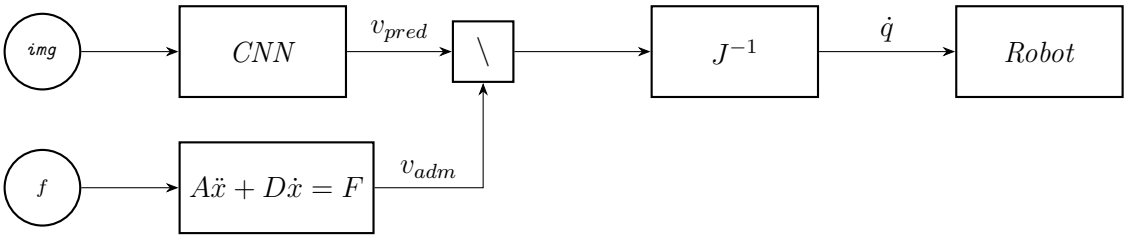

Figure 5.1. Full open loop control scheme

The main control script is then modified, in order to implement this behaviour too. The code realizes an admittance–like behavior in task space as a discrete, first–order mass–damper driven by the external Force, combined with a hard switch that selects

between vision and admittance commands. The main difference with the classical admittance control law, is that the damping matrix K has been set to 0, omitting the related term $K(x - x_{eq})$. Including K would turn the mapping into "push $\rightarrow$ position offset": at steady force the velocity would go to zero and the tool would settle at some displaced x.

That conflicts with the goal (gently keep moving the end-effector while an operator or contact applies a force), and it would also require maintaining a consistent position state x and an equilibrium x_{eq} that wouldn't fight the CNN's vision command. In a velocity architecture like this, a spring introduces bias and "snap-back": when contact is released, stored spring displacement can create unintended motion. By using only M and D, the branch stays strictly dissipative, simple to tune, and provides the desired behavior. For these reasons, the control law is the one shown in Fig. 5.1.
The admittance branch is configured by some parameters, which have been varied to test different scenarios:

- **Matrices M and D.** These two matrices, defined as

$$M = \begin{bmatrix} m_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & m_6 \end{bmatrix} \quad \text{and} \quad D = \begin{bmatrix} d_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & d_6 \end{bmatrix},$$

  have the role of defining the behaviour of the controlled system.
  The *virtual damping* D sets the steady-state sensitivity: under a constant force f_0, the velocity converges to $v_{ss} = \frac{f_0}{D}$. Hence D is the direct "push to speed" knob: larger value of D yields slower drift under the same contact load and increases dissipativity.

  The virtual mass M, instead, shapes the transients and the noise rejection: it acts like inertia term on v. Larger values of M (at fixed D) show how quickly the compliant motion ramps up or decays and attenuates high-frequency wrench spikes.

  Because M and D are diagonal in the implementation, each of the six task-space axes can be tuned independently (for the linear part M in kg and D in $N \cdot s/m$; for the rotational one, M in $kg \cdot m^2$ and D in $N \cdot m \cdot s/rad$), allowing tighter rotational damping and gentler translational yielding while preserving well-behaved interaction.

- **Velocity deadband v_ϵ.** Conceptually, this term adds a tiny region around $v = 0$ where the commanded velocity is forced to exactly zero, making the system quiet when no intentional push is present and preventing micro-motions that would otherwise persist because of noise, quantization, or the integrator memory in the backward-Euler update.

- **Force thresholds f_ϵ and τ_ϵ.** To prevent spurious reactions in free space, the raw force is filtered by *per–axis* thresholds: any linear force component with magnitude below f_ϵ N and any torque component below τ_ϵ $N \cdot m$ is set to zero. This componentwise gate suppresses bias drift, yielding a sparse, reliable force that drives the switch to the admittance branch only when contact is unambiguous.

- **Sampling period** Δt. It parameterizes the implicit discretization of the admittance law, to be better discussed in the following.

For each control cycle, the code reads a wrench $F \in \mathbb{R}^6$ from the Bota sensor, containing three forces and three torques. Since the sensor frame is not aligned with the end-effector frame, a fixed rotation

$$R = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

is applied to both the force and torque components. Let $f = (f_1, f_2, f_3)$ and $\tau = (\tau_1, \tau_2, \tau_3)$ denote the rotated linear and angular parts, respectively.

Very small components are then filtered out componentwise using the *force thresholds* f_ϵ and τ_ϵ:

$$\bar{f}_i = \begin{cases} f_i & \text{if } |f_i| > f_\epsilon, \\ 0 & \text{if } |f_i| \le f_\epsilon, \end{cases} \qquad \bar{\tau}_i = \begin{cases} \tau_i & \text{if } |\tau_i| > \tau_\epsilon, \\ 0 & \text{if } |\tau_i| \le \tau_\epsilon. \end{cases}$$

The filtered wrench actually fed to the admittance law is then

$$\bar{F} \;=\; [\,\bar{f}_1, \bar{f}_2, \bar{f}_3, \bar{\tau}_1, \bar{\tau}_2, \bar{\tau}_3\,]^\top.$$

Backward (implicit) Euler is used to discretize the system in real time. Over one sampling interval Δt, the derivative at t_k is approximated by a backward difference and the damping term is evaluated at the current step:

$$\dot{v}(t_k) \;\approx\; \frac{v_k - v_{k-1}}{\Delta t} \qquad \Longrightarrow \qquad M\,\frac{v_k - v_{k-1}}{\Delta t} + D\,v_k \;=\; \bar{F}_k.$$

Rearranging gives the implicit update solved at each cycle:

$$\left(D + \tfrac{M}{\Delta t}\right) v_k \;=\; \bar{F}_k \;+\; \tfrac{M}{\Delta t}\, v_{k-1}.$$

With $M \ge 0$ and $D > 0$, the left–hand matrix is symmetric positive definite, the solution is unique, and the discretization preserves the intended dissipative behaviour of the admittance element at the practical sampling rates used here.

The raw admittance velocity v_k is then subjected to a small *velocity deadband*:

$$v_{k,i} \leftarrow \begin{cases} v_{k,i} & \text{if } |v_{k,i}| > v_\epsilon, \\ 0 & \text{if } |v_{k,i}| \le v_\epsilon, \end{cases}$$

and finally clipped to a maximum magnitude before being mapped to joint velocities through the DLS inverse kinematics.

A separate *switching threshold* f_S is used to decide when to engage the admittance branch instead of the CNN branch. The decision is based on the norm of the filtered linear force,

$$\|\bar{f}\| = \sqrt{\bar{f}_x^2 + \bar{f}_y^2 + \bar{f}_z^2},$$

leading to

$$v_{\text{total}} = \begin{cases} v_{\text{adm}}, & \|\bar{f}\| > f_S \quad \text{(admittance only)}, \\ v_{\text{CNN}}, & \|\bar{f}\| \leq f_S \quad \text{(vision only)}. \end{cases}$$

Thus, the per–component deadbands $(f_\epsilon, \tau_\epsilon)$ and the switching threshold f_S play different roles: the former reject small noisy readings on each axis, while the latter decides when the interaction is strong enough to give priority to the admittance behaviour.

Although torques do not participate in the switching criterion by default, they could be included via a torque threshold τ_S (engage if $\|\bar{f}\| > f_S$ or $\|\bar{\tau}\| > \tau_S$). Once the admittance branch is active, both linear and angular admittance velocities are generated, deadbanded and clipped as above, and then mapped to joint rates through the DLS inverse kinematics.

Chapter 6

Results

This chapter presents the experimental results obtained with the two proposed control approaches. For each approach, the mechanical and vision setup used during data acquisition and the corresponding workspace ranges are first described. The data-collection phase is then summarized with quantitative descriptors, such as the number of sessions and images. Next, the training phase is detailed for each approach, including the main hyperparameters, the structure of the training/validation splits, and representative images from the experimental setup.

Finally, the performance is reported using task-relevant metrics (per-axis MAE/MSE, overall velocity-vector error, and the percentage of samples within given tolerance bands), together with learning curves and photographs taken from the physical experiments.

6.1 One-Network Approach

This configuration follows the procedure described in Section 4.1. The robot is initialized in a *base pose* [230, 0, 190, 0, 90, 0] (Fig. 4.3), chosen so that the endoscopic camera is approximately normal to the working surface and centered over the region of interest. In this arrangement, the field of view covers a sufficiently large portion of the workspace to permit reliable data collection while still providing adequate image resolution for training and evaluation.

The *target pose* is selected empirically by positioning the die within the camera frame and refining its placement until a repeatable and well-centered image is obtained (TCP reaches $z_a = 17$mm). For this reason, the face showing the number five is used: its symmetric arrangement of dots provides a visually distinctive and easily repeatable reference frame for the final approach. Figure 6.1 shows two representative images acquired at the base pose (left) and at the desired target pose (right), which act as the endpoints of the single-network data-collection trajectories.

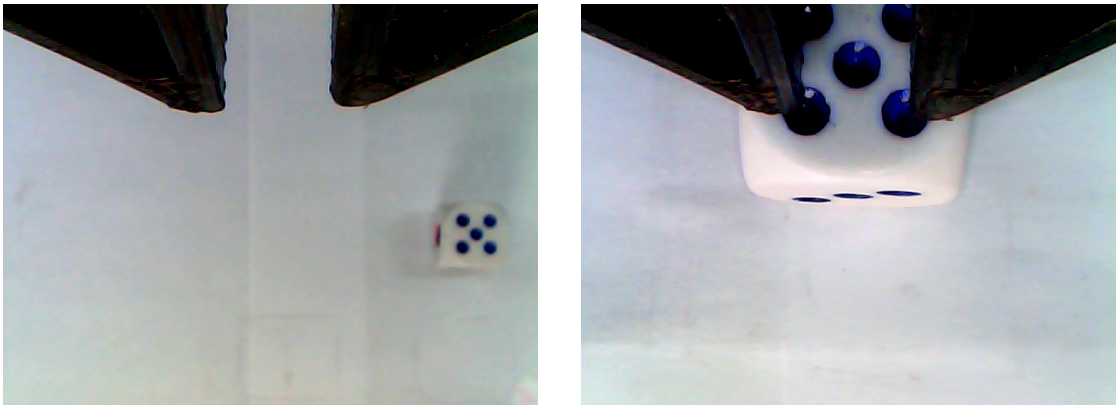

Figure 6.1. Base pose (left) and desired target pose (right) for the first network.

Following the procedure of Section 4.1.2, a fictitious grid is superimposed on the workspace. The initial region considered is a 40×130 mm rectangle in the x–y plane. Using a fine sampling resolution of $\Delta x = \Delta y = 1$ mm would yield a grid with

$$C = 130 \times 40 = 5200$$

cells. While this density would provide excellent spatial coverage and potentially high precision, collecting and labeling data for all 5200 positions would be impractical in terms of time and manual effort.

For this reason, the methodology transitions to a two-network scheme in later sections: a *coarse* network to cover the full workspace and a *refine* network dedicated to precise control in the vicinity of the target. Notice that the orientation of the end-effector is kept constant for all trajectories and all points of the grid, fixed at [0, 90, 0].
In the context of the single-network baseline, however, a coarser sampling grid is adopted for feasibility. The displacements are set to $\Delta x_1 = \Delta y_1 = 10$ mm, resulting in

$$C_1 = 13 \times 4 = 52$$

cells, which drastically reduces the data-collection effort while still ensuring enough spatial variability for training.

With a logging period of $T_\ell = \frac{1}{30}$ s, `SetCartLinVel` set to 15 mm/s, a total of 10221 frames is acquired; an excerpt of the resulting CSV file is reported in Table 6.1. The last three columns are identically zero because the end-effector orientation is kept constant for all trajectories and the threshold $\eta_2 = 0.1$rad/s removes the noise-dominated components (while $\eta_1 = 0.01$rad/s). As expected, the dominant term is the third component, which is consistently negative, corresponding to motion towards the x–y plane.

Inspection of Table 6.1 also shows that consecutive frames exhibit only small variations in the first three components, while the image indices increase sequentially. This confirms the temporal coherence between images and labels, providing a sanity check on the logging pipeline and the consistency of the collected dataset.

image_filename	$v_{\text{task},1}$	$v_{\text{task},2}$	$v_{\text{task},3}$	$v_{\text{task},4}$	$v_{\text{task},5}$	$v_{\text{task},6}$
frame_000000.png	0.04416	-0.60956	-0.78475	0.00000	0.00000	0.00000
frame_000001.png	0.03190	-0.44471	-0.89413	0.00000	0.00000	0.00000
frame_000002.png	0.01400	-0.39077	-0.91772	0.00000	0.00000	0.00000
frame_000003.png	-0.03325	-0.39086	-0.91951	0.00000	0.00000	0.00000
frame_000004.png	-0.01605	-0.38908	-0.92092	0.00000	0.00000	0.00000
...	...	...	...	...	...	...
frame_010221.png	0.14312	0.26956	-0.95113	0.00000	0.00000	0.00000

Table 6.1. Excerpt from the logged dataset used to train the networks.

The training phase was carried out iteratively, starting from a very compact baseline and progressively increasing the model capacity and the number of training epochs. As a first attempt, a shallow network with 10 units in the first layer, 50 hidden units and an output layer of 6 neurons (one per velocity component) was used. This configuration served as a low–capacity baseline: it was sufficiently simple to train quickly and to verify that the image–to–velocity mapping could be learned at all, but it exhibited clear signs of underfitting, with relatively high training and validation errors.

Subsequent experiments increased both the width and the depth of the network, as well as the maximum number of epochs. In practice, this meant adding more convolutional filters and a larger fully connected layer, so that the model could extract richer visual features and represent more complex input–output relationships. At the same time, the number of allowed epochs was raised to give the optimizer enough iterations to exploit this additional capacity. Overfitting was controlled by monitoring the validation loss and by enabling early stopping, so the training was automatically halted once further epochs no longer improved generalization.

The final architecture adopted for the coarse network consists of a stack of five convolutional blocks with [10, 50, 100, 180, 250] filters, each followed by 2×2 max–pooling, a flattening layer, and a fully connected layer with 500 ReLU units, ending in a 6–dimensional linear output. The dataset is split into 80% training and 20% validation samples, and images are resized to 240×320 pixels and normalized in [0,1]. The model is trained with the Adam optimizer (*learning rate* 10^{-3}) and mean–squared error loss, while global and per–component mean absolute errors are tracked as metrics. A combination of early stopping, learning–rate reduction on plateau, and model checkpointing ensures that the best weights (in terms of validation loss) are retained and that the final model balances fitting accuracy with robustness to overfitting.

During training, lasted about 1,5 hours, the network is optimized using the mean–squared error loss, which is also monitored as a global metric. Figure 6.2, however, reports the evolution of the mean absolute error for each task–space component during the training of the *coarse* network, calculated as:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|\hat{v}^{(i)} - v^{(i)}|. \tag{6.1}$$

MAE provides a more intuitive measure of the average absolute deviation between prediction and ground truth for each velocity component, expressed in the same units as the output, which makes the per–component behaviour of the network easier to interpret.

From the curves in Fig. 6.2, it can be observed that the *early stopping* criterion interrupts the training at epoch 58, well before the initial limit of 100 epochs. This indicates that no further improvement in validation MAE was detected beyond that point, and the model parameters were restored to the best-performing epoch.

For the first three components, which correspond to the translational part of the task velocity, both training (solid) and validation (dashed) curves show a rapid drop in the first epochs and then settle to small values, with only mild oscillations. This indicates that the coarse model quickly captures the main image–to–velocity relation without significant overfitting. The remaining components v_4 and v_6 are almost identically zero in the dataset, and their MAE curves therefore converge very close to zero after a few epochs.

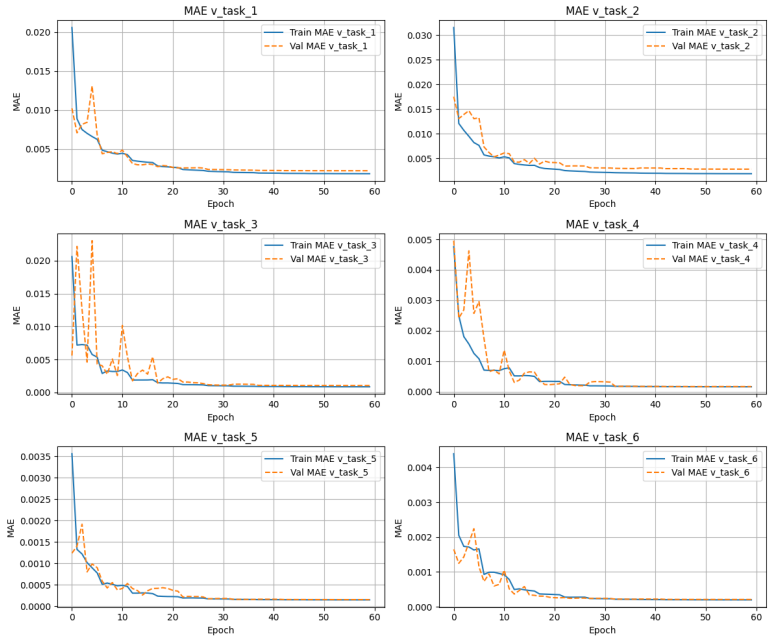

Figure 6.2. Component-wise MAE for the coarse network.

It is worth noting that, despite the very low MAE values achieved by the coarse network, the behaviour of the robot is not sufficiently accurate. The training and validation sets cover only a limited portion of the workspace, so the network learns to interpolate well around the sampled trajectories but does not generalize reliably to unseen positions. As a result, the end–effector reaches only a neighbourhood of the target and still exhibits residual errors, as expected. This motivates the introduction of a second network: the coarse model is used solely to drive the system into a rough vicinity of the goal, while the refine network, trained on a denser dataset around the target region, is then employed to achieve the required positioning accuracy.

6.2 Two-Networks Approach

As previously discussed, the second network is activated when the z-coordinate of the TCP falls below a given threshold. In this work, the threshold is set to 40 mm: once the end-effector is closer than this distance to the workspace plane, the refine network takes over from the coarse one.

Figure 6.3 shows the camera perspective corresponding to the initial pose used for training and to the desired target pose (the same configuration adopted in the one-network approach).

Figure 6.3. Base pose (left) and desired target pose (right) for the second network.

Since the second network is responsible for the final positioning accuracy, the grid parameters are now chosen to reflect this requirement. Tests carried out with the one-network approach showed a maximum positioning error of approximately 10 mm at the target pose. For this reason, the refine dataset is collected over a smaller region centred around the target, defined as a 10×10 mm square in the x–y plane. With $\Delta x_2 = \Delta y_2 = 1$ mm, the resulting grid contains

$$C_2 = 10 \times 10 = 100$$

cells, providing a dense sampling of the area where the coarse network leaves the residual error. As before, the end-effector orientation is kept fixed at $[0, 90, 0]$ for all trajectories.

With the same logging period of $T_\ell = \frac{1}{30}$ s, a total of 10 254 frames are acquired for the refine stage. Although the overall number of samples is comparable to the previous case, the `MoveLin` command is now executed at 5 mm/s, so that more frames are recorded for each grid point along the trajectory. The corresponding `.csv` file has the same structure as the one reported in Table 6.1, containing, for every frame, the image filename and the six task–space velocity components.

Then, the refine CNN is trained using the same data pipeline adopted for the coarse model: images are resized to 240×320 pixels, normalized in [0,1], and the dataset is randomly permuted and split into 80% training and 20% validation samples with fixed random seeds for reproducibility. In order to resolve the small residual errors left by

the coarse network, the architecture is made deeper and wider: six convolutional blocks with [10, 50, 100, 200, 500, 1000] filters (each followed by 2×2 max–pooling) are used, followed by a flattening layer and a fully connected layer with 2000 ReLU units, ending in a 6–dimensional linear output. Training is performed with the Adam optimizer (learning rate 10^{-3}), using the mean–squared error as loss and tracking both the global MAE and the per–component MAE, as discussed previously. Early stopping, learning–rate reduction on plateau, and model checkpointing are again employed to retain the best weights in terms of validation loss.

Figure 6.4 shows the evolution of the mean absolute error for each task–space component during training of the refine network. In this case the model continues to improve throughout the full budget of 100 epochs (total training time approximately 4 hours), so the *early stopping* criterion is not triggered.

For the translational components v_1, v_2 and v_3 the MAE starts around 6–10 $\times$ 10^{-2} and rapidly decreases within the first ten to twenty epochs, then stabilizes below 5×10^{-3}. Training (solid) and validation (dashed) curves almost coincide after the initial transient, indicating good generalization and no evident overfitting. The remaining components v_4, v_5 and v_6 exhibit smaller initial errors and quickly converge to values of the order of 10^{-3} or less, with only mild oscillations attributable to the reduced batch size. Overall, these trends confirm that, on the densely sampled region around the target, the refine network learns an accurate and well–behaved mapping from images to task–space velocities, providing the precision needed to correct the residual errors left by the coarse stage.

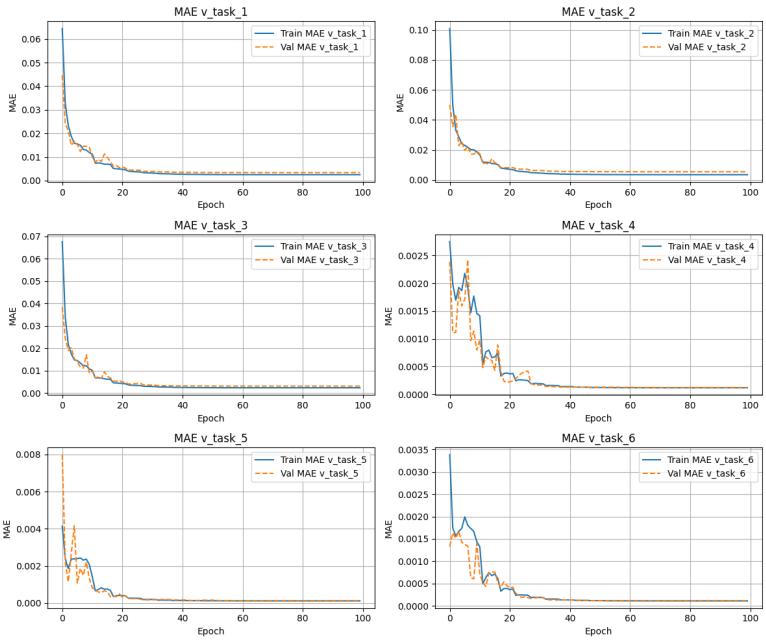

Figure 6.4. Component-wise MAE for the refine network.

6.3 Tests

The overall control pipeline was evaluated through a series of experiments designed to verify its behaviour under different operating conditions and to assess the consistency between expected and observed performance.
As a first baseline scenario, the die is positioned on the workspace and the controller is executed with the robot starting from the same base pose adopted during data collection, namely [230, 0, 195, 0, 90, 0]. This allows assessing how well the system reproduces the behaviour seen during training, both in terms of approach trajectory and final pose accuracy. Subsequently, a range of more challenging configurations is explored.

6.3.1 Moving the die during execution

A first qualitative test evaluates how the controller reacts when the target does not remain fixed in the workspace. Starting from the same base pose used during data collection [230,0,195,0,90,0], the robot begins its image–guided motion towards the die. While the loop is running, the operator repeatedly displaces the die to different positions on the plane, always keeping it within the camera field of view. At each control step, the new image is processed by the CNN, which updates the predicted task–space velocity; through the Jacobian mapping and smoothing, the resulting joint commands continuously bend the trajectory towards the latest die position.

Figure 6.5 shows a sequence of synchronized frames: the top row reports the external view of the robot and operator, while the bottom row shows the corresponding endoscopic camera images. The path of the end–effector is clearly non–straight, reflecting the live changes in the target location, but the controller remains stable and consistently steers the gripper towards the die. The final pose is close to the desired grasp configuration: the face with five dots ends up approximately under the fingertips, although a small residual misalignment is visible (the upper dots are not perfectly aligned with the finger edges).

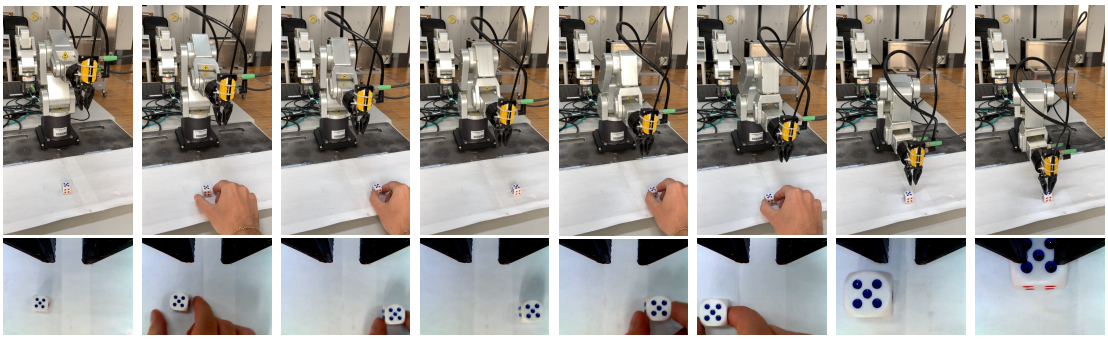

Figure 6.5. Sequence of robot configurations during a motion with live displacement of the die.

6.3.2 Different initial poses of the robot

A second set of tests evaluates how the controller behaves when the die remains fixed at the same target location, while the robot starts from different initial poses. In particular, the base pose is displaced along the Y_b direction, producing three starting configurations with three different y_a:

$$[230, -60, 195, 0, 90, 0], \qquad [230, 0, 195, 0, 90, 0], \qquad [230, 30, 195, 0, 90, 0].$$

Figure 6.6 shows the robot in the three considered configurations together with the corresponding camera views. Although the base position of the TCP has clearly changed, from the vision standpoint these are simply three different images drawn from the same image–velocity mapping learned during training. The CNN has no access to the robot configuration: it only receives the endoscopic image and outputs the associated velocity vector. Consequently, as long as the die remains within the field of view and its appearance is compatible with the training data, changing the base pose does not alter the network's behaviour, since the controller interprets each situation purely on the basis of the observed frame.

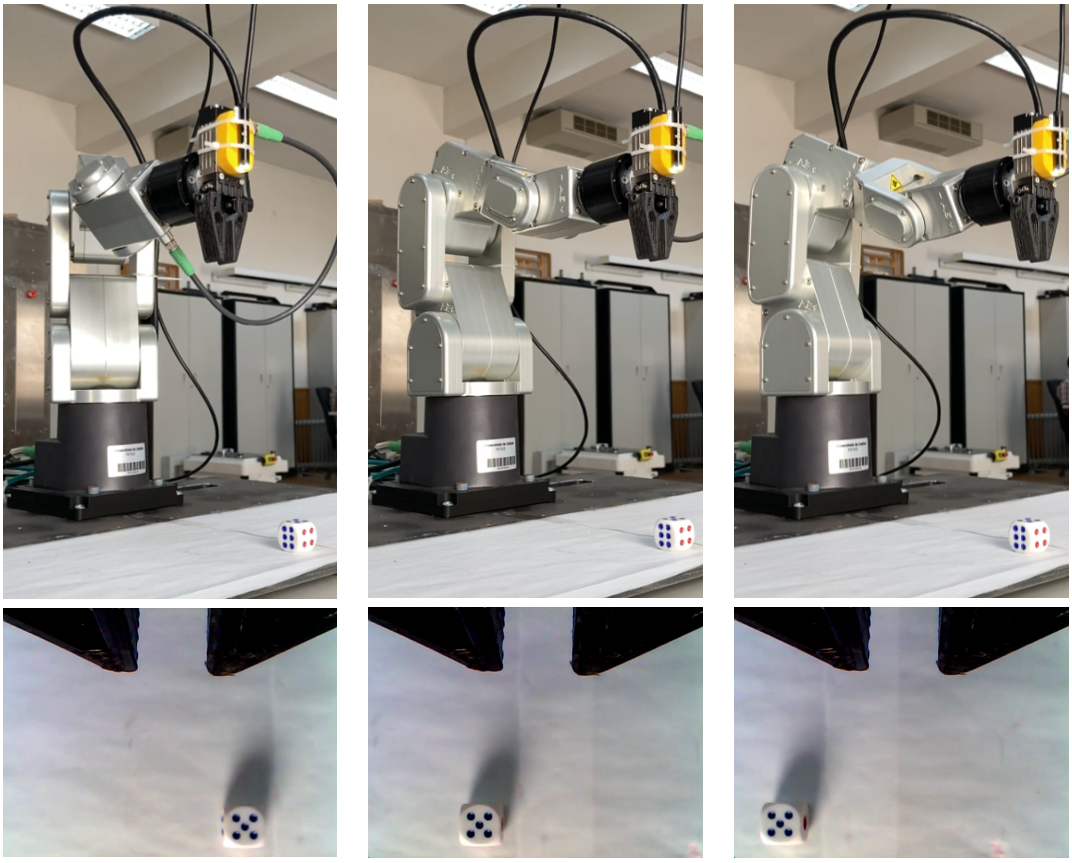

Figure 6.6. Different base poses of the robot and corresponding camera views.

Figure 6.7 shows instead the final poses reached by the robot in the three cases. As expected, the system converges to essentially the same arrival configuration (little displacements can be observed, in the first frame for example), demonstrating that the controller compensates for different starting geometries and uses only the visual information to determine the appropriate motion. This confirms that the network correctly generalizes over changes in the robot's initial configuration and behaves consistently with the principles discussed in Section 4.5.2.

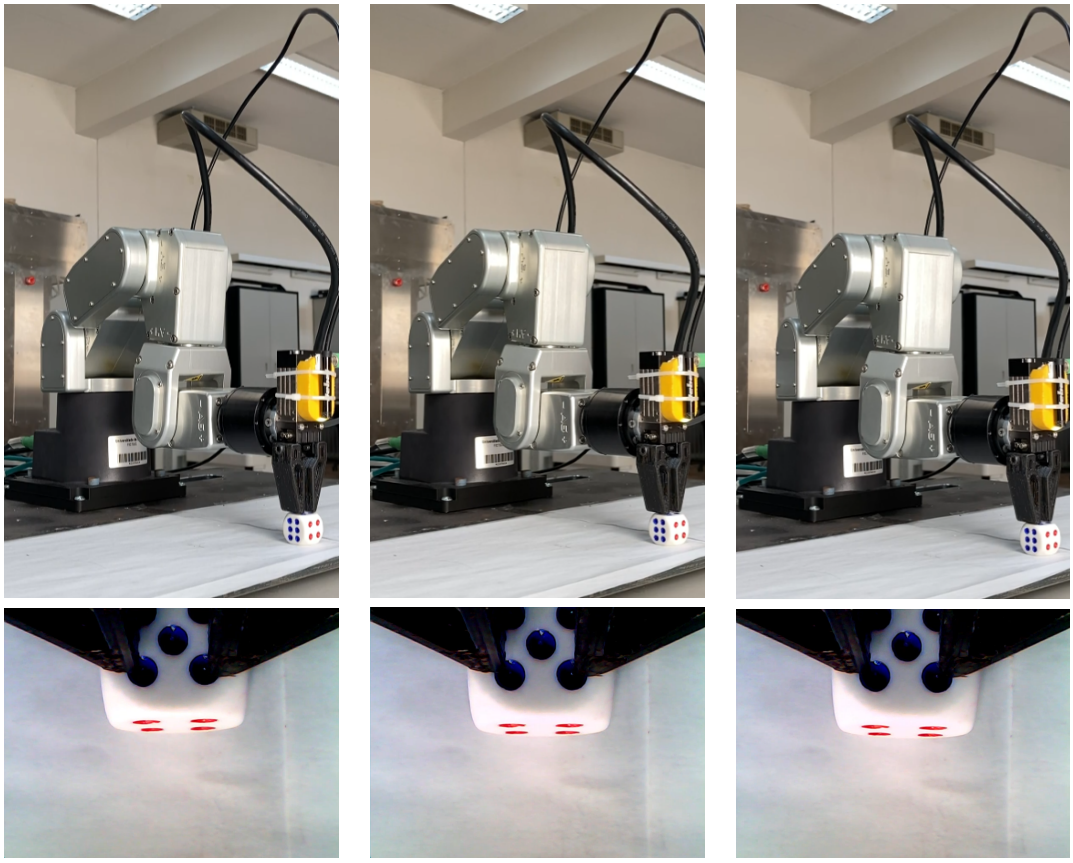

Figure 6.7. Final arrival poses obtained from the three different initial configurations.

6.3.3 Damped Least Squares

A final experiment evaluates the behaviour of the controller when the robot starts from a configuration that that passes close to a kinematic singularity. The base pose is modified by increasing the z–coordinate of the TCP to z_+

$$[230,\ 0,\ 310,\ 0,\ 90,\ 0],$$

so that the wrist is much higher above the workspace than in the training setup. Figure 6.8 shows the corresponding initial configuration together with the camera view. Even though

the CNN has never seen images acquired from this exact height, the target still appears in the upper part of the frame, in a configuration qualitatively similar to those used for training. As a result, the network immediately predicts task–space velocities that drive the TCP towards the X_b-Y_b plane, and the robot starts moving downwards toward the die.

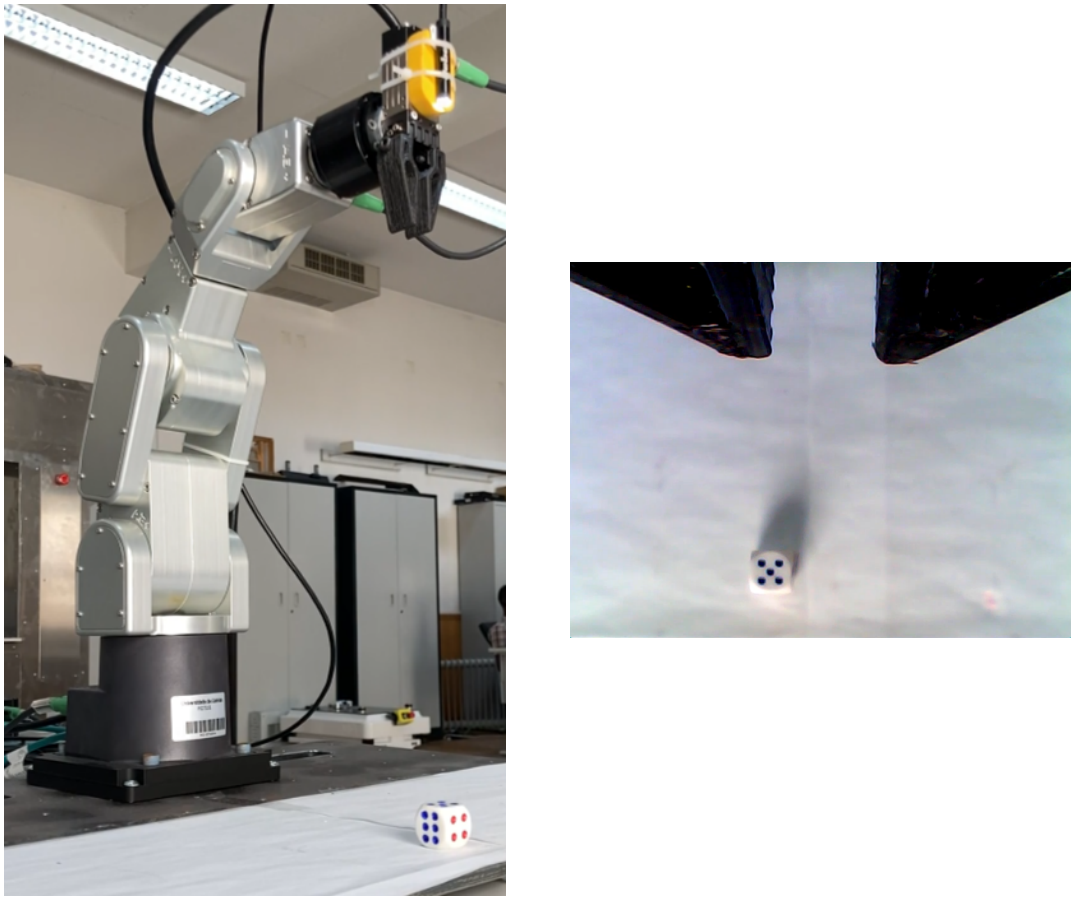

Figure 6.8. Initial pose with the augmented z_+ coordinate.

The critical part of this trajectory occurs when the arm passes near a wrist singularity, i.e. when the fifth joint approaches $q_5 \approx 0°$ and the wrist axes become almost aligned. In this region the Jacobian matrix becomes nearly singular: its determinant (or, equivalently, the manipulability) tends to zero and the condition number grows, so a standard inverse J^{-1} produces very large joint velocities for small commanded twists.

Figure 6.9 compares the outcome obtained without damping (left column) and with the Damped Least Squares inverse ($\lambda = 0.03$). In the *no–DLS* case, as the robot approaches the singular configuration, the fifth joint begins to spin rapidly: the controller "tries" to realize the commanded end–effector motion by generating large $\dot{q}_5$, but the corresponding task–space motion is very small because the Jacobian is ill–conditioned. The arm therefore

exhibits an unintended rotation of the wrist and fails to converge cleanly to the desired grasp pose.

When DLS is enabled, the inverse kinematics uses

$$\dot{q} = J^T \left(JJ^T + \lambda^2 I\right)^{-1} v_{\text{pred}},$$

so that the damping term $\lambda^2 I$ regularizes JJ^T and prevents the smallest singular values from collapsing. Joint velocities remain bounded even near the singularity, and the arm is able to pass through the problematic region and settle above the die. The final pose is slightly less accurate than in the nominal, non–singular tests (the face dots are not perfectly aligned with the gripper fingers), but the task is completed safely and without the large, spurious wrist motion observed with the undamped inverse. This illustrates the expected trade-off of DLS: improved robustness to singularities at the cost of a small loss in asymptotic accuracy.

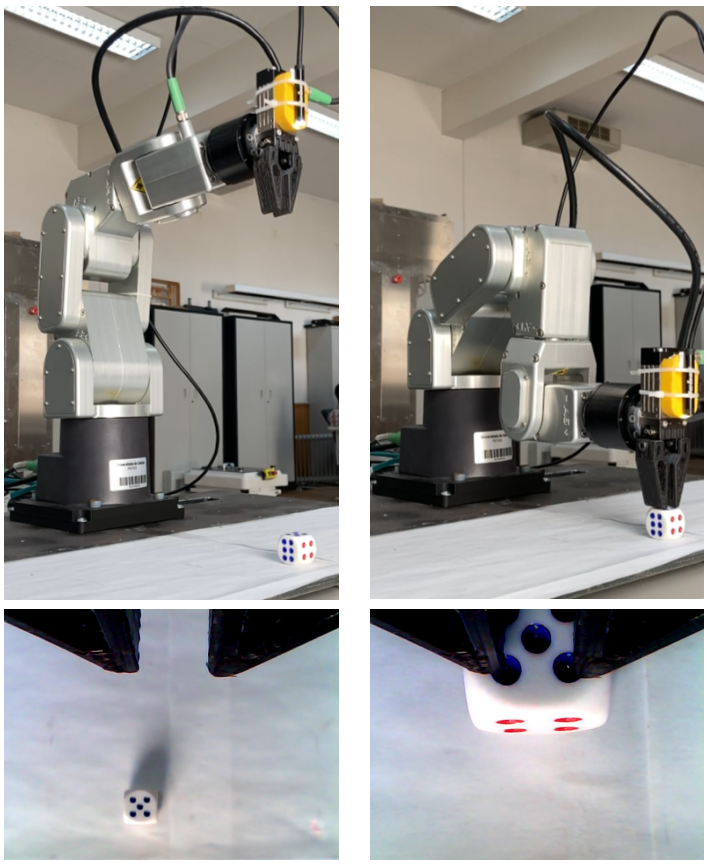

Figure 6.9. Arrival poses for pure inverse (left) and DLS (right).

6.3.4 Admittance-like branch test

The virtual mass and damping used in the admittance-like branch were tuned iteratively in real time, by varying the entries of the matrices and observing the resulting behaviour of the robot. The final choice was a diagonal, isotropic configuration

$$M = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}, \qquad D = \begin{bmatrix} 100 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 100 \end{bmatrix},$$

which provides a relatively light virtual mass and a strongly damped response in all controlled directions. The force and torque thresholds that trigger the admittance mode were set to

$$f_\epsilon = 0.5 \text{ N}, \qquad \tau_\epsilon = 0.5 \text{ Nm},$$

and a small deadband $v_\epsilon = 1 \times 10^{-4}$ was introduced on the commanded admittance velocity to prevent the robot from reacting to sensor noise and very small contact forces.

Figure 6.10 illustrates a typical experiment. Initially, the CNN branch is active and controls the robot exactly as in the previous tests: images from the endoscopic camera are fed to the network, which outputs the desired task–space twist later mapped into joint velocities. When the operator applies a horizontal force on the end-effector such that $\|\bar{f}\| > f_s = 1$ N, the logic switches to the admittance branch. At that moment, image processing is effectively paused and the commanded velocity is generated solely by the admittance law driven by the measured wrench.

The robot yields compliantly in the direction of the applied force, allowing safe physical interaction. As soon as the external force drops below the threshold and the admittance velocity falls inside the deadband, the controller hands control back to the CNN branch, which resumes visual servoing from the new configuration.

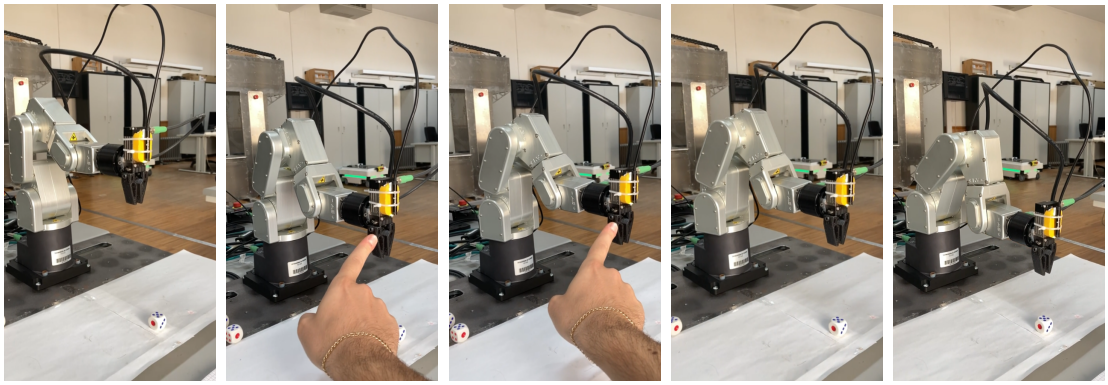

Figure 6.10. Sequence of robot configurations during a motion with an applied force.

The corresponding endoscopic view at the nominal arrival configuration is reported in Figure 6.11. The final pose is qualitatively correct, but less accurate than in the pure CNN tests: the face dots are not perfectly aligned with the gripper fingers. This

is consistent with the design objective of the admittance branch, which prioritizes safety and compliance during physical interaction over sub-millimetric positioning accuracy.

Figure 6.11. Camera view at the arrival configuration with the admittance-like branch.

It is worth noting that the torque threshold τ_S was not included in the current switching logic, which relies only on the norm of the linear force $\|\bar{f}\|$; large wrist torques with small net force therefore do not, by themselves, trigger the admittance mode. Incorporating τ_S in a combined effort measure is left as future work.

Moreover, the admittance parameters were tuned manually; more refined, direction–dependent tuning (for example stiffer normal to the table and softer tangentially) could further improve interaction quality and operator comfort.

Chapter 7

Future work

The controller developed in this thesis demonstrates that a convolutional network can successfully drive a robot toward a visual target using only image–based predictions of task–space velocities. However, several extensions and research directions naturally emerge from the current framework.

A first limitation concerns the treatment of orientation. Throughout all experiments, the end–effector orientation has been kept fixed, and the CNN has never been exposed to rotations of the tool or to different viewing angles of the target. Exploring trajectories in which the orientation is deliberately varied, both during data collection and during loop control, would make it possible to evaluate whether the learned mapping remains consistent under rotational changes, and whether the controller can be extended to predict full six–dimensional twists reliably.

The CNN architecture itself could also be enhanced. The current design is intentionally lightweight to guarantee real–time performance, but deeper or more structured models may improve generalization if paired with a suitably enlarged dataset. Data augmentation represents a simple but effective intermediate step: small rotations, translations, brightness perturbations, or synthetic noise injected before the shuffling phase could increase robustness to variations not seen during collection.

A more fundamental constraint of the present approach is that the dataset was constructed for a single specific target: a die. As a consequence, although the model generalizes well across different spatial placements of *that* object, its predictions are not expected to remain accurate for targets with different shapes, textures, or visual characteristics. Extending the dataset to include multiple objects, or introducing a dedicated classification stage that identifies the target before regression, would allow the system to handle a wider range of tasks.

Finally, the controller does not explicitly reason about the absolute position of either the robot or the target. All decisions are based solely on the appearance of the image, without incorporating geometric priors or state estimates. Integrating position measurements, either through classical visual servoing formulations or through learned state estimators, could significantly improve stability, accuracy, and predictability of the motion, especially in regions where the visual information alone becomes ambiguous.

Overall, the present work provides a complete and functional pipeline, but also opens

the way to numerous extensions that combine learning, geometry, and control to achieve a more robust image–based robotic behaviour.

Below, two objectives for future improvements are examined in greater detail, together with the corresponding solution strategies proposed to address them.

Target presence detection. To make the controller robust when the visual target leaves the camera's field of view or becomes heavily occluded, two simple strategies can be considered.

- A small additional CNN can be trained to decide whether the target is present in each frame. Its output is a confidence score in [0,1]; when this score falls below a chosen threshold, the corresponding velocity command is set to zero instead of being sent to the robot. The classifier is trained with both normal images (target visible) and "negative" examples: blank views of the workspace, different backgrounds and lighting conditions, occlusions and distractors. In this way, the controller moves only when the target is detected with sufficient confidence.
- A lighter alternative is to keep only the regression network and extend the training set with images in which the die is absent, assigning them a zero-velocity label. The model then learns that, in the absence of clear visual evidence, the safest behaviour is to remain still. The main design choice is the proportion of such blank images: too few and the robot may still react to noise; too many and the network becomes overly conservative. In practice, a moderate percentage of blank samples, combined with a simple threshold on the norm of the predicted velocity, can already provide a robust "no target, no motion" behaviour.

The two options are not mutually exclusive and could, in principle, be combined. However, doing so would increase the computational load of the control loop, since it would require running one or more additional networks in parallel with the regression model.

Exploiting kinematic redundancy. Another natural extension is to exploit the kinematic redundancy of the 6-DoF arm in tasks where only the TCP position matters and orientation is largely unconstrained (for example, tracking the centre of a spherical target). In this situation the visual controller specifies only a desired translational motion, leaving several joint configurations that are equivalent from the camera's point of view.

Classical redundancy–resolution schemes can then be layered on top of the learned controller. The CNN output provides the primary task (move the TCP in the right direction), while an additional "null–space" component adjusts the posture without altering the image-based behaviour. This secondary motion can be chosen to keep joints away from their limits, avoid awkward or unsafe configurations, favour high–manipulability poses, or enforce a preferred camera framing. In practice, this would allow the same vision–based controller to produce safer, smoother and more natural trajectories while fully exploiting the extra degrees of freedom of the arm.

Bibliography

[1] Andrew Pierson and Michael Gashler. "Deep learning in robotics: a review of recent research". In: *Advanced Robotics* 31.16 (2017), pp. 821–835.

[2] Heli Deliwala and Dhruv Kadia. "Deep learning in robotic applications: A review". In: *Materials Today: Proceedings* 47 (2021), pp. 163–170.

[3] Seth Hutchinson, Gregory D Hager, and Peter I Corke. "A tutorial on visual servo control". In: *IEEE Transactions on Robotics and Automation* 12.5 (1996), pp. 651–670.

[4] Ashvin Saxena et al. "Exploring convolutional networks for end-to-end visual servoing". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 3817–3823.

[5] Quentin Bateux et al. "Training deep neural networks for visual servoing". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 3307–3314.

[6] Jing Li, Hui Wang, and Ming Zhao. "A Hybrid Visual Servoing Approach Based on CNN for Robot Manipulators". In: *Robotics* 14.5 (2025), p. 66. DOI: 10.3390/robotics14050066. URL: https://www.mdpi.com/2218-6581/14/5/66.

[7] Fumiya Tokuda, Shuji Arai, and Kazuhiro Kosuge. "Convolutional Neural Network-Based Visual Servoing for Eye-to-Hand Manipulator". In: *IEEE Access* 9 (2021), pp. 91820–91835. DOI: 10.1109/ACCESS.2021.3092526. URL: https://ieeexplore.ieee.org/document/9464907.

[8] Jia Guo et al. "Convolutional Neural Network-Based Robot Control for an Eye-in-Hand Camera". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 53.8 (2023), pp. 4764–4775.

[9] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[10] Ravi Raj and Andrzej Kos. "An Extensive Study of Convolutional Neural Networks: Applications in Computer Vision for Improved Robotics Perceptions". In: *Sensors* 25.4 (Feb. 2025), p. 1033. DOI: 10.3390/s25041033. URL: https://doi.org/10.3390/s25041033.

[11] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Advanced Textbooks in Control and Signal Processing. London: Springer-Verlag London Limited, 2009. ISBN: 978-1-84628-641-4. DOI: 10.1007/978-1-84628-642-1.

[12] John J. Craig. *Introduction to Robotics: Mechanics and Control*. 3rd. Pearson Prentice Hall, 2005.

[13] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2006.

[14] Mecademic. *User Manual for the Meca500 Industrial Robot (R3 & R4)*. English. For Firmware Version 9.3.x; Document Revision A. Mecademic. May 22, 2023. URL: https://support.mecademic.com/ (visited on 10/17/2025).

[15] Mecademic. *User Manual for the Electric Parallel Grippers MEGP 25E/25LS*. English. Document ID: MC-UM-MEGP25-EN; For Firmware Version 10.2; Document Revision A. Mecademic. June 13, 2024. URL: https://support.mecademic.com/ (visited on 10/17/2025).

[16] Rebel. *RB-1140 Endoscope Camera — User Manual*. English. Model RB-1140; multilanguage: DE/EN/PL/RO. Lechpol Electronics Leszek Sp.k. Mar. 10, 2023. URL: https://www.lechpol.ro/char/RB-1140.pdf (visited on 10/20/2025).

[17] Bota Systems AG. *Rokubi / Medusa Datasheet*. English. Revision A. Bota Systems AG. 2024. URL: https://www.botasys.com/?utm_source=specificationsheet_2021 (visited on 10/21/2025).

[18] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. 2019.

[19] Mecademic. *Programming Manual for Mecademic Industrial Robots (for Firmware 10.3)*. English. Document ID: MC-PM-EN; Document Revision A; For Firmware Version 10.3. Mecademic. Nov. 7, 2024. URL: https://www.mecademic.com/support (visited on 10/28/2025).

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016. URL: https://www.deeplearningbook.org.

[21] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *arXiv preprint arXiv:1609.04836* (2017).

[22] Lutz Prechelt. "Early Stopping — But When?" In: *Neural Networks: Tricks of the Trade*. Ed. by Geneviève B. Orr and Klaus-Robert Müller. Vol. 1524. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 55–69. DOI: 10.1007/3-540-49430-8_3.

[23] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 2010, pp. 807–814.

[24] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 15. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323.

[25] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations (ICLR)*. arXiv:1412.6980. 2015. URL: https://arxiv.org/abs/1412.6980.

[26] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958.

[27] Géry Casiez, Nicolas Roussel, and Daniel Vogel. “1€ Filter: A Simple Speed-Based Low-Pass Filter for Noisy Input in Interactive Systems”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’12)*. ACM, 2012, pp. 2527–2530. DOI: 10.1145/2207676.2208639.