TÉCNICO LISBOA

Robot hand self-perception via 2D segmentation: a deep learning approach

Alexandre Miguel Fernandes de Almeida

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Alexandre José Malheiro Bernardino
Prof. José Alberto Rosado dos Santos-Vitor

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor: Prof. Alexandre José Malheiro Bernardino
Member of the Committee: Prof.ª Maria Margarida Campos da Silveira

December 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor Prof. Alexandre Bernardino and to Pedro Vicente for their support and availability throughout the last months. Their feedback and input was of the utmost importance to the development of this work.

To my family and friends I would also like to express my thanks for their support that guided me through some of the hardest moments. I give a special thanks to Diogo Oliveira, Catarina Moiçó, Carlos Branco, Miguel Nascimento and João Doroana for their friendly advice and insight.

Abstract

We propose using deep learning to perform segmentation on images containing robotic hands. This serves many applications, *e.g.* helping the robot with tasks requiring accurate knowledge about the positioning of the robot's hands. Detecting its own hands, through vision, is more reliable than using kinematics because, in the latter, each joint of the arm must be properly calibrated, otherwise small offsets will propagate and create large errors when estimating the hand's pose. Maintaining all the joints properly calibrated is impractical and time consuming. To overcome the challenge of collecting and annotating large amounts of images, the datasets are generated in simulation, using domain randomization. We use a model, pre-trained on a large scale dataset, and fine-tune it on our datasets. We make experiments, with different training datasets and strategies, showing results on different types of datasets, with images generated in simulation and real indoor environments, with the real robot. From these experiments we create a final model, trained solely on synthetic images, that achieves an average IoU of 82% on synthetic validation data and $63,5\%$ on real validation data. These results were achieved with only 1000 training images and 3 hours of training time on a single GPU. We do not intend to create a robust model, but rather develop a methodology requiring low amounts of data to achieve reasonable performance, on real data, and give detailed insight on how to properly generate variability in the data and how to fine-tune a complex model to a very different task.

Keywords

Robotic hand, Segmentation, Deep learning, Domain randomization, Indoor environments

Contents

Acronyms

CNN	Convolutional Neural Network
DoF	Degree of Freedom
DR	Domain Randomization
FCN	Fully Convolutional Network
FPN	Feature Pyramid Network
IoU	Intersection over Union
RCNN	Region-based Convolutional Neural Network
RCU	Residual Convolutional Unit
ResNet	Residual Network
RoI	Region of Interest
RPN	Region Proposal Network

1

Introduction

With this work, we propose to use a state-of-the-art Convolutional Neural Network (CNN) to segment robotic hands from the background, in an image. Solving this vision problem serves many applications. By determining the position of each visible point belonging to the hand of the robot, we improve the ability of the robot perceiving its own body, in the environment.

Due to the high level of complexity in this type of networks, *i.e.* the high amount of weights we need to estimate (train), we also require a great amount of data in order to fit the model while maintaining performance on unseen data (*i.e.* to prevent overfitting). However, sometimes, generating this amount of data can be laborious and expensive. To tackle this problem, we build the datasets in simulation, exploiting several hand poses and backgrounds.

We train the network with the simulated data and show that the model is able to retain reasonable performance when validated with real data. We initialize the training process from models, pre-trained on large real datasets, and fine-tune the weights to fit our data and perform the desired task. We explore several transfer learning strategies to achieve a good performance on the target domain, when using pre-trained weights on a source domain.

1.1 Motivation

Self-perception is a necessary step to have truly autonomous robots. The ability of perceiving the surrounding environment has been one of the main problems tackled in computer vision, but perceiving the limits of its own physical boundaries is also crucial for robots, in order to perform precision tasks like object grasping.

Robots need to have a precise estimation of its body boundaries, when performing precision tasks. The hands are the most important to consider, since these parts make most of the interactions with the objects of the environment. Through its direct kinematics and body geometry, the robot could have an estimation of these boundaries and keypoints (*e.g.* the fingertips). However, for some tasks, this is not accurate enough and it is prone to errors. Through vision, we can accurately determine the position of every pixel of the hand, in the 3D space, and help the robot with tasks that need this level of accuracy.

CNNs recently started to solve many problems in computer vision, due to their high efficiency in extracting features from images. Said features can be used to classify objects and make bounding box predictions for each object. One of the most successful works in this area, Mask Region-based Convolutional Neural Network (RCNN) [1], achieved great accuracy at these tasks, while also retrieving a binary mask for each object. However, these algorithms are meant for multi-class prediction and have high accuracy for objects at long distances, making them computationally heavy and difficult to train.

1.2 Objectives and contributions

Our main goal, with this work, is fine-tuning a pre-trained CNN to extract the masks of robotic hands from images, in first person view. In order to achieve this, we must overcome two main challenges: (i) collect and annotate enough data to train the network, and (ii) fine-tuning can fail since both domains and classification tasks differ.

Regarding challenge (i), this type of networks require a large amount of training data, in order to achieve a good performance on unseen images, and to our knowledge, there is no pre-existent annotated datasets of robotic hands available, in first person view. The contributions we bring to solve this challenge can be summarised as follows:

- We developed a framework for Unity [1] (available on github), to generate simulated datasets using Domain Randomization (DR) concepts, that can be used to train deep learning models. We randomize several aspects of the scenes and populate the scenes with distinct parametric objects and a 3D model of the right arm and hand of the robot Vizzy [2].

[1] Our Unity framework https://github.com/alexalm4190/Mask_RCNN-Vizzy_Hand/tree/master/Unity_package

- We train a state-of-the-art CNN, with the datasets generated in Unity, and we show that, by using only simulated images with simple visual properties, we can still retain a reasonable performance when validating with real images that have clutter and different types of lighting.

Regarding challenge (ii), models already pre-trained on large datasets differ both in domain and in the classification task, so it can be a hard task to fine-tune the *weights*, to fit our data and to perform the task of segmenting robotic hands. In order to solve this challenge, we bring the following contributions:

- We evaluate different transfer learning strategies, by breaking down the Mask RCNN [1] architecture into 3 blocks (each containing more general or more task-specific layers) and decide which blocks should start from pre-trained weights and which should be trained from scratch.
- After defining which layers should be transferred, we also make a set of experiments to decide which layers should be frozen during training and which should be fine-tuned to the new task.

1.3 Outline

This document is organized as follows. In Chapter 2, we review (i) some fundamental deep learning and segmentation methods and works that will be used for the development of this work, and (ii) related works that seek to solve the same problem, as this work. In Chapter 3, we define the general methodology that was adopted in the development of this work, explore and give a detailed overview on the methods used to generate the data and the architectures used for the network, as well as giving insight on the strategies, used for training. In Chapter 4, we give some implementation details, define and explain the experimental setup, real validation datasets and evaluation metrics suitable to validate and compare the models. We present our experiments and the results of each, in Chapter 5. Finally, in Chapter 6, we give an overview of the work, draw the most important conclusions and make suggestions for any future work.

2

State-of-the-art

In this chapter, we will cover some state-of-the-art methods that are most relevant to our work. We divide this chapter into two main sections, Section 2.1 explains in detail some important methods and deep learning architectures which will be used in our work for the task of 2D segmentation, and Section 2.2 is where we present works related to our objective, which is giving robots the ability of perceiving their own hand, in the environment.

2.1 Background work

We will start by explaining a standard type of CNN, widely used for 2D segmentation in images (Section 2.1.1). We divide 2D segmentation into two distinct tasks, semantic segmentation (class-aware segmentation, Section 2.1.2) and instance segmentation (class-aware and instance-aware segmentation, Section 2.1.3). We give an overview of some of the most important works, that seek to solve one of these tasks. All these works use the type of CNN explained in Section 2.1.1, to make a pixel-wise classification of the input image, but have several differences in the rest of the architecture. We will expose these differences and explain the advantages and disadvantages in using each of them.

2.1.1 Fully Convolutional Networks

A Fully Convolutional Network (FCN) [3] is a state-of-the-art CNN that, to our knowledge, outperforms current methods in pixel-wise classification tasks (segmentation), without the need for any pre or post-processing (e.g. superpixels or proposals, like in [4, 5]). It differs from a conventional CNN in the sense that it does not have any fully connected layers to aggregate all the information from previous convolutional layers, into a single output. Instead, the fully connected layers are turned into convolutional layers and the image is not downsized to a single output vector, resulting in an output map that has a proportional, but smaller, size than the input image, due to the max pooling operations. The advantage of this feature is that each pixel is classified having only partial view of the object (in a patch around itself) and not the whole image, reducing the number of redundant parameters and the complexity of the model. Finally, to convert the output to the original image dimensions an up-sampling layer, called transpose convolution [6], is used. The advantage of this layer is that it does not use a predefined interpolation filter, instead it learns to up-sample optimally by backpropagation. This architecture can be used as an extension of other deep learning architectures, to extend the classification task to segmentation (pixel-wise classification).

2.1.2 Semantic segmentation

Semantic Segmentation can be described as the task of classifying each individual pixel in an image, rather than assigning a class label to the whole image (image classification) or to smaller regions of the image (object detection).

To obtain a segmentation map (output), segmentation networks have 2 parts (following the architecture of the FCN), the down-sampling path, which is used to capture semantic/contextual information, and the up-sampling path, which recovers spatial information. In simple terms, the down-sampling path interprets the context (what) and the up-sampling path enables precise localization (where).

Some recent works [7, 8] build upon the FCN architecture. In [7], the proposed architecture (called U-NET) follows the encoder-decoder concept of an FCN (the down-sampling and up-sampling paths). The important modification is that, in the up-sampling part, they also have a large number of feature channels. This results in an expansive path that is more or less symmetric to the contracting path (yielding a u-shaped architecture), which allow the network to propagate context information to higher resolution layers. At the end of the expansive path, we end up with the same number of feature channels as in the beginning of the contracting path, *i.e.* 64 channels. For the final layer, a 1x1 convolution is used to map the 64-component feature vector to the desired number of classes (2, for a binary classifier of background and foreground). This 1x1 convolution maps each pixel to the probability of belonging to each of the desired classes, thus giving the desired segmentation of the image. This network was

designed having in mind biomedical applications, (*e.g.* segmentation of moving cells), making it an ideal choice to segment images with low background clutter and clear boundaries between different objects (with no overlapping).

RefineNet [8] was proposed recently to solve the limitation of low resolution prediction in other FCN architectures, similarly to the U-NET. The main difference in this network is that it is divided into 4 blocks, each block receives an input image of a different resolution and maps these images into 4 feature maps of different resolutions, using Residual Convolutional Unit (RCU) blocks [9] pre-trained on the ImageNet dataset [10] and fine-tuned for this task. Each feature map is then up-sampled to the dimensions of the feature map with the highest resolution and are fused through summation. This architecture allows to effectively combine high-level semantics and low-level features to produce high-resolution segmentation maps but this type of networks can only achieve semantic segmentation and not instance segmentation, which is a more complicated task and will be discussed in Section 2.1.3.

2.1.3 Instance segmentation

Instance segmentation differs from semantic segmentation in the sense that labels are also instance-aware, meaning that different classes are assigned different labels and different instances of the same class are also assigned different labels.

In [11], Silberman *et al.* show the differences between semantic segmentation and instance segmentation. In Figure 2.1 these differences are illustrated.

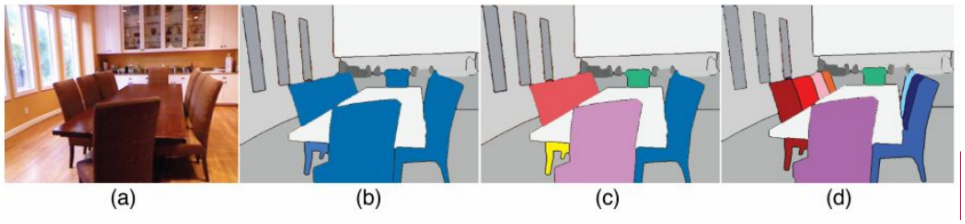

Figure 2.1: Shows the difference between (b) a semantic segmentation of the chairs in the image; (c) a naive instance segmentation, where the hypothesis is that connected components represent different instances of the chair class; and (d) a correct instance segmentation where different instances of the chair class are assigned different labels. Originally from [11].

Given the input image in Figure 2.1 (a), we could predict the labels of each class (chairs or background) and then separate all instances by finding connected components, as shown in figure 2.1 (c). However this is a naive approach. First, because different instances of the same object may be connected. Secondly, an object that is occluded will get assigned more than one instance, because different parts of the object will be separate due to the occlusion. A correct instance segmentation result would be as shown in figure 2.1 (d), where there is a correct reasoning about instances within contiguous segments and across occlusions.

To solve the problems of instance segmentation, Pinheiro *et al.* [12] proposed a network called DeepMask that learns to predict a segmentation mask given an input patch of an image, and assigns a score corresponding to how likely the patch is to contain an object. These two tasks are learnt jointly, with a single cost function that combines them. Compared to models that have a distinct network for each task, this architecture achieves a greater inference speed, at test time.

The methods developed in DeepMask [12] were further improved in [13], where the authors propose to build a top-down architecture which merges the information from low-level features with the high-level knowledge encoded in upper network layers. This refinement is applied to already existent feedforward networks, such as DeepMask [12], improving the accuracy rating of segmentation proposals along with the inference speed per image.

In both these architectures, segmentation precedes recognition, which is slow and less accurate, according to a more recent work [1], where the class labels and segmentation masks are predicted in parallel. This architecture, called Mask RCNN, builds upon a family of recent state-of-the-art networks for object recognition tasks, called RCNNs [14, 15], and extends these networks to the task of instance segmentation by adding an additional branch in parallel that follows the architecture of a FCN. A RCNN (*e.g.* Fast RCNN [14], Faster RCNN [15]) typically has 3 main stages, (i) the feature extraction stage or backbone, which uses notable very deep networks (*e.g.* VGG [16], ResNet [9]) pre-trained on large scale datasets (*e.g.* ImageNet [10], COCO [17]), to map the input images to a high number of feature maps, (ii) the region proposal stage which learns to generate Region of Interest (RoI) proposals (bounding boxes) for regions in the image containing objects (in Fast RCNN it is assumed these proposals are pre-computed, but Faster RCNN proposes a network, called Region Proposal Network (RPN), that learns to make such proposals), and (iii) the classification stage which contains the final classification layers that predict the class labels of the objects contained in each proposal and makes a bounding box regression to align the proposed region with the groundtruth bounding box that contains the object.

Mask RCNN [1] uses the ResNet [9] architecture as the backbone (i), which due to the nature of its residual learning model (*i.e.* some layers have feed-forward connections to deeper layers) mitigates the vanishing gradient problem [1], even when a large number of layers is used (e.g. the ResNet-101 architecture that has 101 layers in total). For the final part of the this stage, a Feature Pyramid Network (FPN) [18] is stacked in order to expand the final ResNet feature maps to 5 different resolutions, creating a multi-scale pyramid of feature maps. Similarly to Faster RCNN, this architecture uses a RPN for stage (ii), but it differs in the way that each RoI is proposed from a feature map of the FPN, depending on its scale, *i.e.* RoIs with smaller sizes are mapped into finer-resolution levels of the pyramid. For the final stage (iii), Mask RCNN extends the architecture used both in Fast and Faster RCNN to add a

[1] As more layers are added to a neural network, the gradient of the loss function approaches zero, when propagating to initial layers. For a more detailed explanation, see https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

segmentation branch, in parallel with the already implemented branches for predicting the class labels and the bounding box offset. This new segmentation branch follows a FCN architecture and adds a new loss to the total cost function, which is the average binary cross-entropy, and a per-pixel *sigmoid* activation function in the last layer.

2.2 Related work

Vision in robots is recently starting to solve a high number of problems, like awareness, *i.e.* the ability to discern different objects and parts of the environment, and self-awareness (or self-perception), which is the ability to position its own physical boundaries in space, *e.g.* determine the position of its own hand to perform specific tasks like object grasping and avoid collisions and interference with other elements of the environment.

To determine the position of the robot's hand, one could estimate its pose, *i.e.* the position and orientation of the reference frame of the hand. A "traditional" way to do this is through the direct kinematics, which consists in determining the position and orientation of certain keypoints in the robot, assuming we are given the angle of each joint that connects to the point of interest. Although fairly simple and efficient, this method depends on the unique geometry of the robot and is very prone to calibration errors. Indeed, small errors in the estimation of initial joints propagate through the whole chain and eventually leads to greater errors in the final joints.

Vicente *et al.* [19] proposes a vision-based method to estimate the pose of the robot's hand and use this estimation to calibrate the kinematic model and thus improving the grasping accuracy of the robot. The position of the other joints can be further calculated from the pose of the hand's reference frame, but this depends on the specific geometry of the robot, and varies even in the same type of robots. Our goal is to have something more general and not specific to a single robot model.

There is a more complete approach to our problem (when comparing to pose estimation), called keypoint estimation. This approach consists in estimating the position of keypoints, in an image, that form the structure of an object. For hand keypoint estimation, the keypoints can be just the fingertips or each joint that grants an extra Degree of Freedom (DoF). In figure 2.2, we show an illustration of this problem.

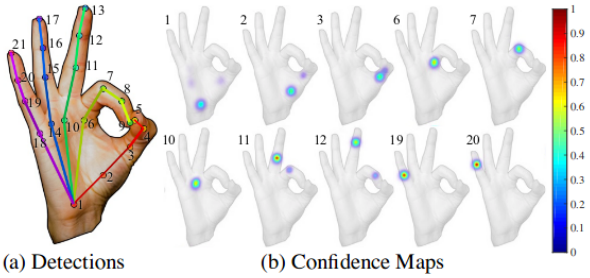

Figure 2.2: Results of estimating confidence maps, for detecting keypoints of a human hand. Originally used in [20].

There are works that estimate the keypoints of human hands [20–22] for applications like human-computer interaction, sign language recognition and augmented/virtual reality technologies. Although a human hand is related to our work, since the model we use for our robot has humanoid hands, there are a few differences in these works, *e.g.* all of them estimate the keypoints in third person view. Estimating the keypoints of a robotic hand, in a first person view, is a more rare task. Usually, research for this problem focuses in improving the ability of perceiving the environment surrounding the robot and not self-perceiving its own physical boundaries.

In a final analysis, the most complete approach to estimate all the boundaries of the hand and achieve self-perception is through segmentation. The problem of segmenting a hand differs from hand keypoint estimation in the way that all visible pixels belonging to the hand are considered keypoints. This can grant a much more detailed information about the spacial layout of the hand, due to the high-resolution of cameras, and does not depend, highly, on the specific geometry of the robot.

Methods for solving the problem of segmenting a hand, through vision, have been mainly focused on human hands, and not robotic hands or arms. Although our robot has humanoid hands, there are some major differences between these methods and our work, namely i) many human hand segmentation methods rely on skin-color information [23, 24], which perform poorly on cluttered domains with different lighting effects, and ii) there is a wide variety of human hand datasets available and already annotated [25] whilst, to our knowledge, there is no pre-existent annotated dataset of robotic hands, in first person view.

The most similar work we've found to ours is [26], where J. Leitner *et al.* propose a machine learning method to detect the hands of two different humanoid robotic models. The authors also propose two approaches for detection: (i) detect the fingertips of the robots' hands, and (ii) detect the full hand. The second task is more desirable, since it provides more information, but it is described as a more complicated task, requiring ten times more training time to reach the best solution, than the first approach. However promising, this work does not present numerical results due to the lack of suitable datasets. In figure 2.3, we show some results of this method. Given that the algorithm was trained with very few images, the results are reasonable. But these results do not prove generality, since all images have a

similar background and do not have much variability.

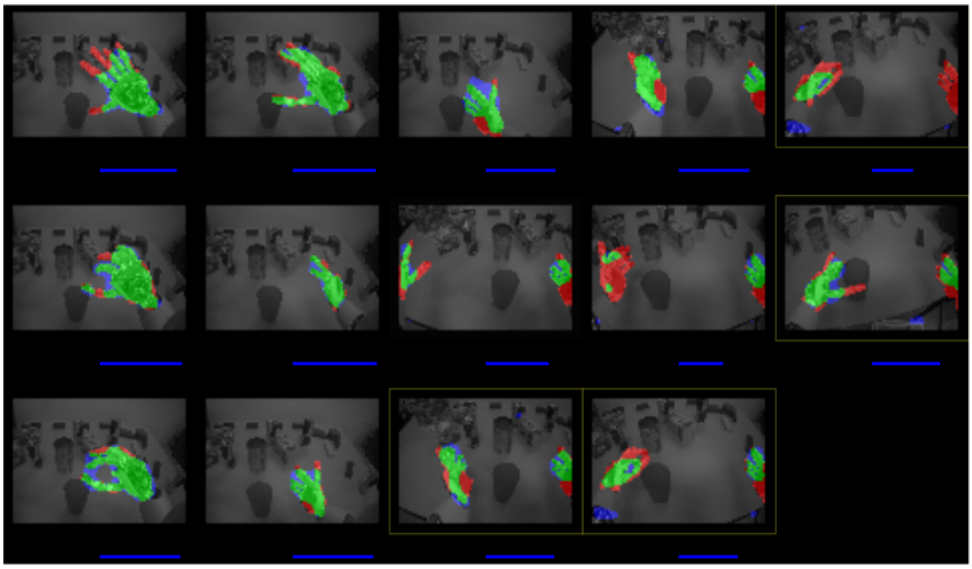

Figure 2.3: Results of segmenting a humanoid robotic hand. The green pixels are the true positives, the red pixels are false negatives and the blue pixels are false positives. Originally used in [26], to show their results in segmenting a robotic hand.

3

Methodology

In this work, we propose to extract the bounding box and a binary mask of the instances of humanoid robotic hands, present in an input image. For the neural network architecture, we use Mask R-CNN [1] with the ResNet-101-FPN [9, 18] backbone, *i.e.* the feature extraction process.

First, to disambiguate some important terms, we define *hyperparameters* as any parameter related to the network's architecture or to the training process, that can be tuned manually without having to make any significant change in the architecture and are not changed during training (*e.g.* learning rate, loss functions, number of train/val epochs). Also, we define *model parameters* or *weights* as the parameters that are adjusted during training through backpropagation, to decrease the training loss (*e.g.* the kernels/filters weights). Finally, a *model* is the name given to an instance of the network trained with a given train and validation dataset and with a selected set of *hyperparameters*.

3.1 Pipeline overview

Our methodology can be divided into three essential processes, as we show, in Figure 3.1, with a simple pipeline.

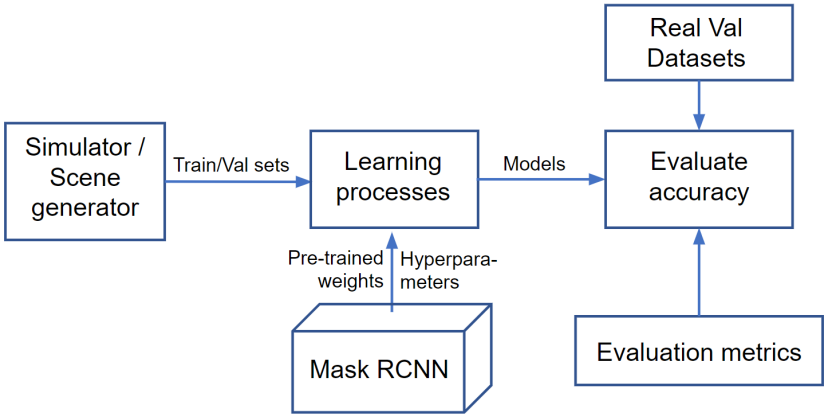

Figure 3.1: General pipeline of this work, using the architecture of Mask RCNN [1].

The data generation (Section 3.2) is where we describe the methods used to generate training and validation images and the corresponding groundtruth data, *i.e.* the bounding boxes and masks, to feed the network during training. The learning process (Section 3.3) can be seen as the strategies that can be adopted to train a model pre-trained on another dataset, to perform a different classification task, with significant changes in the domain. We also give a detailed view of the architecture used for this process and explain some of the choices made in this part. The last process is the evaluation, where we evaluate the accuracy of trained *models*, using appropriate metrics and real validation datasets that will be defined in Chapter 4.

3.2 Data generation

To our knowledge, there is no pre-existent dataset of humanoid robotic hands with annotated groundtruth masks, available for research. Considering the large amount of time required to collect and annotate enough training and validation images, we propose to use a different approach. We build our own custom dataset in simulation, using a 3D model of a humanoid robotic hand and DR concepts [27], to generate a dataset with enough variability to overcome the *reality gap*, *i.e.* to achieve reasonable performance when validating a model, trained only with simulated images, on real images.

3.2.1 Domain Randomization

The idea of this concept (first introduced by Tobin *et al.* [27]), although simple, can be very helpful for closing the gap between the simulated and real worlds. This is done by generating a training set in an environment that simulates the real domain, with such high variability that, when validating with real data, the model retains most of its performance.

The main objective is randomizing the right properties of the training environment. However, defining what these "right properties" are, *i.e.* choosing which properties of the training environment and how they should be randomized so that the *model*, trained with this simulated and randomized environment, achieves maximum performance in real images, is not an easy task. To make this definition more general, the environment that we have full access to (*i.e.* the simulator) will be called the source domain (S) and the environment that we would like to transfer the model to (*i.e.* the images captured from the real world) will be called target domain (T). Training happens in the source domain S, where we can control a set of N randomization parameters with a configuration $\gamma = [\gamma_1, \gamma_2, ..., \gamma_N]$, where $\gamma_i \in [\gamma_i^{lower}, \gamma_i^{upper}] \subset \Re$, with $i = 1, 2, ..., N$. The goal is to define a randomization space $\Gamma \subset \Re^{2\times N}$, in the source domain S, from which to sample a distribution of configurations that generates the groundtruth data, to feed the learning process, that achieves maximum performance, when validating in the target domain T. Thus we have a formulation,

$$\Gamma^* = \underset{\Gamma}{\operatorname{argmax}}\, \mathcal{A}[\ell(\gamma \sim \Gamma; S); T], \tag{3.1}$$

where $\mathcal{A}(X; T)$ is the accuracy when evaluating a model X in the domain T and $\ell(Y; S)$ is the learning process that fits the model to a set of data points Y, extracted from domain S, using a set of environment configurations γ that follows a distribution of configurations Γ. Finding the distribution of configurations (randomization space Γ), in the domain S, that maximizes the accuracy when evaluating in domain T is defining the "right properties" that should be randomized and how they should be randomized.

One of the key aspects of this concept is to achieve enough variability in the simulator such that the real world may appear to the model as just another variation and therefore, common strategies focus on randomizing the position, orientation, texture, shape and number of the objects present in a scene (positive and negative objects); randomizing the background textures, the lighting effects, the position and orientation of the cameras, the type and amount of random noise added to the image, and other texture effects that can be manipulated in the simulator. However the best strategy for applying DR to a simulated environment, highly depends on the learning process and the restrictions of the target domain.

The simulator can be divided in 4 main parts, (i) the foreground, (ii) the background, (iii) the lighting effects, and (iv) the cameras. In Sections 3.2.1.A, 3.2.1.B and 3.2.1.C, we define and explain each of the three first parts and their corresponding sets of randomization parameters and configurations, which can be tuned. Regarding the cameras, we do not apply any randomization, to the components of this part. This is mainly because we already randomize the position and orientation of all the objects, in the scenes, so we would not bring any new effect, to the images, if we randomize the position and orientation of the cameras (which are the main randomization parameters we could tune, in this part).

3.2.1.A Foreground randomization

The foreground is the part that contains the positive instances of the image, *i.e.* the pixels that belong to the 3D model of the robotic hand we want to detect.

In this part of the simulator, possible randomization parameters are (i) the textures used to render the hand, (ii) the color, (iii) the number of hands, (iv) the position and orientation of the hand, relative to the world frame, and (v) the orientation of the hand, relative to the arm. We do not randomize parameters (i), (ii) and (iii). Instead, we fix those to best resemble the hand of the real robot, since we only validate our model on a single real robot. In the case of any future work trying to extend this methodology to detect different robotic hands at the same time, these can be relevant parameters to randomize. The major parameters we need to consider, in this part, are parameters (iv) and (v).

In Figure 3.2, we show the reference frame of the wrist joint. The initial configuration for the orientation of the hand, relative to the arm, is $\gamma_{x,y,z} = [0, 0, 0]$, in angular units, which means the hand starts aligned with the reference frame of the wrist.

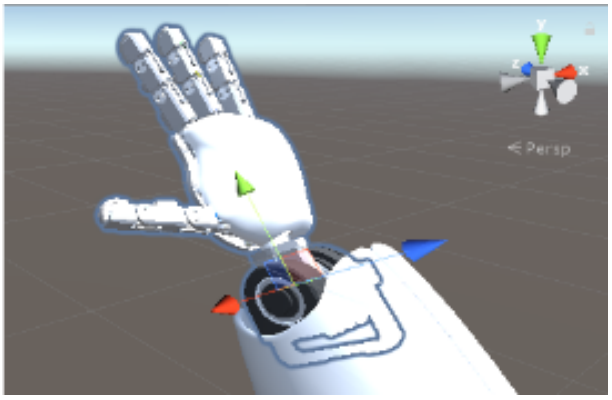

Figure 3.2: Reference frame of the wrist joint. The red, green and blue axes correspond to the x, y, z axes, respectively.

In the context of this work, a real robot has physical limitations that restrict the relative orientation of its own hand, with respect to its own arm. If we consider the orientation of the hand, relative to the orientation of the arm, as 3 separate randomization parameters (one parameter for each axis), the possible configurations, for these parameters, are bounded by the limits of the real robot, *i.e.* $[\gamma_1^{lower}, \gamma_2^{lower}, \gamma_3^{lower}]$ and $[\gamma_1^{upper}, \gamma_2^{upper}, \gamma_3^{upper}]$ are defined as the lower and upper bounds for the angles of the 3 joints that control these DoF, in the real robot. In this case, we can define ranges for the values of each parameter, that avoid configurations which would be unfeasible, in the target domain T and would not bring relevant data to the learning process. In this case, we can define limits for each parameter, from which increasing the variability only decreases the performance of the model, when evaluated in domain T. This is due to prior knowledge, about the target domain, which is not always possible to have.

3.2.1.B Background randomization

The background is the part that contains the negative instances of the image, *i.e.* all pixels that do not belong to the 3D model of the robotic hand we want to detect.

In this part of the simulator, we create a simplified scenario, in order to reduce the complexity of randomization, *i.e.* limit the number of randomization parameters. This scenario contains two main components, a background wall and distractor objects. Regarding the former, possible randomization parameters are (i) texture, and (ii) color. For the distractor objects, possible randomization parameters are (i) texture, (ii) color, (iii) shape, (iv) scale, (v) number of objects, (vi) position, and (vii) orientation, of each object.

For the background wall, we mainly considered 3 types of textures, (i) solid color, (ii) Perlin noise, and (iii) real images. We later show, in Section 5.1.1, that these 3 types of textures for the background wall are very important, in order to deal with the high amount of clutter, in some real validation images. In Figure 3.3 we give a visualization of these textures.

Figure 3.3: Solid color, Perlin noise and real backgrounds, from left to right, respectively.

The solid background is simple to randomize. It consists of 3 parameters, where each parameter represents a color channel, in RGB where $R, G, B \in [0, 255] \subset \Re$. Each time a RGB value is sampled, we assign this value to all the pixels belonging to the background wall.

Perlin noise [28] has already been previously used for DR, in [29, 30]. We show, in this work, the impact such textures can have when validating *models* on real images, with high amounts of clutter. This method of generating textures is much more complex than the solid backgrounds, so we will provide a brief explanation.

The purpose behind Perlin noise is to generate very convincing representations of real textures like clouds, fire, water, stars, marble, wood, rock and crystal. The general method begins by selecting a shape that can be repeated to fill the entire space; for a one-dimensional space, the shapes used are intervals of equal length placed one after the other, while for two-dimensional spaces, *e.g.* our background wall, the most obvious repeatable shapes are squares. Taking the two-dimensional example, for each grid point a *pseudo-random 2D gradient vector* is generated. *Pseudo-random*, in this case, means picking gradients of unit length but different directions. For 2D, 8 or 16 gradients, distributed

around the unit circle, are good choices. The next step is to compute the 4 distance vectors from each point P in the surface to the surrounding points of the square. The dot product between each gradient vector and its corresponding distance vector, gives the final *influence* values. Next, the blending of the noise contribution from the four corners is performed in a manner similar to bilinear interpolation, using a fifth order blending curve to compute the interpolant,

$$f(t) = 6t^5 - 15t^4 + 10t^3, \tag{3.2}$$

where $t \in [0, 1]$. The function described in equation 3.2 has zero first and second derivatives at both $t = 0$ and $t = 1$, making the second derivative of the noise function continuous in all its domain, which is highly desirable for computer graphics tasks.

The method explained is commonly known as classic Perlin noise. We use a variant of this method, known as simplex Perlin noise. The main difference in this variant is that, for a space with N dimensions, it picks the simplest and most compact shape that can be repeated to fill the entire space. For a one-dimensional space, the shape is the same as in classic Perlin noise, *i.e.* equally spaced intervals. But for a two-dimensional space, a square has more corners than necessary so, instead, an equilateral triangle is used to form a 2D grid. Since triangles have 1 less corner than squares, we have one less gradient vector to compute, making the algorithm more efficient. This efficiency increases even more with the number of dimensions; *e.g.* in 3D, instead of using a cube, which has 8 corners, we can use a tetrahedron, which only has 5 vertices.

We later show, in Section 5.1.1, that the third type of backgrounds (the real backgrounds) also give a boost in performance when validating the *models* in high cluttered real images. This is due to the presence of natural real textures in training images, that serve as negative samples in the learning phase. The process of collecting these images, although simple, will be explained below, since there are important details to keep in mind.

We start by searching for images with the supercategory “indoor”, in the COCO dataset [17]. We collect a number of random images from this supercategory and, for each of these real images, we overlap a 3D model of the robotic arm and hand, generated in the simulator. The reason we overlap the model of the hand, in each of these images, may not seem obvious because, since the main purpose of these images is to feed the network with real textures as negative samples, we could just feed these images without the need of having any positive samples in it (pixels belonging to the robotic hand). However, due to the architecture and implementation of Mask RCNN used, we cannot feed purely negative images, during training (but it is possible in inference mode). Details about this architecture will be explained further, in Section 3.3.1, but the intuition behind this restriction is due to the RPN, the stage in the network

where RoIs are proposed and classified as negative (does not contain an object) or positive (contains an object).

Regarding the other component of the background, *i.e.* the distractor objects, the arm to which the hand is attached to is considered a distractor object, but with fixed color, texture, shape and scale. We only randomize the position and orientation of the arm, which consequently also varies the position and orientation of the hand relative to the world frame.

For the remaining distractor objects, we start by generating 4 types of objects, each with a different parametric shape. The chosen parametric shapes are ellipsoids, parallelepipeds, elliptic cylinders and spherocylinders (see Figure 3.4, for a view of their default shapes).

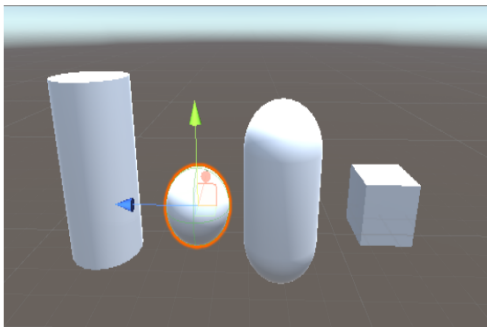

Figure 3.4: Default shape of each type of parametric objects.

For each image, we randomize the color, shape, position and orientation of each parametric object. For the texture, we only use solid color textures, *i.e.* all pixels of each separate object have the same color, at a given image. For the position and orientation, we rotate each object around its own reference frame without any restriction, and vary the frame's XYZ coordinates restricted by a defined parallelepiped volume. Since this volume goes beyond the boundaries of the camera, there is a significant probability of an object not appearing in an image, so, in practice, this also randomizes the number of distractor objects that appear in an image. Regarding the scales, each parametric object is defined by a reference frame and we can randomize the scale, separately, along each Cartesian axis forming different shaped objects with different scales along different axes.

3.2.1.C Lighting effects randomization

The lighting effects is the part that controls the lighting of the scene. For this part, we generate 3 lights and randomize the position and orientation of each light, relative to the world frame. The position parameters, for each light, only take values in a cube around the initial position of their respective light. Regarding the orientation of each light, we randomize the orientation of the x and y axes, but not the z axis because this one points in the direction of the light. In Figure 3.5, we show a visualization of these components, in the simulator.

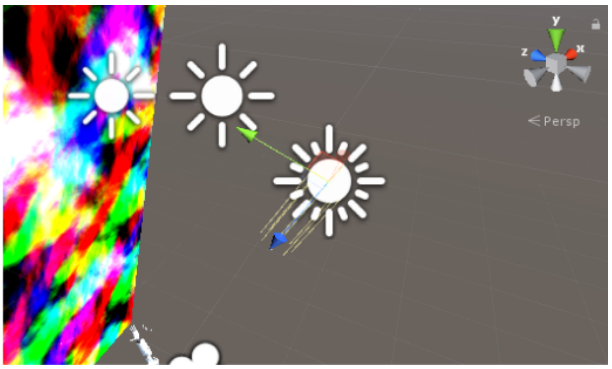

Figure 3.5: Light component randomization, in the simulator.

3.2.2 Groundtruth annotation

The groundtruth data is represented by binary masks, *i.e.* images where pixels can take the value of 1 (positive pixels) or 0 (negative pixels). These binary masks, along with the RGB images, are automatically generated by projecting the robotic hand's 3D surface coordinates onto the 2D image plane. We also use the depth coordinates of the distractor objects and the hand, to check if there are any overlaps. There is no need to include the bounding box coordinates, in the annotations, because these can be calculated from the binary masks, by finding the positive pixels that have the highest and lowest pixel coordinates. Since our approach allows for instance segmentation, we need a separate binary mask for each instance of a robotic hand, present in the image. However, in this work, we only sample images with a single visible hand, from the simulator.

Figure 3.6: Example of an image generated in the simulator (left) and its corresponding groundtruth mask (right).

In order to generate the groundtruth masks from the images taken with the real robot, we have developed an annotation tool (available on our public github repository [1]) that opens the image and lets the user manually create a polygon shaped perimeter around objects. The areas inside the polygons are filled with the *RGB* black color $[0, 0, 0]$ and the rest of the image is filled with *RGB* values that represent

[1] Annotation tool github https://github.com/alexalm4190/Mask_RCNN-Vizzy_Hand/blob/master/utils/annotate_masks.py

white color, $[255, 255, 255]$. In Figure 3.7, we provide the result of annotating a real image, with this tool.

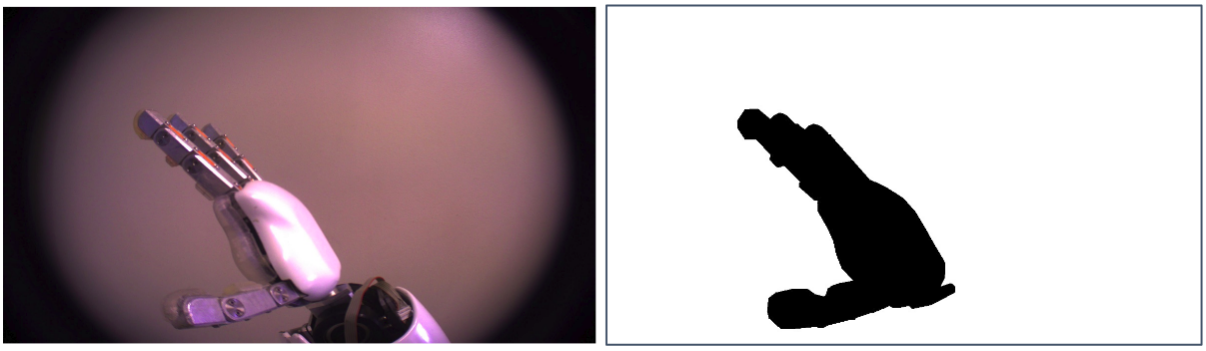

Figure 3.7: Image belonging to one of the real validation datasets (left) and its corresponding annotated groundtruth mask (right).

3.3 Learning

With a data generation process defined, we now need to define how to fit a model to the data and learn to make correct predictions on real images. For this purpose, in this Section we have to define and explain three main components, namely (i) the chosen CNN architecture (Section 3.3.1), (ii) the structure of the training and validation datasets (Section 3.3.2), and (iii) the transfer learning process (Section 3.3.3), *i.e.* since we are training a deep neural network with low amounts of data, training from scratch is not a good strategy, instead we use models, pre-trained on large amounts of data, to initialize the training process.

3.3.1 Network architecture

For the architecture of the network, we decided to use the work of Mask RCNN [1] (already introduced in Chapter 2). The main reasons that led us to this choice were, (i) Mask RCNN is a recent CNN architecture that outperformed the 2016 COCO challenge winners, in both instance segmentation and bounding box detection, and (ii) there are several *models*, available online, that follow this architecture and they were pre-trained on large scale datasets, to detect a large number of classes.

Since this is the core structure of our work, we will give a detailed explanation about the architecture of this network. In Figure 3.8, we provide a simplified diagram of this architecture, for a better understanding on the main components and outputs.

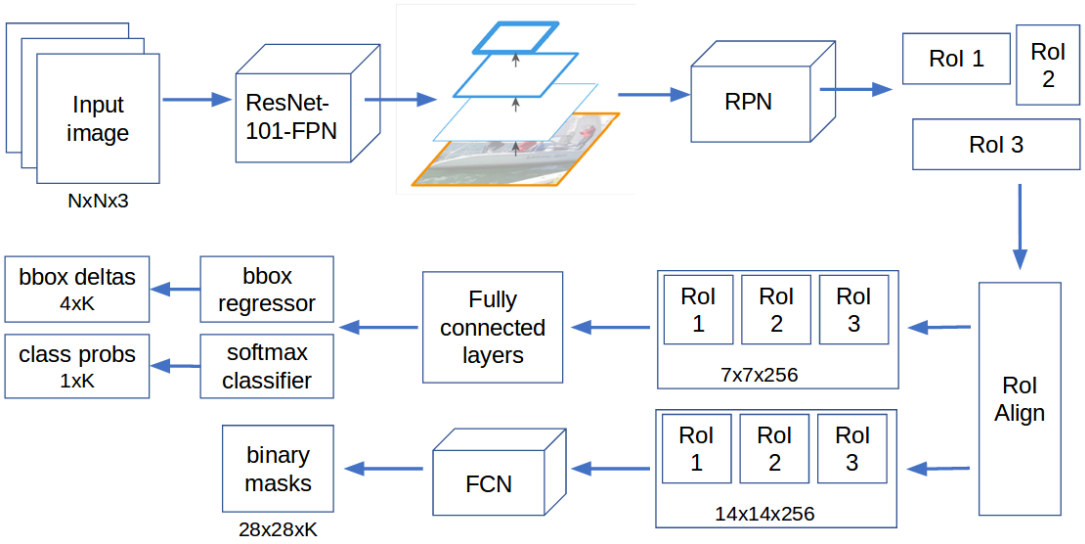

Figure 3.8: Architecture of Mask RCNN [1], using a ResNet-101-FPN [9, 18] as the backbone.

The input image has 3 channels, RGB, and a square shape of dimension $N \times N$. Both the number of channels and the dimension are *hyperparameters* and as such, can be changed. The input image then passes through a Residual Network (ResNet)-101-FPN, *i.e.* a 101 layer deep ResNet [9] with a FPN [18] at the end to transform the feature maps into 4 levels of feature maps (each level with a unique resolution). This stage is called the backbone and its task is learning to extract the most relevant features from the input image and aggregate this information into feature maps that will be used for classification.

The second important stage, the RPN, has the task of learning to propose RoIs from different levels of the FPN. For each pixel of the feature maps, a total of 15 anchors, of different scales and ratios (as shown in Figure 3.9), are centered in that pixel.

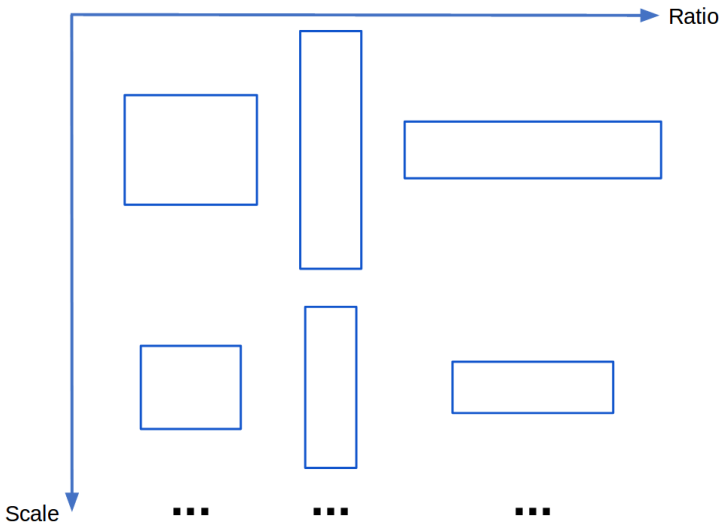

Figure 3.9: RPN anchors, combinations of different scales and ratios.

Each anchor is classified as background or foreground and a bounding box regression is made to adjust the positive anchors to their respective groundtruth bounding boxes. The class label of each anchor is generated by a softmax layer that predicts the probability of each anchor being background or foreground, and the anchor is refined, *i.e.* 4 other values are predicted, for each anchor ($4 \times A$, where A is the total number of anchors placed in the feature maps). These 4 values contain the displacement along the x and y axes between the centers of the anchor and the groundtruth box, and the differences in scale of the height and width of both boxes.

From all the anchors, the network chooses N RoIs (by applying non-maximum supression [2]), where N is a *hyperparameter*, to be passed into the next stage to backpropagate the final class, bounding box and mask errors. Out of these N RoIs, there is a fixed ratio of negatives (*i.e.* RoIs with objectness score lesser than 0.3) and positives (*i.e.* RoIs with objectenss score greater than 0.7), also set as a *hyperparameter*. This means that all images implicitly have negative samples, so there is no need for purely negative images, during training.

This stage has 2 loss functions, (i) rpn_class_loss accounts for the error in predicting the class labels of each anchor, and (ii) rpn_bbox_loss accounts for the error in predicting the deltas that best adjust the anchor to its corresponding groundtruth bounding box. For the rpn_class_loss, the cross-entropy loss is used (equation 3.3, K represents the total number of classes, in our case $K = 2$), and for the rpn_bbox_loss the Huber loss is used (equation 3.4).

$$CE(y, \hat{y}) = -\sum_{i=1}^{K} y_i log(\hat{y_i}) \tag{3.3}$$

$$Huber(y, \hat{y}) = \begin{cases} 0.5||y - \hat{y}||^2, & \text{if } ||y - \hat{y}|| < 1 \\ ||y - \hat{y}|| - 0.5, & \text{if } otherwise \end{cases} \tag{3.4}$$

The last stage picks all the proposed RoIs, after applying the non-maximum suppression, and each RoI is down-sampled to a fixed dimension (7×7 for the classification and bounding box regression branches and 14×14 for the mask branch). This stage predicts 3 outputs, (i) the final bounding box deltas, with dimension $4 \times K$, (ii) the final class labels $1 \times K$ vector, and (iii) a $28 \times 28 \times K$ binary mask, where K is the total number of classes (in our case, $K = 2$). Each output has a loss function associated; output (i) uses the Huber loss (equation 3.4), output (ii) uses the cross-entropy function (equation 3.3), and output (iii) also uses the cross-entropy to compute a per-pixel loss.

[2] This algorithm filters RoIs that contain the same object, to guarantee there is only one proposal for each object instance. For a more detailed explanation, see https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

3.3.2 Training and validation datasets

In order to obtain a good model, we need 2 types of datasets, (i) the training set is the data used to fine-tune the *weights* of the network, and (ii) the validation set is where we check the performance of the *model* on unseen data and it is used to tune *hyperparameters*, prevent overfitting and evaluate our final *models*, to test our hypotheses. In this section, we describe the composition of the training and validation datasets, generated in simulation. We must also generate data to evaluate how the *models* performs on the real robot, with real images, but the datasets used for this will be explained later, in Section 4.4

The simulated data used for training and validation is all generated in the simulator, as was already explained in Section 3.2. To train a *model*, we sample a total of 1000 images from the simulator, approximately, in which 80% are used for training and 20% are for validation. In the literature, it is common to choose a split between the values of $60/40\%$ and $80/20\%$. There is no obvious choice (it highly depends on the structure of the dataset, the data distribution and the number of samples), but it is important to have the largest possible training set while also keeping a statistically meaningful validation set. Since we are short on training data, we chose the $80/20\%$ split.

Another aspect, in this split, is that the validation set follows the same data distribution than the training set, *e.g.* if the training set has images with real backgrounds and perlin noise, the validation set also contain images with these properties. In other words, both datasets are sampled, randomly, from the same data generation process. This randomization is made with a given seed, because we want this process to be reproducible, *i.e.* if we train the network more than once, with the same dataset, the split must always be identical, in order to produce similar results.

Finally, in order to feed the data to the network, all images need to be resized to a fixed resolution (the resolution defined in the architecture of the network). This is done with bilinear interpolation.

3.3.3 Transfer learning

During training, the goal is to minimize the error of the predictions in the train set, while also keeping a high generalization capacity, *i.e.* the capacity of retaining good results when validated with data that was not used to adjust the *model parameters* (*e.g.* the validation set). More importantly, the generalization capacity must also be extended when the domain changes substantially, *e.g.* when we validated with real images. According to J. Yosinski *et al.* [31], that can be achieved by using the knowledge of previously trained *models* in large scale datasets and by adopting a good training strategy, *i.e.* choosing the correct layers to freeze and those to fine-tune, during the learning process.

In theory, initial layers extract features that represent more general aspects of the dataset and, as such, the *weights* tend to not change much in these layers, as in deeper layers, even when the dataset

and the classification task are both changed. As such, initializing the *weights* to already trained (and accurate) values will almost always give good convergence. However, the more different the tasks and datasets get, the less we gain in transferring the *weights* from a pre-trained *model*. There is a point in which we start to achieve better performance if we train layers from scratch or if we transfer the *weights* but fine-tune all layers.

In the case when the datasets and the classification tasks are very similar, *e.g.* when transferring *weights* trained to classify cats to another model that will be trained to classify another feline-like race in a similar environment, the most efficient way to achieve an optimal solution is to transfer the *weights* from all layers, fine-tune the classification layers and freeze the remaining layers, to keep extracting the same type of features. The more differences the datasets and classification tasks present, the more layers we will have to fine-tune, but we can still benefit from transferring the *weights*. However, if the differences are too strong we will start to benefit more if we train some layers (or even all layers, in the most extreme case) from scratch, *i.e.* initializing the layers with random *weights*.

In the case of our work, our task is to train a *model* on the source domain S (the domain of the simulator), using the architecture explained in Section 3.3.1, and retain the highest performance possible when validating the *models* on the target domain T (the domain of the real robot, in an indoor environment). However, we have access to another *model*, pre-trained on a large scale real domain R. Our hypothesis is that, even if domain S and domain R do not share many relevant features and the classification tasks are completely different (in domain R, the model is trained to classify many different classes, while in domain S we train the model to classify a robotic hand, with textures generated in simulation), we can still benefit from initializing from a pre-trained model, because domain R and domain T might share some, more general, features (both have real textures and indoor environments).

Our hypothesis raises a few questions to which we seek to give the answers, namely (i) which layers should be initialized with *weights*, pre-trained on domain R, and which layers should be trained from scratch, (ii) from the layers initialized with pre-trained *weights*, which should be fine-tuned to the new task and which should be frozen, and (iii) which dataset should be chosen to define the domain R. Yosinski *et al.* [31] already gave answers to some of these questions, but they use a CNN with only 8 layers to conduct their experiments, while we are using a much more complex architecture, with more than 100 layers deep and where the outputs of 5 different layers are contributing to the total loss.

In order to simplify the problem, we break down the architecture into three stages, as shown in Figure 3.10, and perform experiments by choosing which stages should start from pre-trained *weights*, which stages should be trained from scratch and which should be fine-tuned to the new task or frozen. For the last classification layers, *i.e.* the softmax classifier and the last convolutional layers for the binary masks and the bounding box regressions, since we changed the architecture of those layers for binary classification, we cannot initialize them with *weights* pre-trained for a multi-class architecture, so these

layers are always trained from scratch.

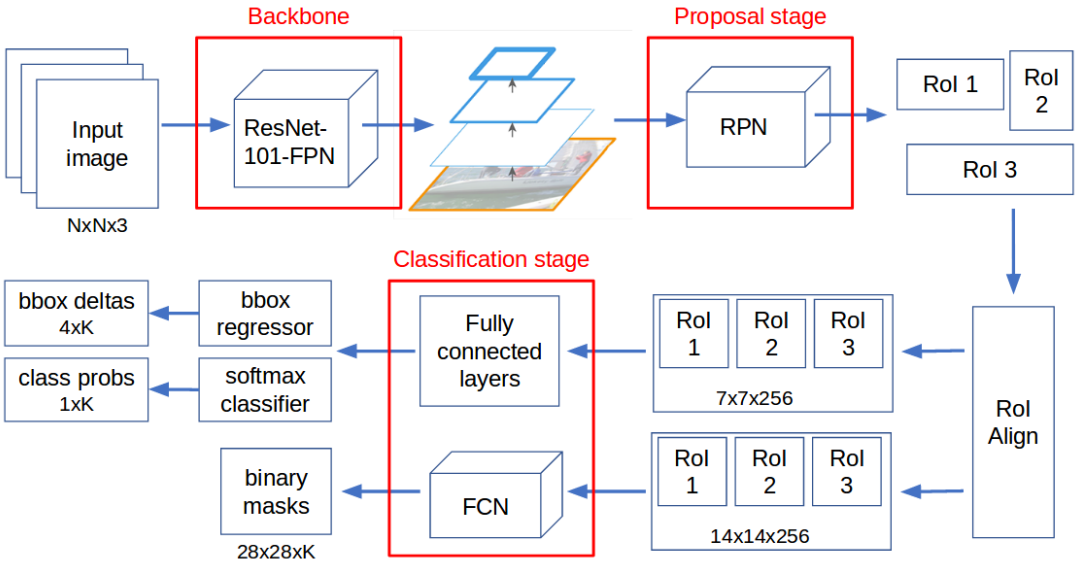

Figure 3.10: Mask RCNN architecture divided by blocks, for transfer learning experiments.

4

Experimental setup

In this chapter, we will explain our experimental setup, *i.e.* the platforms used to implement and validate our processes, as well as the real validation datasets and evaluation metrics used to produce the numerical results.

We develop a framework, using Unity Engine [1], to create a simulation environment from where we sample our training and validation images. The real data is generated by taking images from a real humanoid robot platform, in real environments.

4.1 Real robot

In order to generate real data and validate our approach on real images, we use the humanoid robot *Vizzy*, developed in [2] for the purpose of assistive robotics and social interaction. The robotic platform contains dexterous arms and hands (each hand with a total of 4 fingers) and 3 cameras, one in the chest and the other two in the eyes (see Figure 4.1 for a better understanding on the robot's structure).

[1] Unity webpage https://unity.com/

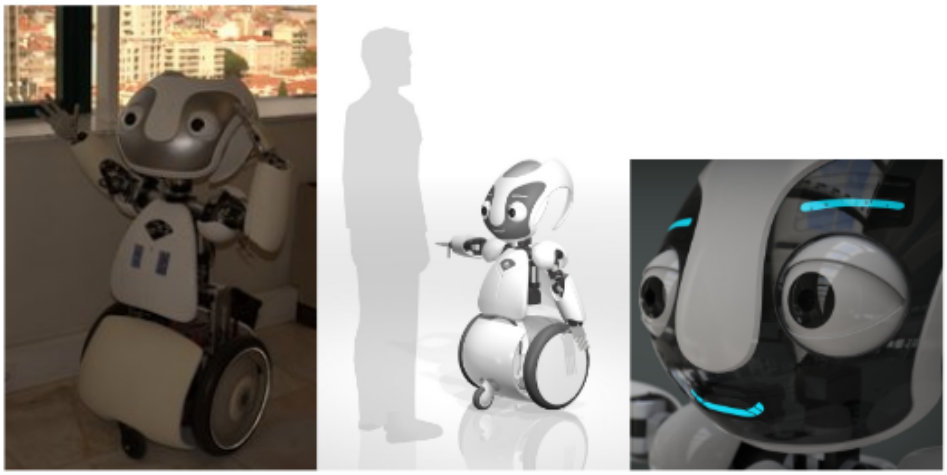

Figure 4.1: Vizzy, a humanoid robot used for assistive robotics and social interaction. Originally used in [2].

Both eyes and the head's movement can be controlled. We use the cameras of the eyes to collect images of the own robot's hand(s) and the environment surrounding them. Furthermore, we use a package [2] to interact with the real robot, that uses both ROS [3] and YARP [4], as middlewares.

4.2 Unity engine

We chose Unity, a game development platform, as our engine to implement our DR based approach, to sample training and validation data. The reason we chose this platform, instead of other simulation platforms, *e.g.* Gazebo [5], is due to both the time it takes to generate scenes and the quality of the textures. In [29], Borrego *et al.* generate one simulated scene, in Gazebo, with parametric objects and background textures (*e.g.* with perlin noise and checkerboard patterns), in roughly a second. With Unity, we are able to generate, approximately, five simulated scenes, with roughly the same complexity, in terms of textures and with a laptop with similar CPU and GPU. Also, being a platform designed for game development, Unity also exceeds other simulators (most, like Gazebo, are designed for robotic simulations, where the most optimized aspect is the physics' simulation) in the quality and complexity of the textures it generates.

We integrate a 3D model of *Vizzy* in Unity, and explore the processes of generating the scenes, sampling images with different parametric objects and complex textures (*e.g.* Perlin noise) and generating the groundtruth masks, needed to train the models.

[2] Vizzy github repository https://github.com/vislab-tecnico-lisboa/vizzy
[3] ROS webpage http://www.ros.org/
[4] YARP webpage http://www.yarp.it/
[5] Gazebo webpage http://gazebosim.org/

4.2.1 Vizzy 3D model

In order to generate images with positive samples (*i.e.* pixels belonging to *Vizzy's* hand), we use a 3D model of the robot *Vizzy* hand and wrist (we show the 3D model design and its corresponding model, in Unity, in Figure 4.2). Each individual component can be imported, without undoing their initial structure, but the links between each component (*e.g.* if the wrist rotates, the whole hand must also rotate since it is attached to it) had to be remade, in Unity, following an hierarchical logic, *i.e.* a rotation or translation of one component affects all the components that are bellow, in the hierarchy.

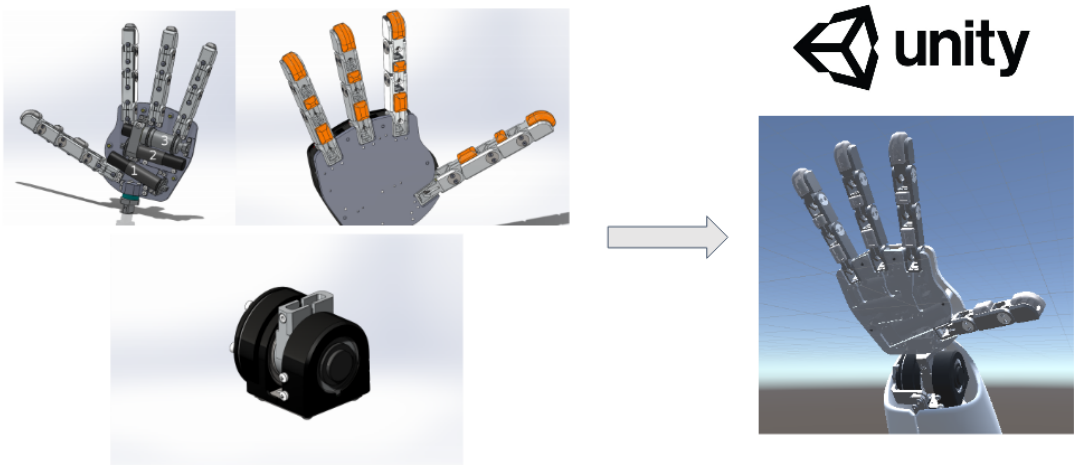

Figure 4.2: *Vizzy* hand 3D model imported to Unity, for data generation.

4.3 Mask RCNN implementation

For the learning process, we use an implementation of Mask RCNN [6] that follows the architecture of the original work [1]. This implementation is open-source and it is the most widely used for Mask RCNN, from what we found so far. Also, we published the implementation of our methods and experiments on github [7]

The implementation used for Mask RCNN allows several *hyperparameters* to be tuned. In Table 4.1, we give a description and the domain for each *hyperparameter*. Note that this table does not contain all the tunable *hyperparameters* of the network, but it contains all those that were changed for our implementation.

[6] mask rcnn github https://github.com/matterport/Mask_RCNN

[7] Python and Unity implementations of our methods and experiments https://github.com/alexalm4190/Mask_RCNN-Vizzy_Hand

Table 4.1: Description of all the *hyperparameters* that were changed, in our implementation of Mask RCNN

Hyperparameter	**Domain**	**Description**
Learning rate	$\in \mathbb{R}_{>0}$	Learning rate value of the optimizer (stochastic gradient descent).
GPU count	$\in \mathbb{N}$	Number of GPUs used to process each batch of images.
Images per GPU	$\in \mathbb{N}$	Batch of images that will be processed by each GPU.
Number of epochs	$\in \mathbb{N}$	Number of training epochs. Defined with a stop criteria, by monitoring the validation loss.
Steps per epoch	$\in \mathbb{N}$	Number of training steps, per epoch.
Validation steps	$\in \mathbb{N}$	Number of validation steps, per epoch.
Backbone	$\in \{50, 101\}$	Number of layers for the ResNet backbone.
Number of classes	$\in \mathbb{N}$	Number of output channels of the network.
RPN anchor scales	$\in \mathbb{N}^5$	Length of square anchor sides, in pixels, for each scale.
Train RoIs per image	$\in \mathbb{N}$	Number of RoIs selected for training, after the proposal stage.
Input image resolution	$\in \mathbb{N}^2$	Width and height dimensions to which the input images are resized.
RoI positive ratio	$\in [0, 1]$	Ratio of positive RoIs used to train classifier/mask heads.

For the first tries and experiments, we set the learning rate to 0.0001, but this *hyperparameter* is further tuned. We explain how it is tuned and show the results on the validation sets, in Section 5.1.4.

We use two different machines, in which we run the learning processes, one with two GeForce GTX 1070 GPUs (each with $8GB$ of memory space) and the other with a GeForce GTX 1080 Ti, as the GPU (with $12GB$ of memory space), but we only use one GPU, at a time, for each learning process. For our implementation, we set both GPU count and images per GPU to 1. If we increase the images per GPU, we start lacking memory space. A $12GB$ GPU can typically only handle 2 images of 1024×1024 pixels, in this implementation of Mask RCNN.

We always set the steps per epoch and the validation steps to the sizes of the training and validation datasets divided by the batch size (which is 1, in our case), respectively. We make this to ensure that, in each epoch, we pass all the training and validation images through the network, exactly one time.

For the RPN stage, we use 5 different scales for the anchors, namely $8, 16, 32, 64, 128$. In each image, we only use 4 RoIs to train the layers of the classification stage, in which 1 is a positive RoI (with a score

of more than 0.7, from a scale of 0 to 1) and the remaining 3 are negative RoIs (with a score of less than 0.3). Neutral RoIs, *i.e.* with a score between 0.3 and 0.7, are not used for training.

Regarding the remaining *hyperparameters*, we use a ResNet backbone with a 101 layers deep architecture, the number of classes is 2 (background and Vizzy's hand) and, for the input image, we use a resolution of 256×256.

4.4 Real validation datasets

For the real validation datasets, we take images with the real robot in three different environments, (i) a low cluttered environment, with a simple solid background, (ii) a medium cluttered environment, with several distractor objects, and (iii) a high cluttered environment in an office-like scene and with different lighting effects. In Figure 4.3, we show two images of each dataset.

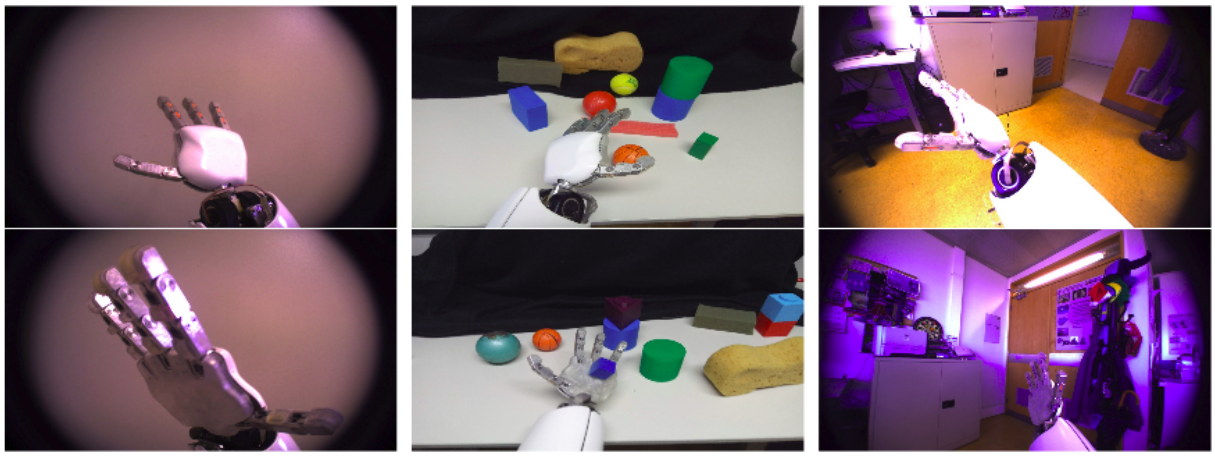

Figure 4.3: Real validation datasets, in low cluttered, medium cluttered and high cluttered environments, from left to right, respectively.

Dataset (i) consists only in 12 images, where we only vary the position and orientation of the robot's hand and arm. This is a simple dataset, with the purpose of validating the ability of the model to detect the hand, without much interference from background textures. This allows us to evaluate how similar the real robot's hand is to the 3D model, that is used in Unity to generate the positive instances of the training and validation data. If the model learns to detect almost as accurately in this dataset, as in the validation dataset, it means the 3D model has the appropriate textures, colors, size and shape to represent the hand of the real robot, in simulation.

For dataset (ii), we setup a scene with a relatively simple background, but with multiple distractor objects that, sometimes, occlude part of the hand. We vary the configuration of the objects, in a tabletop scenario, and the pose of the robot's hand and arm and sample a total of 17 images. Most of the distractor objects have parametric shapes and solid colors, as the ones generated in the simulator. This

dataset allows us to validate the model's ability to deal with occlusions and interactions between the robot's hand and other objects.

For the real validation dataset (iii), we sample a total of 30 images, with the real robot. We take the images in an office-like environment, with high amounts of clutter in the background and different lighting effects. We vary the orientation of the cameras, to capture different parts of the environment, and the pose of the hand and arm of the robot. This dataset allows us to evaluate how the model performs in extreme situations, with real textures very different from the textures generated in the simulator.

4.5 Evaluation metrics

In order to validate and compare different models, it is not enough to visualize the results and compare the images. Due to the scale of some experiments and datasets, visualization, although useful, becomes insufficient. This is why there is a need to define evaluation metrics, *i.e.* numbers that give insight on how the network performs on a certain aspect.

Note that the following methods are all semantic segmentation metrics, *i.e.* all the instances of a class present in an image are merged into a single binary mask and the accuracy is measured by comparing the groundtruth with the predicted binary mask.

First, we define some mathematical terms. P and G are the predicted and groundtruth masks, respectively, and are both binary matrices (*i.e.* the value of row i and column j is either 0 or 1) of size $h \times w$ (h and w are the height and width of the input images of the network). The operator $\cup$ is a logical or and the operator $\cap$ is a logical and.

4.5.1 Average Intersection over Union

This metric gives insight on how accurate the masks predicted by the network are, with respect to the groundtruth masks. The Intersection over Union (IoU) is the area defined by the intersection of both masks (predicted and groundtruth) over the union of the masks. This ratio gives us a notion of the accuracy, *i.e.* the amount of pixels that were correctly predicted, according to the groundtruth data.

Given as input the predicted masks of all positive classes and the corresponding groundtruth masks, for each image inferred in the network, this metric outputs the IoU, averaged across all inferred images. More detailed, the IoU for each image, is calculated as follows,

$$\text{IoU} = \frac{\sum_{i=1}^{h} \sum_{j=1}^{w} P_{ij} \cap G_{ij}}{\sum_{i=1}^{h} \sum_{j=1}^{w} P_{ij} \cup G_{ij}}, \tag{4.1}$$

Despite this metric being a good translation of accuracy, we cannot distinguish where the model fails the most, *i.e.* if it has more false positives or more false negatives.

4.5.2 Average *precision*

This metric represents the percentage of the predicted mask (only the positive instances) that is correct, according to the groundtruth mask. This is a ratio between the true positives and the total number of positives (true or false). The average *precision* gives us a notion of how good the model is at ignoring false positives, *i.e.* not detecting pixels that belong to the background.

Given as input the predicted masks of all positive classes and the corresponding groundtruth masks, for each image inferred in the network, this metric outputs the *precision*, averaged through all the inferred images. More detailed, the *precision* for each image, is calculated as follows,

$$precision = \frac{\sum_{i=1}^{h}\sum_{j=1}^{w} P_{ij} \cap G_{ij}}{\sum_{i=1}^{h}\sum_{j=1}^{w} P_{ij}}, \tag{4.2}$$

However this does not translate in accuracy because imagining the model detects just 1 pixel belonging to the hand, it would have 100% *precision* since all the pixels detected belong to the hand. If the hand has a total size of 1000 pixels, this means the model would have misclassified a total of 999 pixels.

4.5.3 Average *recall*

This metric represents the percentage of the groundtruth mask (only the pixels belonging to the hand) that was successfully detected, in the predicted mask. This is a ratio between the true positives and the number of pixels with value 1, in the groundtruth mask. The average *recall* gives us a notion of how good the model is at detecting the groundtruth mask, *i.e.* classifying as positives the pixels belonging to the hand.

Given as input the predicted masks of all positive classes and the corresponding groundtruth masks, for each image inferred in the network, this metric outputs the *recall*, averaged through all inferred images. More detailed, the *recall* for each image, is calculated as follows,

$$recall = \frac{\sum_{i=1}^{h}\sum_{j=1}^{w} P_{ij} \cap G_{ij}}{\sum_{i=1}^{h}\sum_{j=1}^{w} G_{ij}}, \tag{4.3}$$

This metric also does not translate in accuracy because imagining the model detects the whole image (*i.e.* assigns the value 1 to every pixel in the predicted mask), it would have 100% *recall* since all the

pixels belonging to the hand were detected. However, if the hand only occupies one third of the total image size, the other two thirds would have been misclassified.

5

Experiments and results

In this chapter, a series of experiments are conducted to produce both numerical and visual results, in order to test our hypotheses and compare different approaches. These experiments also aim to explore, in an empirical way, the effects of different training strategies and DR parameters. We use the architecture of Mask RCNN and different train and validation datasets are generated, in Unity. The processes of these experiments are shaped so that all the aspects of our approaches are validated. This is done, not only to validate the methods used, but also to provide insight on how to fine-tune this type of neural networks to work on different environments, with different goals. For instance, in our case, we show how to fine-tune a model pre-trained on a large scale dataset with 80 different classes, to work in a specific indoor environment and detect a different class (Vizzy's hands).

5.1 Experiments

In this section, we describe a set of planned experiments (see Table 5.1 for details on each experiment), on the real validation datasets defined in Section 4.4, using evaluation metrics explained in Section 4.5. The results of each experiment are shown and explained further, in this Section.

Table 5.1: Description of each experiment, and important conclusions that can be drawn from each.

Experiments	Tested methods	Description	Tested hypotheses
A	Domain randomization.	Train Mask RCNN with different simulation datasets.	Which randomization parameters affect most the performance, when validating the model with real images.
B	Transferred weights.	Divide the network into three stages (backbone, proposals, classification) and initialize each stage with pre-trained weights or train it from scratch.	Evaluate which stages should be trained from scratch and which should be initialized from pre-trained models.
C	Fine-tuning.	Divide the backbone into 4 blocks of layers and, after transferring the weights, each block is either frozen or fine-tuned to the new task.	Evaluate which blocks should be frozen, during training, and which should be fine-tuned to the new task.
D	*Hyper-parameters.*	We test different combinations of *hyperparameters*.	Evaluate the decrease/increase in performance, when changing some *hyperparameters*, in order to understand the effects of each one.

Each time we train the network, we stop the learning process when the total validation loss (the sum of the 5 validation losses) does not decrease for 15 epochs and we pick the model with the lowest validation loss for evaluation. We set the maximum number of training epochs to 150, in case the validation loss does not stop decreasing.

5.1.1 Results of experiment A

For this experiment, we generated 6 different training and validation datasets (see the bottom part of Figure 5.1, to see which components each dataset adds) in Unity. All the datasets have, approximately, the same number of images (1000 images), but in each new dataset it is randomized a new parameter, *e.g.* perlin noise, distractor objects, lighting effects and real images as background (check Section 3.2.1 for more detail on the random parameters). For each new dataset, the data generation process is repeated, but with the additional parameters, *e.g.* the dataset that contains real images as backgrounds also contains other images with perlin noise. The exception to this are the the first two datasets. In the first dataset we generate images with only the hand of the robot, while in the second we only generate images with the arm attached to the hand. This is due to the fact that we want to avoid scenarios that go too far from reality. We only use the first dataset as the baseline dataset (only contains a positive instance and a simple type of background) to which we add new properties, in order to validate the increase in the performance.

Furthermore, we train 6 *models* (with fixed *hyperparameters* and initializing with the same pre-trained weights), each using one of the 6 training sets generated in Unity. For each *model*, we infer the real validation datasets and finally, apply the evaluation metrics to the outputs of the inferences, using the groundtruth of the real images. The results of this experiment are show in Figure 5.1 (top part).

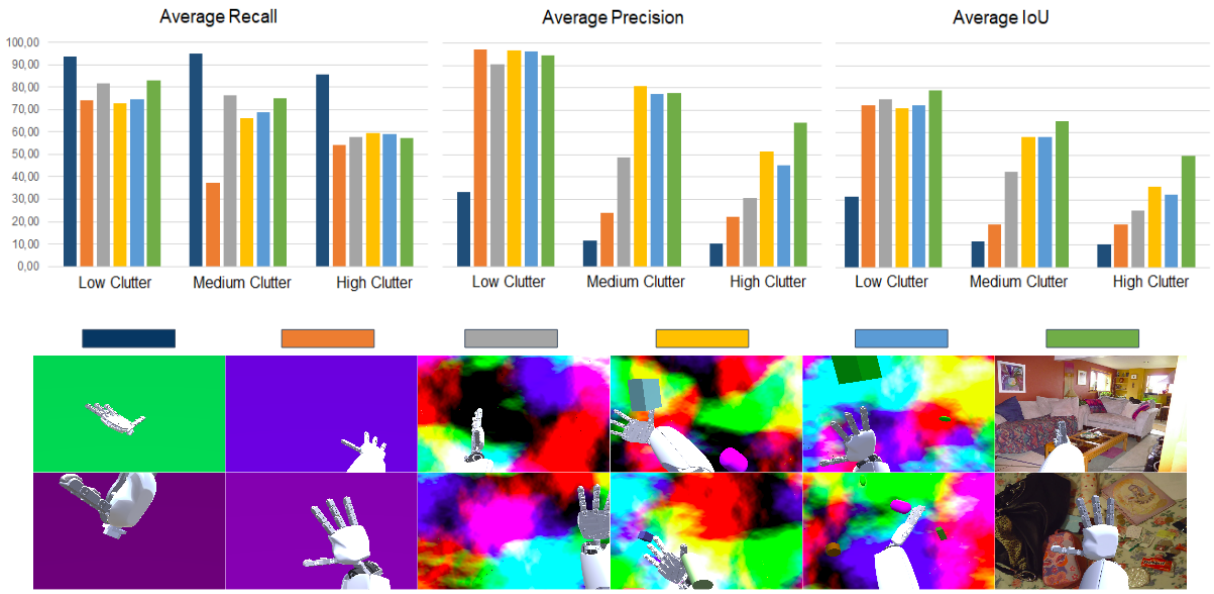

Figure 5.1: Results when validating 6 models on 3 real validation datasets with 3 evaluation metrics (histograms at the top). Each model is trained with a different training dataset, each dataset is shown in the bottom part of this figure.

By evaluating the average *recall* (Figure 5.1, top left), we can see that there is a significant decrease in this value when attaching the arm to the hand (from the dark blue dataset to the orange), meaning the second *model* detects less pixels belonging to the hand than the first. However, the average *precision* (top middle of Figure 5.1) increases by a large margin, meaning the second *model* also detects less pixels outside the boundary of the hand (*e.g.* belonging to the arm of the real robot). In the average IoU (top right, in Figure 5.1), we can see the accuracy of the *model* increases, meaning that the increase in average *precision* compensates for the decrease in average *recall*.

The 4 remaining training datasets mostly address the low average *precision* value, in the medium and high clutter datasets, without significantly affecting the recall. In Figure 5.1, we can see that the perlin noise (gray dataset) and the distractor objects (yellow dataset) are the components that most increase the average *precision* and IoU, in the medium clutter dataset. These components also increase the performance in the high clutter dataset, but the greatest performance increase in this dataset is from the addition of real images as backgrounds (green dataset).

The addition of lighting effects as randomization parameters (light blue dataset), contrary to our expectations, did not significantly affect the *model's* performance in neither of the real validation datasets. The high clutter dataset contains images with different lighting effects (with artificial lights turned on, with and without sunlight), so the *model* should benefit from being trained with different types of lighting effects, but we did not explore many types of lighting effects and we only randomize the orientation of the lights, in the simulator.

In Figure 5.2, we show the results of inferring 3 images from the low clutter real validation dataset, in the 2 first *models* (trained with the dark blue and orange datasets). As expected, by attaching the arm to the hand, in the training dataset, the model learns to ignore this part in the real robot as well. However, there are two unexpected results, (i) the *model* also learns to ignore other parts of the image that do not belong to the hand, and (ii) the *model* becomes more restrictive in what it considers to be part of the hand leading to a lower recall value.

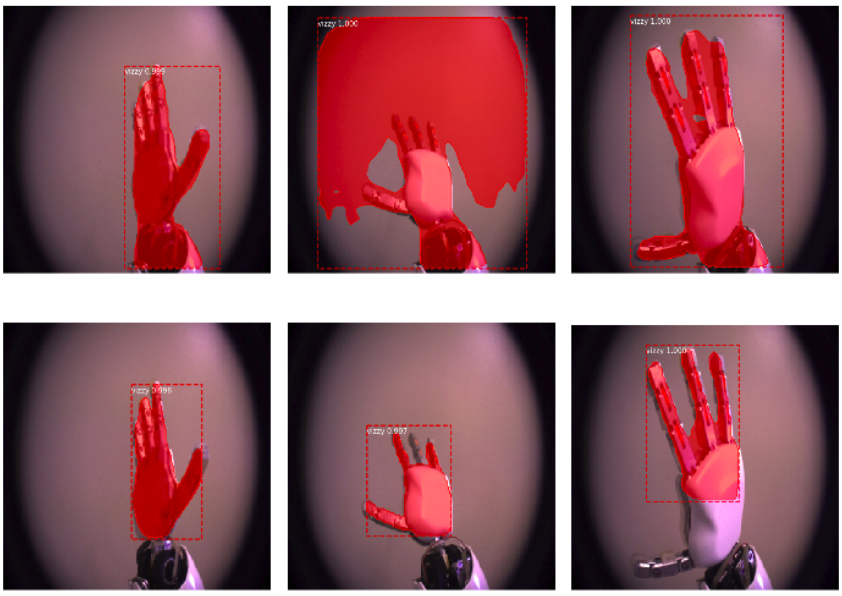

Figure 5.2: Results when inferring three images with 2 different *models* (the top images are results from the model trained on the dark blue dataset, and the bottom ones are from the *model* trained on the orange dataset).

5.1.2 Results of experiment B

In order to conduct this experiment, we divide the network into 3 stages, as was already explained in Section 3.3.3 (Figure 3.10). We use the *weights* of a Mask RCNN model, pre-trained on COCO [17], to initialize the layers of the network, and we use the green dataset (see Figure 5.1) for training and validation. For the real validation dataset, we merge our 3 real datasets (see Figure 4.3 of Chapter 4) into a single set and apply the evaluation metrics to it.

We train 4 models, each initialized in a different way. The goal is to evaluate different combinations of initializations, by transferring pre-trained *weights* to the given stage(s) and train the remaining stage(s) from scratch. The results of inferring the real and simulated validation datasets on these 4 models are shown in Table 5.2.

Table 5.2: Results of inferring 4 *models*, each trained from different initialization points, on the real and simulated validation datasets, using the evaluation metrics.

Transferred weights	Simulation			Real		
	Rec	*Pre*	*IoU*	*Rec*	*Pre*	*IoU*
Backbone.	85,8	88,4	77,5	64,3	71,8	54,4
Backbone and proposal stage.	**90,0**	93,5	**84,8**	62,5	**86,9**	58,2
Backbone and classification stage.	89,3	**93,8**	84,5	55,2	68,1	50,8
All stages.	87,2	92,8	82,0	**72,2**	79,5	**63,4**

From Table 5.2, we can extrapolate an interesting result. Previous work in transfer learning [31] has shown that if a CNN is pre-trained on a large dataset, for a given task, and we want to train the same CNN, but for a very different task and domain, then we achieve greater performance if we train deeper layers from scratch, instead of initializing with the pre-trained *weights*. In a first analysis, we might think that this would apply to our case, since our task is to detect a robotic hand in an indoor environment, with distractor objects, and Mask RCNN was pre-trained on a large scale dataset with 80 different classes and many types of environments. However, when using Mask RCNN to train a model with our training dataset, we actually achieve higher IoU ($63,4\%$, averaged across all real validation images), when inferring the real validation dataset, if we initialize all the layers with *weights* pre-trained on COCO. But since the validation dataset mostly contains textures generated in simulation, we achieve a higher IoU, when inferring the simulated validation dataset, if we train the proposal ($84,5\%$) or the classification ($84,8\%$) stages from scratch (but not both). This means that our target domain (*i.e.* the domain of our real validation dataset) shares even task specific features with the COCO domain, and if we use a small dataset (approximately 1000 images) for training we cannot train from scratch even deeper layers without causing the model to overfit to the data generated in the simulator.

5.1.3 Results of experiment C

For this experiment, we use the *weights* of a Mask RCNN model, pre-trained on COCO [17], to initialize all the stages of the network (picking from the best result of experiment B, Section 5.1.2), and we use the green dataset (see Figure 5.1) for training and validation. For the real validation dataset, we merge our 3 datasets (see Figure 4.3 of Chapter 4) into a single set and apply the evaluation metrics to it.

We divide the backbone into 4 blocks of layers, where the last layer of each block is the feed-forward connection to a different level of the FPN. More in detail, a 1×1 convolutional layer is applied to the output of each block of the backbone and the result is added to the feature map of the upper layer (except for the layer of the last block). Finally, a 3×3 convolutional layer is applied to each level, resulting in 4 different feature map levels.

We train 5 different models, each with a different number of backbone stages (or blocks) frozen (during training). After this process, we apply the evaluation metrics (just the IoU, because now we are only interested in measuring the accuracy) to each model, when inferring the real and simulated validation sets. The results of this experiment can be seen in Figure 5.3.

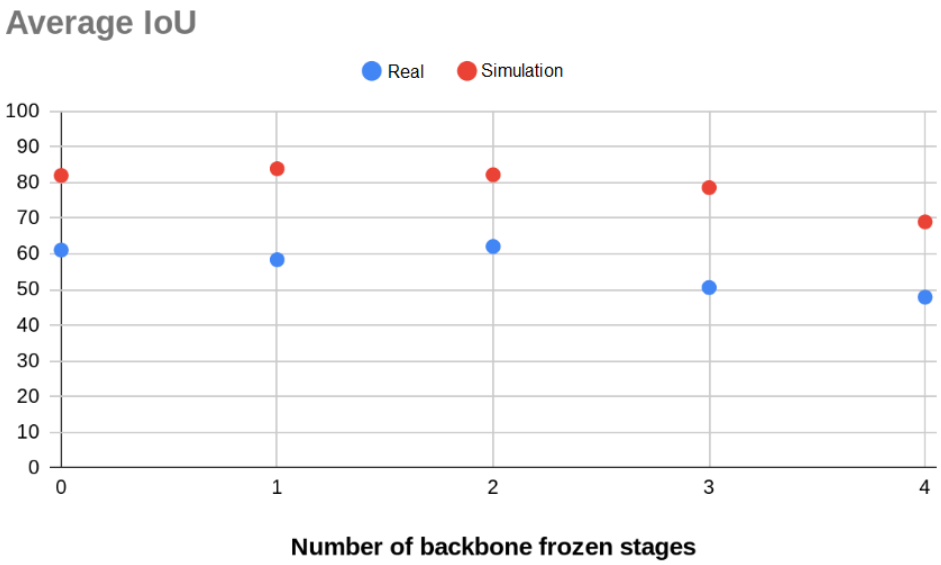

Figure 5.3: Results (average IoU) of inferring the real and simulated validation sets on 5 different *models*, each trained with a number of backbone stages (or blocks) frozen.

From the results of Figure 5.3, we can see that we gain no significant benefit in freezing backbone layers, during training, and we actually start loosing performance (in both the real and simulated validation datasets) when we freeze 3 or 4 backbone stages. So it is better, given the chosen pre-trained model, our classification task and our datasets, to fine-tune all layers to the new task.

In Figures 5.4 and 5.5, we can see the validation losses, for each training epoch, when fine-tuning all the layers of the network and when freezing the 4 stages of the backbone, respectively. The difference is mainly on the validation loss associated with the prediction of bounding boxes for the proposals of the RPN. This means that the RPN cannot properly learn to predict bounding boxes for the objects present in the image, without fine-tuning backbone layers.

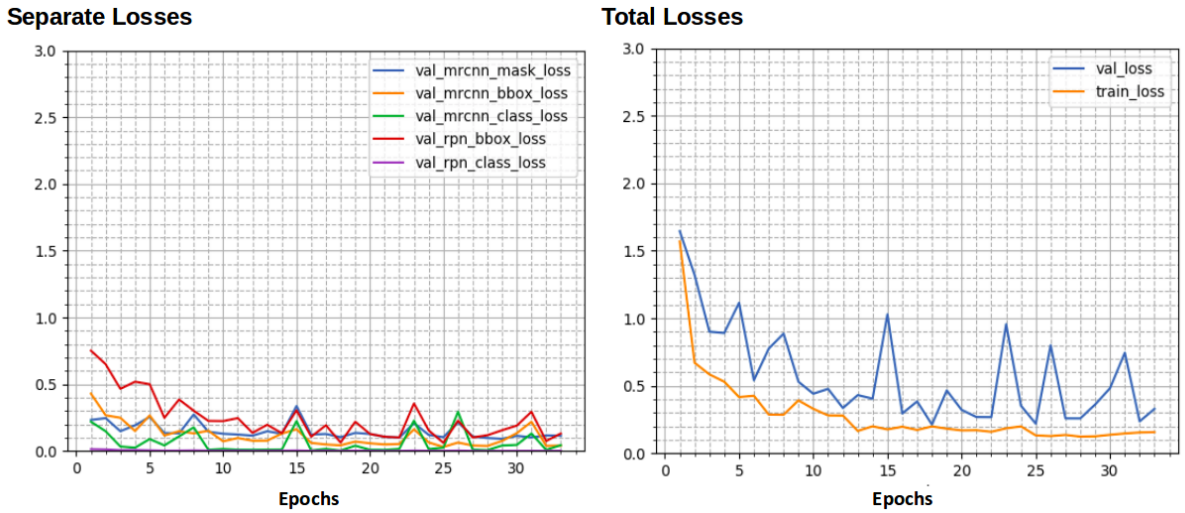

Figure 5.4: Plots showing each separate validation loss (left) and showing the total validation and training losses (right), for each training epoch, when fine-tuning all the layers of the network to the new task.

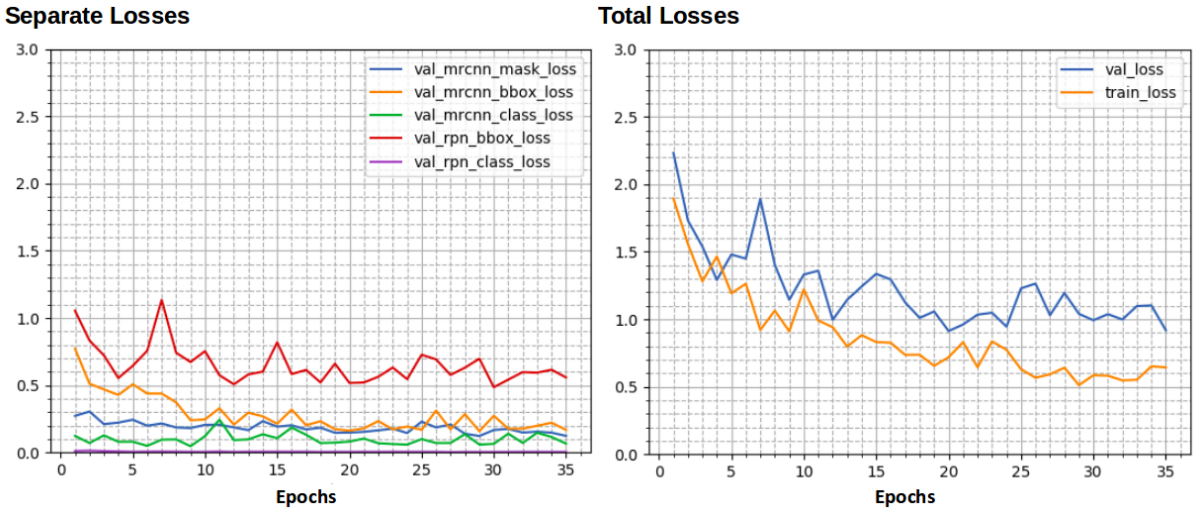

Figure 5.5: Plots showing each separate validation loss (left) and showing the total validation and training losses (right), for each training epoch, when freezing the backbone of the network and fine-tuning the remaining layers.

5.1.4 Results of experiment D

In this experiment, we also use the green dataset (see Figure 5.1) for training and validation and merge the 3 real validation datasets.

In Chapter 4, Table 4.1, we have already explained the *hyperparameteres* that were changed, for our implementation of Mask RCNN, and justified some of these changes. We found the learning rate to be the *hyperparameter* that brings the most variance to the results and, because of this, we make a

line search to evaluate how it affects the performance of the model and to find a suitable value for this *hyperparameter*.

We show, in Figure 5.6, the average IoU of each *model* trained with a different learning rate, on both the real and simulated validation sets. We search the learning rate in the range of values $[10^{-5}, 10^{-3}]$, with 10 equally spaced intervals, in a logarithmic scale, *i.e.* the learning rate is given by 10^{x}, where $x \in [-5, -3]$ with increments of 0.2, so $x \in \{-5, -4.8, -4.6, ..., -3\}$.

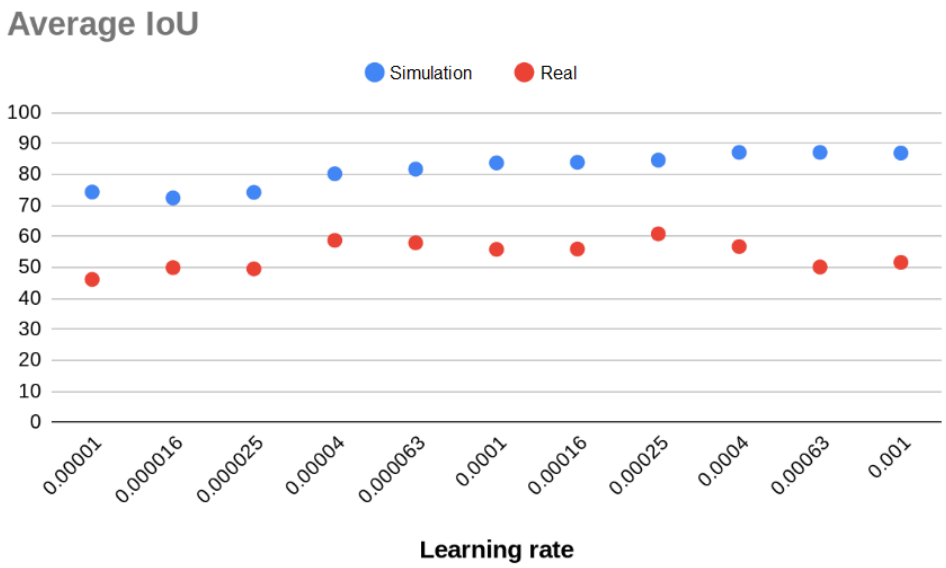

Figure 5.6: Results (average IoU) of inferring the real and simulated validation sets on several models, each trained with a different learning rate.

The results of our search (as we can see in Figure 5.6) show that the performance, in the simulated validation set, tends to increase with the learning rate, for the given interval of values. However, the performance in the real validation set starts to decrease again from $10^{-3.2}$ learning rate up. So, given the current configurations (train/val sets, pre-trained model), the optimal value for the learning rate is in the interval $]10^{-4.6}, 10^{-3.2}[$. For our line search, the optimal value (in the real domain) for the learning rate is $10^{-3.6}$.

5.2 Results of best model

We train a *model* with the configurations that yielded the best results, in Section 5.1, *i.e.* we use the green dataset (see Figure 5.1) for training and validation, we initialize all the stages of Mask RCNN with a model pre-trained on COCO and we fine-tune all the layers in the network, with a learning rate of 0.00025. We have validated this model on the validation datasets. Moreover, we display the IoU metric as a histogram, according to the amount of images within a given IoU interval (with a total of 10 equally

spaced intervals, from 0 to 1). We have a histogram for each validation dataset (i.e., low, medium and high clutter real datasets and the simulated dataset), all shown in Figure 5.7.

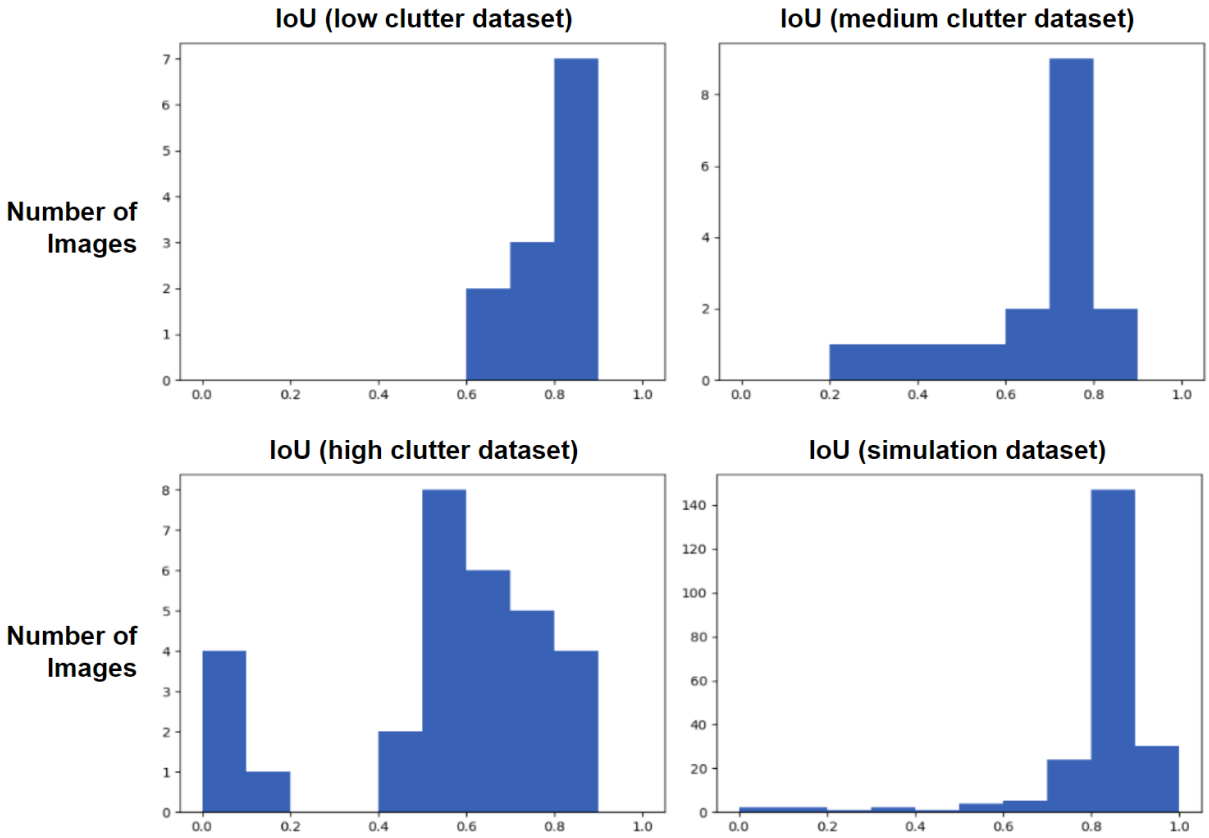

Figure 5.7: Results of inferring the low clutter (top left, average IoU of 80%), medium clutter (top right, average IoU of 66%), high clutter (bottom left, average IoU of $55,4\%$) and simulated (bottom right, average IoU of 82%) datasets, on the model.

The histograms of Figure 5.7 show that most of the images have an IoU value around the average value of the corresponding dataset, but there are some images that have very low IoU (between $0-0.4$). This mostly happens because, in some of those images, the bounding box predicted by the RPN is either too small (it only contains a small part of the hand, like the bottom left image of Figure 5.8) or it is not even predicted. However, there are also cases where the mask is not properly predicted, even when the bounding box is (like the top middle image of Figure 5.10).

We also show a visualization of some results, on some real and simulated validation images, by overlapping the predicted masks, bounding boxes and class scores on these images. We show results on the simulated validation (Figure 5.8), low clutter (Figure 5.9), medium clutter (Figure 5.10) and high clutter (Figure 5.11) datasets.

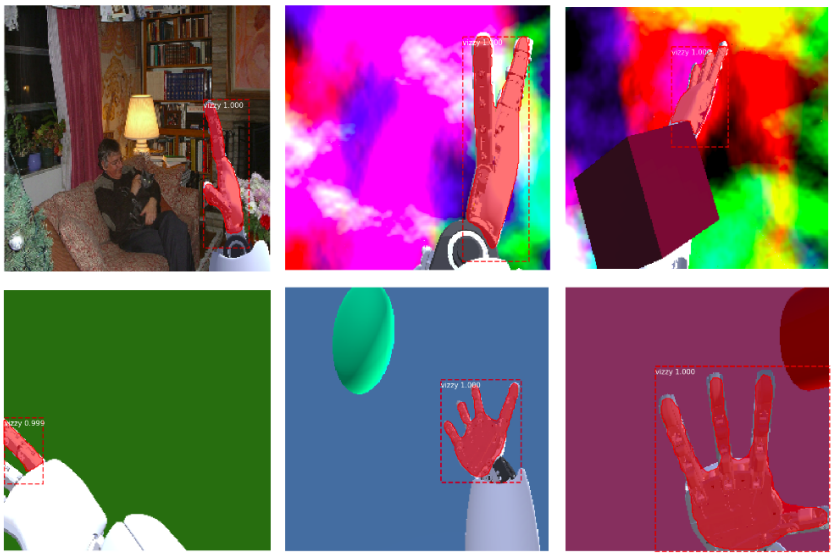

Figure 5.8: Visualization of the predicted masks, bounding boxes and class scores, for 6 images of the simulated validation set.

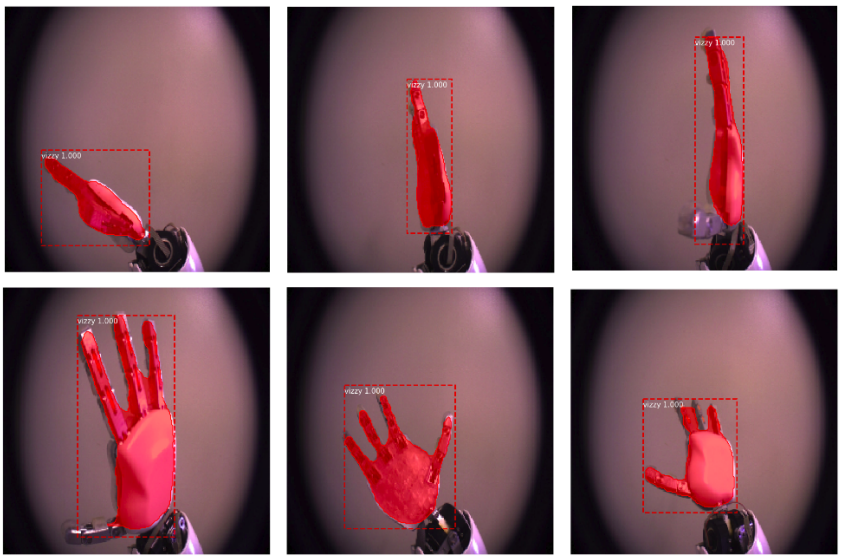

Figure 5.9: Visualization of the predicted masks, bounding boxes and class scores, for 6 images of the low clutter real validation set.

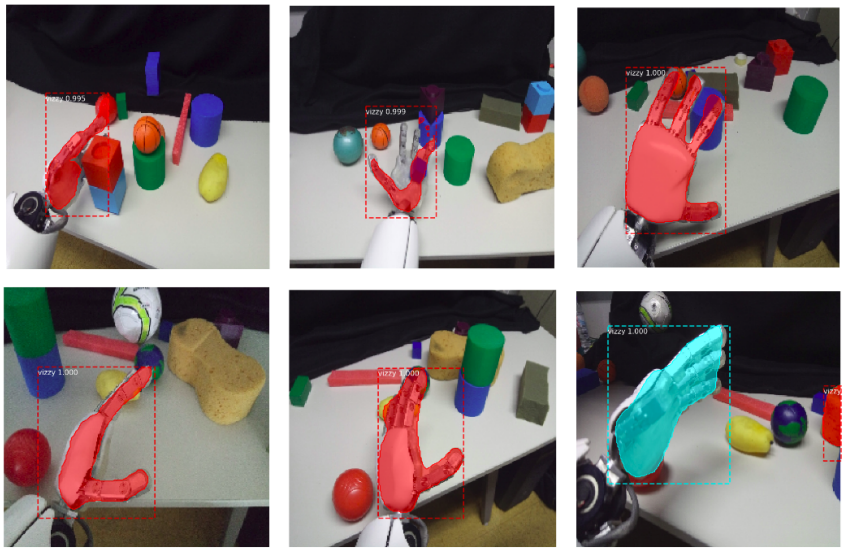

Figure 5.10: Visualization of the predicted masks, bounding boxes and class scores, for 6 images of the medium clutter real validation set.

Figure 5.11: Visualization of the predicted masks, bounding boxes and class scores, for 6 images of the high clutter real validation set.

5.3 Results on instance segmentation with test data

In order to test our final model's capacity to generalize to data that was neither used in training nor in validation, we collected and annotated 21 more images, with the real robot, in an environment similar to the one in which the high clutter dataset was collected. We inferred these images on our final model, yielding an average IoU of $56,3\%$. When comparing to the results of inferring the high clutter dataset in

this model (which yielded an average IoU of $55,4\%$), we can conclude that our model has the capacity of generalizing to unseen data, that was not used in the process of adjusting the methodology. These results can be visualized in Figure 5.12.

Figure 5.12: Visualization of the predicted masks, bounding boxes and class scores, for 6 images of the real test dataset.

Furthermore, we show, in Figure 5.13, results of inferring images where both the real robot's hands are visible. We use the same model that obtained the results of Section 5.2. These images were not used in neither training nor validation, making them part of the test dataset used to obtain the final results for our *model*.

Figure 5.13: Visualization of the predicted masks, bounding boxes and class scores, for 6 images taken with both hands of the robot visible.

The model used to obtain these results was trained on a training set where each image only contains 1 positive instance of the robot's hand (the 3D model of the right hand). We show that, even when trained with exactly 1 positive instance, in each training image, the *model* is able to detect more than 1 positive instance (left and right hands of the real robot) when testing with real images, without any data augmentation. The top images of Figure 5.13 even show reasonable results for both hands, but when their boundaries start to get close (as we can see in the bottom images) their masks start to misclassify too many pixels (however, the bounding boxes are still correctly predicted, in most cases).

6

Conclusions

With this work, it was intended to give a real robot the ability to perceive all the points belonging to its own hand, in an image. This is done by fine-tuning a pre-trained CNN to extract a binary masks of the robot's hand, from images. To do this, we had to face two main challenges: (i) collect and annotate enough images, to train a deep CNN, and (ii) overcome the differences in the domains and tasks between the pre-trained model and our target model.

To overcome the challenges inherent to this work, we developed a process for generating training and validation images, using Unity and DR concepts. Also, we break down the architecture of a complex CNN into a simplified structure and apply transfer learning methods, in order to find a good strategy that fine-tunes the *model* to the new task.

With our approach, we were able to create a simple process for fine-tuning the *weights* of a complex model, using only, approximately, 1000 images for training and validation, each image only containing 1 positive instance (*i.e.* 1 robotic hand) and simple scenes, without any need for data augmentation or other pre-processing methods. Our best model can achieve an average IoU of 82%, on simulated validation data. When evaluating on some real environments we can retain most of this performance, *e.g.* for the low and medium clutter datasets, we retain $97,6\%$ and $80,5\%$ of the performance, respec-

tively. However, when evaluating on the high clutter dataset, we are only able to retain $67,6\%$ of the performance on the simulated dataset.

Furthermore, we only evaluate the *models* on, approximately, 80 real images, since it is a very time consuming task to collect and annotate a dataset with a size significant enough. Due to the small size of our datasets, we are not able to make strong conclusions and evaluate the performance in many types of environment, with a high variability in the all the components of the environments.

Overall, we give a detailed insight on different learning strategies and data generation methods, which can be used to train a state-of-the-art network (Mask RCNN) and achieve the objective of this work, with reasonable performance and the ability to generalize to real test data.

6.1 Future work

With respect to the data generation process, future work can explore more randomization parameters and more suitable configurations for each parameter. For example, in the background wall we also applied two other patterns to the texture (besides the ones explained in Section 3.2.1.B), (i) gradient effect, and (ii) checkerboard pattern, but found these to have little effect or even worsen the performance, due to having no similarity to our target domain. However, other domains may benefit from these types of background. Also, for the lighting effects, we found our strategy to be ineffective (or insufficient), when evaluating in the real domain. It would also be interesting to explore other randomization parameters in this component, like the intensity of each light, the number of lights and the type of each light, as well as different ranges for each parameter, in order to achieve a good solution.

Concerning the learning process, although the implementation of Mask RCNN used allows for a large number of *hyperparameters* to be tuned, we only tune the learning rate. This is due to a limitation regarding the available number of GPUs that we could use for training, making it unfeasible (in terms of time) to train the network with several combinations of *hyperparameters*. If any future work has the possibility of training the network with several GPUs, at the same time, we recommend that a more detailed *hyperparameter* search is made, *e.g.* random search or even bayesian optimization, for a more efficient search, with more than 2 *hyperparameters* and several possible values for each. Finally, another important addition to this work would be to include the information of the robot's kinematics, in the region proposal stage, in order to relieve the burden of generating proposals, from the network.

Bibliography

[1] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, "Mask r-cnn," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2980–2988, 2017.

[2] P. Moreno, R. Nunes, R. Figueiredo, R. Ferreira, A. Bernardino, J. Santos-Victor, R. Beira, L. Vargas, D. Aragão, and M. Aragão, "Vizzy: A Humanoid on Wheels for Assistive Robotics," in *Robot 2015: Second Iberian Robotics Conference*, L. P. Reis, A. P. Moreira, P. U. Lima, L. Montano, and V. Muñoz-Martinez, Eds. Cham: Springer International Publishing, 2016, pp. 17–28.

[3] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 640–651, Apr. 2017.

[4] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915–1929, Aug 2013.

[5] B. Hariharan, P. Arbeláez, R. Girshick, and J. Malik, "Simultaneous Detection and Segmentation," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 297–312.

[6] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[7] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.

[8] G. Lin, A. Milan, C. Shen, and I. Reid, "Refinenet: Multi-path refinement networks for high-resolution semantic segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 5168–5177.

[9] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[10] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and L. Fei-Fei, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 12 2015.

[11] N. Silberman, D. Sontag, and R. Fergus, “Instance segmentation of indoor scenes using a coverage loss,” in *Computer Vision, ECCV 2014 - 13th European Conference, Proceedings*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8689 LNCS, no. PART 1. Springer Verlag, 2014, pp. 616–631.

[12] P. O. Pinheiro, R. Collobert, and P. Dollár, “Learning to segment object candidates,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’15. Cambridge, MA, USA: MIT Press, 2015, pp. 1990–1998.

[13] P. O. Pinheiro, T.-Y. Lin, R. Collobert, and P. Dollár, “Learning to Refine Object Segments,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 75–91.

[14] R. Girshick, “Fast r-cnn,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 1440–1448.

[15] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, 2015.

[16] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014.

[17] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft coco: Common objects in context,” *Lecture Notes in Computer Science*, p. 740–755, 2014.

[18] T.-Y. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, “Feature pyramid networks for object detection,” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 936–944, 2017.

[19] P. Vicente, L. Jamone, and A. Bernardino, “Robotic hand pose estimation based on stereo vision and gpu-enabled internal graphical simulation,” *Journal of Intelligent & Robotic Systems*, vol. 83, no. 3, pp. 339–358, Sep. 2016. [Online]. Available: https://doi.org/10.1007/s10846-016-0376-6

[20] T. Simon, H. Joo, I. Matthews, and Y. Sheikh, “Hand keypoint detection in single images using multiview bootstrapping,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 4645–4653.

[21] B. Doosti, “Hand pose estimation: A survey,” *CoRR*, vol. abs/1903.01013, 2019. [Online]. Available: http://arxiv.org/abs/1903.01013

[22] F. Yin, X. Chai, and X. Chen, “Iterative Reference Driven Metric Learning for Signer Independent Isolated Sign Language Recognition,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 434–450.

[23] P. Kakumanu, S. Makrogiannis, and N. Bourbakis, “A survey of skin-color modeling and detection methods,” *Pattern Recognition*, vol. 40, pp. 1106–1122, 03 2007.

[24] M. J. Jones and J. M. Rehg, “Statistical Color Models with Application to Skin Detection,” *International Journal of Computer Vision*, vol. 46, no. 1, pp. 81–96, jan 2002. [Online]. Available: https://doi.org/10.1023/A:1013200319198

[25] A. U. Khan and A. Borji, “Analysis of hand segmentation in the wild,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2018, pp. 4710–4719.

[26] J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber, “Humanoid learns to detect its own hands,” *2013 IEEE Congress on Evolutionary Computation, CEC 2013*, 06 2013.

[27] J. Tobin, R. H. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain randomization for transferring deep neural networks from simulation to the real world,” *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 2017.

[28] K. Perlin, “Improving noise,” *ACM Trans. Graph.*, vol. 21, no. 3, pp. 681–682, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566654.566636

[29] J. Borrego, R. Figueiredo, A. Dehban, P. Moreno, A. Bernardino, and J. Santos-Victor, “A generic visual perception domain randomisation framework for gazebo,” in *2018 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, April 2018, pp. 237–242.

[30] S. James, A. Davison, and E. Johns, “Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task,” 07 2017.

[31] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, “How transferable are features in deep neural networks?” in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds.

Curran Associates, Inc., 2014, pp. 3320–3328. [Online]. Available: http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf