UNIVERSIDADE
BEIRA INTERIOR

Deep Reinforcement Learning for 3D-based Object Grasping

Ricardo André Galhardas Vermelho

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

(2º ciclo de estudos)

Orientador: Luís Filipe Barbosa de Almeida Alexandre

junho de 2021

Acknowledgements

To my supervisor Professor Dr. Luís Alexandre, for his availability, dedication and unconditional support. For all the teachings and suggestions given and for the indispensable help he gave me, as well as for his words of encouragement and understanding showed upon my work, during my academic path here at University of Beira Interior.

To my parents João and Dina, whom I love and to whom I thank very much, for the education, principles and values transmitted, and the possibility of following my professional and academic dreams and for the support given and demonstrated every day, to continue to face the adversities and fight for my goals, without forgetting the difficulties that both had in order for me to be able to finish my Master's.

To the rest of my family, my very real thanks, for each tender word, the positive energy that they gave me daily, for the encouragement, for the way they taught me to grow over this time. And, above all in memory of my aunt Maria João, my great friend, who I do not forget her contagious sympathy, generosity and delicacy and all the daily support for 21 years of my life. To her I express my love, affection and gratitude. Thank you, dear aunt for everything!

Colleagues and friends are also worthy of a note of appreciation for their friendship, availability in raising pertinent questions and all the ideas they shared with me during the Master's.

To all those who made it possible to carry out this work directly or indirectly, my thanks.

Resumo alargado

Hoje em dia, ouve-se falar mais de robôs e do crescimento da robótica do que se ouviria há duas décadas atrás. A indústria da robótica tem vindo a evoluir imenso e a prova disso é a existência de robôs em estações de trabalho e laboratórios, cujo seu propósito é colaborar nas tarefas dos seus colegas trabalhadores humanos. A este tipo de robôs dá-se o nome de *Cobot* ou robô colaborativo.
Estes robôs têm de suporte algoritmos da Inteligência Artificial para os ajudar a tomar as decisões mais corretas nas tarefas que têm de desempenhar. Contudo, este tipo de robôs já começa a ser adotado para tarefas domésticas.
O tema desta dissertação envolve três grandes áreas: Inteligência Artificial, Visão Computacional e Robótica e tem como principal objetivo o desenvolvimento de um algoritmo de Aprendizagem por Reforço, que dê suporte a um robô universal, versão 3, na tomada de decisões para apanhar e separar objetos de cozinha por tipo.
Assim sendo optou-se pelo uso de um algoritmo já desenvolvido, chamado Visual Pushing-for-Grasping, que permite simular robôs colaborativos a empurrar e apanhar objetos. Todavia, os objetos utilizados por este algoritmo em simulação não eram objetos de cozinha e o algoritmo apenas realiza apanha de objetos sem realizar a separação dos mesmos.
Como tal, propomos uma nova abordagem com base no algoritmo anteriormente referido, e que passará a utilizar modelos 3D de objetos de cozinha, fará a deteção do tipo de objeto no cenário com recurso a um modelo de deteção de objetos exterior ao algoritmo base e que procederá à separação dos objetos por tipo.
Os resultados experimentais permitem concluir que esta nova abordagem ainda precisa de ser melhorada, contudo e por ser uma abordagem nova tanto no ramo da Robótica como no ramo da Inteligència Artificial, para uso com o robôs universais da versão 3, afirmamos que os resultados estão melhores do que o esperado e expectamos que um dia esta possa ser aplicada a um robô físico em contexto real.

Abstract

Nowadays, collaborative robots based on Artificial Intelligence algorithms are very common to see in workstations and laboratories and they are expected to help their human colleagues in their everyday work. However, this type of robots can also assist in a domestic home, in tasks such as separate and organizing cutlery objects, but for that they need an algorithm to tell them which object to grasp and where to it.
The main focus of this thesis is to create or improve an existing algorithm based on a Deep Reinforcement Learning for 3D-based Object Grasping, aiming to help collaborative robots on such tasks. Therefore, this work aims to present the state of the art and the study carried out, that enables the implementation of the proposed model that will help such robots to detect, grasp and separate each type of cutlery objects and consecutive experiments and results, as well as the retrospective of all the work done.

Keywords

Reinforcement Learning, Object Detection, Object Grasping and Pushing, Point Cloud, Cutlery Objects.

Contents

List of Figures

List of Tables

Listings

Acronyms

AP	Average Precision
CBL	Convolution with Batch Normalization and Leaky-ReLu
CDNA	Convolutional Dynamic Neural Advection
COCO	Common Object in Context
CNN	Convolutional Neural Network
CSP	Cross Stage Partial Network
DNA	Dynamic Neural Advection
FCN	Fully Convolutional Network
FCLSTM	Fully Connected LSTM
GPD	Grasp Pose Detection
GPG	Grasp Candidate Generation
GPU	Graphics Processing Unit
GWS	Grasp Wrench Space
IFR	International Federation of Robotics
IoU	Intersection over Union
LSTM	Long Short-Term Memory
LTS	Long-term Support
mAP	mean Average Precision
MDFGPP	Mean Distance between Final and Goal Pixel Position
MPC	Model Predictive Control
PANet	Path Aggregation Networks
POV	point of view
PSNR	Peak Signal-to-Noise Ratio
QNET	Deep Q-Network

RAHP	Recall-at-High-Precision
RGB	Red-Green-Blue
ROS	Robot Operating System
SOCIA	Soft Computing and Image Analysis Laboratory
SPP	Spatial Pyramid Pooling
SSIM	Structural Similarity Index Measure
STP	Spatial Transformer Predictors
TCP	Transmission Control Protocol
TD	Temporal Difference
TTA	Test Time Augmentation
UBI	University of Beira Interior
UNET	U-Net Convolutional Network
UR3	Universal Robot 3
VPG	Visual Pushing-for-Grasping
YCB	Yale-CMU-Berkeley
YoLo	You-Only-Look-Once

Chapter 1

Introduction

1.1 Context

This thesis is being developed in the sequence of second year of Master's Degree. It was a proposed theme by teacher Luís Alexandre, called **Deep Reinforcement Learning for 3D-Based Object Grasping** and it is a follow-up from the past work developed by my colleague Gabriel Esteves, called **Grasping Cutlery using a Universal Robot**.

As years go by, technology evolves exponentially and the Human being is constantly trying to find solutions to the problems he has to face.
The field of Industry as seen huge revolutions in its core over the past decades, but the undergoing revolution might be the biggest of all. With the introduction of robots in the field of Industry, it has become more clear that they are a great addition to ease mankind's daily tasks, since humans are slower and less accurate [24]. Thus, robots play a key role in today's manufacturing industry. The **IFR** stated that up until the year of 2019, the supply of industrial robots would increase 13% per year [64].

With the introduction of robots, two new concepts regarding interaction between robots and humans, were introduced. The first is called *Non Collaborative Robots* and the second is *Collaborative Robots*. We can establish a differentiation, simply by saying that *Non Collaborative Robots* do not interact physically with a human being since they lack in safety and are located in designated areas just for them. As for *Collaborative Robots*, more commonly known as **Cobots**, they interact with humans while performing their tasks, are lightweight and can be easily moved [64].

So, Collaborative Robots have been adopted in the Domestic area, for tasks like floor and windows cleaning, lawnmowers and also as robotic toys and games. This market has grown over the past years. As example, the cleaning robots'market has seen an increase of over 24% in 2018 when compared to the values of 2017 [43].
Collaborative Robots, as any other robot, have some basic actions like grasping and pushing, but sometimes they need the objects to be in a certain angle, orientation or position. In this work, we want to develop a software model that might give these robots the capability to grasp or push objects that is independent from the object's position and orientation.

1.2 Motivation

Robotics has evolved a lot since it was first researched 50 years ago. Nowadays, robotics reached major fields like manufacturing, transportation of goods and medicine.
There has been a major investment in robotics in a diversity of areas. Some of the most known are the

Space Exploration [63], which involves robots in space explorations, in maintenance of shuttle and stations, among other uses; the **Marine** field, reports investments in robotics for underwater exploration [28] and in the **Medicine** field, which counts with more than *200000* successful operations [35].
With the technological evolution, new concepts arise in the Robotics field. Some of them came to help people in need or at their work, for example: a new concept regarding robots is called Service Robots that work in private homes, helping people in they daily life [4].
Another rather new concept is the Collaborative Robots. These robots are safe and lightweight, making interactions with people possible, which helps their cooperation with humans in their tasks [11].
All of what was said before is a part of the motivation to develop this work. Although, at a personal level, this theme has come to expand my knowledge in Artificial Intelligence, regardless of the fact that I have never work in the Robotics' field and it makes me curious.

1.3 Objectives

The main objective of this thesis is to endow a robotic arm with the capability of grasping objects in the simplest way possible. For a human being, grasping objects from any kind of container is an easy task, but for a robot it involves complex steps and programmed moves, since usually they are controlled rigidly to make precise and repetitive movements. In order for a robot to grasp any kind of cutlery object, first it is necessary to know the precise orientation and position of the object, and with this knowledge, compute the robot's movements to a point where it can grasp that object.
This thesis wraps three main fields of Artificial Intelligence: **Reinforcement Learning**, **Computer Vision** and **Robotics**. To achieve the goal of this work, it is necessary to understand the basis of the intermediate goals embedded in it: First, a simple understanding of Reinforcement Learning is needed, in order to contextualize the bigger task that lies ahead; Second, read about what was done in this area until present day in terms of research and/or approaches; Third, develop some approach to tackle the main objective; Fourth, learn how to manipulate the robotic arm and its gripper and Finally, incorporate the developed architecture with ROS and get it to work in the actual robotic arm, after the proposed method has been tested in simulation.

1.4 Document Organization

Reflecting the work that has been done, this document has the following structure:

1. Chapter 1: **Introduction**
   Formally and briefly presents the project, its context in the modern world, motivation, objectives regarding its choice, as well as the structure of the document;
2. Chapter 2: **Background**
   Describes the proposed problem and approaches subjects related to the proposed problem;
3. Chapter 3: **Related Work**
   Describes the summary of papers related to this project divided into three categories: Grasping, Pushing and Pushing with Grasping;
4. Chapter 4: **Proposed Work**
   This chapter describes how the VPG algorithm works, its architecture, the metrics used to eval-

uate it, the metrics we chose for evaluate the model and an introduction to the changes we want to make to this algorithm;

5. Chapter 5: **Proposed Changes to VPG**
   In this chapter we described all the proposed changes to the original algorithm of VPG to speed up the process of localize and grasp specific objects;
6. Chapter 6: **Simulation with CoppeliaSim**
   Describes details about the simulation, the usage of container or not for the simulation and the obtained results;
7. Chapter 7: **Conclusions and Future Work**
   This chapter contains our final thoughts on this project, what was involved in the making of this work and what was left undone;
8. Appendix A: **Installations Guide**
   On this appendix you may find the instructions on every step made to test each approach and instructions about the steps taken to install the needed packages;
9. Appendix B: **Glossary**
   This last appendix contains descriptions of methods that are embedded to some of the approaches referred in the Related Work's chapter.

Chapter 2

Background

2.1 Introduction

This chapter approaches the proposed problem, as well as fundamental and theoretical subjects related to the proposed problem, for example, what is Reinforcement Learning and what it involves and some of the metrics used in the Reinforcement Learning field, which are the background of our thesis.

2.2 Proposed Problem

In the laboratory, we have a robotic arm (Figure 2.1) that we want to be able to grasp objects within any position, orientation and from any angle and in any container. Without a designed Reinforcement Learning algorithm, that would control its movements, the robot's movements would be rigid, precise and repetitive movements. As stated in section Objectives, this thesis covers three main areas of **Artificial Intelligence**, crucial to this work: **Reinforcement Learning**, **Computer Vision** and **Robotics**. Thus, it is imperative to have a better understanding of the basis of Reinforcement Learning, since this kind of algorithms will be the ground of the system that will be supporting the robot's decision by providing it with a value expressing how good the outcome from applying an action over an object would be. Then, resorting to a 3D-depth camera we will be able to provide guidance, support and vision to the robot in making its decisions.
Since in the Robotic field there aren't any approaches, for UR3 robot, oriented to grasp cutlery and sort it in baskets, then the result must be an algorithm capable of supporting the robot in making actions for grasping cutlery objects.

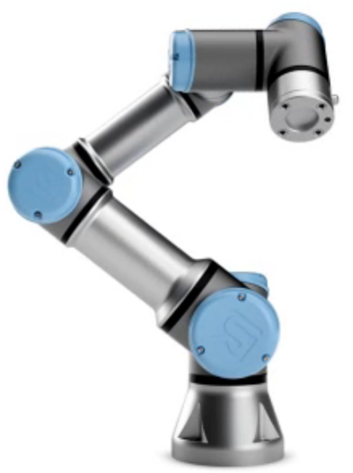


Figure 2.1: UR3 from [73].

2.3 Related Subjects

By introducing the proposed problem, there are important subjects that need to be introduced as well to help contextualize the fields where the problem of this thesis falls in.
These subjects are Machine Learning, Reinforcement Learning, Q-Learning, Deep Q-Learning, Evaluation metrics used in Reinforcement Learning field and an Object Detection model.

2.3.1 Machine Learning

The first important subject that wraps some of the other subjects, if not all, we will present here is called Machine Learning. This field is mostly applied as a form of statistics that uses computers to estimate statistic values for complicated functions without the need to prove confidence intervals for them [34].

What are Learning Algorithms?
Machine Learning is composed of algorithms that are capable of learning through data given as input [34].
The last sentence raises an essential question: ”What does it mean that an algorithm can ’learn’”?
It is said that only through an Experience E, an algorithm is capable of ”learning”. However, these Experiences are related to the class or kind of Tasks T and the Performance measures P, manifested by the algorithm during those Experiences. Obviously, the Performance P improves with more tasks T made, thus obtaining more Experience E [34].
The Performance P is a statistic value describing how good an algorithm performed towards an Experience E over a Task T [34].

A Machine Learning Task is described as the way the system processes a collection of features, which were previously measured so we may have information about an event we want the system to process [34].
There are a vast number of types of Machine Learning Tasks, even though the most common are the following:

- Classification: In this domain, the program has to specify which k categories, the input belongs to [34];
- Classification with missing inputs: In this domain, it is not guaranteed that with every measurement, the input will be given and in that case it might appear some missing inputs. That said, the program will have to learn from a set of functions which ones will give him the classification of the x missing inputs [34];
- Regression: In this domain, given an input it is asked that the program predicts a numeric value for the given input [34];
- Anomaly detection: In this domain the intention is to go through a set of data and find those that aren't common or are atypical [34].

For tasks like Classification and Classification with missing inputs, we measure the model's Accuracy and Error Rate onto the test set [34]. The Accuracy of a model corresponds to the proportion of the inputs to which the model produced the correct output [34]. The Error Rate corresponds to the proportion of the inputs to which the model produced an incorrect output [34].

We say that a model generalizes well if it obtains a good Performance in inputs it has never seen before [34]. However, with Generalization comes a set of errors that can be measured.

- Training Error: This error is measured on the training dataset. By knowing its value we can perform improvements to reduce it [34];
- Generalization Error or Test Error: This error is the value expected on a new input. Ideally we want it to be low and is generally calculated on the test dataset [34].

The Capacity of a model determines if it is more likely to Overfit or Underfit and can be described as the aptness to cover a big set of functions [34].
Underfit is more likely to occur on the training set if the error obtained by the model isn't low enough. Overfit is more likely to occur if the difference between test and training errors is too big [34].

- Models with low capacity are more likely to Underfit, because they might fail fitting the training set [34];
- Models with high capacity might Overfit, since they memorize properties that do not apply well on the test set [34].

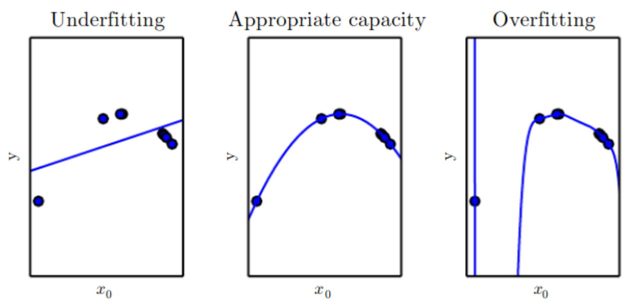


Figure 2.2: This image resembles the Underfit, Optimal Solution and Overfit for Linear Regression, from [34].

A way to control the capacity of a model is to choose the correct ”Hypothesis Space” the model. Hypothesis Space is the set of functions that an algorithm is allowed to go to in order to select one that is suitable to solve the problem [34].
From figure 2.2 we can somehow generalize the problem:

- The problem would Underfit if a function is not capable of capturing the curvature of the representative dots of the data. In the Linear Regression case, we are talking about a Linear Function [34];
- When a function is capable of capturing the curvature of the dots representing the data, we can't see any problem of Underfit or Overfit [34];
- Lastly, the problem would Overfit if a function is larger than it should be in a way it can fit much more functions (dots) than those needed. In the Linear Regression case, a Polynomial Function of degree 9 would cause the Overfitting [34].

Anyhow, in Machine Learning there are eight types of approaches, but the most important/common are [34]:

- Unsupervised Learning Algorithms: This type of algorithms use data sets that contain many categories. They must learn the data set's properties [34];
- Supervised Learning Algorithms: These algorithms use data sets in which the examples are all

labeled [34];
- Reinforcement Learning Algorithms: These algorithms aim to help an agent conduct actions according to how good a certain action proves to be [58].

Our work falls over one of the three major approaches previously referred called, Reinforcement Learning.

2.3.2 Reinforcement Learning

In Reinforcement Learning we learn to map situations to actions, to have a reward value as large as possible. The agent is told to try every action to see which one yields the highest reward. Challenging environments may have possible actions that not only affect the current reward but also the next situation [58].

Features of Reinforcement Learning
This field of Machine Learning has two important features called ”Trial and Error” and Outcome.

- ”Trial and Error” $\rightarrow$ The Reinforcement Learning algorithm together with a robot and a 3D camera, will decide if a specific action appears to be better than another [34];
- Outcome $\rightarrow$ The outcome will be good or bad according to an action realized by the agent and this will help the agent in making decisions when it encounters an identical/similar situation in the future, whether the agent decides to use a previous action or to use a new one [34].

Stakeholders of a Reinforcement Learning situation
Approaches based on Reinforcement Learning algorithms usually involve three components:

- An **agent** to interact with its surroundings by sensing the state of the environment to achieve a goal [58];
- The **environment** which is in a specific state and within which the agent introduces an action to alter its state, receiving a **reward** telling it how good was its action [34];
- And lastly, the **objectives** or goals. These objectives will help the agent in making decisions in a way they will affect the environment surrounding the agent [34].

Many times, these approaches follow a Markov Decision Process, although any method capable of solving problems involving the previous stakeholders in a real environment would be considered a Reinforcement Learning method [34].

What is a Markov Decision Process?
A Reinforcement Learning environment resembles a Markov Decision Process considering that in an environment of a Reinforcement Learning situation, the agent has to decide which action is best according to the current state of the environment and the process of deciding which action is best repeats through out the whole problem.
Therefore, a Markov Decision Process contains the following stakeholders [58]:

- A state S_t;
- A possible action A_t;
- A reward R_t received after transitioning from one state to another;
- An agent that interacts with its surroundings;
- An environment that the agent has to explore and exploit to cause changes;

- A policy $\pi(S)$ that maps the solution to the Markov Decision Process of state S_t using action A_t.

A state S at a given time t is considered as some sort of signal given to the agent like a perception of "how the environment is" at a given moment. These states or signals are produced by a system that is part of the environment [58].
The agent has a set of possible actions - Actions A, that it has to exploit to obtain a reward or to explore to make a better action, to transit from state S_t to state S_{t+1} of the environment [58].
From applying an action A_t to the state S_t of the environment, the agent receives a reward indicating how well he performed while applying that action. The main objective is that the agent must maximize this value in order to make the right choice when facing the environment's current state [58].
The reward function $R(S_t, a, S_{t+1}')$ returns a expected value from applying an action a while being in state S_t and ending up in state S_{t+1} [58].
A Policy $\pi(S)$ is a function that maps the correct action a to apply while in state S [58], i.e., it defines the way the agent will act at the present moment by mapping of the environment states onto actions and letting the agent choose which action to perform according to the environment's current state [34]. Depending on the situation, a policy may be an easy function and in other situations may require a high number of computations [58].

Challenges towards Reinforcement Learning
Reinforcement Learning has some demanding challenges. However, there is one that is more challenging than the others. It is called the Exploration-Exploitation dilemma.
This dilemma concerns mainly the agent present in the environment in which it has to take actions to obtain rewards. The agent has to decide between exploiting actions he has already taken in the past and gave him a good reward and exploring new actions that might give him better rewards than the previous and known actions [58].
By having to explore or exploit all the possible actions in order to alter the environment's state, the agent is kind of "obliged" to do more work rather than stick with the suitable options that in the past gave him good results [58].

2.3.3 Q-Learning

Reinforcement Learning field has many algorithms including Q-Learning [58].
Q-Learning is a model-free algorithm of Reinforcement Learning, based on the state-action-reward-state premise, and was developed by Watkins in 1989 [58]. It is considered an off-policy TD algorithm, because it is independent from the policy while directly estimating the optimal q_* value for the pair action-state. However, the policy π is still important to help the algorithm decide which pairs of state-action are visited and updated [58]. The next formula defines Q-Learning algorithm:
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma * max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
$Q(S_t, A_t)$ defines the optimal value for the current state-action pair, whereas α is the step-size parameter with value between 0 and 1, γ is the discount factor between 0 and 1 and $Q(S_{t+1}, a)$ is the expected optimal value of applying an action a in state S_{t+1} [58].
An algorithm is considered to be model-free if it is based in trial-and-error and does not compute possible changes of the environment in response to an action taken by it [58].
A Q-Learning algorithm can be defined by figure 2.3 [58]:

```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 2.3: Q-Learning Algorithm pseudo-code, from [58].

This kind of algorithm has a simple pipeline [58]:

1. For each environment state:
   (a) Get possible set of next states S;
   (b) For each possible S:
      i. Using a policy, choose the action with highest q_* value for the current state S and observe the change produced S';
      ii. Visit and update each pair of state-action $Q(S, A)$
      iii. Update state S as new state S';

The above process executes until the algorithm reaches the final possible environment state S.

However, this algorithm is not an optimal solution in all cases. Take for example ”The Cliff” problem represented in figure 2.4.
The concept behind this problem is that the Q-Learning agent has to avoid the gray zone called ”The Cliff” and go from starting point S to goal position G. The red line called ”Optimal path” represents a Q-Learning algorithm and the blue line represents another algorithm called Sarsa, which is not independent from the policy [58].
As we can see, the Q-Learning learns an optimal policy but, now and then, it may result in its fall to the cliff and receiving a negative reward of -100, whereas Sarsa algorithm chooses a path much safer with rewards that may be -1 but never -100, because it never gets closer to the cliff [58].

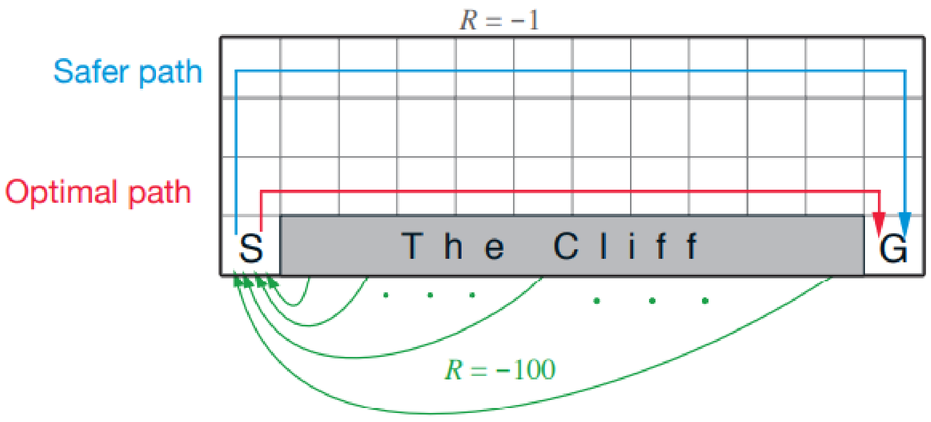


Figure 2.4: Q-Learning Algorithm Process [58].

2.3.4 Deep Q-Learning

Q-Learning is a model-free algorithm and its performance problem demonstrated in figure 2.4 only shows that, by not caring about the changes produced by its actions, it underperforms [58].
To address this defect, Deep Q-Learning was introduced as a combination of Q-Learning with a convolutional network, that allows the possibility to store experience replay and eliminated the instability described with the "The Cliff" problem [58].
As an example of the usage of Deep Q-Learning, these algorithms were trained to learn how to play Atari 2600 games such as Bellemare and Bowling, by being in contact with them for 38 days or 50 million frames of interaction with the games. As comparison, the Deep Q-Learning algorithm was tested against a professional human in these games and its performance was marked as 75%, which is above the performance of the professional human [58].
While the human sees images with resolution of 210x160 pixels image with 128 colors at 60Hz, the algorithm sees an array of size 84x84x4 representing the size of a frame (84x84) and the four recent frames, turning the game into a Markov Decision Process, where each frame represents a possible state [58].

2.3.5 Evaluation Metrics used in Reinforcement Learning Field

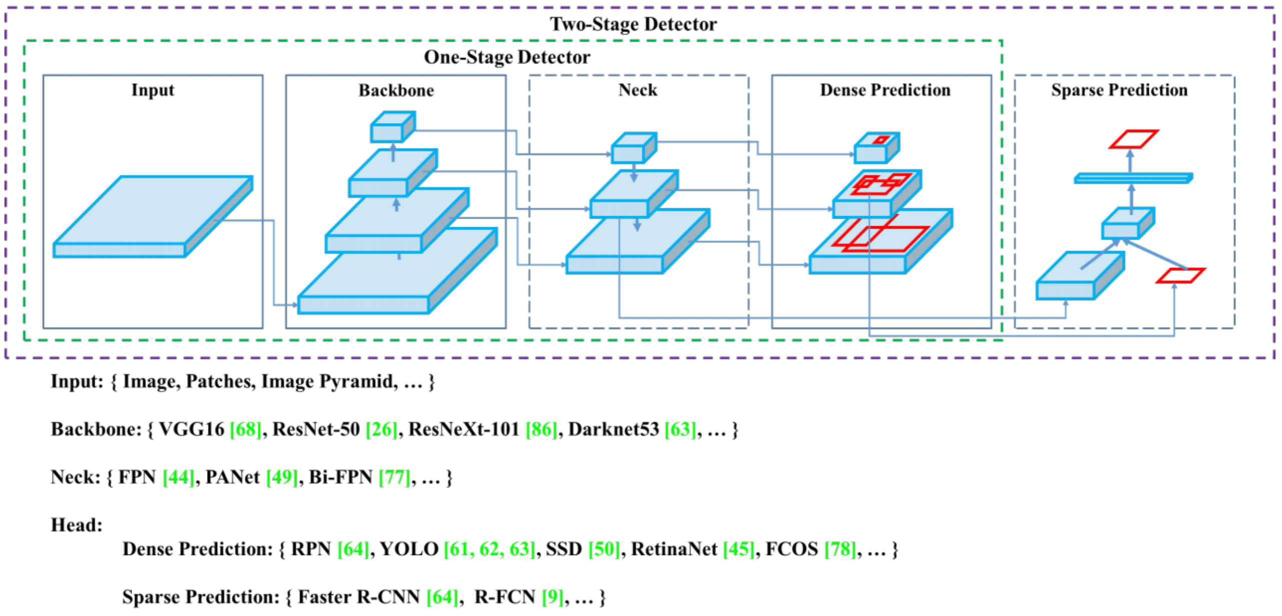
Reinforcement Learning models also need some way to be evaluated. Thus, the existence of some metrics based on the usage of a Reinforcement Learning algorithm for robotic tasks divided by three categories:

1. **Grasping objects**:
   - Grasp Success - Describes the performance of the algorithm in grasping tasks, usually given by successful grasps divided by total grasps made;
   - Average Grasp Success Rate - Described in [38] as the number of objects grasped during all the grasp attempts;
   - Completion Rate - Described in [38] as number of objects grasped during experiment divided by total number of objects in test;
   - Suction Success Rate - Described in [17] as the number of objects successfully grasped divided by the number of lift operations.
2. **Pushing Objects**:
   - Mean Distance between Final and Goal Pixel Position - Normally expressed in millimeters, this metric tells us how close the final pixel position of an object resulting from pushing actions is to the original goal pixel position;
   - Peak Signal-to-Noise Ratio - Evaluate image quality. Acceptable values would be between 30 and 50dB;
   - Structural Similarity Index Measure - Measures the similarity between two images. Good values are between 0.97 and 0.99.
3. **Whole Experiment**:
   - Test Success Rate - Described in [17] as the number of successful tests divided by the total number of realized tests;
   - Recall-at-High-Precision - Referenced in [52] and described in [33] as the measure of grasp detection performance when the number of false positives is small;

- Action Efficiency - Described in [38] as the number of objects grasped divided by the number of actions required to grasp them.

2.3.6 Object Detection Model

The purpose of our work is to endow the UR3 with the capability of grasping objects, thus we have chosen an already developed algorithm, VPG. However, the idea is to add something new to this approach. Therefore, we though of adding an Object Detection model and by doing so, the algorithm would be capable of detecting an object and tell the robot to grasp it.
YoLo is the name of the model we adopted to implement this detection phase onto the VPG. It is a set of models specialized in object detection [3] trained on the COCO dataset [10] and each of its models can be divided in four components [1].



Input: { Image, Patches, Image Pyramid, … }

Backbone: { VGG16 [68], ResNet-50 [26], ResNeXt-101 [86], Darknet53 [63], … }

Neck: { FPN [44], PANet [49], Bi-FPN [77], … }

Head:

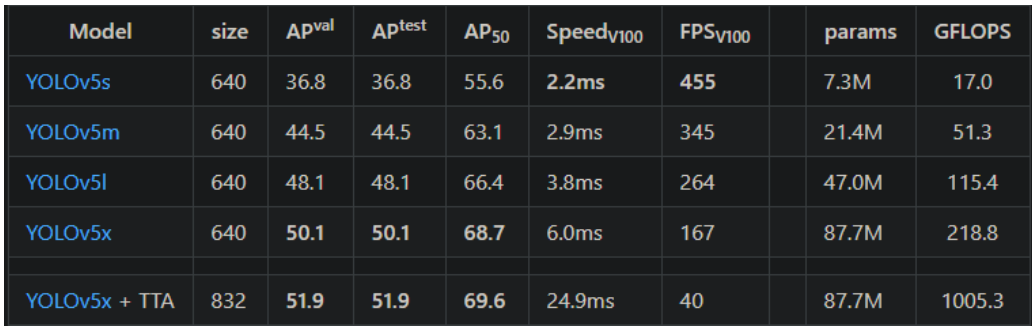
Dense Prediction: { RPN [64], YOLO [61, 62, 63], SSD [50], RetinaNet [45], FCOS [78], … }

Sparse Prediction: { Faster R-CNN [64], R-FCN [9], … }

Figure 2.5: YoLo version 4 structure, from [81].

The YoLo version 5 has an identical structure to that present in the previous version of YoLo, version 4, and although the components of each of the 4 parts may vary, the structure is unchanged and is composed of:

1. The first element being the input terminal. It receives the images and enhances the data contained in those images;
2. Second is the Backbone structure of the model [2]. This structure is responsible for the feature extraction phase, i.e., extracting more info from the images to be able to predict the classes of the objects in the analyzed images later on and is composed basically by CSPs [B.9] and one SPP [B.11];
3. After the Backbone structure comes the Neck structure composed by PANet [B.10], which are used as neck to get feature pyramids [2];
4. Lastly, YoLo has the Prediction or Head of the model. This structure is responsible for the final detection part of the whole process. It generates the probabilities of the classes of the objects, object scores and bounding boxes around the objects present in the image 2.6 [2].



Model	size	AP^{val}	AP^{test}	AP_{50}	$Speed_{V100}$	FPS_{V100}	params	GFLOPS
YOLOv5s	640	36.8	36.8	55.6	**2.2ms**	**455**	7.3M	17.0
YOLOv5m	640	44.5	44.5	63.1	2.9ms	345	21.4M	51.3
YOLOv5l	640	48.1	48.1	66.4	3.8ms	264	47.0M	115.4
YOLOv5x	640	**50.1**	**50.1**	**68.7**	6.0ms	167	87.7M	218.8
YOLOv5x + TTA	832	**51.9**	**51.9**	**69.6**	24.9ms	40	87.7M	1005.3

Figure 2.6: YoLo version 5 let us choose from 4 options, from [82].

YoLo provides four different options of models with its version 5. As figure 2.6 shows, we can choose from the smallest version (version s) up until extra-large version (version x) with Test Time Augmentation - which increases the size of the images to obtain better results. The differences between them, excluding the last version with Test Time Augmentation, are the number of parameters used by each of the models and the number of layers implemented within each option of the YoLov5.

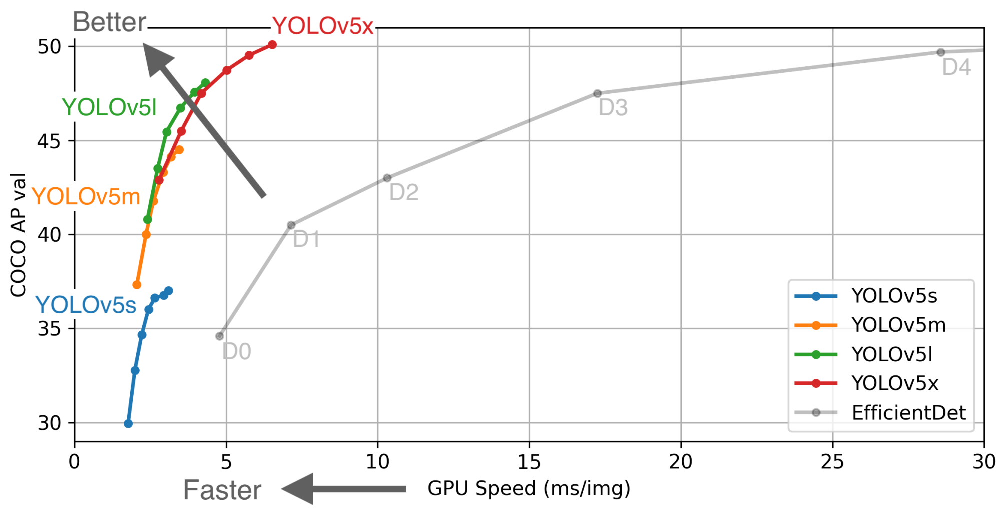


Figure 2.7: YoLo version 5 - The 4 options and comparison between how fast and accurate they are, from [82]

Since this algorithm is based on the evaluation of the best prediction to execute an action (push or grasp), we propose the addition of an Object Detection model to make the process of grasping an object more selective and therefore, hope to improve the grasping ability of the algorithm.

This Object Detection model will be trained to detect cutlery objects such as forks, knives and spoons, because the images from the simulation had very low quality. The metrics used by YoLo are precision, recall, mAP@.5 and mAP@.5:.95 [67].
In Image Classification, precision is the positive predicted value of a class and is basically the ratio of true positives to the sum of true positives and false positives, and the recall is the ratio of true positives to the ground truth of positives (true positives + false negatives).
When having a set of detections, the mAP is the mean value over all classes, i.e., each class has a value representing the AP of the calculations of the detector and the mAP is the interpolation of all APs. The AP of a class corresponds to the area below the precision/recall curve [44].

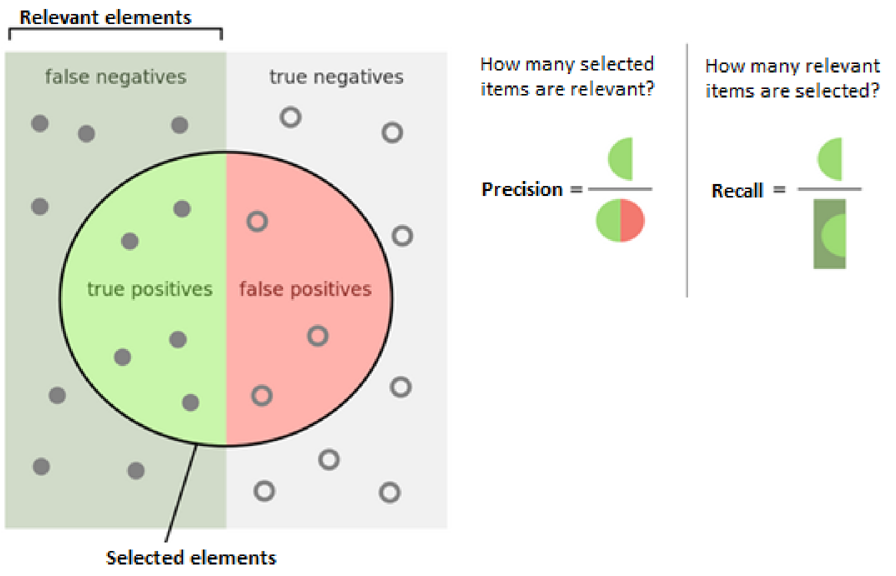


Figure 2.8: Precision and Recall relationship, from [25].

IoU is a ratio between the area of intersection and the area of union of the predicted bounding box and the ground truth bounding box. This way, IoU will only be used to assess if the predicted bounding box is true positive, false positive or false negative.
The predicted bounding box will never be evaluated for true negative since it is assumed that an image is not to be empty of objects.

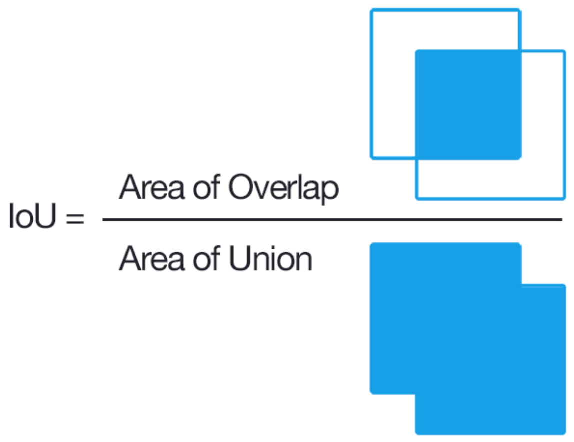


Figure 2.9: IoU Equation for Object Detection, from [37].

Using IoU we can be sure of when mAP is correct or not. For example, when looking for mAP@.5 we are actually looking for values of mAP where IoU is greater than 0.5.

2.4 Conclusion

In this chapter we introduced the proposed problem, related subjects to the proposed problem such as Reinforcement Learning and briefly presented the object detection model and some of the metrics used to evaluate its performance.

Chapter 3

Related Work

3.1 Introduction

This chapter presents a summary of the studied methodologies separated into three possible motions that the robotic arm can do: Pushing, Grasping and Pushing with Grasping and a comparison between these approaches.

3.2 Pushing Studies

3.2.1 Deep Visual Foresight for Planning Robot Motion

Chelsea Finn and Sergey Levine [19] introduced in this paper a Reinforcement Learning approach for planning robot motion without human supervision. Thus they want the robot to predict the effects of its actions in advance and therefore allow for the creation of flexible models that can be applied to a wide variety of situations and tasks.

They propose combining learning a predictive model of raw observations with MPC, without the need of a reward function or image goal. It requires little human intervention, uses unlabeled training data, and also doesn't require a calibrated camera or precision sensor. This kind of problem typically has a difficult task tied to it, which is related to pushing an object to the desired position, which contains some challenges such as estimating and modeling the physical world and might be unnecessary for the desired task.

The main goal of this approach is to move pixels from a position to another. Thus, an approach that optimizes each sequence of actions to make the pixels move as desired is needed. Summarizing, the technique involves an MPC algorithm based on probabilistic inference that allows the robot to make a plan for its actions regarding moving user-specified objects to user-defined locations.

The proposed model, figure 3.1, is composed of a series of LSTM [B.3] layers that receive an RGB image and computes a prediction on the following image containing the next movement of the user-specified pixel(s). This allows to continuously predict the movements of the pixels until they arrive at the desired goal.

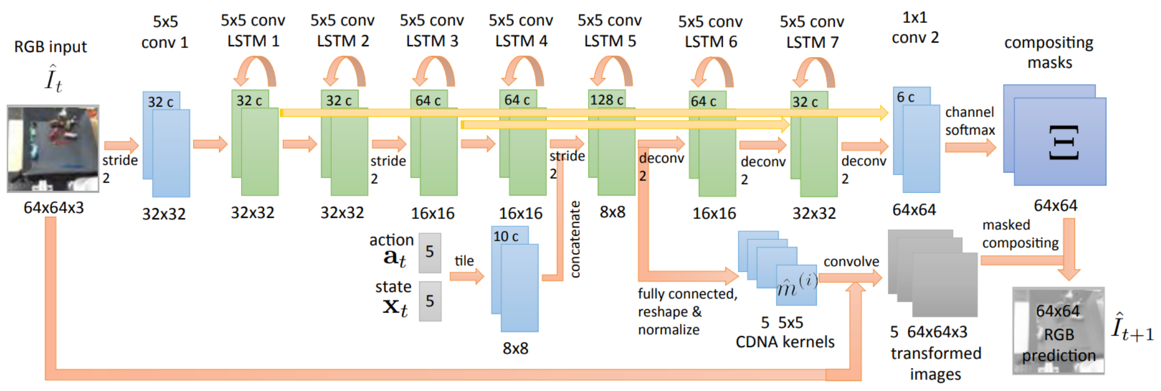


Figure 3.1: Proposed Model composed by a series of LSTMs, from [19].

The above model receives as input the current and the previous images $I_{0:1}$, the gripper pose X_t, both the end-effectors (previous and present) and a sequence of actions a_t from which the robot may execute one.
As we can see, the model is composed of a Convolutional layer followed by a series of LSTM [B.3] layers and one last Convolutional layer.
The first convolutional layer will extract features from the images and output them to the LSTM layers. The LSTM layers will process the features alongside the data about the gripper pose X_t and the available actions a_t (concatenated with the output from LSTM layer number 4) and output this information to the last convolutional layer. The last convolutional layer will convert the data from the LSTM layers to the original image's resolution, resulting in a full-resolution mask (full map of the pixels that contain objects).
From LSTM layer 5, a model of the current pixels is outputted, normalized, and reshaped into 10 CDNA (Convolutional Dynamic Neural Advection) kernels or affine transformation matrices. These matrices will then suffer a convolution, given place to 10 transformed images.
The resolution mask is compared to the 10 produced images from LSTM layer 5. Parts that are the same are not replaced in the mask. The other pixels that are different will be multiplied by the corresponding pixels from the 10 transformed images, resulting in the predicted frame.
This model differs from the modules referred in 3.2.2, since it applies a normalization layer after each layer of the neural network.

The **Visual Model-Predictive Control** has three major steps embedded in it. The first is called **Specification of Goals** in which the position of the goal pixels is specified. The second is called **Evaluate Actions** - we consider moving from a pixel to a goal pixel location upon a series of actions. And lastly, **Sampling-Based MPC** optimizes a set of actions at each time step.

The results where obtained under experiments with a 7-DOF robotic arm and evaluated with the mean distance between final and goal pixel position. The proposed method achieved the smallest value for the used metric, with a value of 2.52±1.06mm.

As conclusion, the proposed method intends to predict the goal pixel location of a desired object in an image by using a convolutional LSTM [B.3] model with unlabeled data to determine which actions the robot must take to make the designated pixel achieve its goal.

There is a lack of some data in results, for example: execution times, obtained accuracy and hardware used to run the proposed model. This work doesn't handle occlusions too well, meaning that objects that have some part under other objects are not taken into account by the model.

3.2.2 Unsupervised Learning for Physical Interaction through Video Prediction

When interacting with the ambient surrounding it, the agent has to learn how to predict the effect of its actions. Replicating this type of interactions in the real-world is challenging, because the environment is full of different scenes. Thus having labeled data becomes more and more difficult to acquire. Chelsea Finn, Ian Goodfellow and Sergey Levine [71] proposed a method involving an action-conditioned video prediction model that works with the pixel motion. This approach predicts a motion from a previous image. When applying a motion onto the previous image it is able to create the next frame.

The main problem regarding the Video Prediction theme, is the fact that most of the previous works are based in short-term prediction using LSTM [B.3] models and the need a ground truth in order to know if they achieved a good result or not. Thus, the proposed model is a combination of **CDNA** and **STP** to transform previous images in new pixels.

The authors of this paper proposed three modules very alike, but with some differences between them:

1. **DNA**: computes a distribution over pixel locations in previous frames for each pixel in the new frame. Each predicted value is seen as an expected value.
   This module can be described as the application of the following formula:
   $\hat{I}_t(x, y) = \sum_{k\in(-k,k)} \sum_{l\in(-k,k)} \hat{m}_{xy}(k, l)\hat{I}_{t-1}(x - k, y - l)$.
   To get the next prediction $\hat{I}_t$, we apply a motion transformation $\hat{m}$, more simply described as a vector, for each pixel (x,y) of the previous image prediction $\hat{I}_{t-1}$ and k represents how wide the distribution computed for each pixel really is;
2. **CDNA**: computes parameters of normalized convolution kernels. These parameters are then applied to the previous images forming new pixel values.
   $\hat{J}_t(x, y) = \sum_{k\in(-k,k)} \sum_{l\in(-k,k)} \hat{m}_{xy}(k, l)\hat{I}_{t-1}(x - k, y - l)$
   From the above formula, we retrieve the idea that via one predicted object transformation $\hat{m}$ applied to the previous image I_{t-1}, we compute a new image $\hat{J}_t$ for each pixel of the previous image. Thus, to get the full prediction we must apply multiple transformations $\hat{m}^i$ onto the previous image, which will then be obtained by combining all transformations $\hat{J}^i_t$;
3. **STP**: computes parameters of various matrices of transformation. These parameters are then applied to the previous images.

Furthermore, their objective is to predict an object's motion without reconstructing its appearance.

In the experiments, all modules were evaluated by two metrics' PSNR, which evaluates the image quality and acceptable values would be between 30 and 50dB and SSIM, which is used to measure the similarity between two images and its good values are between 0.97 and 0.99. All modules were trained for 8 time steps and tested for up to 18 time steps. 5% of training set was used for validation. The results show that the models lost quality on the predicted images over time, since PSNR was lower than 30dB and SSIM was below 0.97.

In summary, this paper [71] is a new approach for motion prediction using robotic arms is proposed. It is based on three modules with some little differences, though in their essence they had LSTM layers connected to each other [B.3] and generate new predictions for the images fed to the network. Although, they manage to obtain some decent results and far greater than normal methods like FC LSTM [B.4]. We would have like to see a measurement of the accuracy of the models, how much time the models took to run and what hardware they used to run the models.

3.3 Grasping Studies

3.3.1 PointNetGPD: Detecting Grasp Configurations from Point Sets

Uncertainty may come from a wide variety of aspects, even for a robot that operates in real world conditions. The main focus of this research resides with imprecision and deficiency in sensing, usually associated with the sensor for robotic perception.
There is a suggestion in [15] and [74] to use deep neural networks for planning grasps directly with sensor input like images or point clouds. The proposed method [54] is an end-to-end grasp evaluation that addresses the problem of finding the configurations for robot grasp. This model receives raw point clouds as input to capture geometric structures of the area in between the gripper and the object.

The problem of this research resides in the fact that grasping objects is a difficult task. In [54], the main objective is to learn how to grasp objects using robotic hands with a specific gripper configuration and a reinforcement learning algorithm, given a point cloud of the objects we want to grasp. Therefore, the proposed model has a module to generate a dataset composed of sampling and scoring stages, and has a module to generate the grasp candidate, called GPG.

The GPG was adopted in order to compute sampling of heuristic grasps from point cloud and was modified to reduce collisions between generated grasps and surface.

1. Discard generated grasp that are close to the surface;
2. Remove configuration that is approaching away from support surface;
3. Pull colliding grasps to opposite direction of their approach until collision disappears.

The Sampling stage of the model samples the existing meshes in the dataset to possibly prevent generating unfeasible grasps, and the Scoring stage is a way of labeling grasps, so distinguish good from bad grasps is facilitated.

The proposed method will evaluate grasp quality by analysing point cloud with *PointNet*.
Generating the dataset involves two steps: sampling and scoring.

Sampling: YCB provides registered point cloud for most objects, however sampling the meshes may prevent the generation of unfeasible grasps. These are the steps:

1. For each grasp, choose two random surface points p_1, p_2 as contact points and an angle between $[0, 0.5\pi]$ for approach;
2. Construction of grasp g $((p_1 + p_2)/2, r)$;

3. Simulate an approach to see if it will collide, so unfeasible grasps may be eliminated;
4. The remaining grasps will be transformed from meshes into point cloud coordinates.

Scoring: There are two different grasp quality metrics to label a grasp, given the grasp g and object state s.

1. **Force-closure** metric Q_{fc}. Requires the coefficient of friction γ and returns a binary outcome indicating if grasp is force-closure or not.
   Modification: γ increases gradually from 0.4 until grasp is antipodal. $1/\gamma$ is recorded as its score. The lower the γ, the better, robust and feasible will be the grasp.
2. **GWS** Q_{gws}. Uses the radius of GWS as a score of grasp quality.

To combine this two metrics, a weighted sum is adopted, and produce the final score given as follows: $Q(s, g) = \alpha Q_{fc}(s, g) + \beta Q_{gws}(s, g)$.

To learn the quality of the grasp, PointNet will take as input the grasp represented by point cloud within closing are of the gripper.
For learning and inference, the point cloud will be transformed into unified local gripper coordinate to eliminate ambiguity. Then, these N points will go through the network to estimate the level of quality.

The generated dataset is used to train the grasp quality evaluation model. The quantitative quality is now non-binary labels, so assigning labels and enable multi-class grasp quality classification is more flexible.
Equal number of grasps are sampled with Q_{fc} to balance different qualities and real point cloud is used instead of simulated point cloud.

The model will use a C-class cross-entropy loss as classifier. To optimize the network, the Adam optimizer and other parameters are used. The data is augmented by adding random offset to the point cloud.

This approach experiments' are divided into simulation and real-world categories. The simulation architecture counts with 200 epochs of training with GPG and obtained a classification accuracy of 91.81% for a full point cloud set. Real-world experiments were carried out on a UR5 robotic arm with Robotiq 3-finger gripper and obtained a 90% completion rate for a 3-class classification on proposed model.

Concluding, the presented model, PointNetGPD, is a novel approach for detecting grasp configurations, addressing the grasp quality evaluation over imprecise and deficient point clouds. Experiments show that PointNetGPD outperforms state-of-art grasp detection methods.

3.3.2 PointNet++ Grasping: Learning An End-to-end Spatial Grasp Generation Algorithm from Sparse Point Clouds

Ever since the evolution of technology, mankind has been eager to study new ways to apply and improve robotic grasping in high challenging environments. The most demanding problems are related to stacked objects which are a costly issue when looking for grasps; the data generated from the camera may be noisy and sparse and finally, the difficult choice of which metric to use when looking for the best grasp among all candidates.

In [52], the main objective is how to tackle the grasping action without resorting to traditional algorithms involving prebuilt databases, since they are pretty difficult to adapt to unseen objects and use CNNs [B.1] for extracting features, since they only extract local features and are not capable for 3D point classification tasks. Thus, the proposed model is based on **PointNet++** [52] approach with a multi-mask loss to qualify scores, categories and poses of grasps and uses 79 objects from YCB database from which 20k point cloud annotations are generated.

There are two ways to generate and select grasps. One is using metrics to evaluate a single object and its boundaries, according to every possible rotation of the gripper and respective collision distance to the object. The other is related to stacked objects and is able to compute the grasps for objects within the camera region following the previous principle.

The experiments used 8192 points from resampling the point clouds of the taken photos. These points are then trained using the same parameters of PointNetGPD [54] and a GeForce 840M GPU in simulation, however using a different metric called RAHP to evaluate grasps until they reach 99% precision. In the real-world, the model was put to test with an UR10, an SR300 RealSense camera and a total of 30 objects were used from which 50% correspond to unseen objects.
As the authors of this research said and we can see from the results, this model achieves good results both in *success rate* and *completion rate* for both known objects and unseen objects.

In conclusion, the multi-object grasp evaluation algorithm is probably the best so far according to documented results, since for novel objects it is able to achieve 71.43% success rate and 91.60% completion rate and the fact that this algorithm is prepared to handle object occlusion from point cloud data, identifying parts of objects that are occluded by others.

3.4 Pushing with Grasping Studies

3.4.1 Learning Synergies between Pushing and Grasping with Self- Supervised Deep Reinforcement Learning

Andy Zeng *et al.* [16] introduced a method for robotic manipulation involving grasping and pushing objects. Besides making a simple differentiation between **Prehensile** and **Non-Prehensile** motions, they show that it is possible to learn these motions from scratch with a reinforcement learning model.

The challenging task of manipulating a robotic arm for grasping and pushing is formulated over a Markov Decision Process.
Given a state s_t in an instant t, execute action a_t according to a policy reward $\pi(S_t)$. After the application of the previous action, the Markov decision process shifts to the next state s_{t+1}. A state in this problem's case is a projection of the **RGB-D heightmap** onto a 3D Cloud Point. The main goal of this problem is to learn a policy π that maximizes the future reward. As figure 4.1 suggests, at the

beginning of each epoch the algorithm formulates an heightmap of the scenario and sends it through the neural network to obtain the best grasp or push prediction to execute and this process is repeated until all objects were grasped.

The proposed approach to solve this challenging task is called **Visual Pushing for Grasping**. This method uses two Fully Convolutional Networks [B.2] composed of 121-layer DenseNets. ϕ_p takes as input 16 different images with 22.5 degrees of orientation between each other and will output 16 Q-value maps corresponding to locations where pushes might have the best effect, and ϕ_q receives 16 different rotations of the original image and will output 16 Q-value maps corresponding to locations where grasps might produce a good change in the environment [B.7]. According to the Q-value of a specific pixel, we know if a grasp motion is better than a push.

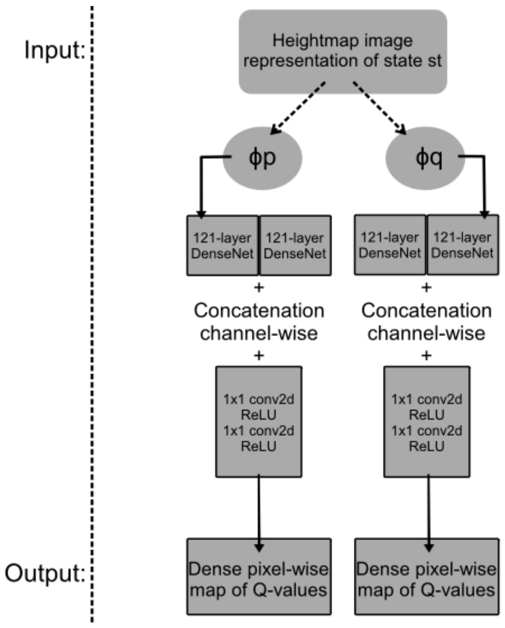

Input:
Heightmap image representation of state st
ϕp
ϕq
121-layer DenseNet
121-layer DenseNet
121-layer DenseNet
121-layer DenseNet
+
Concatenation channel-wise
+
Concatenation channel-wise
+
+
1x1 conv2d ReLU 1x1 conv2d ReLU
1x1 conv2d ReLU 1x1 conv2d ReLU
Output:
Dense pixel-wise map of Q-values
Dense pixel-wise map of Q-values

Figure 3.2: Simplified way to represent the layers of the FCN. Made with [50].

The model follows the next reward scheme to transit from one state to the next. If the grasping action is successful then $R_g(s_t, s_{t+1}) = 1$, otherwise the reward will be given according to the produced changes in the environment [B.7] $R_p(s_t, s_{t+1}) = 0.5$ and uses the following metrics:

1. Average % of **Completion** of test runs that finishes a task after ten attempts successfully, i.e., number of simulation cases that saw all the objects grasped after ten attempts;
2. Average % of **Grasp Success** per completion, i.e., Grasp Success per successful case;
3. % of **Action Efficiency** describing how good the policy is of finishing a task, in other words, is given by the number of objects yet to grasp divided by the sum of the number of attempts made before successfully grasp each object that isn't in the scene anymore.

In simulation experiments, the **Visual Pushing for Grasping** obtained better results when comparing to **Grasping-Only** method, achieving 67.7% of *Grasp Success*. Using challenging arrangements of objects, the results for real-world experiments managed to achieve 83.3% *Grasp Success* for the proposed method.

In sum, the Visual Pushing for Grasping proves to be a great model to endow robotic arms with the ability to grasp objects. However, this grasping ability fails in early steps, mainly because it is based

on performing a push to evaluate if a grasp is doable. It achieves very high performances compared to other variants of it.
Andy Zeng *et al.* stated that they only tried synergies like pushing and grasping and that it would be interesting to try some other types of synergies. We agree with them since seeing results for rolling or stacking objects would be interesting. Lastly, another point that would have been interesting to see results would be different shapes of objects, rather than only block-shaped objects.

3.4.2 Learning efficient push and grasp policy in a totebox from simulation

When discussing object manipulation in environments that are not structured, we stumble upon an open problem due to objects that might not be known. Attempts have been made to improve grasping performance in pushing, active vision, etc. However, these are built assuming that the objects are in an open platform, what not always happens.
In [38], the grasping ability is put to test in a different environment called totebox (Figure 3.3), what might have two drawbacks: first, the object could be at the edge and second, the object could be in a corner.

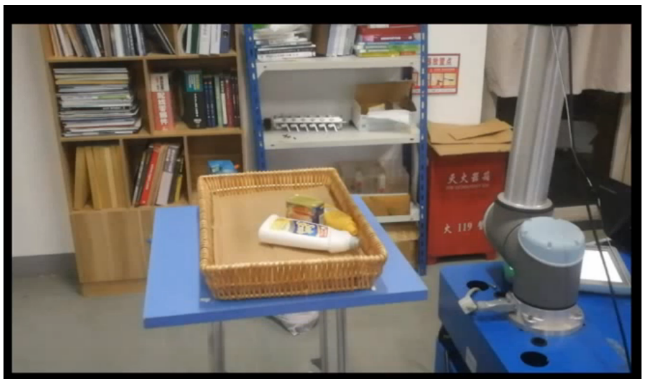


Figure 3.3: Totebox and robot setup, from [38].

Since the Pushing with Grasping approach is taken in account in this paper, it is normal that the next assumption is valid: *"The objects have rigid bodies and are randomly placed in a totebox. Some of them may be in a corner with no possible grasp pose and push pose. The direction of the approach for either grasping or pushing is perpendicular to the plane of the totebox."* - [38], page 4, section 2.3. Thus, the pushing motion should meet the next requirements:

1. Do not attempt pushes if robust grasps can be taken.
2. Pushes must increase availability of parts to be grasped.
3. Lastly, pushes should not damage the totebox nor make it move.

The proposed model in [38] is divided in two stages:

- Pre-Training - As reducing training cost is a priority, a Deep Q Learning [B.5] algorithm was used for this stage. The target of the pushing actions is to increase grasp access.
  The rewards for pushing actions that increase grasp access are given as follows:
  $R_{p,1} = R_{g,1} = \lambda_1 R_{suc} - \lambda_2 - \lambda_3 |S - S_0|$
  $R_{suc} \in \{0, 1\} \rightarrow$ gives the reward according to if action $u_{p/g}$ has been successful or not.
  S represents the area of inner boundaries of the totebox and S_0 is the area of the rectangle.

$|S - S_0|$ denotes punishments according to changes within the totebox.
λ_1, λ_2 and λ_3 are coefficients. The first two are equal to 2 and 1 respectively, to make sure $\lambda_1 R_s uc - \lambda_2$ stays between -1 and 1. $-\lambda_2$ represents the step cost and λ_3 represents a large punishment that will bestow upon the reward if the totebox suffers any movement;

- Combining Pushing with Grasping - The last stage involves training pushing with grasping. A reward will be attributed to the grasp's quality function Q_g based on how a push increased the access to a grasp.
  The reward method is changed to allow an improvement of grasp access and is given as follows:

$$R_{p,2} = R_{p,1} + \lambda_4 \sum_{\substack{p_g \in p'_g(o) \\ r_g \in r'_g(o)}} (\text{relu}(Q_{g,a}((p_g, r_g), s)) - \text{relu}(Q_{g,b}((p_g, r_g), s)))$$

Figure 3.4: To improve grasp access, the rewards method is given by $R_{p,2}$, from [38].

The activation function **ReLU** aims to ignore negative values. $p_g^{'}(o)$ and $r_g^{'}(o)$ denotes all possible positions and rotations for an observation o.
The designed policy $\pi(s)$ is defined as follows:

$$\pi(s) = \begin{cases} \text{argmax } Q_g & \text{if } \max(Q_g) > Q_g^* \\ \text{argmax } Q_p & \text{if } \max(Q_p) > Q_p^* \\ & \text{and } \max(Q_g) < Q_g^* \\ \text{argmax}\{Q_p, Q_g\} & \text{otherwise} \end{cases}$$

Figure 3.5: Designed policy $\pi(s)$ of proposed algorithm, from [38].

The threshold Q^*_g is accepted as threshold if at least a grasp is tried and might get the best success rate to pick up an object, making sure the grasp is robust and safe. The threshold value will be raised from 0 gradually, until (true positive/(true positive + false positive)) reaches 99%, when it will be fixed. We can see *true positives* as successful grasps and *false positives* as failed grasps. The threshold Q^*_p is obtained in a similar way.

And is evaluated using four metrics:

1. Average % completion $\frac{\text{\#objects that be cleared}}{\text{\#objects in test}}$.
2. Average % grasp success rate $\frac{\text{\#objects that be cleared}}{\text{\#grasping attempts}}$.
3. Average % action efficiency for cleared objects $\frac{\text{\#objects that be cleared}}{\text{\#action attempts for objects that have been cleared}}$.
4. Average % collision rate $\frac{\text{\#collision with the totebox}}{\text{\#action attempts}}$.

The experiments were conducted both in simulation and in the real world.
In simulation, the results show that the proposed method can improve action efficiency from 4% to 5% and the completion rate is not up to 100% since there are some objects that can hardly be picked.
In real world experiments, the results show that the proposed algorithm can adapt to real environment with a completion rate of 83.37% in novel objects.

Concluding, the proposed a method [38] combines pushing with grasping. This method applies a policy to help decide if a pushing action is actually a good contribution for the success of a grasping action. The designed algorithm follows the same logic described in [16] for feature extraction from the heightmaps. This method outperforms some of the famous works in the field with 83.37% of completion both in simulation and real world, also for unseen objects.

3.4.3 Deep Reinforcement Learning for Robotic Pushing and Picking in Cluttered Environment

Autonomous robots have been gradually occupying the logistic sector due to the rapid development of e-commerce. Another example is the use of mobile robots for product transportation in warehouses. Over the years, many designs of robotic hands have been studied. The robotic hand with a suction cup is trendy for robotic grasping tasks due to its simple structure.
In [17], a novel approach for robotic grasping in cluttered environments is proposed. The robotic hand, composed of a suction cup and a gripper, must grasp objects even in cluttered environments. This approach contains a deep Q-Network [B.5] to compute the proper grasping points of the objects.

Developing grasping systems for robotic hands is a challenging task, but even more if the environment where the objects are inserted is cluttered. Thus, a new design for the robotic hand must be approached and a new system to tackle cluttered environments must be developed.

The proposed system has the following pipeline, also shown in figure 3.6:

1. Obtain RGB and Depth images;
2. Compute affordance map using affordance ConvNet;
3. Evaluate the quality of computed map using metric ϕ.
    - If ϕ > threshold $\rightarrow$ grasp object accordingly;
    - Otherwise, feed affordance map to deep Q-Network [B.5] to compute disturbances (pushes) using Active Affordance strategy.

This pipeline is repeated until all objects are cleared from the environment.

Active Exploration generates better results both in Suction and Test Success Rates, allowing the probability of repeating lifts that had failed to be minimized, when compared to when a Static Affordance Map is used.

The proposed model is evaluated using the following metrics:

- Average number of operations per test;
- Average increment of metric ϕ per push;
- Test success rate - number of successful tests divided by total number of tests.

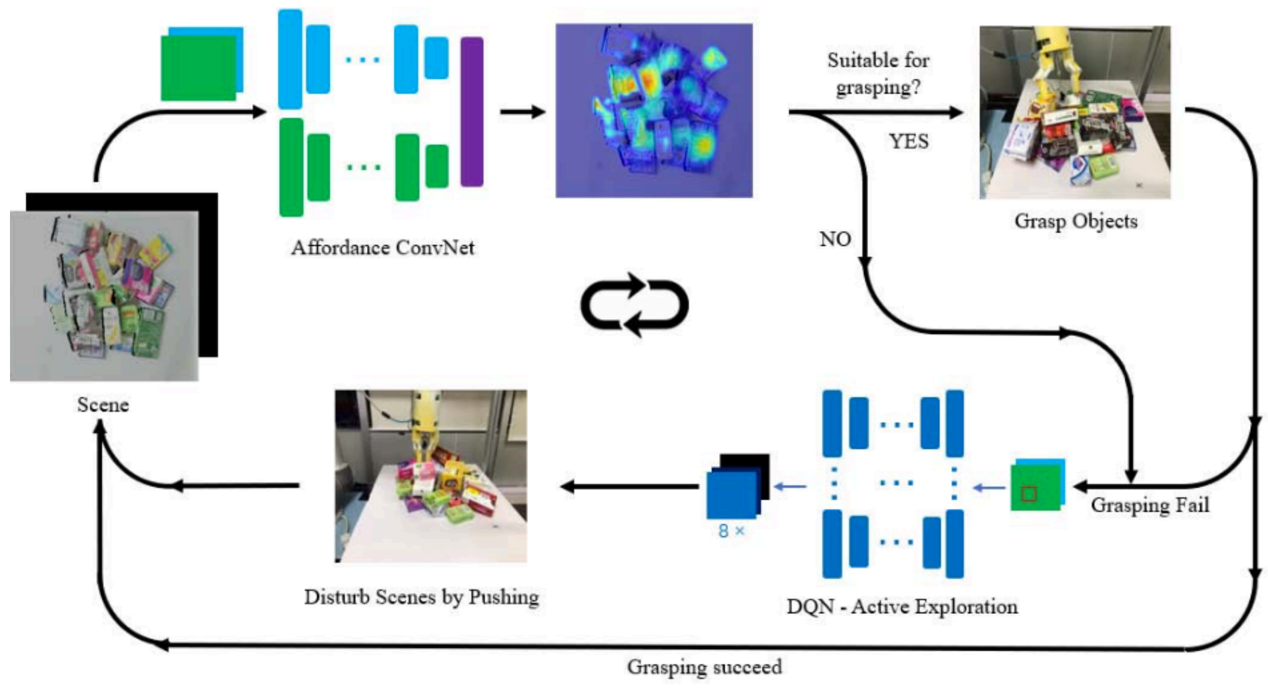
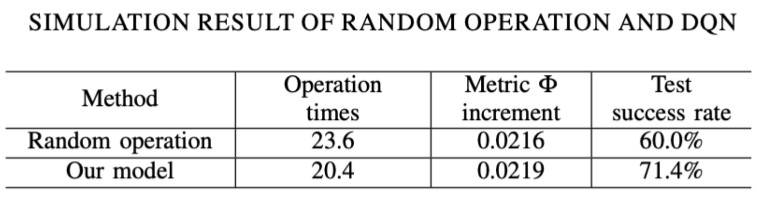


Figure 3.6: Deep Q Network pipeline, from [17].

Simulation experiments of this research were made under CoppeliaSim using an UR5 robotic arm. The proposed architecture achieved a success rate of 71.4%, as shown in figure 3.7. The real world results are given in two rates: suction success rate: - Number of objects successfully grasped divided by number of lift operations and test success rate - Number of successful tests divided by the total number of realized tests, both seem better for *Active Affordance Map* - a map computed from the RGB-D image, showing where good grasp are located.


SIMULATION RESULT OF RANDOM OPERATION AND DQN

Method	Operation times	Metric Φ increment	Test success rate
Random operation	23.6	0.0216	60.0%
Our model	20.4	0.0219	71.4%

Figure 3.7: Simulation results, from [17].

In conclusion, this novel approach for grasping objects combines a suction cup with a two-finger gripper. Using Deep Q-Network [B.5], similar to a U-Net [B.6], the method can know if the current state of the environment needs a change (push motion at a designed location) or if a grasp is feasible. However, it still has some flaws, such as making pushes that do not produce a change (in places without objects) and not recognizing all types of objects.

3.5 Summary of the Studied Works

In this section, we made a quick and straightforward review of the studied models and refer those we think maybe the best approaches to tackle our main problem.

Paper	Year	Main Focus	Robot used	# Objects	Dataset	Simulation	Real world	Results
[20]	2017	Pushing	7-DoF	10 to 20	Own (Images)	✗	✓	2.52±1.06mm MDFGPP
[72]	2016	Pushing	7-DoF	N/A	Own (Images)	✗	✗	< 0.97 SSIM
[53]	2018	Grasping	UR5	22	YCB	✓	✓	90% Completion Rate
[51]	2020	Grasping	UR10	79	YCB	✓	✓	91.6% Completion Rate
[38]	2020	Pushing w/ Grasping	UR10	79	YCB	✓	✓	83.37% Completion Rate
[75]	2018	Pushing w/ Grasping	UR5	9	N/A	✓	✓	67.7% Grasp Success
[18]	2019	Pushing w/ Grasping	UR5	40	N/A	✓	✓	71.4% Suction Success

Table 3.1: Table containing the summary of all studied works.

As present in the above table, the studied papers have different focuses. They are divided into Pushing, Grasping and Pushing with Grasping.
These three categories are:

1. Pushing: Approaches based on frame prediction given raw video footage, to predict the movement of an object from location x to location y, just by making pushing action;
2. Grasping: Models for grasping prediction according to the orientation and position of the gripper and the distance between the gripper and the object;
3. Pushing with Grasping: This last category is the category embraced by this thesis's primary focus. Since this area's approaches are oriented to models that combine pushing with grasping, this category may be the most researched for robotic grasping.

We can say that Visual Pushing-for-Grasping may be the best approach to follow in our specific case because:

- Only moving objects from one place to another, isn't the focus of this research. However, the approaches proposed by Action-Conditioned Benchmark [20] based on the paper [19], and the Unsupervised Learning Prediction Video [72] based on the paper [71] are representative of good studies on the Pushing category since predicting frames is not always the followed path in this field;
- Only computing grasping actions for the robot to execute may lead to some new problems. For example, when using a box as the environment for the objects, we may stumble upon two cases:
    1. The objects selected to grasp may be at a corner of the box;
    2. Or the objects may be at one of the edges of the box.

  These two cases make grasping nearly impossible without trying to get the objects in a safe grasping position.
  However, the research made by PointNetGPD [53] based on the paper [54] and PointNet++ [51] based on the paper [52], focus the issues related with the Grasping category. These issues are related with the investigation of the orientation and position of the objects and the distance between the gripper and the object to be grasped;
- We think that combining pushing and grasping may be the optimal approach to follow. As referred previously, the objects might be in unreachable positions or even may be obstructing the path to other objects that we want to grasp.
  These are more than enough reasons to want to approach a model that decides whether making a push or a grasp is the best action for a specific situation. Although the approaches proposed by Learning Efficient Push and Grasp [38] and Active Perception [18] are good options to tackle the problems arisen by the Grasping matter, they may not be 100% ready for the robotic arm used in this thesis work.

3.6 Conclusion

In this chapter, were introduced some methodologies used over the last years on this field. Lastly, we presented our opinion on these papers and why we have chosen Visual Pushing-for-Grasping as the primary approach.

Chapter 4

Proposed Work

4.1 Introduction

In this chapter we will explore more in depth the VPG algorithm presented in section 3.4.1. More specifically, we will approach the following points:

1. First of all, we will explain the concept behind this algorithm;
2. Then, we will show a simple architecture that resembles the behavior of the VPG algorithm;
3. Some important critical decisions made by this algorithm, that influence which action the robot takes;
4. Soon after, we talk about the adopted evaluation metrics and show a fictional example;
5. Then, we discuss why we won't follow their metrics and scrutinize the ones we think that most suit this kind of work;
6. Finally, we introduce the changes we made to this algorithm that are related to the main purpose of the thesis, with the objective of making the process of grasping objects much faster and hopefully more accurate and present the structure of the approach.

4.2 Visual Pushing-for-Grasping

The proposed work is mainly based on the approach and research made by *Andy Zeng et al.* [16] and available at Github [75], called VPG and is an algorithm designed to help a robotic arm perform two types of action, push or grasp, in an environment full of objects, with the objective of grasping all of them.

4.2.1 Model's Architecture

The model chosen is called VPG. Its streamline in Figure 4.1, can be divided in the following parts:

1. Obtain a valid depth heightmap of the current scene by taking two types of images: an RGB and a depth image;
2. Process the valid depth heightmap through the algorithm's neural networks to obtain configurations for grasp and push;
3. Execute an action according to the obtained configurations and approach a critical decision;
4. Get new images to repeat the whole process again, until all the objects have been grasped.

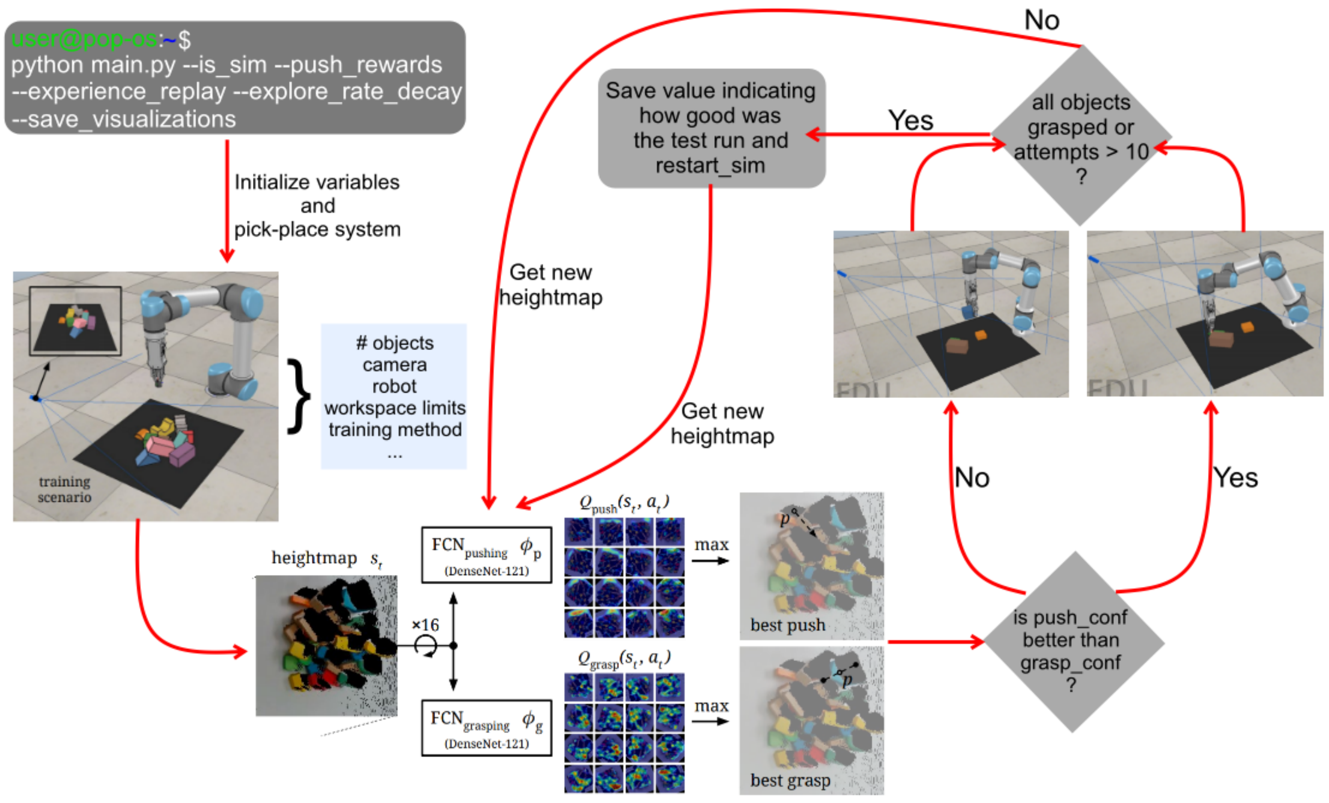

user@pop-os:~$
python main.py --is_sim --push_rewards
--experience_replay --explore_rate_decay
--save_visualizations
Initialize variables and pick-place system
training scenario
# objects
camera
robot
workspace limits
training method
...
Get new heightmap
Save value indicating how good was the test run and restart_sim
Yes
No
all objects grasped or attempts > 10 ?
Get new heightmap
heightmap s_t
×16
FCN pushing ϕ_p (DenseNet-121)
FCN grasping ϕ_g (DenseNet-121)
$Q_{push}(s_t, a_t)$
$Q_{grasp}(s_t, a_t)$
max
max
best push
best grasp
No
Yes
is push_conf better than grasp_conf ?

Figure 4.1: VPG streamline process, from [16].

4.2.2 Obtaining a valid depth heightmap

After initializing the pick-place system, i.e. the robot and the camera, the model needs to obtain a valid depth heightmap [B.8], so it can generate the best configurations for both push and grasp actions. This process is rather simple to explain:

1. First, we obtain both color and depth images of the current state of the environment;
2. Then, we proceed to transform or convert these images into heightmaps;
3. And finally, we combine these two recently obtained heightmaps to get the valid depth heightmap that represents the current state of the environment.

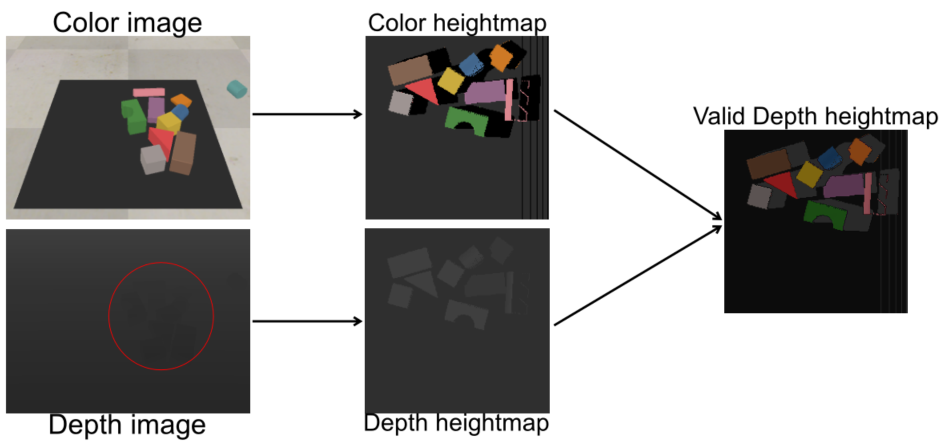


Figure 4.2: Obtaining the valid depth heightmap, made with [50].

The process of obtaining the color and depth heightmaps follows the next steps:

1. Obtain Color and Depth Images;
2. Get a Point Cloud;
3. Get Color and Depth Heightmaps.

4.2.3 Obtain Color and Depth Images

The capture and consecutive processing of the RGB image follows these steps:

1. Capture the raw color image and respective resolution from the Vision Sensor camera;
2. Define the shape of the color image through the obtained resolution values ”color_img.shape = (resolution[1], resolution[0], 3)”, i.e. the shape of the image is defined as (width, height, color channels);
3. Divide the values contained in the array of the image by 255 and apply a correction (if value x < 0 then x += 1) to ensure that those values stay within the range of 0 and 1;
4. Multiply the color image array by 255, thus all the values are converted to rgb range of 0 to 255;
5. Flip the image from left to right, so the image is visualized as if it was taken by the robot instead of the camera;
6. Convert the color image array to unsigned integer 8, since to save the image it is the only supported format;
7. Save the image.

Figure 4.3 represents the color images taken by VPG algorithm.

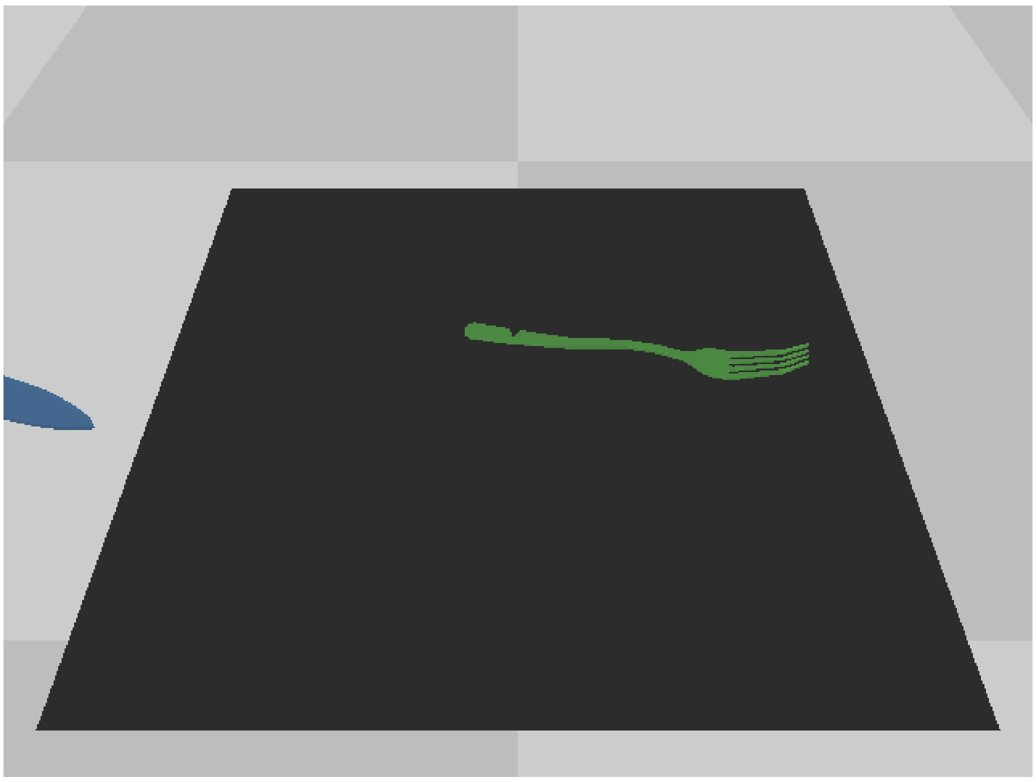


Figure 4.3: Color image taken by Vision Sensor camera in simulation.

The capture and consecutive processing of the depth image follows these steps:

1. Capture the buffer depth array and respective resolution from the Vision Sensor camera;

2. Define the shape of the depth image through the obtained resolution values ”depth_img.shape = (resolution[1], resolution[0])”, i.e. the shape of the image is defined as (width, height) since the depth image only has 1 channel not 3 as the Red-Green-Blue image;
3. Flip the image from left to right, so the image is visualized as if it was taken by the robot instead of the camera;
4. Apply a correction to the depth values, so they are framed inside the farther (10) and nearest (0.01) distance of the camera;
5. The depth image suffers a 10000 multiplication, thus all the values are in millimeters instead of meters, before being saved;
6. Save the image.

Figure 4.4 represents the depth images taken by VPG algorithm. Within a dashed rectangle is the same object present in the color image, just for better understanding of this kind of images.

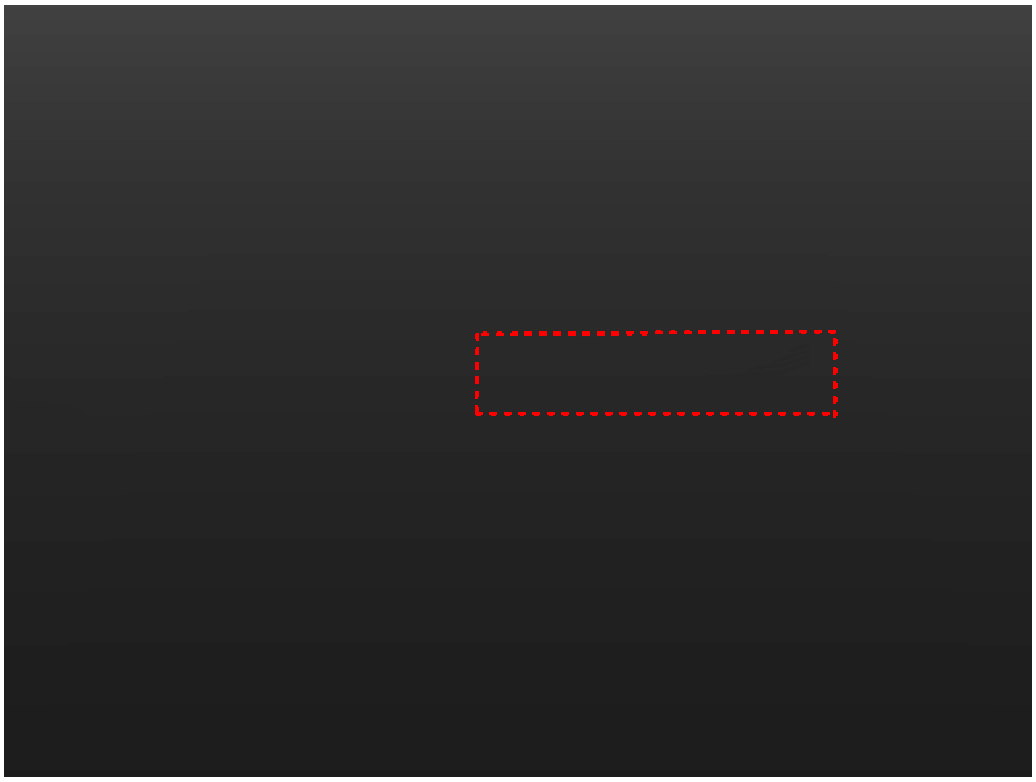


Figure 4.4: Depth image taken by Vision Sensor camera in simulation.

4.2.4 Get a Point Cloud

After obtaining the RGB and depth images, the next step is to obtain a point cloud to more easily create the heightmaps afterwards.
By projecting the depth image into a 3D point cloud, it is possible to compute the final point cloud.

1. Retrieve depth image's resolution (width, height);
2. Compute x, y and z camera points based on the intrinsic values of the camera. These intrinsic values represent the optical center and the focal length of the camera and they allow mapping camera coordinates to pixel coordinates in an image and vice-versa;
3. Shape each array of x, y and z coordinates to have height*width lines by 1 column;

4. Concatenate the arrays to have the complete point cloud.

Then, we just need to reshape the color image to be able to represent the colors in the point cloud.

1. Fill each array corresponding to the values of Red (0's), Green (1's) and Blue (2's) from the color image;
2. Shape each array of Red, Green and Blue values to have height*width lines by 1 column;
3. Concatenate the arrays to have the complete set of colors for each point of the point cloud.

4.2.5 Get Color and Depth Heightmaps

Having computed the point cloud and the color arrays containing the color values of each point in the point cloud, we can obtain the color and depth heightmaps.

1. The points are sorted by z coordinate in both point cloud and color array;
2. Filter the points such that all points outside the robot workspace limits are discarded;
3. Last, the top-down heightmaps are created using the point cloud and color arrays. By top-down heightmaps, we mean that these heightmaps represent the scenario seen from above.

4.2.6 Neural networks and their process

The model is composed of two identical neural networks called DenseNet-121. These two networks are divided in ϕ_p and ϕ_g and they evaluate 16 different images from the inputted depth heightmap [B.8] of the current state of the environment, for pushes and grasps respectively.

In the end, the model has two best predictions, one corresponding to the best push configuration and the other to the best grasp configuration.

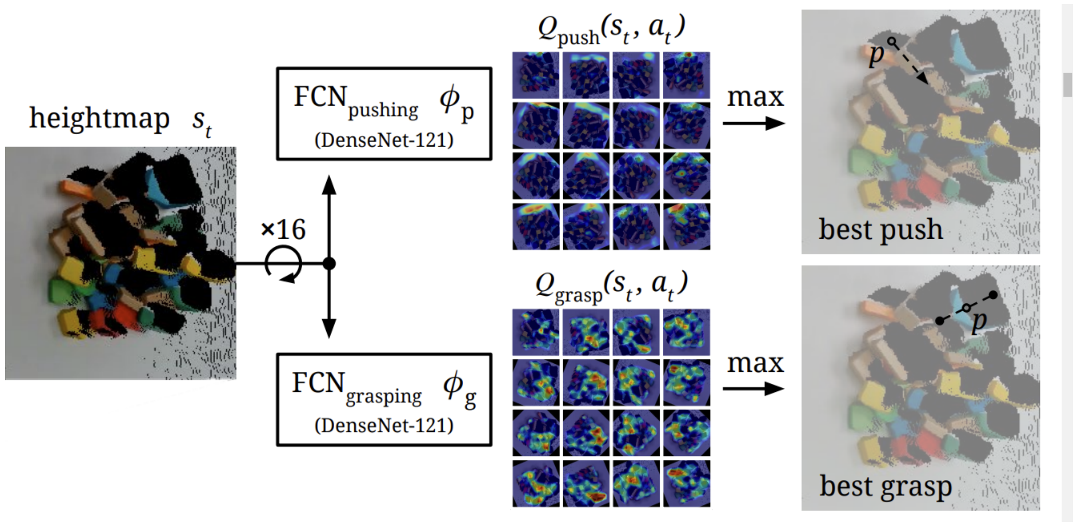

heightmap s_t
×16
FCN$_{\text{pushing}}$ ϕ_p
(DenseNet-121)
FCN$_{\text{grasping}}$ ϕ_g
(DenseNet-121)
$Q_{\text{push}}(s_t, a_t)$
$Q_{\text{grasp}}(s_t, a_t)$
max
max
p
best push
p
best grasp

Figure 4.5: Essential architecture model of VPG, from [16].

4.2.7 VPG critical decisions

The model also has two other important decisions embedded in it.

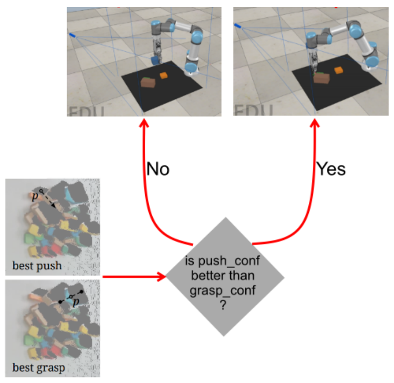
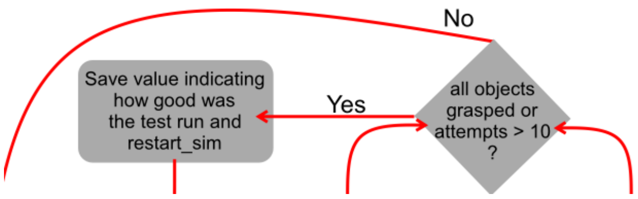


Figure 4.6: First critical decision of the VPG, made with [50].


No
Save value indicating how good was the test run and restart_sim
Yes
all objects grasped or attempts > 10 ?
Figure 4.7: Second critical decision of the VPG, made with [50].

The first critical decision (on the left) has to do with the decision of either executing a push or a grasp action. Basically, the model will just compare if the recently obtained best push configuration is greater than the best grasp configuration and if so, the push action is executed. Otherwise, the grasp action is executed.

The second and final critical decision (on the right) has to do with the decision of whether saving the number of the iteration that satisfies the condition ”Table empty or Maximum possible attempts that the model can fail consecutively when making an action greater than 10?” and restart simulation or continue without restarting. Either way, the next step involves obtaining a new depth heightmap of the environment’s current state.

4.2.8 Used Metrics

To evaluate the performance of the model, the authors proposed four metrics. **Average % of Clearance**, **Average % of Grasp Success per Clearance**, **Average % of Action Efficiency** and **Average Grasp to Push ratio**.
There are other parameters involved in these metrics that need explaining.

1. Clearance Log: This parameter will contain the numbers of all iterations that meet the following: ”Table empty or Maximum possible attempts that the model can fail consecutively when making an action greater than 10?”;
2. Maximum Trials: Is initially set to 30 trials, but in evaluation context, it assumes the size of the *Clearance Log*, i.e., the number of rows that this log file contains;
3. Number of Grasp Success: Essentially represents the number of grasps that had a reward greater than 0.5 in the trial currently under evaluation and also, the number of pushes that were successful in changing the position/orientation of any of the objects;
4. Valid Clearance: It is an array of booleans, i.e., a list containing true and false values. Each boolean indicates if the correspondent value in the list of *Number of Grasp Success* is greater than 10;
5. Number of Objects Complete: Represents the number of objects we want to use to evaluate the results from simulation or real-world experiments. Usually, it is the same as the number of objects used in those experiments.

Therefore, we can introduce the Average % of Clearance as the % of trials with more than 10 successful grasps:

$$\text{Average \% Clearance} = \frac{\text{\# Valid Clearance == True}}{\text{Max Trials}} * 100$$

The Average % of Grasp Success per Clearance metric involves two parameters that need explaining. They are called **Grasp Success Rate** and **Total Grasp Attempts**.

- Grasp Success Rate: This parameter is a list containing the success rates of each trial. The grasp success rate of a trial is given by the *Number of Grasp Success* of that trial divided by *Total Grasp Attempts* made in that trial;
- Total Grasp Attempts: Corresponds to the number of grasps made in a trial, both successful and non-successful grasps.

Hence, the Average % of Grasp Success per Clearance is the average of grasp success rate of the trials that had more than 10 grasps successful and is given by:

Average % Grasp Success per Clearance = avg($\sum$ *Grasp Success Rate[Valid Clearance == True]*) * 100

To understand the Average % of Action Efficiency metric, we first need to understand the **Number of Actions before Completion**.

The *Number of Actions before Completion* contains the number of actions made between trials that reached completion and is given by: Number of Actions before Completion = *Clearance Log*[1:Max Trials] - *Clearance Log*[0:Max Trials-1]

Therefore, the Average % of Action Efficiency is the average of each value from *Number of Actions before Completion* that managed to grasp more than 10 objects:

$$\text{Average \% Action Efficiency} = \sum \frac{\textit{Number of Objects Complete}}{\textit{Number of Actions before Completion}[\textit{Valid Clearance}]} * 100$$

The last metric is the Average Grasp to Push Ratio and involves knowing what the **Grasp to Push Ratio** is and how we obtain it for each trial.

Therefore, the *Grasp to Push Ratio* of 1 trial is given by the *Total Grasp Attempts* made in that trial divided by the *Number of Actions before Completion* of that trial.

That said, the Average Grasp to Push Ratio is the average *Grasp to Push Ratio* of all trials that had more than 10 successful grasps and is given by:

Average Grasp to Push Ratio = avg($\sum$ *Grasp to Push Ratio[Valid Clearance == True]*) * 100

4.2.9 Fictional Example

The following is not a truthful example, is just to express how the previously explained metrics are related and how to compute each one of them.

Let's say that our model achieved clearance at the iterations 8, 35, 79, 115 and 205. Our *Clearance Log* = [8 35 79 115 205].

The next step is to concatenate the *Clearance Log* with 0, so computing the *Number of Actions before Completion* will follow the formula explained previously and to initialize all the other parameters we will need.

Clearance Log = concatenate(0, *Clearance Log*) = [0 8 35 79 115 205]

Number of Object Complete = 10, because we want to evaluate for 10 objects.

Let's say that the *Number of Grasp Success* is given as follows: *Number of Grasp Success* = [3 5 20

13 9]. Thus, the *Valid Clearance* for this result is *Valid Clearance* = [F F T T F].
Therefore, the *Average % of Clearance* = 2True/size(*Clearance Log*) * 100 = 40%

If we have a *Total Grasp Attempts* in each trial given by: *Total Grasp Attempts* = [6 20 32 17 21].
With this information, we can calculate the *Grasp Success Rate* for each trial following the formula *Number of Grasp Success/Total Grasp Attempts*. *Grasp Success Rate* = [0.5 0.25 0.625 0.765 0.429].
After obtaining the success rates for the grasps, we can compute the *Average % Grasp Success Rate per Clearance* = ((0.625+0.765)/2)*100 = 69.50%.
To obtain the *Average % of Action Efficiency*, we need to compute the *Number of Actions before Completion.*
Number of Actions before Completion = *Clearange Log*[1:*Max Trials*] - *Clearance Log*[0:*Max Trials*-1] = [8 27 44 36 90]. So, the Action Efficiency is given as follows:

$$\text{Average \% of Action Efficiency} = (\frac{\frac{10}{44}+\frac{10}{36}}{2}) * 100 = 25.25\%$$

What's left is to obtain the last metric. For that we first need to compute every *Grasp to Push Ratio*.
Grasp to Push Ratio = [6/8 20/27 32/44 17/36 21/90] = [0.75 0.741 0.727 0.472 0.233].
Average Grasp to Push Ratio = ((0.727+0.472)/2)*100 = 59.95%

4.2.10 Our Adopted Metrics

In our case, we thought that some of these metrics don't make sense since they are evaluating if the table is empty, meaning if all the objects present in the table were grasped, or if the robot failed to complete a task of grasping an object during ten consecutive attempts. In our vision, either we evaluate for something positive or something negative, not both at the same time for the same metric.

Given that there are some incongruities with the first used metric **Avg % of Clearance** by the authors of VPG [75] and thus with the second metric **Avg % of Grasp Success per Clearance**, related to the fact that Clearance counts as both something good (table empty, meaning all objects grasped) and bad (the case where the model can fail at least 10 consecutive actions). We propose three simple and easily understandable metrics:

1. Initially, we were going to propose the usage of a metric called Action Efficiency, which could be described as the sum of all actions needed (grasps and pushes) to successfully grasp all the objects during one training case: given by "number objects/(grasps + pushes)";
   - By training case, we mean the set of iterations since the drop off of the objects in the simulation until the last iteration where the final object was grasped. After this last iteration, the simulation will restart the simulation and a new training case will start.

   However, this metric wasn't put to use since making pushes that could be helping the action of making a successful grasp and taking those pushes into account could end up in penalizing the actions of the robot;
2. Therefore, the first metric proposed and in use is called Average Grasp to All Ratio, and is described as the number of successful grasps made in the universe of all grasps. The second metric proposed is called Average Push to All Ratio, and is similar to the first one but related to pushing motions;

Accompanied with these two metrics we also display the number of all made pushes and grasps;

3. In terms of knowing how accurate the algorithm was while sorting the objects by bins, we propose a metric that uses the following pipeline:
   (a) When there aren't more objects in the current training case or the simulation reached the maximum number of attempts to grasp the objects, we use a second vision sensor to take images from the place where the objects are dropped after being grasped;
   (b) Then, we apply the YoLo model to the RGB images;
   (c) The final part of this evaluation has to be done by us, since the Object Recognition model can fail to identify correctly the objects present in the images. Therefore, knowing the number of objects that were in simulation we count how many were grasped and how many were placed in the correct spot.

   Therefore, there are three ways of evaluating the accuracy of the model given by the following formulas:
   (a) Evaluate the whole grasp experience by getting the percentage of successful grasps in the entire grasp universe, which in turn is basically the Grasp to All Ratio evaluated on the final iteration of the simulation, and is given by:

   $$\frac{\text{total \# successful grasps}}{\text{\# grasps made (successes or not)}} \quad (1);$$

   (b) Evaluate the number of successful and well placed grasps in the whole universe of all made grasps, given by the formula:

   $$\frac{\text{\# successful and well placed grasps}}{\text{total \# successful grasps}} \quad (2);$$

   (c) Evaluate the amount of successful and well placed grasps in the whole universe of all made grasps, given by the formula:

   $$\frac{\text{\# successful and well placed grasps}}{\text{\# grasps made (successes or not}} \quad (3).$$

   Note that the metrics (2) and (3) refer to " successful and well placed grasps", because inside the category of successful grasps, we have two subcategories: those that were grasped successfully, but not placed in the correct container and those that were grasped successfully and placed in the correct container.

4.3 Object Detection

Since the purpose of the thesis was to grab an already developed and functioning software and apply changes to it in order to be able to grasp and sort cutlery objects, we thought of merging an Object Recognition model with the VPG algorithm, to help recognize the objects present in the scene during each epoch and therefore, speeding up the process of grasping the objects while sorting them by category.

The Object Recognition model we chose to adopt was the YoLo version 5, available at [82], because it is already prepared to recognize cutlery objects and promises high accuracy when evaluating each object in an image.

4.4 Structure of the Approach

The structure of this thesis and its approach is divided into two parts. The first part is related to the simulation environment, and the second part is associated with the real-world experiments.

4.4.1 Simulation Environment

The first part of the approach involves a simulation environment.
Besides wanting to secure that the model is complete and working correctly before testing with a real robot, we want to avoid problems in the real world. These problems are related to either someone getting hurt or to breaking the robot. Thus, testing in simulation reassures us that real-world experiments may run without problems.
CoppeliaSim [59] software will handle the simulation environment, where the VPG model [75] will be put to the test alongside with a UR3 [73], to see if we can obtain results closer or even better than those from the state of the art of the paper in [16].

4.4.2 Real-World Environment

The second and last part of this approach involves real-world experiments. These real-world experiments imply that the simulation phase went well and that we still have time to do them.
To make them we will use the developed approach [75], ROS [60] and the real UR3 [73]

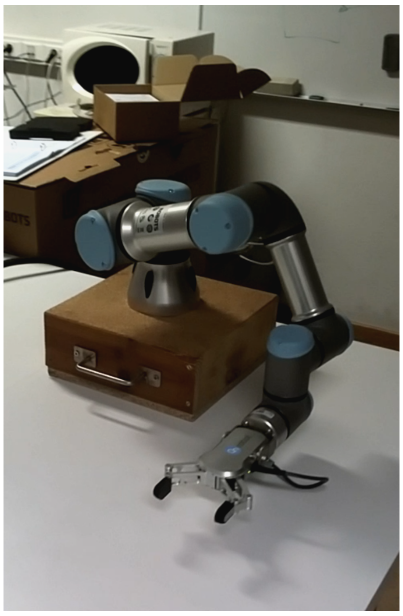


Figure 4.8: UR3 from SOCIA Laboratory.

Chapter 5

Proposed Changes to Visual Pushing-for-Grasping

5.1 Introduction

In this chapter, we will approach our primary purpose for this thesis, which is all the changes applied onto the original algorithm of VPG to make it faster in the field of object detection, thus making the process of grasping and sorting objects much faster than before.

5.2 Recap of the Visual Pushing-for-Grasping Algorithm

The baseline algorithm we are using for our work is called VPG [75]. Its concept revolves around the idea of helping a robotic arm taking actions over objects. These actions may be pushes, thus helping create spaces between clusters of objects, or grasps, thus helping clean up areas with many objects. The notion "Making a push may help on the success of a future grasp" is built-in to this algorithm hence the system designed to attribute rewards according to each action.

1. A push would receive a reward of 0.5 if it had caused a significant change in the environment. By significant change, we mean that the position of the objects from one iteration/epoch to another has changed;
2. A grasp receives a reward of 1.0 if the action of grasping an object with success has been made.

As explained in section 4.2.2, at the beginning of each epoch, the algorithm captures an RGB and a depth image. These two images will eventually be converted to heightmaps, i.e., images containing depth values, which in turn will be combined into one depth heightmap.
The algorithm can compute possible push and grasp configurations from the last heightmap and choose the best configuration to apply to the current scenario.

5.3 From Block-Shaped Objects to Cutlery Objects

This section approaches the changes made to use VPG with cutlery objects. These cutlery objects were retrieved from YCB dataset [79] and they contain forks, knives, and spoons.

Initially, the VPG was running under simulation with the out-of-box objects. These objects were block-shaped, that may vary from pyramids to parallelepipeds.

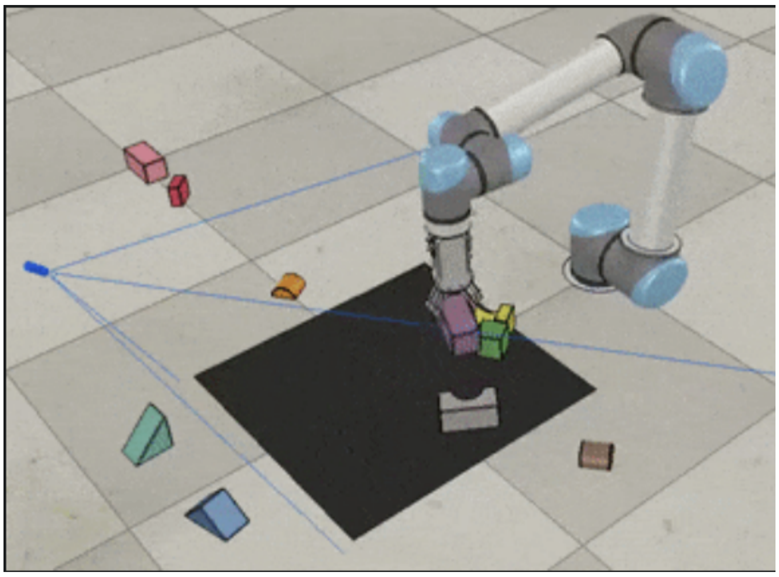
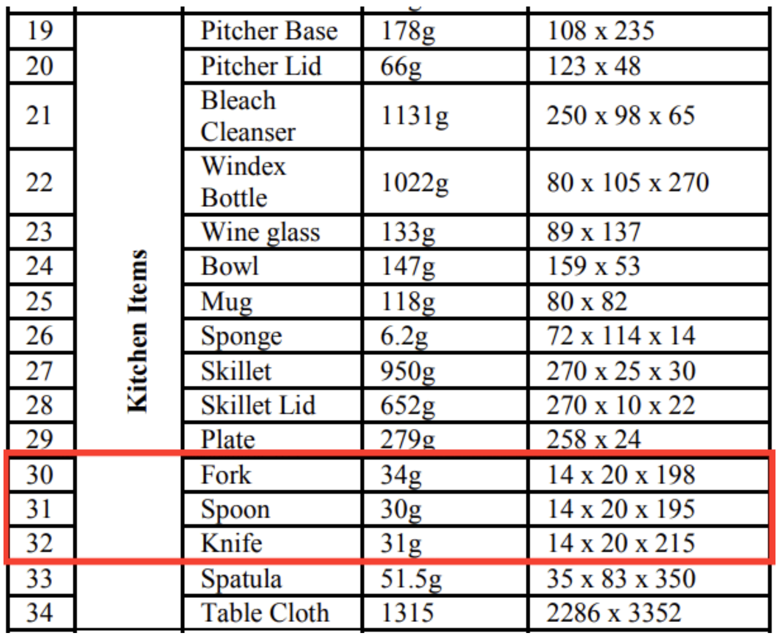


Figure 5.1: The VPG algorithm [75] came with block-shaped objects out-of-box.

Since we want to work on the UR3 with cutlery objects, we needed to adapt its simulation to cutlery objects.
After some careful search, we found out that the best dataset containing cutlery objects for this kind of work was the YCB dataset. This dataset is very popular in Artificial Intelligence and has various objects for various activities. Figure 5.2 contains all the objects covered by this dataset.



19	Kitchen Items	Pitcher Base	178g	108 x 235
20		Pitcher Lid	66g	123 x 48
21		Bleach Cleanser	1131g	250 x 98 x 65
22		Windex Bottle	1022g	80 x 105 x 270
23		Wine glass	133g	89 x 137
24		Bowl	147g	159 x 53
25		Mug	118g	80 x 82
26		Sponge	6.2g	72 x 114 x 14
27		Skillet	950g	270 x 25 x 30
28		Skillet Lid	652g	270 x 10 x 22
29		Plate	279g	258 x 24
30		Fork	34g	14 x 20 x 198
31		Spoon	30g	14 x 20 x 195
32		Knife	31g	14 x 20 x 215
33		Spatula	51.5g	35 x 83 x 350
34		Table Cloth	1315	2286 x 3352

Figure 5.2: The YCB dataset contains some kitchen objects. From those present in the image, the ones used were, the fork, knife and spoon. All of them 16k laser scan objects, from [80].

Later on, we also tried other 3D models of cutlery objects retrieved from TurboSquid [68], figure **??**.

The main goal of trying out 3D models of cutlery objects like those on figure **??** was because the models from YCB dataset had some deformations and therefore did not resemble so much the real-world

cutlery objects.
The models from TurboSquid [68] do not have visible deformations and are free to download from the website.

5.4 Object Detection Model

Previously, on section 5.1 we said that we wanted to apply some changes to the original VPG algorithm to distinguish the different objects in the scene and be able to separate them by type. For that, we thought of introducing an Object Detection model called YoLo.
Therefore, in the next subsections, we will talk about the changes made to this module in order to support images from the simulation of VPG, how it was incorporated with VPG and what information it does provide us.

5.4.1 Fine Tuning YoLov5

Upon testing YoLo with some of the images from simulation, example figure 6.6, we noticed that the model wasn't getting any matches for the objects present in the image. This problem was related to the fact that the quality of the images used to train YoLo from COCO dataset was far superior to those obtained from simulation and therefore implying a fine-tuning of this model.

Furthermore, we trained and tested all four versions of the YoLo model to make a comparison. We gathered a dataset composed of 300 images from the simulation, divided into two parts: train - composed of 80% of the total images from the simulation and test/val - composed of the remaining 20% of the total dataset.
YoLo makes a fine-tuning stage in two steps:

1. First, the Object Detection model undergoes a training step to train the model with the given images for the desired images. At the end of this step, the recently trained model is saved, and the last step starts immediately;
2. The last step of YoLo is the test or validation component where the recently trained model infers over the images of the validation dataset. The final results come from this stage of fine-tuning.

Below are displayed figures containing the obtained results for 3 classes (fork, knife and spoon), that the last stage of the fine-tuning outputs.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.98	0.968	0.988	0.701
Fork	60	58	0.982	0.931	0.976	0.704
Knife	60	111	0.99	0.991	0.994	0.669
Spoon	60	59	0.967	0.98	0.994	0.729
Total time:	0.496h = 29min45sec				GPU Mem:	0.841GB
Epochs:	300				Image size:	640

Table 5.1: Obtained results for YoLov5s under our custom dataset of images from simulation.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.967	0.985	0.988	0.689
Fork	60	58	0.944	0.966	0.974	0.704
Knife	60	111	0.991	0.99	0.995	0.656
Spoon	60	59	0.967	1	0.995	0.708
Total time:	0.911h = 54min39sec				GPU Mem:	6.38GB
Epochs:	300				Image size:	640

Table 5.2: Obtained results for YoLov5m under our custom dataset of images from simulation.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.984	0.986	0.991	0.688
Fork	60	58	0.982	0.966	0.982	0.699
Knife	60	111	0.989	0.991	0.995	0.676
Spoon	60	59	0.982	1	0.995	0.69
Total time:	1.454h = 1h27min14sec				GPU Mem:	5.06GB
Epochs:	300				Image size:	640

Table 5.3: Obtained results for YoLov5l under our custom dataset of images from simulation.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.977	0.977	0.989	0.684
Fork	60	58	0.982	0.983	0.988	0.695
Knife	60	111	0.982	0.982	0.994	0.686
Spoon	60	59	0.968	0.966	0.993	0.671
Total time:	2.563h = 2h33min46sec				GPU Mem:	7.77GB
Epochs:	300				Image size:	640

Table 5.4: Obtained results for YoLov5x under our custom dataset of images from simulation.

With the above training results, we can say that we have chosen YoLov5x version to work with, because in theory it has the best results, although in practice they are very close to the values obtained with the other versions of YoLov5.
However, we also made inference tests with each version of YoLov5, but using TTA, which is a method that increases the size of the used images to obtain better results, and using the same images of the test/validation dataset to have way to compare these results and see how different they are.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.979	0.991	0.995	0.703
Fork	60	58	1	1	0.996	0.722
Knife	60	111	0.987	0.991	0.995	0.678
Spoon	60	59	0.95	0.983	0.994	0.709
Total time:	7 seconds				GPU Mem:	2293MB = 2.24GB
Epochs:	NaN (inference)				Image size:	640*1.3 = 832

Table 5.5: Obtained TTA results for YoLov5s under our custom dataset of images from simulation.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.98	0.992	0.996	0.712
Fork	60	58	0.983	0.984	0.995	0.73
Knife	60	111	0.991	0.991	0.996	0.701
Spoon	60	59	0.966	1	0.996	0.705
Total time:	8 seconds				GPU Mem:	3381MB = 3.30GB
Epochs:	NaN (inference)				Image size:	640*1.3 = 832

Table 5.6: Obtained TTA results for YoLov5m under our custom dataset of images from simulation.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.987	0.986	0.994	0.714
Fork	60	58	0.998	0.983	0.995	0.706
Knife	60	111	0.981	0.991	0.995	0.687
Spoon	60	59	0.983	0.983	0.994	0.749
Total time:	8 seconds				GPU Mem:	3627MB = 3.55GB
Epochs:	NaN (inference)				Image size:	640*1.3 = 832

Table 5.7: Obtained TTA results for YoLov5l under our custom dataset of images from simulation.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	228	0.981	0.982	0.987	0.693
Fork	60	58	0.981	0.966	0.972	0.713
Knife	60	111	0.98	0.982	0.996	0.677
Spoon	60	59	0.983	1	0.994	0.69
Total time:	12 seconds				GPU Mem:	4391MB = 4.28GB
Epochs:	NaN (inference)				Image size:	640*1.3 = 832

Table 5.8: Obtained TTA results for YoLov5x under our custom dataset of images from simulation.

As tables 5.5 to 5.8 show, we are using augmentation with increased image size of more 30% than the original size. The obtained results are very similar to those obtained during train, which show that the retrain of the YoLo was well applied, once also using TTA, the model was capable of inference over novel images.

5.4.2 Information provided by this Object Detection Model

When applying the Object Detection model to the current scene, we expect to get an image containing bounding boxes surrounding the objects present at the scene and the respective label and score attributed to them by YoLo.

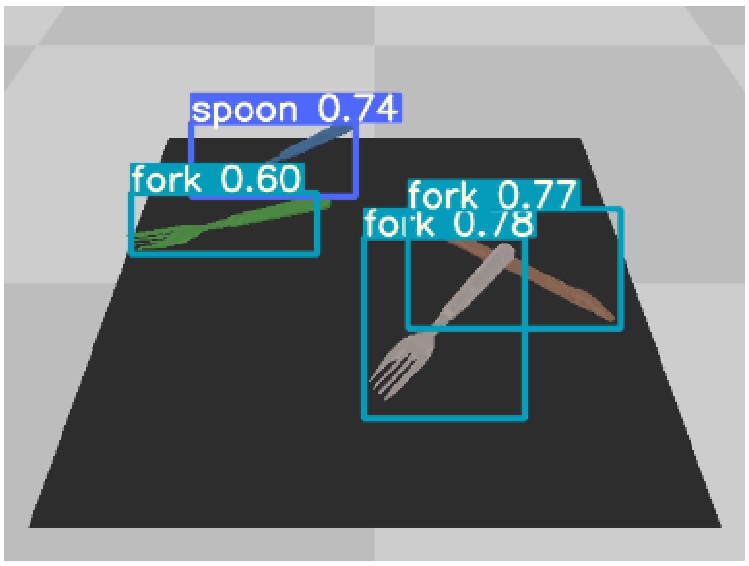


Figure 5.3: Resulting image from YoLo's evaluation of the current scene.

With some little changes we can save to files the information of the bounding box of each object in scene and the label and score attributed by YoLo. The structure of this information is as follows in the figure 5.4:

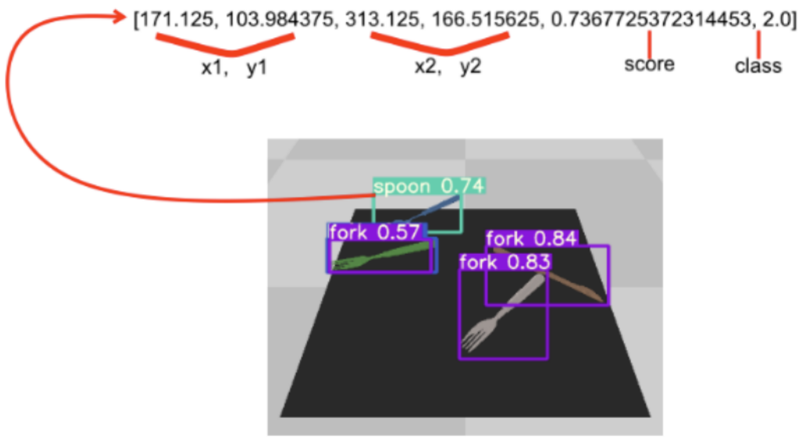
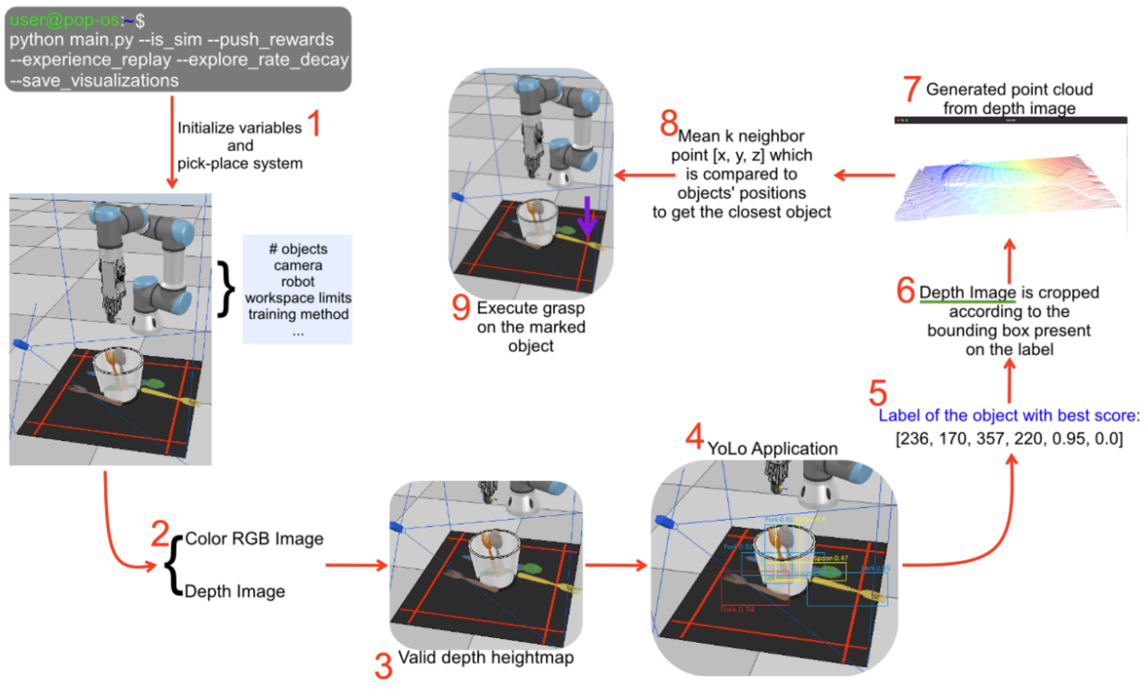


Figure 5.4: From YoLo we can obtain the label of each object.

The first four numbers are related to the starting and ending point of the bounding box surrounding the object, the next number indicates the confidence score attributed to the object (meaning, how much the object recognition model, from 0 to 1, recognizes the object for it says it is.) and lastly, the class of the object (0 - fork, 1 - knife and 2 - spoon).

5.4.3 Incorporation of Object Detection Model in Visual Pushing-for-Grasping

The incorporation of the Object Detection model was simple. Since it is a way for us to evaluate the current iteration's scene and get the coordinates of the object with better score attributed by YoLo, the procedure regarding its incorporation is shown in figure 5.5:



```
user@pop-os:~$
python main.py --is_sim --push_rewards
--experience_replay --explore_rate_decay
--save_visualizations
```

1 Initialize variables and pick-place system
# objects
camera
robot
workspace limits
training method
...
2 Color RGB Image
Depth Image
3 Valid depth heightmap
4 YoLo Application
5 Label of the object with best score:
[236, 170, 357, 220, 0.95, 0.0]
6 Depth Image is cropped according to the bounding box present on the label
7 Generated point cloud from depth image
8 Mean k neighbor point [x, y, z] which is compared to objects' positions to get the closest object
9 Execute grasp on the marked object

Figure 5.5: Application of YoLo with the Visual Pushing-for-Grasping algorithm, made with [50].

From figure 5.5, this incorporation attends to 9 steps, described as follows:

1. Initialization of the scenery, robot, vision sensor camera, objects and workspace limits;
2. Get RGB and Depth images of the current scene;
3. Get the valid depth heightmap of the workspace;
4. Apply YoLo onto the RGB image of the scene;
5. Obtain the label and bounding box of the object with the best score attributed by YoLo model;
6. Given the bounding box coordinates, crop the Depth image of the current iteration;
7. Generate the point cloud according to the Depth image;
8. Using the k-means algorithm, get the mean k point of the point cloud. Since the coordinates of this point are seen from the vision sensor's point of view, when comparing to the coordinates of the objects in scene, we have to retrieve the object with coordinates closer to the camera;
9. Execute grasp or push on the specified object in order to grasp it.

5.4.3.1 Conversion from Depth Image to Point Cloud

By using the Object Detection model called YoLo, we want to be able to differentiate the objects present in the simulation, thus telling the robot to apply an action to a specific object until all the objects are separated by type.
To endow the algorithm with this kind of mechanism, we proposed a module that would grab the depth image and convert it to point cloud to calculate a mean point from that point cloud. This module follows the below pipeline:

1. Retrieve the depth image of the current epoch/iteration and crop it according to the bounding box of the object with better confidence value attributed by YoLo;
2. Initialize the camera intrinsic values - [[618.62 0 w/2], [0 618.62 h/2], [0 0 1]]. The value 618.62 is the focal length in x-axis and y-axis for the camera, and w/2, and h/2 represent the central point of the image;
3. Create point cloud from depth image, with depth_scale=10000 and depth_trunc=10000. This way, we secure that all the values in the point cloud are calculated for meters;
4. Initialize camera pose using camera position, camera orientation and camera rotation matrix, i.e., simulation world position [x, y, z], its orientation angles with each axis and the rotation matrix given as a list of values in Euler angles [B.12];
5. Convert point cloud from camera coordinate system to robot coordinate system;
6. Apply zFar=10 and zNear=0.01 transformation to all points in the point cloud;
7. Sort surface points of point cloud by z value;
8. Collect the point with largest z from the point cloud;
9. Apply K-mean neighbors algorithm to obtain the average point from point cloud;
10. Get average coordinates for the point obtained from k mean neighbors algorithm.

The coordinates of the k-means point [x, y, z] are coordinates closer to the camera in the simulation world, thus when comparing to the coordinates of the objects in simulation, we want to get the object with the largest coordinates in terms of x and y-axis, therefore being closer to the camera.

5.4.3.2 K-Means Point to Robot Action

Figure 5.6 contains the last part of the changes applied to the original Visual Pushing-for-Grasping algorithm.

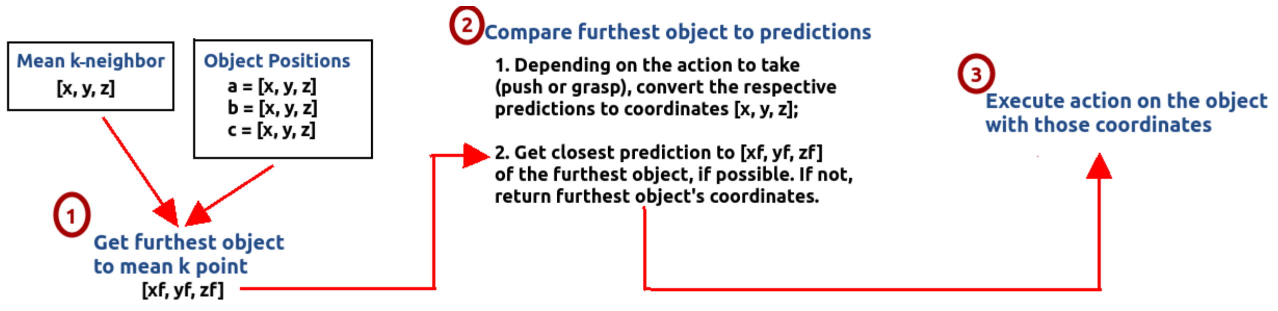

Mean k-neighbor
[x, y, z]
Object Positions
a = [x, y, z]
b = [x, y, z]
c = [x, y, z]
1
Get furthest object to mean k point
[xf, yf, zf]
2
Compare furthest object to predictions
1. Depending on the action to take (push or grasp), convert the respective predictions to coordinates [x, y, z];
2. Get closest prediction to [xf, yf, zf] of the furthest object, if possible. If not, return furthest object's coordinates.
3
Execute action on the object with those coordinates

Figure 5.6: K-means point to robot action, made with [50].

We can describe each step as:

1. The first step involves the computed mean k point and all the object positions.
   We know that the coordinates of the mean k point are given as if they were seen from the camera,

therefore to obtain the coordinates of an object [x_f, y_f, z_f] from the object positions, we need to look for the object with furthest coordinates relatively to the coordinates of mean k point;

2. Then, depending on which action the robot will make (grasp or push), we convert either the push or the grasp predictions to [x, y, z] coordinates;
3. Next we compare the coordinates [x_f, y_f, z_f] to those of the predictions. If in fact there is a prediction close enough to [x_f, y_f, z_f] coordinates, we will use that prediction otherwise we use [x_f, y_f, z_f];
    - A prediction is close to [x_f, y_f, z_f] if the difference between them is less than 0.005, which is the value we chose as maximum difference.
4. Last step is to execute the action in the location with the coordinates resulting from the previous step.

5.4.4 Conclusion

In this chapter, we presented all the proposed and integrated changes made to the Visual Pushing-for-Grasping algorithm with the hope of having a way to detect which object are we grasping and to make the grasp more straightforward, instead of trying to grasp at a random location, and to be more accurate. These changes were mainly directed to detect the objects in the scene and apply an action to a specific object.

The next chapter will describe the simulation with CoppeliaSim [59], with and without using a container for the objects, and all the experiments made and their results.

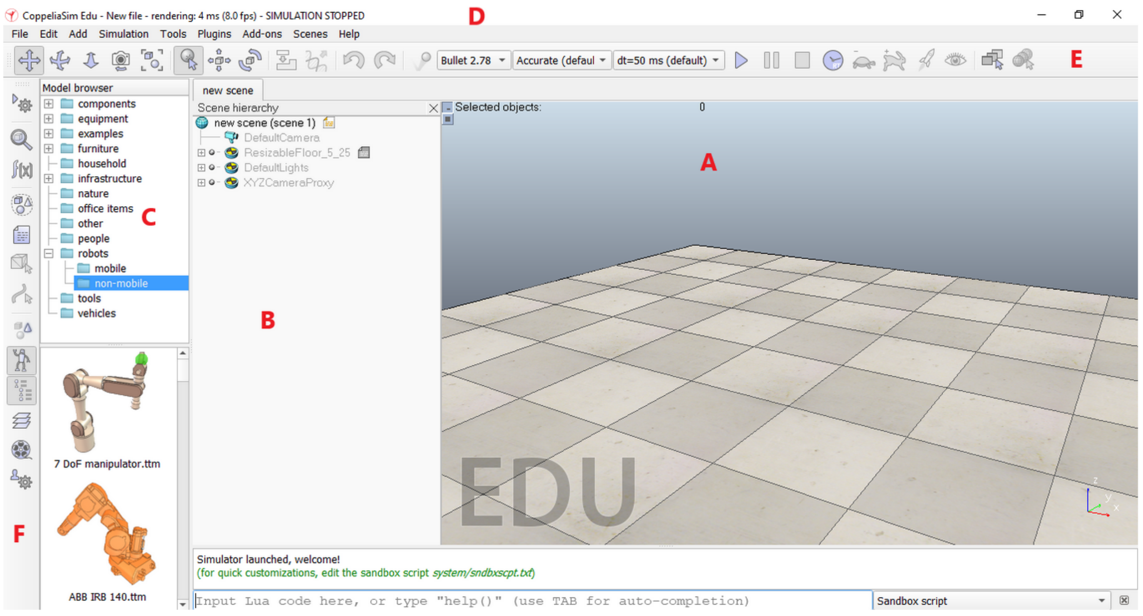
Chapter 6

Simulation with CoppeliaSim

6.1 Visual Pushing-for-Grasping and CoppeliaSim Interaction

VPG is a Reinforcement Learning algorithm [75] developed to assist a real robot arm in grasping objects. This algorithm analyzes every scene and computes the best action.
The best action can be a push - where the robot tries to separate objects to create space between them or can be a grasp - where the robot tries to grasp an object.
This algorithm connects to a robot via TCP connection or to a simulation via localhost connection (127.0.0.1) through port 19997. As soon as the connection is established, the simulation starts and the objects start being placed in the robot's environment.

We are using the CoppeliaSim [59] simulator. It allows us to simulate various types of robots, from UR3 to 7-DOF robotic arm and any kind of simulation environment. Figure 6.1 represents the Coppeliasim environment.



Simulator launched, welcome!
(for quick customizations, edit the sandbox script *system/sndbxscpt.txt*)

```
Input Lua code here, or type "help()" (use TAB for auto-completion)
```

Figure 6.1: CoppeliaSim simulation environment.

The environment can be divided by parts:

1. In A is the place where the simulation scene will be displayed;
2. As for B is where the hierarchy of objects of the scene are shown;
3. C is the list of all the modules and robotic arms available;
4. Part D shows the status of the current scene - resume/stopped/paused;
5. E and F are quick access tabs, for example to access an object's properties.

Figure 6.2 shows a scene of the robot and the coordinate axes. The measurement unit used from the International System of Units is the meter and the robot is centered in [0, 0.0251, 0.1430].

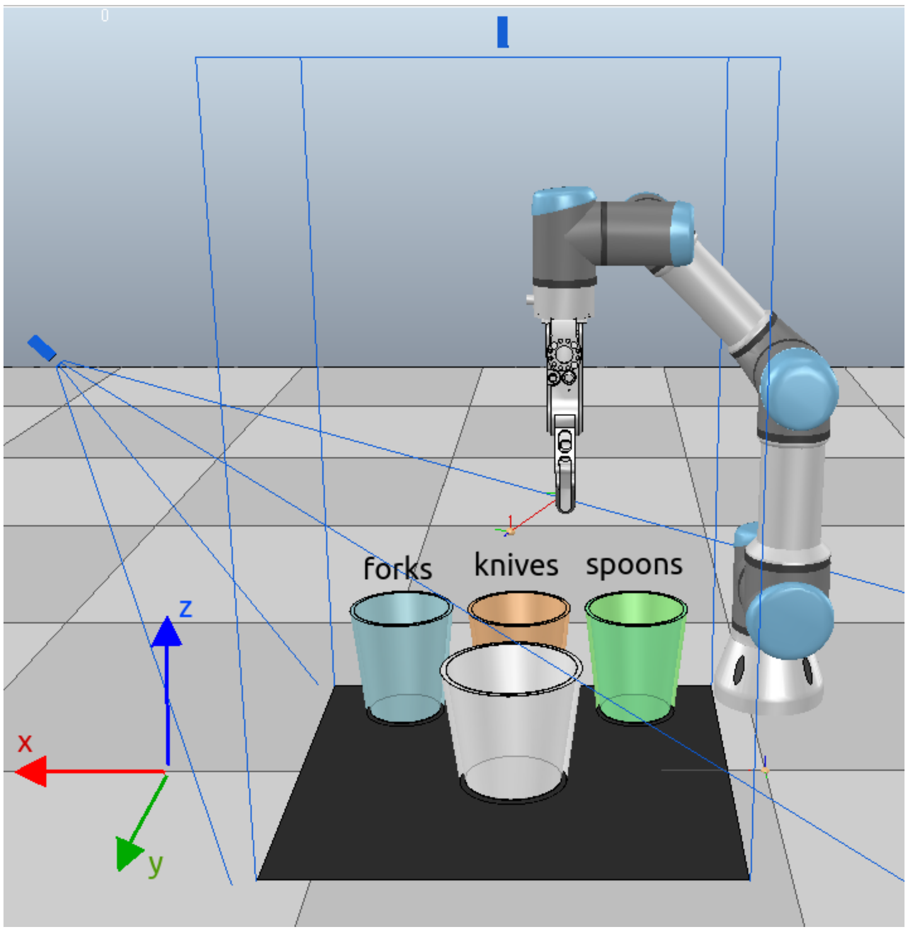


Figure 6.2: Simulation and coordinate axes.

6.2 Problems Found

The simulation with CoppeliaSim [59] and the UR3 [73] had some problems to address at first. These problems were:

1. The simulation randomly froze due to gripper position calculations;
2. When restarting the first time after clearance to add new objects, it wouldn't stop restarting and adding objects;
3. And lastly, not all the simulation area was 100% reachable for the robot. Therefore, it would have "freak out" moments, causing the scattering of the objects still in simulation, which eventually would count as a clearance.

The first two problems were easier to solve, although they involved understanding the code first. The last point is related to what we can call the "safe zone" where the robot could actually reach objects or get to locations where it had to execute actions. This safe area is discriminated in figures 6.3 and 6.4.

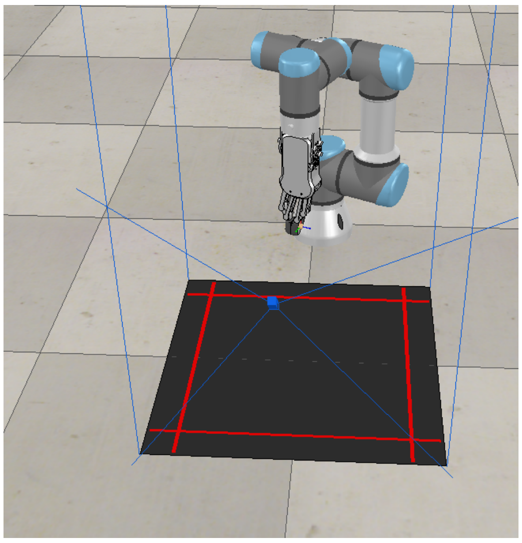


Figure 6.3: Created an area that was thought to be safe for carrying out actions.

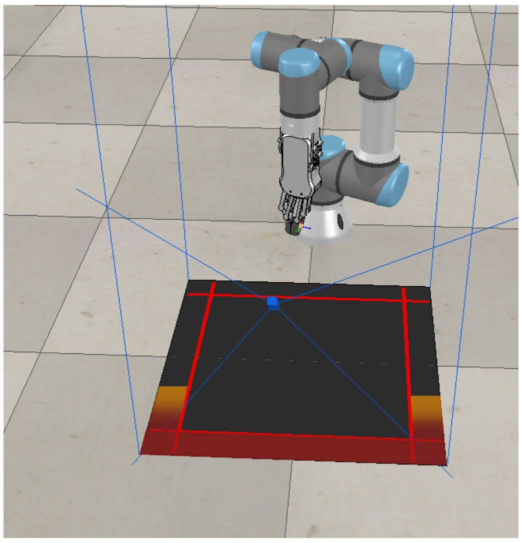


Figure 6.4: Representation of the places where realizing actions would much faster ”freak out” moments during simulation, made with [50].

In figure 6.4, we represent the locations where realizing actions outside the ”safe area” would cause the robot to lose control, thus spreading out the objects through the simulation environment and outside it. In this situation, a dark red color means an area much more conducive to this problem.
The solution found to this problem involves two parts:

1. Adding objects to the environment would stop being in a random location of the table and start being in the center of it, right below the robot;
2. Evaluate the target locations where the actions were to occur and apply a correction to these, so they would occur inside the safe area and finish inside of it.

6.3 Adaptation to Cutlery Objects

This section will discuss the simulation using cutlery objects. First, we will synthesize the key aspects of using cutlery objects, but without using a container for them. Then, we briefly introduce the aspects of using cutlery objects with a container for them. And, finally we introduce other cutlery objects retrieved from TurboSquid [68] and the results from fine-tuning YoLo [82] for these objects.

6.3.1 Simulation without Bin

In order for us to achieve the main goal of this thesis, the Visual Pushing-for-Grasping simulation had to stop using the block-shaped objects (figure 5.1) and start using the cutlery objects that it was supposed to be using.
So, the first major step was to incorporate these objects with the simulation. For this step, we just use the .obj files related to the objects we want to use, from the YCB dataset.
To make the simulation start using cutlery objects instead of the original block-shaped objects, we just need to use the following command on a new terminal window:

```
            python3 -W ignore main.py --is_sim --obj_mesh_dir 'path_to_folder' --
    push_rewards --experience_replay --explore_rate_decay --training_iters 250 --
    save_visualizations
```

Listing 6.1: Adding the tag "obj_mesh_dir" and telling the path to a folder containing the new meshes to use makes the simulation run with the new objects.

Below is the standard command line to run the simulation with the default object meshes and without ignoring warnings.

```
            python3 main.py --is_sim --push_rewards --experience_replay --
    explore_rate_decay --save_visualizations
```

Listing 6.2: Run VPG without specifying folder containing meshes will make the simulation use the common block-shaped meshes.

The following figure demonstrates the usage of these new cutlery objects from YCB dataset.

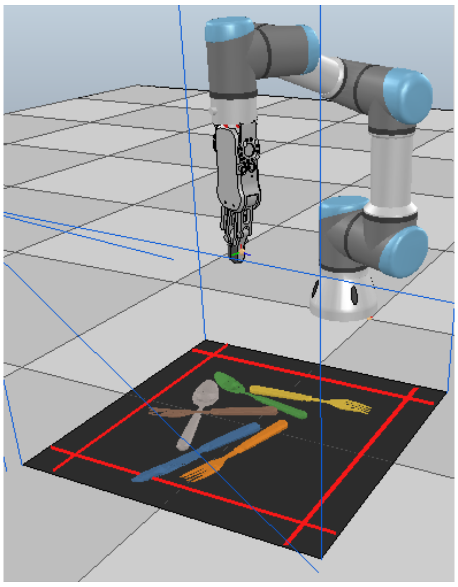


Figure 6.5: Visual Pushing-for-Grasping simulation with cutlery objects from YCB dataset, but with no bin.

6.3.2 Simulation using Bin

This part is the second major step of the change from block-shaped objects to cutlery objects. It involves the addition of a bin to act has a place where the objects will stay and wait for the robot to grasp them, instead of being flat on the table surface. It doesn't work well, since some of the objects go through the sides of the bin, i.e., either as soon as the objects would drop in the container, they would go through the sides of the container landing on the simulation table or by any action made by the robot, they would also go through the container. This might occur because of some bug in the software, because even raising the thickness of the bin through software in the CoppeliaSim [59], won't solve the problem.
However, this kind of problem doesn't occur in reality, so we chose to ignore its existence.

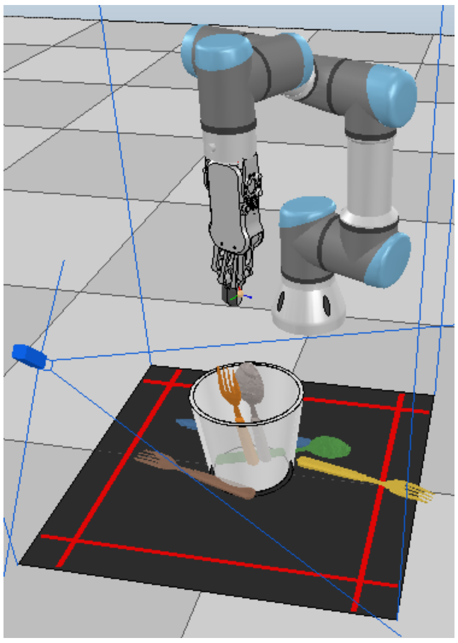


Figure 6.6: Visual Pushing-for-Grasping simulation with cutlery objects from YCB dataset, but using the bin as container for the objects during simulation.

6.3.3 Cutlery Objects from TurboSquid

We said we used objects from YCB dataset [79], however they have some deformations along their structure, as figures 6.7 and 6.8 show.

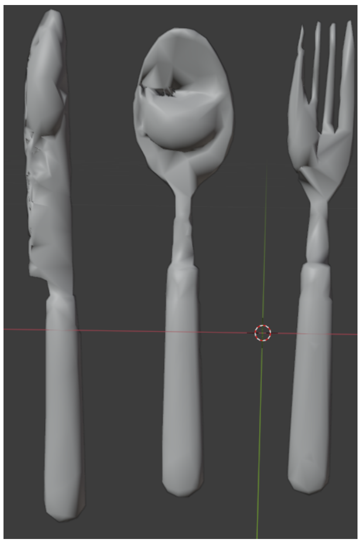


Figure 6.7: YCB models seen from front. Image taken from Blender.

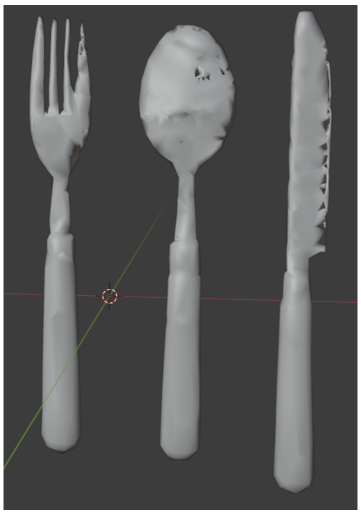


Figure 6.8: YCB models seen from back. Image taken from Blender.

Therefore, and to test the simulation with cutlery objects that resemble the real world objects, we searched for this kind of objects on TurboSquid [68] website and found the following objects:

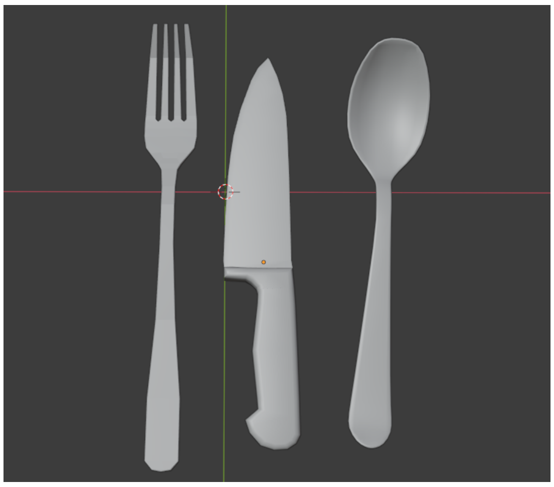


Figure 6.9: Cutlery downloaded from TurboSquid - front view [68].

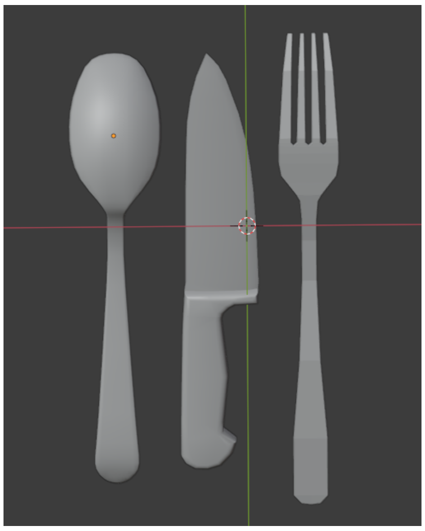


Figure 6.10: Cutlery downloaded from TurboSquid - back view [68].

These new objects also do not hold much longer inside the cup without falling from its sides, however the simulation is able to run without any problems with these objects. Though, to be able to completely use them, a fine-tuning of YoLo was necessary.
This fine tuning counted with the same number of images as the other we did with objects from YCB dataset, 300 images total divided over 80% train and 20% validation sets. The results, table 6.1, from the fine tuning were obtained over the validation set, containing 60 images.

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	144	0.914	0.92	0.934	0.722
Fork	60	60	0.993	0.933	0.995	0.733
Knife	60	55	0.999	1	0.996	0.806
Spoon	60	29	0.75	0.826	0.812	0.626
Total time:	2.826h = 2h49min33sec				GPU Mem:	7.77GB
Epochs:	300				Image size:	640

Table 6.1: YoLo version 5x fine tuning (train) on new images with TurboSquid models [68].

These results show that there exists high precision > 0.9 to detect forks and knives and that most of those detections of forks and knives were well executed, because the respective recall value was above 0.9 and their respective mAP.5 was also above 0.9 indicating that the IoU of the prediction bounding boxes over the ground truth bounding boxes was greater than 0.5.
When comparing to the results of training YoLo with YCB models [79], table 5.4, we see that the results for Precision, Recall and mAp.5 were similar for fork and knife classes. However, the train with YCB models obtained better results in those three metrics also for the spoon class, making the result of the line representing all classes far better than the one representing TurboSquid [68] results. We also performed a test on the recently trained model of YoLo version 5X with TTA and the obtained results were a bit better than on those obtained with the objects from YCB dataset (table 5.8), since there was very high precision for both fork and knife classes (= 1) and a similar mAP.5 for both the previous

classes. However, the spoon class had worst results with TurboSquid [68] models, which shows that the model had more difficulty in detecting spoons from these models than with YCB models [79].

Class	# Images	Targets	Precision	Recall	mAP.5	mAP.5:.95
All	60	144	0.986	0.914	0.945	0.718
Fork	60	60	1	0.924	0.995	0.725
Knife	60	55	1	0.99	0.996	0.768
Spoon	60	29	0.958	0.828	0.843	0.66
Total time:	12 seconds				GPU Mem:	4391MB = 4.28GB
Epochs:	NaN (inference)				Image size:	640*1.3 = 832

Table 6.2: YoLo version 5x fine tuning (inference) on new images with TurboSquid models [68].

This new fine tuning was only made over the version 5X of YoLo, since it is the version we will be using for simulations.

6.3.4 Simulation using TurboSquid's 3D Model Cups

The experiments and respective results, on section 6.4, were conducted using a 3D model cup from CoppeliaSim [59] itself. However, these experiments had a problem related to the container where the objects would drop out of it.

This problem can be described as a ”Go-through situation”, and as explained in section 6.3.2, the objects either go through the sides of the cup as soon as they enter the simulation or when the robot applies an action on them.

In an attempt to see this problem solved, we though of searching for other 3D models of cups on TurboSquid [68] and see if the problem would persist. We looked over 20 models, some weren't compatible as they would be in formats like:

1. ”.c4d” which is a 3D modelling format of the program Cinema 4D, not compatible with CoppeliaSim [59] nor with Blender [6]. If Blender could open it, we would've converted it to one of the supported formats;
2. ”.max”. Although Blender [6] can import this kind of files, it gave us an error of not supported version for the format file;
3. ”.fbx”. We could open this kind of files with Blender [6] and then convert them to one of the supported formats, however they would be too big for CoppeliaSim [59] to import them.

CoppeliaSim supports the following format files for mesh import [12]:

1. OBJ: Wavefront file format. Currently the only format file supported for mesh import;
2. DXF: Autodesk file format (AutoCad);
3. STL (ASCII or binary);
4. COLLADA;
5. URDF.

From those 20 or so models, only 12 3D models were in the correct file format supported by the CoppeliaSim [59], ".obj" file format. From these 12 3D models, 11 of them didn't show any sign of being able to avoid the described problem, even after performing the following attempts/changes to them:

1. Increase the skin thickness of the 3D model in CoppeliaSim;
2. Increase the compliance feature of the 3D model, also in CoppeliaSim. This feature is said to represent the softness of the material and is calculated by 1/stiffness, being stiffness the value that indicates how much stiff is an object;
3. Increasing the skin thickness of an object using Blender, using the solidify modifier of this tool; This modifier provides two modes: Simple and Complex, both have the Offset option, which is a value between [-1, 1] and helps to locate the solidified output on both in and out the original mesh. If set to 0, the output will be centered on the original mesh [7]. And, Rim which is an option to fill or not the edges between the inner and outer areas;

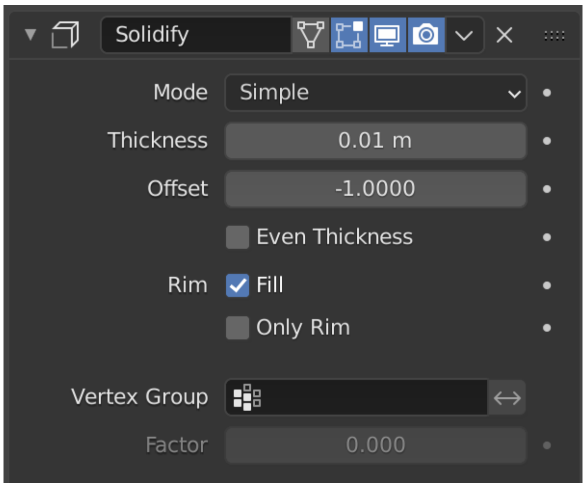

Solidify
Mode: Simple
Thickness: 0.01 m
Offset: -1.0000
Even Thickness
Rim: ✓ Fill
Only Rim
Vertex Group
Factor: 0.000

Figure 6.11: Blender's solidify modifier.

4. Increasing the skin thickness using Blender and reducing the number of polygons of the object, also with Blender using the Decimate modifier. This modifier gives us the option to set the ratio of polygons we want the object to have, i.e. how much % of polygons from the total we want.

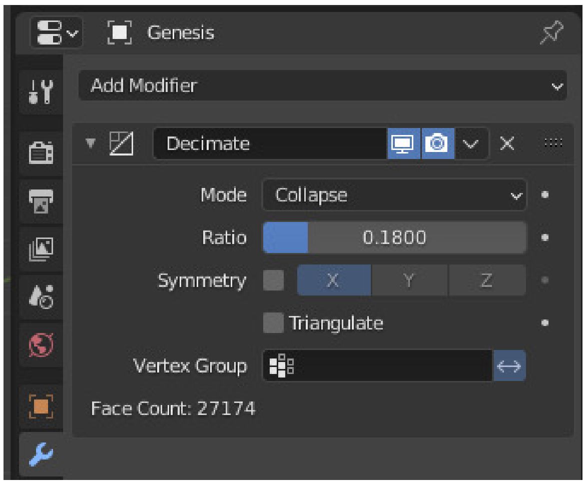

Genesis
Add Modifier
Decimate
Mode: Collapse
Ratio: 0.1800
Symmetry: X Y Z
Triangulate
Vertex Group
Face Count: 27174

Figure 6.12: Blender's decimate modifier.

The 12 3D models can be seen in the figure 6.13. The objects used didn't even stood a second inside the models 1 to 9 and 11 and 12, they would instantly drop on the floor going through the cup. The 3D model number 10 offer a bit of hope, because the objects would stand in it for a bit longer without going through it, but as soon as the robot would make an action they instantly would drop on the floor.

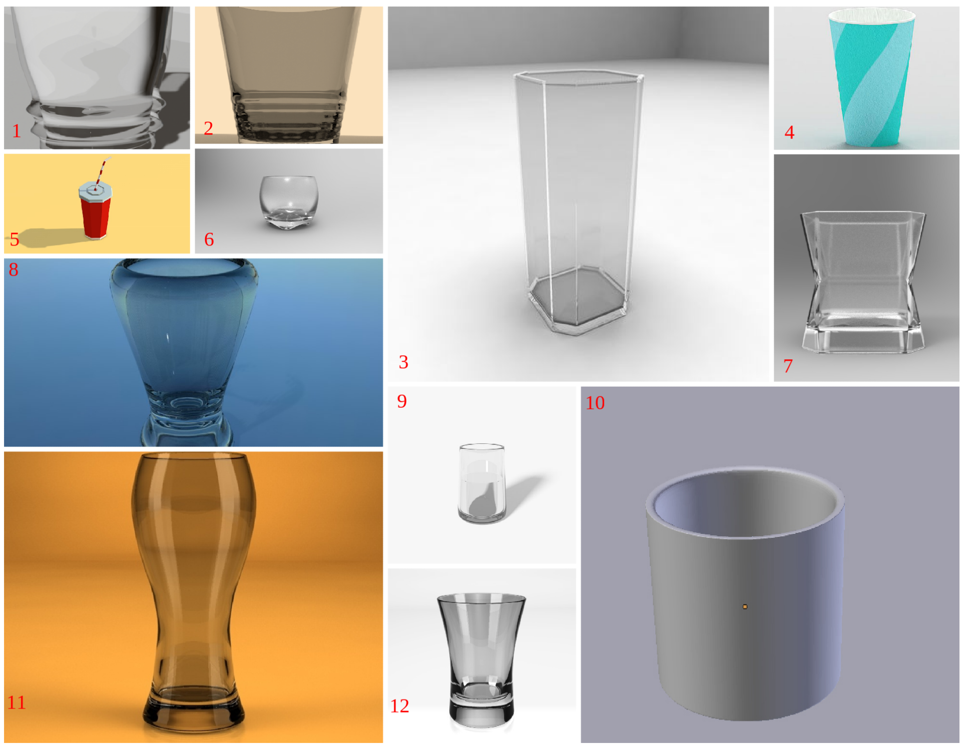


Figure 6.13: The 12 3D models of cups tested. The respective reference for each model is: 1 [31], 2 [22], 3 [29], 4 [48], 5 [8], 6 [77], 7 [78], 8 [61], 9 [76], 10 [14], 11 [5], 12 [30].

6.4 Simulation Experiments

In this section we will approach the conducted simulation experiments and the respective results. They will be divided into three major sections:

1. Simulation experiments and respective results using YCB and TurboSquid models and without using container;
2. Simulation experiments and respective results using container for YCB models [79];
3. Simulation experiments and respective results using container for TurboSquid 3D models [68];

The simulations ran on CoppeliaSim environment [59] under Ubuntu operating system [70], since is quite simple to run them on a Linux based system through a terminal window.

The majority of the simulations ran for 250 epochs, since it takes less time to obtain results. However, we also ran the simulation two times during 500 epochs, just to analyze if the results would be better or not. The supposed and ideal stop criteria would have been 2500 epochs, since the authors of Visual Pushing-for-Grasping [16] ran the original code for this number of iterations, however running over 2500 epochs takes roughly 12 to 14 hours to finish and that is why we decided to start by obtaining results with a lower number of epochs/iterations.

To run the experiments we are using an Nvidia Geforce RTX 3060 12GB vram, supported by the following Python packages:

- Torch developer version 1.9.0 [65] - A library for machine learning, mostly used for Computational Vision and Natural Processing Language;
- Torchvision developer version 0.9.0 [66] - A package of the library Torch, used in computer vision to apply transformations on images, etc;
- Open3d 0.11.0 [47] - Modern open-source library for 3d data processing. Allows transformations such as RGB image to Point Cloud, read and write Point Clouds and draw Point Clouds and display them on screen.

6.4.1 Simulation Experiments Using YCB and TurboSquid Models and Without Using Container

In this section we will approach the results of the simulations made without container for the objects, using 3D models from YCB dataset [79] and TurboSquid [68].

6.4.1.1 Experiments Using YCB Object Models

For this type of models and without using container, we made two tests. The first only used 2 objects and the second uses 6 objects, resembling a more cluttered environment.

Figure 6.14 contains the results of the simulation ran for 250 epochs with two 3D models from the YCB dataset [79] and without using a container for the objects.

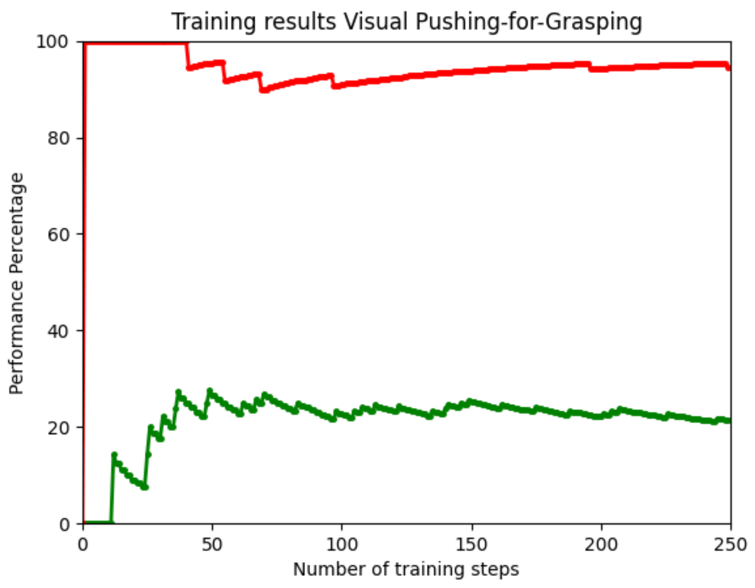


Figure 6.14: Obtained results for two YCB objects without container and during 250 epochs. In red is the Average Push to All Ratio and in green is the Average Grasp to All Ratio.

As we can see, the performance of the made pushes (Average Push to All Ratio) manages to stay above 90.0%, with its average value from the sum of all iterations around 94.24% and the Average Grasp to All Ratio keeps above 20.0%, with its average value from the sum of all iterations around 21.49%. This training run took around 1 hour and 7 minutes to complete.

To know how accurate this algorithm was, we counted 21 epochs where it had to restart and therefore take image of the place where successful grasped objects go. So, from these 21 epochs, 2 correspond to a restart given that no grasps were successful and the maximum number of attempts to grasp all objects was reached.
However, during the other 19 epochs, 26 grasps were successfully made, and there were 6 times where the objects were placed on the correct cup after being grasped. That said, there are 3 ways to describe the accuracy of this model:

1. According to (1), knowing that the simulation performed 140 grasp iterations over 250 iterations total, we can calculate this metric as $\frac{26}{140} \simeq 18.57\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{6}{26} = 23.07\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{6}{140} \simeq 4.28\%$.

Figure 6.15 contains the results of the simulation ran for 250 epochs with six 3D models from YCB dataset [79] and without using a container for the objects.

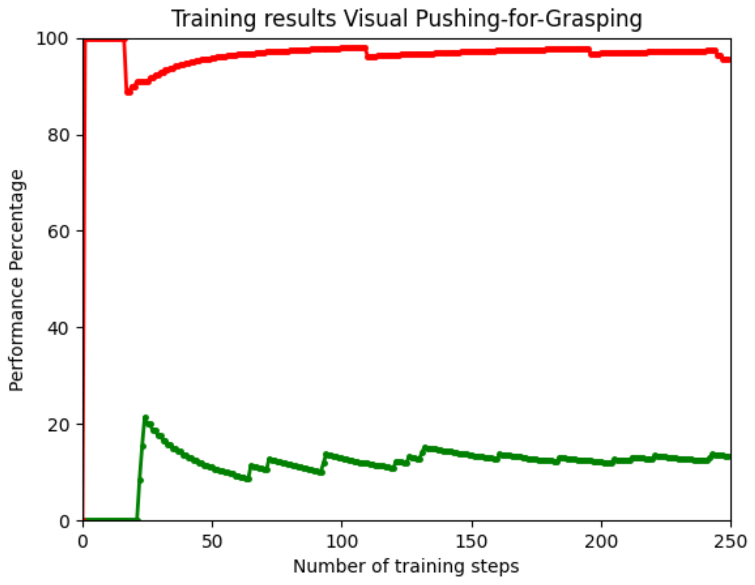


Figure 6.15: Obtained results for six YCB objects without container and during 250 epochs. In red is the Average Push to All Ratio and in green is the Average Grasp to All Ratio.

The performance of the made pushes (Average Push to All Ratio) manages to stay above 90.0%, with its average value from the sum of all iterations around 96.36% and the Average Grasp to All Ratio keeps above 10.0%, with its average value from the sum of all iterations around 11.63%. This training run took around 1 hour and 24 minutes to complete.
To know how accurate this algorithm was, we counted 9 epochs where it had to restart and therefore take image of the place where successful grasped objects go. So, from these 9 epochs, 3 correspond to a restart given that no grasps were successful and the maximum number of attempts to grasp all objects were reached.
However, during the other 6 epochs 11 grasps were successfully made, and there were 6 times where the objects were placed on the correct cup after being grasped. That said, there are 3 ways to describe

the accuracy of this model:

1. According to (1), knowing that the simulation performed 134 grasp iterations over 250 iterations total, we can calculate this metric as $\frac{11}{134} \simeq 8.20\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{6}{11} = 54.54\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{6}{134} \simeq 4.47\%$.

6.4.1.2 Experiments Using 3D Object Models from TurboSquid

For this type of models and without using container, we also made two tests. The first only used 2 objects and the second uses 6 objects, resembling a more cluttered environment.

Figure 6.16 contains the results of the simulation ran for 250 epochs with two 3D models from TurboSquid [68] and without using a container for the objects.

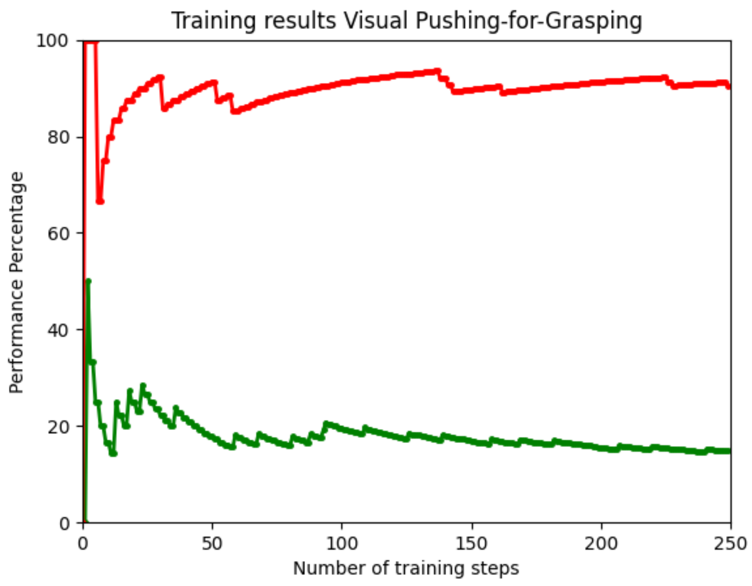


Figure 6.16: Obtained results for two 3D models from TurboSquid without container and during 250 epochs. In red is the Average Push to All Ratio and in green is the Average Grasp to All Ratio.

The performance of the made pushes (Average Push to All Ratio) stays around 90.0%, with its final average value around 89.65% and the Average Grasp to All Ratio keeps above 10.0%, with its final average value around 17.86%. This training run took around 1 hour and 14 minutes to complete.
To know how accurate this algorithm was, we counted 16 epochs where it had to restart and therefore take image of the place where successful grasped objects go. So, from these 16 epochs, 7 correspond to a restart given that no grasps were successful and the maximum number of attempts to grasp all objects were reached.
However, during the other 9 epochs 9 grasps were successfully made, and there were 5 times where the objects were placed on the correct cup after being grasped. That said, there are 3 ways to describe the accuracy of this model:

1. According to (1), knowing that the simulation performed 135 grasp iterations over 250 iterations

total, we can calculate this metric as $\frac{9}{135} \simeq 6.66\%$;

2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{5}{9} = 55.55\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{5}{135} \simeq 3.70\%$.

Figure 6.17 contains the results of the simulation ran for 250 epochs with six 3D models from TurboSquid [68] and without using a container for the objects.

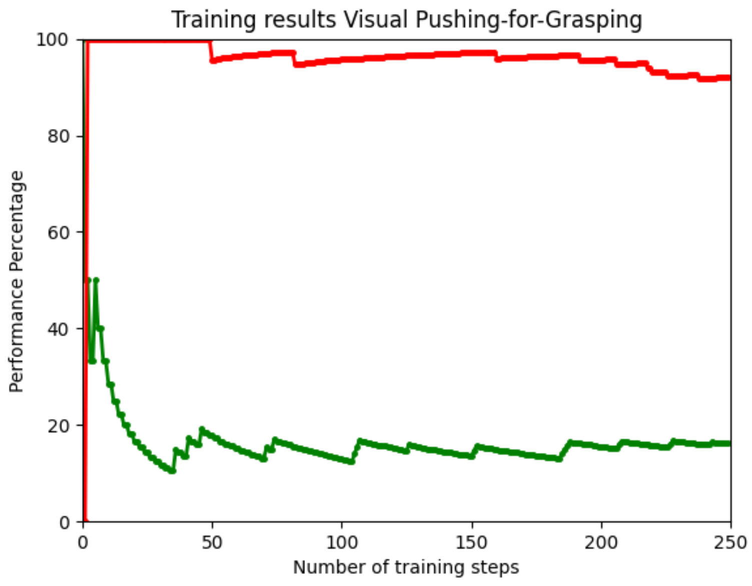


Figure 6.17: Obtained results for six 3D models from TurboSquid without container and during 250 epochs. In red is the Average Push to All Ratio and in green is the Average Grasp to All Ratio.

The performance of the made pushes (Average Push to All Ratio) stays around 90.0%, with its final average value around 95.67% and the Average Grasp to All Ratio keeps above 10.0%, with its final average value around 16.65%. This training run took around 1 hour and 45 minutes to complete.
To know how accurate this algorithm was, we counted 7 epochs where it had to restart and therefore take image of the place where successful grasped objects go. So, from these 7 epochs, 4 correspond to a restart given that no grasps were successful and the maximum number of attempts to grasp all objects were reached.
However, during the other 3 epochs 9 grasps were successfully made, and there were 3 times where the objects were placed on the correct cup after being grasped. That said, there are 3 ways to describe the accuracy of this model:

1. According to (1), knowing that the simulation performed 136 grasp iterations over 250 iterations total, we can calculate this metric as $\frac{9}{136} \simeq 6.62\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{3}{9} = 33.33\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{3}{136} \simeq 2.21\%$.

6.4.2 Simulation Experiments Using a Container for YCB Models

In this section we will approach the results of the simulations made with container for the objects from YCB dataset [79]. There are two results regarding two tests, one ran for 250 epochs and the other ran for 500 epochs. These two results were an attempt to see if there would be any difference in results using different number of total epochs.

Figure 6.18 contains the results of the simulation ran for 250 epochs with two 3D models from YCB dataset [79] and using a container for the objects.

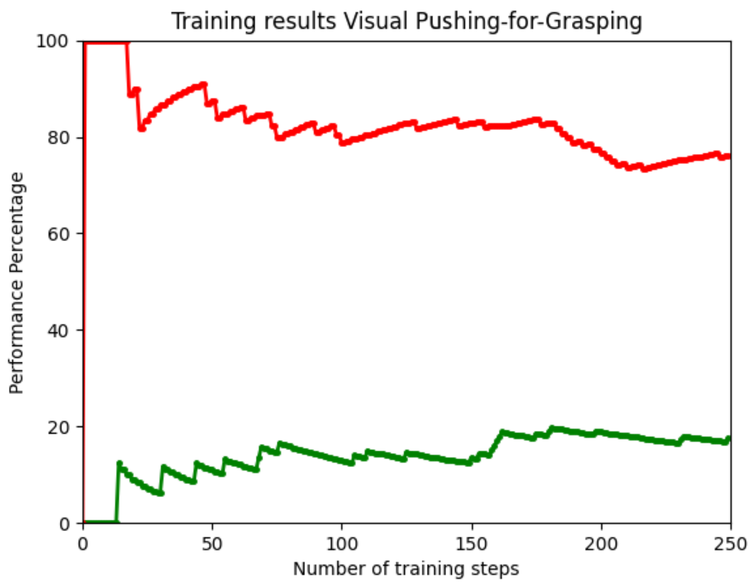


Figure 6.18: Obtained results for our adaptation of Visual Pushing-for-Grasping under 250 epochs for YCB models. In red is the Average Push to All Ratio and in green is the Average Grasp to All Ratio.

The results from our first run tell us that, the performance of the made pushes (Average Push to All Ratio) manages to stay above 75.0%, with its average value from the sum of all iterations around 82.3% and the Average Grasp to All Ratio keeps above 15.0%, with its average value from the sum of all iterations around 14.0%. This training run took around 1 hour and 8 minutes to complete.
To know how accurate this algorithm was, we counted 24 epochs where it had to restart and therefore take image of the place where successful grasped objects go. So, from these 24 epochs, 10 correspond to a restart given that no grasps were successful and the maximum number of attempts to grasp all objects were reached.
However, during the other 14 epochs 21 grasps were successfully made, and there were 19 times where the objects were placed on the correct cup after being grasped. That said, there are 3 ways to describe the accuracy of this model:

1. According to (1), knowing that the simulation performed 145 grasp iterations over 250 iterations total, we can calculate this metric as $\frac{21}{145} \simeq 14.48\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{19}{21} = 90.00\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{19}{145} \simeq 13.10\%$.

Figure 6.19 contains the results of the simulation ran for 500 epochs with two 3D models from YCB dataset [79] and using a container for the objects. This was an attempt to see if the results would improve or not, by making the simulation run for more epochs.

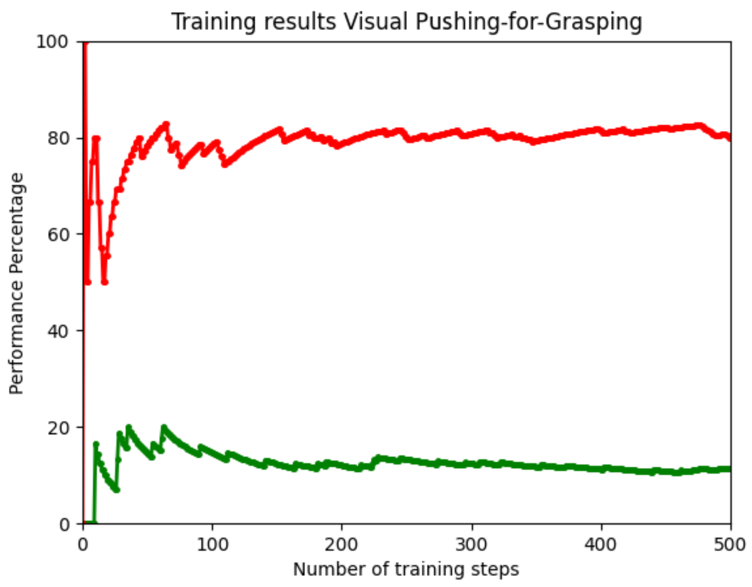


Figure 6.19: Obtained results for our adaptation of Visual Pushing-for-Grasping under 500 epochs using YCB models. In red is the Average Push to All Ratio and in green is the Average Grasp to All Ratio.

The values for Average Push to All Ratio and Average Grasp to All Ratio kept stable around the same as in figure 6.18. The average values from the sum of the results of all iterations are respectively 79.10% and 12.45%. This training run took around 2 hours and 17min to complete.
To know how accurate this algorithm was, we counted 35 epochs where it had to restart and therefore take image of the place where successful grasped objects go. So, from these 35 epochs, 16 correspond to a restart given that no grasps were successful and the maximum number of attempts to grasp all objects were reached.
However, during the other 19 epochs 24 grasps were successfully made, and there were 13 times where the objects were placed on the correct cup after being grasped. That said, the accuracy of this training run can be classified according to:

1. According to (1), knowing that the simulation performed 200 grasp iterations over 500 iterations total, we can calculate this metric as $\frac{24}{200} \simeq 12.00\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{13}{24} \simeq 54.16\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{13}{200} \simeq 6.50\%$.

6.4.3 Simulation Experiments Using a Container for TurboSquid Models

In this section we will approach the results of the simulations made with container for the objects from TurboSquid [68]. There are two results regarding two tests, one ran for 250 epochs and the other ran for 500 epochs. These two results were an attempt to see if there would be any difference in results using different number of total epochs.

Figure 6.20 contains the results of the simulation ran for 250 epochs with two 3D models from TurboSquid [68] and using a container for the objects. The red line indicates the Average Push to All Ratio, which in overall had a mean value of 58.76%. This result shows that a bit more than half the pushes made by the robot were well made causing changes on objects in simulation.
The green line shows the Average Grasp to All Ratio, which had a mean value of 11.58%, indicating that in the universe of all made grasps, only a small percentage was correctly made resulting in successful grasps.
This training simulation took 73 minutes to finished, i.e. around 1 hour and 13 minutes.
To know how accurate this simulation was we evaluate the images YoLo computes before the simulation restarts, so we know how many objects the algorithm/robot got right in the universe of all iterations/epochs that this event occurred.
The algorithm took these images 23 times, i.e. there were 23 epochs in which the simulation had to restart by not having objects in it to grasp or the maximum number of attempts made to grasp the objects successful were reached. In these 23 times, 13 of them the simulation restarted without having grasped objects. From the other 10 times, there were 3 epochs were 1 object was grasped and placed in the correct cup, being that during these 10 epochs the robot grasped successfully 10 times
That said, there are 3 ways to describe the accuracy of this model:

1. According to (1), knowing that the simulation performed 106 grasp iterations over 250 iterations total, we can calculate this metric as $\frac{10}{106} \simeq 9.43\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{3}{10} = 33.0\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{3}{106} \simeq 2.83\%$.

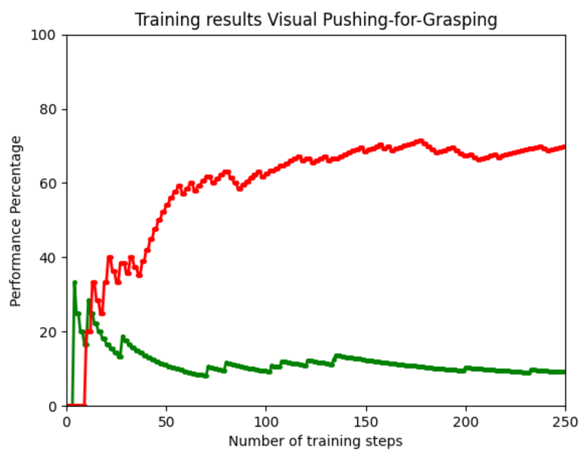


Figure 6.20: 250 epochs of simulation with 3D cutlery models from [68].

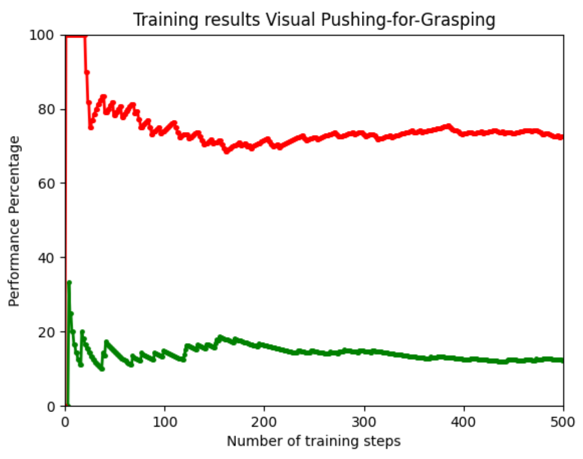


Figure 6.21: 500 epochs of simulation with 3D cutlery models from [68].

In figure 6.21 we have the results of running the simulation with the new object models from TurboSquid [68], during 500 epochs. This simulation took around 144 minutes, i.e. 2 hours and 24 minutes to finish.

As said previously, the red line indicates the Average Push to All Ratio, which had a mean value of 74.59% and shows that almost three-quarters of the made pushes were successful.
The green line shows the Average Grasp to All Ratio, which had a mean value of 14.1%, indicating that in the universe of all made grasps, almost one-quarter of the made grasps were successful.
There were 41 epochs in which the simulation had to restart by not having objects in it to grasp or the maximum number of attempts made to grasp the objects successful were reached. In these 41 times, 26 of them the simulation restarted without having grasped objects. From the other 15 times, there were 11 epochs were at least 1 object was grasped and placed in the correct cup, being that during these 15 epochs the robot grasped successfully 24 times.
That said, there are 3 ways to describe the accuracy of this model:

1. According to (1), knowing that the simulation performed 170 grasp iterations over 500 iterations total, we can calculate this metric as $\frac{24}{170} \simeq 14.10\%$;
2. As for the number of successful and well placed grasps in the whole universe of all successful grasps (2), can be calculated as $\frac{11}{24} \simeq 45.83\%$;
3. Finally, for the percentage of successful and well placed grasps in the whole universe of made grasps (3), can be calculated as $\frac{11}{170} \simeq 6.47\%$.

6.5 Conclusion

In conclusion, the accuracy of the training run depends on how many objects were grasped and how many were grasped and placed on the correct cup. Since we see very little different from the Average Grasp to All Ratio and Average Push to All Ratio results, it makes no sense in using more than 250 epochs for simulations.
Although the results show that even in cluttered environments, this new approach manages to keep the same or acquire better results than those obtained with non-cluttered environments, it is sad to say that we do not have a good value for the Average % of Grasp Success, because of the following two markers:

1. The ”go-through” situation explained in section 6.3.2, where the objects pass through the sides of their container landing on the floor of the simulation;
2. The fact that the robot collides with the containers in simulation and therefore, sometimes the actions are not completely made.

Test #	Epochs	Grasp Rate	Pushing Rate	Metric #1	Metric #2	Metric #3	Container	Type Objects	# Objects	Total Time
1	250	21.49%	94.24%	18.57%	23.07%	4.28%	No	YCB	2	67min
2	250	11.63%	96.36%	8.20%	54.54%	4.47%	No	YCB	6	84min
3	250	17.86%	89.65%	6.66%	55.55%	3.70%	No	TurboSquid	2	74min
4	250	16.65%	95.67%	6.62%	33.33%	2.21%	No	TurboSquid	6	105min
5	250	14.00%	82.30%	14.48%	90.00%	13.10%	Yes	YCB	2	68min
6	250	11.58%	58.76%	9.43%	33.00%	2.83%	Yes	TurboSquid	2	73min
7	500	12.45%	79.10%	12.00%	54.16%	6.50%	Yes	YCB	2	137min
8	500	14.10%	74.59%	14.10%	45.83%	6.47%	Yes	TurboSquid	2	144min

Table 6.3: Summary of simulation experiments using YCB [79] and TurboSquid [68] models and with and without using container for the objects.

The above table 6.3 shows a table comparing all the results from the simulation.
As we can see the Grasp Success Rate had its best result using 3D models from YCB dataset (21.49% -

test number 1), even the Push Success Rate is better when using 3D models from YCB dataset (96.36% - test number 2) than with models from TurboSquid and both values were obtained without usage of a container for the objects.

The metric (1) obtained its best result using YCB models and without using a container for them (18.57% - test number 1). The metric (2) obtained its best result using YCB models and using a container for them (90.00% - test number 5). The last metric (3) obtained its best result using YCB models and using a container for them (13.10% - test number 5).

Therefore, we can conclude that using models of objects from YCB dataset [79] gives us better results than using models from TurboSquid [68], because the models from YCB dataset [79] were better developed for simulations, meaning that they were scanned from objects of the daily life and were designed to work for robotic manipulation in simulation such as grasping simulations, despite the deformations they have. Forasmuch as we saw during the simulations, the robot was capable of making better grasps with them than with the models taken from TurboSquid [68], once they would't slip from the gripper when already grasped, something that sometimes happened with the other models [68].
The results also show that some metrics obtain better results when using the container, while others have better results when not using it.
This might happen due to what referred to previously, about the fact of YCB models being more prepared to be used in simulation than models from TurboSquid [68], but also because:

1. When not using the cup as a container, the robot won't get stuck in it as sometimes it would happen and therefore, making the desired action without any problems;
2. The object detection model might detect better the YCB models [79] than for TurboSquid models [68], i.e., for example, when detecting a spoon from the TurboSquid models, it would be detected as a fork. In contrast, for the YCB models, this kind of error wouldn't occur.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Along this whole year, we developed a project related to three fields: Robotics, Artificial Intelligence and Computer Vision. Besides being a very challenging task, it was a very enriching project since we learned so much about how robots work and how complex a software developed for them must be.
Robotics is an area with a growing demand for specialists and researchers to look for solutions to a wide variety of problems regarding the robots, their way of functioning, where they are applied and for what they are used.
Another big field is Artificial Intelligence. This field is in constant growth with new neural networks always surfacing and competing with each other. It is used in a variety of fields from medicine to robotics.
To make this project, it was critical to overcome various steps with success to achieve its ending. The first most challenging steps were to understand how to work with a simulator and to understand how the original code behind our project works. Other steps such as, understanding the whole process of getting images from the simulation to process them to select which object to grasp or push and how to integrate this process in the existing code, were necessary to understand and implement to achieve with success the end of our project.
Personally, we are not completely satisfied with what we achieved, since there were some problems with the simulation that could not be overcome and therefore, the results obtained are not what we expected and also the fact that we did not reached the phase where we would interact with the real robot.

7.2 Future Work

For future work, there is a workable, solid base that works with cutlery objects, that also contains a sorting system, although not 100% accurate.
As of now, we are capable of simulating the robot separating and sorting cutlery objects using an object detection model. This new approach promises to evaluate the simulation differently from what the authors of Visual Pushing-for-Grasping did, since we are not making grasps or pushes ”at random” locations and we are evaluating for objects that were grasped successfully and placed in the correct container.
We won't deny that it might need improvements, such as:

1. Correct the z coordinate of the grasp or push action, when the object is inside the container. We tried applying this correction, but the robot sometimes wouldn't use the correct value, maybe due to badly coded conditions;
2. Create a mechanism to help the robot avoid getting stuck in the objects' container.

However, we think this project can be adapted to work with the real-world robot UR3 present at SOCIA laboratory.

Appendix A

Installations Guide

A.1 PointNetGPD: Detecting Grasp Configurations from Point Sets - Tutorial

This tutorial is intended to serve as a guide through in case of an error comes up.
Some of the commands in this tutorial come from https://github.com/lianghongzhuo/PointNetGPD

A.1.1 Before install

```
mkdir -p $HOME/code/
cd $HOME/code/
```

Listing A.1: Create code folder for PointNetGPD

A.1.2 Requirements

There are two packages that will be needed in order to proceed with installation.

First, clone a repository to the recently created **code** folder.

```
cd $HOME/code
git clone https://github.com/lianghongzhuo/PointNetGPD.git
```

Listing A.2: Clone PointNetGPD github repository

Then, we need to install some requirements.

```
cd $HOME/code/PointNetGPD
pip3 install -r requirements.txt
```

Listing A.3: Install requirements for PointNetGPD

Now, to install the first package called **meshpy**.

```
cd $HOME/code/PointNetGPD/meshpy
python3 setup.py develop
```

Listing A.4: Install meshpy

And lastly, the other package called **dex-net**.

```
cd $HOME/code/PointNetGPD/dex-net
python3 setup.py develop
```

Listing A.5: Install dex-net

The next step might be need some intervention, if the gripper we are working with is different from **robotiq_85** (in our case, is **rg2**).

```
"finger_width":
"real_finger_width":
"hand_height":
"hand_height_two_finger_side":
"hand_outer_diameter":
"hand_depth":
"real_hand_depth":
"init_bite":
```

Listing A.6: Parameters used for grasp pose generation.

```
    "min_width":
    "force_limit":
    "max_width":
    "finger_radius":
    "max_depth":
```

Listing A.7: Parameters used for dataset generation.

A.1.3 Generated Grasp Dataset Download

The dataset can be downloaded from:
https://tams.informatik.uni-hamburg.de/research/datasets/PointNetGPD_grasps_dataset.zip

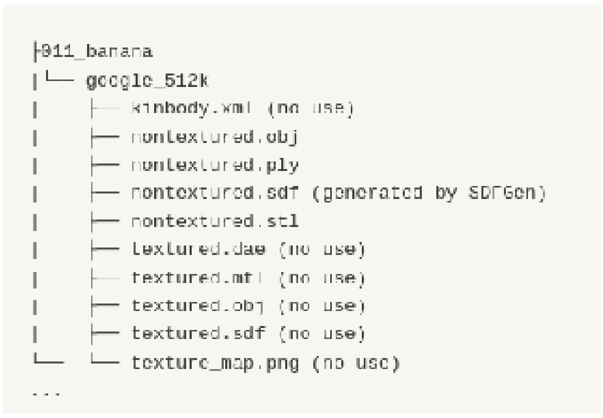
A.1.4 Generate Our Own Grasp Dataset

1. For first experiments, we will use the Banana Dataset of Google512k. It must be added within the next configuration:

   ```
               mkdir -p $HOME/dataset/ycb_meshes_google/objects

   ```

   The structure of the folders and objects after pasting the dataset in the objects'folder should be like this:



```
├011_banana
| └── google_512k
|       ├── kinbody.xml (no use)
|       ├── nontextured.obj
|       ├── nontextured.ply
|       ├── nontextured.sdf (generated by SDFGen)
|       ├── nontextured.stl
|       ├── textured.dae (no use)
|       ├── textured.mtl (no use)
|       ├── textured.obj (no use)
|       ├── textured.sdf (no use)
└──     └── texture_map.png (no use)
...
```

Figure A.1: Banana Dataset Structure.

2. Next step is to install SDFGen, in the code folder in home directory, using the next steps:

```
            git clone https://github.com/jeffmahler/SDFGen.git
            cd SDFGen
            sudo sh install.sh

```

3. Then, we must install the pcl package. It can be one of the next versions: 1.6, 1.7, 1.8 or 1.9. For example, we used version 1.8.1 downloaded from: `https://github.com/PointCloudLibrary/pcl/archive/pcl-1.8.1.tar.gz` , extracted the file and executed the following commands:

```
            cd pcl-pcl-1.8.1 && mkdir build && cd build
            cmake ..

```

   Since this last step might fail we need to install some packages:

   - It might fail, because you have yet not installed the **cmake** library.

   ```
                        sudo apt-get update -y
                        sudo apt-get install -y make
                        make -v

   ```

     The last command will show the version of **make** package.
   - Or it might fail, because you may be missing some other packages rather than **cmake**. Run the following command:

   ```
                        sudo apt -y install libeigen3-dev libflann-dev libboost1
   .58-all-dev
                        sudo apt -y install cmake cmake-gui doxygen mpi-default-
   dev openmpi-bin openmpi-common libusb-dev libqhull* libusb-dev libgtest-dev
                        sudo apt -y install git-core freeglut3-dev pkg-config
   build-essential libxmu-dev libxi-dev libphonon-dev libphonon-dev phonon-
   backend-gstreamer graphviz mono-complete qt-sdk

   ```

   After this troubleshoot, we can continue with pcl build.

```
                cmake -DCMAKE_BUILD_TYPE=Release ..
                make -j2
                sudo make -j2 install

```

   → j2 means we will want to use two threads of the cpu. If you might want it to use all the threads, just replace 2 by the number of threads your cpu have.
   The full guide to compile **pcl** can be followed through here: `https://pcl.readthedocs.io/en/latest/compiling_pcl_posix.html`.

4. Next step is to install **python pcl** library python-pcl:
   For this, we can head to the home directory, just by doing

```
            cd $HOME
            or just simply...
            cd

```

   and run, some other commands:

```
            sudo apt install python3 python3-pip python3-opencv
            pip3 install --upgrade pip
            pip3 install cython
            pip3 install numpy
            sudo apt install python3-pcl
            cd python-pcl
            python setup.py build_ext -i
            python setup.py develop

```

- If by any chance, you get some error while compiling the python-pcl, just run these commands:

  ```
                          sudo apt install libpcl-dev
                          sudo apt install gcc-8 g++8
                          sudo update-alternatives --install /usr/bin/gcc gcc /usr
  /bin/gcc-8 8
                          sudo update-alternatives --install /usr/bin/g++ g++ /usr
  /bin/g++-8 8
                          sudo update-alternatives --config gcc
                          pip3 install vtk

  ```

  The above commands can be found in this page:
  `https://github.com/kroglice/Pop-OS-Setup`

5. Then, we need to generate sdf file for each nontextured.obj.

   ```
               cd $HOME/code/PointNetGPD/dex-net/apps
               python3 read_file_sdf.py

   ```

   After this, alter the lines in the file grasp_sampler where time.clock() appears, to time.time(). This file is located in /code/PointNetGPD/dex-net/src/dexnet/grasping.

6. Next step is to generate the dataset.

   ```
               cd $HOME/code/PointNetGPD/dex-net/apps
               python3 generate-dataset-canny.py [prefix]

   ```

   The [prefix] is optional, but if you want you can put whatever you like and that prefix will be in the generated files.
   If an error occurs, because of a missing folder called generated_grasps,
   just create it in /code/PointNetGPD/dex-net/apps/ , using:

   ```
               cd /code/PointNetGPD/dex-net/apps/ && mkdir generated_grasps

   ```

A.1.5 Visualization tools

For the next two steps, you will need to create some folders and copy some files to another location:

- First, create these two folders ycb_meshes_google and ycb_grasp in $HOME/code/PointNetGPD/PointNetGPD/data/;
- Then, copy the generated files in $HOME/code/PointNetGPD/dex-net/apps/generated_grasps

to $HOME/code/PointNetGPD/PointNetGPD/data/ycb_grasp.
Note: If you had add a prefix in those files, just remove it;

- Following, copy the dataset's folder 011_banana from $HOME/dataset/ycb_meshes_google/objects to $HOME/code/PointNetGPD/PointNetGPD/data/ycb_meshes_google.
  Note: Add the same sufix to this folder, as in the files from $HOME/code/PointNetGPD/dex-net/apps/generated_grasps. In our case it was "**_140**", leading to something like "**011_banana_140**";
- Then, run this command in a terminal window:

```
                sudo apt-get install python3-pyqt5*
                sudo apt-get install python3-pyqt5

```

1. To visualize the grasps, alter the 198th line in read_grasps_from_file.py to:
   npy_names = glob.glob(home_dir + "/PointNetGPD/data/ycb_grasp/*.npy")
   And run the following commands:

```
                cd $HOME/code/PointNetGPD/dex-net/apps
                python read_grasps_from_file.py

```

2. To visualize the object normals, run these commands:

```
                cd $HOME/code/PointNetGPD/dex-net/apps
                python Cal_norm.py

```

   This code will check the norm calculated by **meshpy** and **pcl** library.

A.1.6 Notes

To see which processes are running in CPU, just run these commands:

```
  sudo apt install htop
  htop
```

To see which processes are running in GPU, just run these commands (Nvidia drivers required):

```
watch -n 1 nvidia-smi
```

A.2 Nvidia CUDA installation Guide

To install Nvidia CUDA, first we need to do some preparations just by following this guide [55]:

```
    lspci | grep -i nvidia
```

Listing A.8: Verifying if system has CUDA capable GPU

Verify if your UNIX system is supported by CUDA installation:

```
    uname -m && cat /etc/*release
```

Listing A.9: Verifying if system is supported

The next output is a sample of a correct output. If you see something similar starting with $x84\_64$ you are good to go.

```
    x86_64
```

```
Red Hat Enterprise Linux Workstation release 6.0 (Santiago)
```

Listing A.10: Output from previous command

Please verify if you have gcc package installed:

```
gcc --version
```

Listing A.11: Verifying if system has gcc installed

The kernel headers and development packages for the currently running kernel can be installed with:

```
sudo apt-get install linux-headers-$(uname -r)
```

Listing A.12: Install linux kernel headers

Now, we are going to actually install all the necessary CUDA packages, through the next commands executable via a terminal window in your UNIX-like system:

```
cd Downloads
wget https://developer.nvidia.com/compute/cuda/10.1/Prod/local_installers/cuda-repo-ubuntu1804-10-1-local-10.1.105-418.39_1.0-1_amd64.deb
wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.0.5/10.1_20201106/cudnn-10.1-linux-x64-v8.0.5.39.tgz
wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.0.5/10.1_20201106/Ubuntu18_04-x64/libcudnn8_8.0.5.39-1+cuda10.1_amd64.deb
wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.0.5/10.1_20201106/Ubuntu18_04-x64/libcudnn8-dev_8.0.5.39-1+cuda10.1_amd64.deb
wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.0.5/10.1_20201106/Ubuntu18_04-x64/libcudnn8-samples_8.0.5.39-1+cuda10.1_amd64.deb

sudo dpkg -i cuda-repo-ubuntu1810-10-1-local-10.1.105-418.39_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.105-418.39/7fa2af80.pub
sudo dpkg -i cuda-repo-ubuntu1810-10-1-local-10.1.105-418.39_1.0-1_amd64.deb
sudo apt-get update
sudo apt-get install cuda

sudo nano /etc/profile.d/cuda.sh
export PATH=$PATH:/usr/local/cuda/bin
export CUDADIR=/usr/local/cuda
(save with ctrl+o and exit with ctrl+x)
sudo chmod +x /etc/profile.d/cuda.sh

sudo nano /etc/ld.so.conf.d/cuda.conf
/usr/local/cuda/lib64
(save with ctrl+o and exit with ctrl+x)
sudo ldconfig

reboot

tar -xzvf cudnn-10.1-linux-x64-v8.0.5.39.tgz
sudo cp cuda/include/cudnn*.h /usr/local/cuda/include
sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
sudo chmod a+r /usr/local/cuda/include/cudnn*.h /usr/local/cuda/lib64/libcudnn*
sudo dpkg -i libcudnn8_8.0.5.39-1+cuda10.1_amd64.deb
sudo dpkg -i libcudnn-dev_8.0.5.39-1+cuda10.1_amd64.deb
sudo dpkg -i libcudnn9-samples_8.0.5.39-1+cuda10.1_amd64.deb

(Para verificar se o cudnn ficou bem instalado...)
cp -r /usr/src/cudnn_samples_v8/ $HOME
cd $HOME/cudnn_samples_v8/mnistCUDNN
make clean && make
./mnistCUDNN
Se obtiver uma mensagem "TEST PASSED!", então o cudnn ficou bem instalado.
```

Listing A.13: Install CUDA and cudnn packages version 10.1

A.3 Robot Simulator CoppeliaSim

A.3.1 Where and what version of CoppeliaSim download?

In order to download the simulator CoppeliaSim, we must head to this website [59] and download the version we desire. We have downloaded the Educational version, because it might have more features that we may need for my thesis project.

A.3.2 How to run CoppeliaSim?

→ In Linux and Unix-Like Systems (tested on Ubuntu 18.04.5 LTS [70]):
After your CoppeliaSim's download finish, you just have to follow these steps in order to get it running.

1. First, head to the downloads folder and unpack the CoppeliaSim file;
2. Second, open the new unpack folder regarding CoppeliaSim;
3. Next, open a new terminal window from inside that folder, just by clicking with the right mouse button anywhere inside that folder;
4. And finally, use the following commands and CoppeliaSim should just open as shown in the image below the commands.

```
chmod u+x ./coppeliaSim.sh
```

Listing A.14: Give permissions to execute CoppeliaSim

```
sudo ./coppeliaSim.sh
```

Listing A.15: Start CoppeliaSim

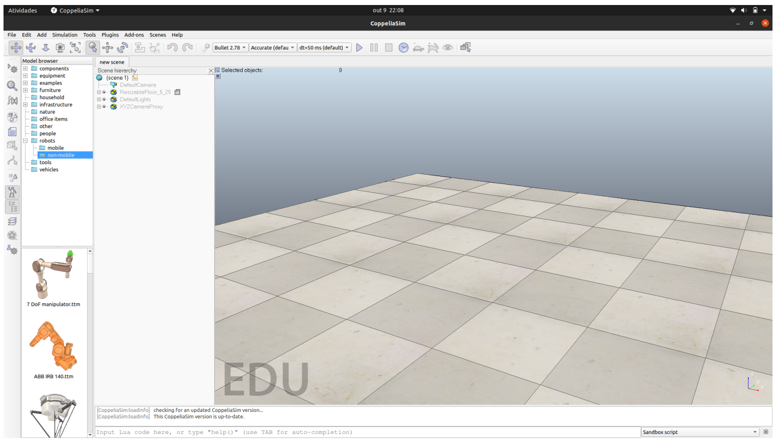
In this last command, only use "sudo" if needed.

→ In macOS Systems (tested on Catalina 10.15.7):
After unzipping the downloaded file, head over to the downloads folder and open the CoppeliaSim's folder. When in there, you will notice a package called "CoppeliaSim.app", just right click over it and select **Open with terminal**.
In prompted terminal window, type the following command:

```
sudo xattr -r -d com.apple.quarantine *
```

Listing A.16: Required step to run CoppeliaSim on macOS

After you have run this command, you can now open the *.app* file.


```
[CoppeliaSim:loadinfo]   checking for an updated CoppeliaSim version...
[CoppeliaSim:loadinfo]   This CoppeliaSim version is up-to-date.
```

EDU
Input Lua code here, or type "help()" (use TAB for auto-completion)
Sandbox script

Figure A.2: CoppeliaSim running after using the commands above.

A.4 ROS - Robot Operating System

1. **What is ROS?**
   ROS is a framework developed to help build software for the robotic arm and simulators, like **CoppeliaSim**.
   This framework is composed by tools, libraries and conventions that want to simplify the way we design and develop complex and robust robot behaviors.
2. **How to install ROS**
   At first we tried to install **ROS** in a Linux Distribution [21] that apparently is not supported, even tough it is based on Debian.
   That aside, we installed **Ubuntu 18.04.5** [70] and managed to install **ROS** [60] . After these installations, we also proceeded with the installation of some packages that might be necessary, for example: pyTorch [56] and CUDA [55][36].

A.4.1 ROS Installation Guide

This guide only applies to **Ubuntu 20.04 Focal Fossa** and **ROS Noetic**.

```
 sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /
etc/apt/sources.list.d/ros-latest.list'
```

Listing A.17: Setup your sources

```
 sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Listing A.18: Setup your keys

```
 sudo apt update
 sudo apt install ros-noetic-desktop-full
```

Listing A.19: Installing ROS Noetic

```
 echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
 source ~/.bashrc
```

Listing A.20: Environment setup

A.4.2 pyTorch Installation following this Guide [56]

In order to install pyTorch in our UNIX Environment, we just have to run one single command in a terminal window.

```
pip install torch torchvision
```

Listing A.21: pyTorch installation

A.5 Deep Reinforcement Learning for Robotic Pushing and Picking in Cluttered Environment

The paper cited in [17] has a Github repository [18], which requires some installations before testing the provided code with the simulator CoppeliaSim [59]. These requirements are: Python 3.6, tqdm, SciPy, OpenCV3, Numpy, Pillow, TensorFlow 1.4.1, urx, pyserial, pypcl and h5py.

```
sudo apt-get install python3 python3-dev python3-pip
```

Listing A.22: Python3 Installation on Ubuntu via sudo.

```
pip install tqdm
```

Listing A.23: tqdm Installation on Ubuntu via pip.

```
pip install scipy
```

Listing A.24: SciPy Installation on Ubuntu via pip.

```
pip install opencv-python
```

Listing A.25: OpenCV3 Installation on Ubuntu via pip.

```
pip install numpy
```

Listing A.26: Numpy Installation on Ubuntu via pip.

```
pip install pillow
```

Listing A.27: Pillow Installation on Ubuntu via pip.

```
pip install --upgrade https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow
-1.4.1-py3-none-any.whl
```

Listing A.28: TensorFlow 1.4.1 Installation on Ubuntu via pip.

```
pip install urx
```

Listing A.29: urx Installation on Ubuntu via pip.

```
pip install pyserial
```

Listing A.30: pyserial Installation on Ubuntu via pip.

```
pip install -U setuptools
pip install python-pcl
```

Listing A.31: pypcl Installation on Ubuntu via pip.

```
pip install h5py
```

Listing A.32: h5py Installation on Ubuntu via pip.

After these steps, CUDA is needed to run simulation with graphics card. Thus, refer to **CUDA Installation** guide on Ubuntu UNIX system.

In order to run the code related to this research, we need to install a package called **pylibfreenect**, however it can be a bit tricky to do so, since the command $\$ pip\, install\, pylibfreenect2$ might fail due to an error linked to the nonexistence of a file named *config.h*. Thus, we present a tutorial [39] to workaround this problem.

1. Clone the **libfreenect2** repository:

   ```
   git clone https://github.com/OpenKinect/libfreenect2
   ```

   Listing A.33: Clone libfreenect2 repository.

2. Head to the recently clone directory:

   ```
   cd libfreenect2/depends/
   ```

   Listing A.34: Change to libfreenect2 directory.

3. Now, run the following **script**:

   ```
   ./download_debs_trusty.sh
   ```

   Listing A.35: Download important deb packages.

4. Install **build tools**:

   ```
   sudo apt-get install build-essential cmake pkg-config
   ```

   Listing A.36: Install build tools Ubuntu UNIX System.

5. Install **libusb** 1.0:

   ```
   sudo apt-get install libusb-1.0-0-dev
   ```

   Listing A.37: Install libusb package.

6. Install **TurboJPEG** package:

   ```
   sudo apt-get install libturbojpeg0-dev
   ```

   Listing A.38: Install TurboJPEG package on Ubuntu.

7. Install **OpenGL** library:

   ```
   sudo apt-get install libglfw3-dev
   ```

   Listing A.39: Download and install OpenGL library for UNIX systems.

8. Install **OpenCL** library

   ```
   sudo mkdir -p /etc/OpenCL/vendors; echo /usr/lib/arm-linux-gnueabihf/
   mali-egl/libmali.so >/etc/OpenCL/vendors/mali.icd; apt-get install opencl-
   headers
   ```

   Listing A.40: Download and install OpenCL library for UNIX systems.

9. Install **Nvidia CUDA**. Follow the CUDA Installation Guide;
10. Install **VAAPI** packages:

```
            sudo apt-get install libva-dev libjpeg-dev
```

Listing A.41: Download and install VAAPI package for UNIX systems.

11. Install **OpenNi2** package:

```
            sudo apt-get install libopenni2-dev
```

Listing A.42: Download and install OpenNi2 package for UNIX systems.

12. Build **libfreenect2**:

```
            cd ..
            mkdir build && cd build
            cmake .. -DCMAKE_INSTALL_PREFIX=$HOME/freenect2
            make
            make install
```

Listing A.43: Build libfreenect2 package.

13. Set up **udev rules** for device access: sudo cp ../platform/linux/udev/90-kinect2.rules /etc/udev/rules.d/, then replug the Kinect;
14. Run the test program: ./bin/Protonect;
15. Run OpenNI2 test (optional): sudo apt-get install openni2-utils && sudo make install-openni2 && NiViewer2.

After building the **libfreenect2** package, head to root of your user account by doing $\$cd$ in a terminal window and then execute the following commands:

```
cd
sudo cp -r libfreenect2/build/libfreenect2/ /usr/local/include/
export LIBFREENECT2_INSTALL_PREFIX=~/usr/local/include/libfreenect2/
sudo cp -r freenect2/ /usr/local/include/
export LIBFREENECT2_INSTALL_PREFIX=~/usr/local/include/freenect2/include/libfreenect2
/
```

Listing A.44: Install pylibfreenect2 package.

Then edit the file called **setup.py** in pylibfreenect2 directory, more specifically line 16 to:

```
libfreenect2_install_prefix = os.environ.get("LIBFREENECT2_INSTALL_PREFIX", "/usr/
local/include/freenect2/")
```

Listing A.45: Line 16 setup.py of pylibfreenect2 - edited.

Finally, just run the setup.py by opening a terminal window on that directory and typing **sudo python3 setup.py install**.

A.6 Open-AI Gym SpaceInvadersv0

This section explains what might be called - Level 0 research on how QNET works - and some results obtained with this research.

A.6.1 Introduction

We have recently started researching the complex structure of QNET, resulting from investing time in constructing a neural network from scratch to work with a robotic hand and learning in-depth how a neural network applied to this concern actuates.

A.6.2 Methodology Used

We had to start somewhere, to learn how these neural networks work. Thus, we searched for tutorials from Open-AI [45] and tried the tutorial for an Atari Game [46].

A.6.3 Implementation

The implementation is divided into Deep Neural Network, Agent, and Main Loop.
Summarizing, the Deep Neural Network is a set of convolutional layers to process the tensor array containing the values of the image's frames; the Agent's class will handle the learning method, and the Main Loop will iterate through n epochs and compute the observations and scores for each iteration, with help from the other classes.

A.6.4 Results

Results were obtained under the following setup characteristics: AMD Ryzen7 2700X CPU, 32GB RAM, NVIDIA GTX1070 8GB and Ubuntu 18.04.5LTS [70]. This little experience obtained a mean score of 245 (indicating how well the agent is capable of learning from its errors), from two test runs and under the home made setup.

A.6.5 Conclusion

In conclusion, the neural network can learn over time. However, it was way easier and less time consuming to test in an home made setup than using Google Colab, since with Colab we are subject to the characteristics that it attributes to us.

A.7 YoLo Fine-tuning

The object detection model used, was prepared to detect cutlery objects on good quality images. However, the images we get from the simulation have very low quality, making hard for the detector to distinguish the objects. The following process was conducted using a custom pc build with a Geforce RTX 3060 (12GB VRAM) and 32GB of RAM, in order to fine-tune the model for object detection with simulation images. The steps taken to realize this process can be seen in this tutorial [32]:

1. Make sure the latest Torch, Torchvision, and PyYaml are installed. In our case, Torch and Torchvision are the developer version since we are using a fairly recent Graphics Processing Unit of 2021.

```
            pip3 install --pre torch==1.9.0.dev20210312+cu111 torchvision torchaudio
    -f https://download.pytorch.org/whl/nightly/cu111/torch_nightly.html

            pip3 install -U PyYAML==5.1

```

Listing A.46: Installing necessary packages

2. Clone YoLov5 repository and download configuration files;

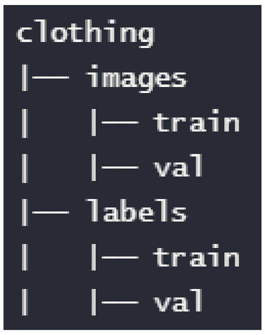
```
            git clone https://github.com/ultralytics/yolov5
            cd yolov5/
```

```
git checkout 0a09882e4f51d29c1739c18d409d5d1d5e9559f8

```

Listing A.47: Clone YoLo repository

3. In the YoLo root folder, we must have a folder called clothing containing the images to train the model and respective labels. To make the labels for each image we used a website called makesense [42]. It provides an option to export the labels to YoLo format.



```
clothing
|── images
|   |── train
|   |── val
|── labels
|   |── train
|   |── val
```

Figure A.3: Division of the dataset used to train YoLo on the images from simulation

4. To train the YoLo model on a custom dataset, the following command line is needed. Pay attention that we are training during 300 epochs, since this model was firstly trained with that number of epochs before public release.

```
python3 train.py --img 640 --batch 4 --epochs 300 --data ./data/clothing.yaml --cfg ./models/yolov5x.yaml --weights yolov5x.pt --name yolov5x_clothing --cache

```

Listing A.48: Fine Tuning YoLo

5. Finally, to test the newly trained model with TTA, we just need to run the following command line on a new terminal. When using TTA, by standards we increase the size of the images by 30%.

```
python3 test.py --weights ./runs/train/folder_to_trained_model/weights/best.pt --data ./data/clothing.yaml --img 832 --augment

```

Listing A.49: Fine Tuning YoLo

Appendix B

Glossary

B.1 Convolutional Neural Network

A CNN [9] is an algorithm from **Deep Learning** that receives an image as input, designates weights and biases to specific objects/pixels in the image and is able to differentiate them.
In CNN context, an image is nothing more than pixels represented by *width*, *height* and *3 color channels*. Thus, when an image is to be processed by a convolutional network, it is represented by three color panes (**Red**, **Green** and **Blue**) as shown in figure B.1.

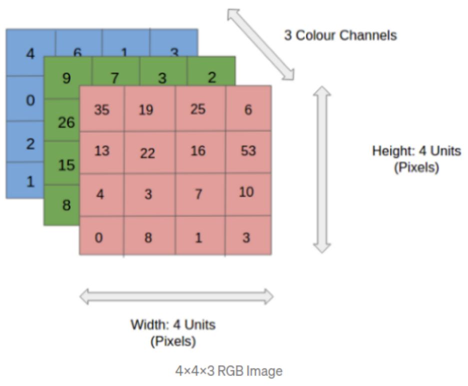


Figure B.1: RGB Image representation, from [9].

A CNN has some characteristics that make her good for problems where processing images, without these loosing features, is important such as in Image Classification, Natural Language Processing, etc. These characteristics are:

1. **Kernel** [9] - Let us say that we have an image with 5x5x1 dimensions. With the application of the kernel/filter k with dimensions 3x3x1, we obtain a convolved feature (Figure B.2);
2. **Stride Length** [9] - Distance according to which the kernel will shift. Example: Stride=1, then kernel will shift 1 to the right.
   Two results are upcoming from the extraction of features: either it has the **Same Padding** - meaning if the original image was augmented, when applying a kernel with same dimensions

$$\begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\\1 & 0 & 1\end{bmatrix} => \begin{bmatrix}1 & 1 & 1 & 0 & 0\\0 & 1 & 1 & 1 & 0\\0 & 0 & 1 & 1 & 1\\0 & 0 & 1 & 1 & 0\\0 & 1 & 1 & 0 & 0\end{bmatrix} = \begin{bmatrix}4 & 3 & 4\\2 & 4 & 3\\2 & 3 & 4\end{bmatrix}$$

Figure B.2: Application of kernel k 3x3x1 to image features 5x5x1, from [9].

as the non-augmented image, it will result in a feature with the same dimensions of the non-augmented image - or **Valid Padding** - meaning when a kernel is applied to a non-augmented image, we get a feature with same dimensions as of the kernel's;

3. **Pooling Layer** [9] - Reducing spatial size of the recently obtained feature from the application of the kernel. There are two ways of doing this: **Max Pooling** - for each iteration of the filter, retrieves the maximum value - or **Average Pooling** - for each iteration of the filter compute average of values.

B.2 Fully Convolutional Network

A FCN [27] transforms image pixels into categories using a CNN. It also transforms the dimensions of the feature map, in the intermediate layer, to the size of the image received as input, with the help of a transposed convolution layer.

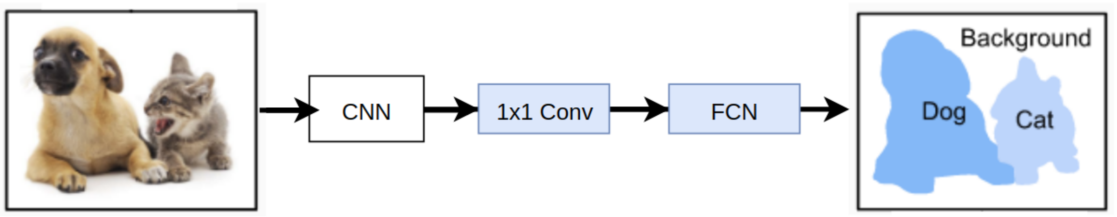


Figure B.3: FCN transforming pixels into categories, from [27].

B.3 Long Short-Term Memory

The LSTM [40] was born from the necessity to address **long-term** information preservation alongside with skipping **short-term** input.

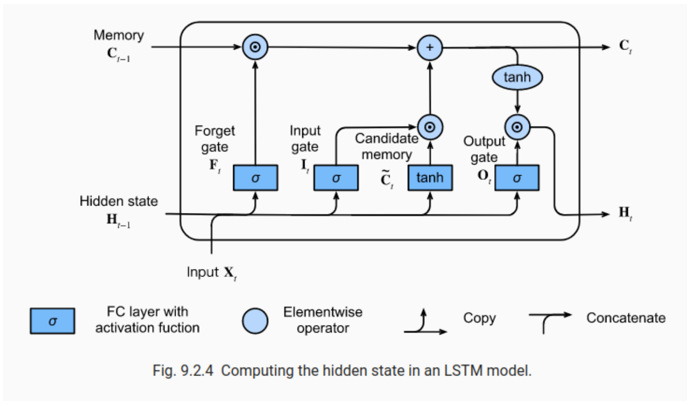


Figure B.4: LSTM representation, from [40].

The components of an LSTM can be described as follows:

1. **Input Gate, Forget Gate, and Output Gate** [40]:
   Input Gate: Gate that decides when data is to be read to the cell;

Output Gate: This gate decides when the data is to be retrieved from a cell;
Forget Gate: This mechanism is in charge of resetting the cell's memory.

2. **Candidate Memory** [40]: Similar computation to that present on the last three cells, but with an activation function *tanh*;
3. **Memory Cell** [40]: This cell captures long range values present in sequences of values;
4. **Hidden State** [41]: Keeps data from previous state, that neural network has seen before.

B.4 Fully Connected LSTM

A FCLSTM [26] is an LSTM followed by a fully connected layer that will process the output of the LSTM and gives an output according to the used loss functions or targets.

B.5 Deep Q-Network

QNET [57] is a reinforcement learning algorithm that evaluates and learns from the actions that makes, informing the agent which action, under the circumstances, is better. It is mostly used in problems with stochastic transitions and rewards. Thus, for problems resembling a Markov Decision Process, this algorithm uses an optimal policy to maximize the reward's expected value, storing the computed values in tables. However, this kind of algorithm may have some flaws, like when we have a bigger set of state/actions, the probability of the algorithm visit a specific state is smaller.

B.6 U-Net Convolutional Network

The architecture of a UNET [69] is essentially a FCN that was modified to train with less images on the field of image segmentation. As the name says, the network architecture is a **U** shape consisting of repeated convolutions followed by an activation function **ReLU** (each) and **max pooling**.

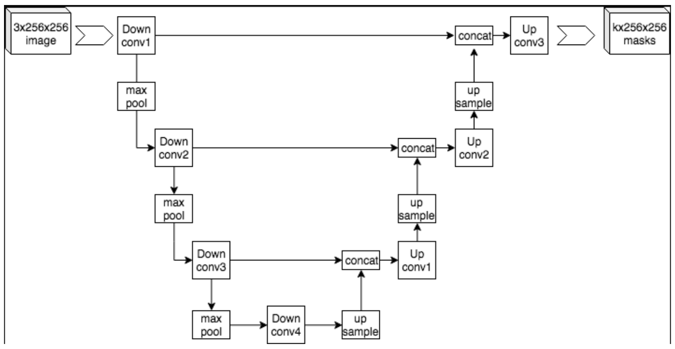


Figure B.5: UNet representation, from [69].

B.7 What characterizes a good change in environment?

A change in the robot's environment can be induced by a push in one or more objects present in the scene or by a successful grasp. Thus, when we refer to a good change in the environment, we are saying that the environment changed from the previous state to the current, employing a push that altered the location of one or more objects or by a grasp that removed one object from the scene.

B.8 What is a good heightmap?

An heightmap is an image that contains color regions and is used to store values about the elevation of a surface. These color regions vary from dark-blue color to dark-red color, which can be seen as -1 for a dark-blue color and 1 to a dark-red color. The closer to a dark-red color is a region, the better will be the grasp represented in that area.

B.9 What is a Cross Stage Partial Network?

CSPs are new networks proposed to mitigate a known problem of some of the computer vision's networks. This problem is related to the fact that these networks would require high inference computations. Therefore, the newly proposed network architecture aims to reduce the number of computations required by 20% [13].

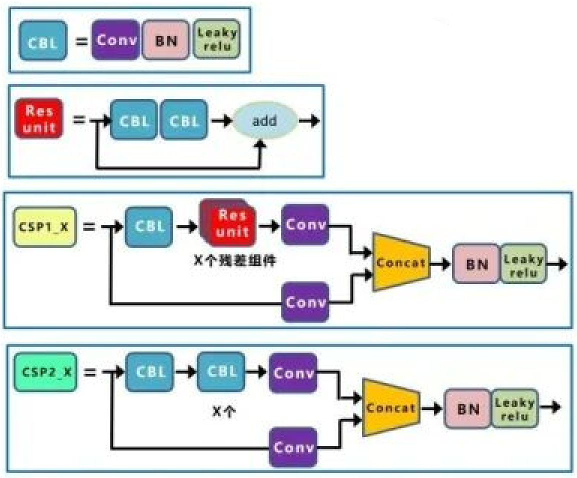


Figure B.6: CSP representation, from [82].

As figure B.6 shows the CSP receives the feature map from another network and partitions it into 2 parts.

1. The first part passes through a single convolutional layer;
2. The second part passes through two CBL layers or a CBL layer and a Res Unit, followed by a final convolutional layer to consolidate the features of the map.
   The Res Unit separated the received input into two parts. The first part passes through two CBL layers and will then be added to the second partition of the input, creating new and different features on the inputted feature map.

The two feature maps are then concatenated and will suffer a batch normalization with Leaky-ReLu as an activation function.

B.10 What are Path Aggregation Networks? [49]

PANet is designed to improve the flow of the information through the neural network.
This kind of network work in a similar way of the SPP network when extracting features from an image

to then reconstructed the feature map, however it involves more steps as shown in figure B.7.

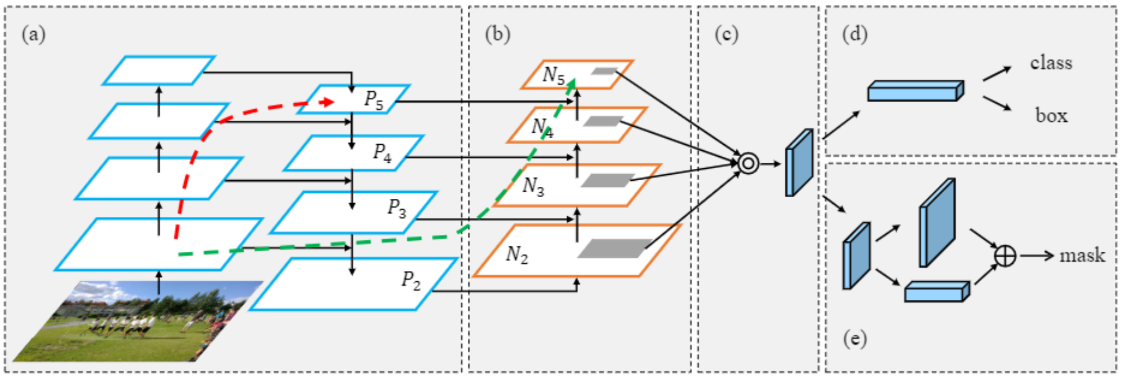


Figure B.7: PANet representation, from [49].

We can explain each step in figure B.7 as follows:

1. In (a), the network extracts features from the image using a bottom-up architecture. Then, it creates feature levels P5 to P2 using a top-down architecture. These feature levels have the same spatial size as that present in the bottom-up approach that extracts features from the image;
2. In (b), the network applies a bottom-up approach that augments the feature levels of P5 to P2 to create feature maps N2 to N5 with even richer features. To create the N_i map is used the previous feature map N_{i-1} and the feature level P_i;
3. In (c), the network retrieves features given the number of levels used for extracting features and creating the features maps.

Therefore, from the figure B.8 let's say we have four levels of extraction. The network would extract only the maximum of the features from each level and fuse (e) them along with the respective box and class of each feature (d) to create a better feature map.

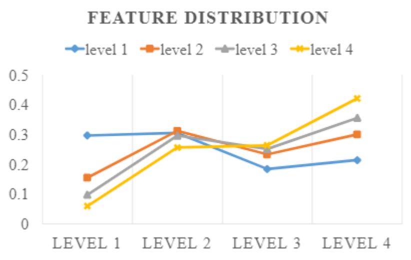


Figure B.8: Adaptive feature pooling (c), from [49].

B.11 What is a Spatial Pyramid Pooling?

The SPP is a layer that pools all the received features and generates outputs with the same length, and in total have the the same as the input. This layer is used in computer vision since it is a robust method for working on features from images with objects that have deformations [62].

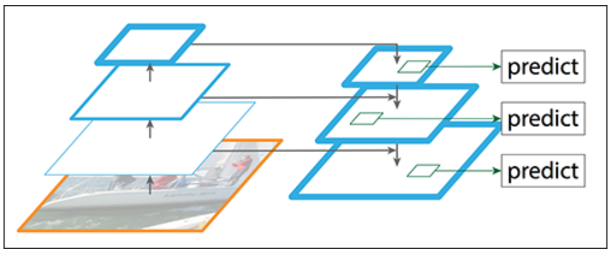


Figure B.9: SPP representation, from [62].

In figure B.9 is a representation of SPP layer which has a similar way of working of PANet. We can say that the SPP has a pyramid structure with two ways of functioning: one is a bottom-up where it extracts features from the image and the other is a top-down pathway which grabs the best features with higher semantic value from each level and reconstructs the best prediction with higher resolution [62]. It is important to refer that, higher the level of the features higher is the semantic value of them and less resolution they will have.

B.12 What are the Euler Angles? [23]

Formulated by Leornard Euler, the Euler Angles are three angles that describe the orientation of a rigid body. This orientation is given with respect to a three-dimensional fixed coordinate system. The following image represents the classic Euler Angles [α, β, γ].

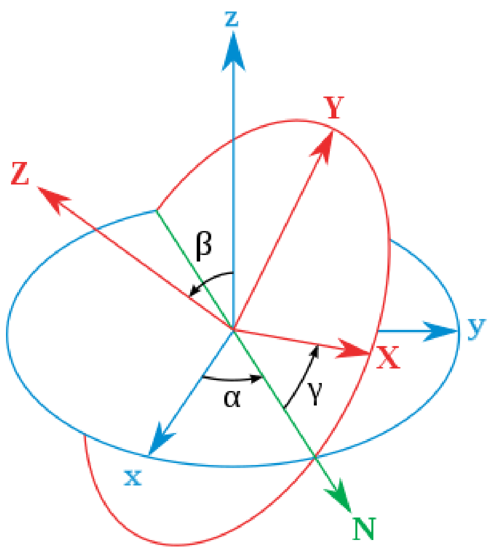


Figure B.10: Euler Angles representation, from [23].

Bibliography

[1] About yolov5. [online]. URL: https://www.programmersought.com/article/20545272716/ [cited 02nd April 2021]. 11

[2] About yolov5. [online]. URL: https://pub.towardsai.net/yolo-v5-explained-and-demystified-4e4719891d69 [cited 02nd April 2021]. 11

[3] About yolov5 - pytorch. [online]. URL: https://pytorch.org/hub/ultralytics_yolov5/ [cited 02nd April 2021]. 11

[4] Advances in robotics (review) [online]. URL: https://www.researchgate.net/publication/225533120_Advances_in_robotics_Review [cited 09th December 2020]. 2

[5] Beer cup - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/beer-glass-3d-c4d/714148 [cited 25th May 2021]. 54

[6] Blender application. [online]. URL: https://www.blender.org/ [cited 25th May 2021]. 52

[7] Blender solidify modifier. [online]. URL: https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/solidify.htm [cited 25th May 2021]. 53

[8] Cartoon low poly 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/3d-design-illustration-browser-model-1373493 [cited 25th May 2021]. 54

[9] Cnn explained [online]. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 [cited 25th December 2020]. xiii, 79, 80

[10] Coco dataset - object recognition dataset [online]. URL: https://cocodataset.org/#home [cited 02nd April 2021]. 11

[11] Collaborative robots: Frontiers of current literature [online]. URL: https://www.researchgate.net/publication/342347381_Collaborative_Robots_Frontiers_of_Current_Literature [cited 09th December 2020]. 2

[12] Coppeliasim supported file formats for mesh import. [online]. URL: https://mde.tw/copsim/content/importExport.htm [cited 25th May 2021]. 52

[13] Cross stage partial network. [online]. URL: https://openaccess.thecvf.com/content_CVPRW_2020/papers/w28/Wang_CSPNet_A_New_Backbone_That_Can_Enhance_Learning_Capability_of_CVPRW_2020_paper.pdf [cited 14th May 2021]. 82

[14] Crystal cup - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/free-crystal-glass-3d-model/781348 [cited 25th May 2021]. 54

[15] Deep learning for detecting robotic grasps [online]. URL: https://arxiv.org/pdf/1301.3592.pdf [cited 08th June 2021]. 17

[16] Deep reinforcement learning for 3d-based object grasping [online]. URL: https://arxiv.org/pdf/1803.09956.pdf [cited 9th October 2020]. xii, 19, 23, 27, 28, 31, 36, 54

[17] Deep reinforcement learning for robotic pushing and picking in cluttered environment [online]. URL: https://ieeexplore.ieee.org/document/8967899 [cited 16th December 2020]. xii, 10, 23, 24, 74

[18] Deep reinforcement learning for robotic pushing and picking in cluttered environment - github repository [online]. URL: https://github.com/weiyx16/Active-Perception [cited 29th December 2020]. 25, 74

[19] Deep visual foresight for planning robot motion, 2017 [online]. URL: https://arxiv.org/abs/1610.00696 [cited 21st October 2020]. 14, 15, 25

[20] Deep visual foresight for planning robot motion approach on github [online]. URL: https://github.com/m-serra/action-inference-for-video-prediction-benchmarking [cited 12th February 2021]. 25

[21] Deepin os linux distribution [online]. URL: https://www.deepin.org/en/ [cited 7th October 2020]. 73

[22] Drink glass - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/blender-glass-3ds/862051 [cited 25th May 2021]. 54

[23] Euler angles - simple explanation. [online]. URL: https://en.wikipedia.org/wiki/Euler_angles [cited 28th May 2021]. xi, xiv, 84

[24] Evolution of industrial robots [online]. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.413.8024&rep=rep1&type=pdf [cited 07th December 2020]. 1

[25] F-measure: Precision and recall. [online]. URL: https://onlinehelp.explorance.com/blueml/Content/articles/getstarted/mlcalculations.htm?TocPath=Get%20started%7C_____3 [cited 23rd April 2021]. xii, 13

[26] Fclstm in pytorch explained [online]. URL: https://towardsdatascience.com/lstms-in-pytorch-528b0440244 [cited 26th December 2020]. 81

[27] Fcn explained [online]. URL: http://d2l.ai/chapter_computer-vision/fcn.html [cited 26th December 2020]. xiii, 80

[28] Future trends in marine robotics [online]. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7059360 [cited 08th December 2020]. 2

[29] Glass cup - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/free-glass-3d-model/444520 [cited 25th May 2021]. 54

[30] Glass cup (2) - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/c4d-glass-modeled-lighting/714145 [cited 25th May 2021]. 54

[31] Glass of water - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/glass-blender-eevee-3d-model-1642287 [cited 25th May 2021]. 54

[32] Google colab tutorial on yolov5 fine tuning [online]. URL: https://youtu.be/XNRzZkZ-Byg [cited 02nd April 2021]. 77

[33] Grasp pose detection in point clouds [online]. URL: https://arxiv.org/pdf/1706.09911v1.pdf [cited 04th June 2021]. 10

[34] Ian goodfellow and yoshua bengio and aaron courville. 2016. machine learning basics. [online]. URL: http://www.deeplearningbook.org/contents/ml.html [cited 02nd June 2021]. 5, 6, 7, 8

[35] Industrial robotic solutions for interventional medicine [online]. URL: https://www.researchgate.net/profile/Tamas_Haidegger/publication/254992519_INDUSTRIAL_ROBOTIC_SOLUTIONS_FOR_INTERVENTIONAL_MEDICINE/links/0deec53b23afa6664c000000/INDUSTRIAL-ROBOTIC-SOLUTIONS-FOR-INTERVENTIONAL-MEDICINE.pdf [cited 08th December 2020]. 2

[36] Install nvidia cuda guide [online]. URL: https://developer.nvidia.com/cuda-downloads?target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=2004&target_type=debnetwork [cited 7th October 2020]. 73

[37] Intersection over union for object detection. [online]. URL: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ [cited 23rd April 2021]. xii, 13

[38] Learning efficient push and grasp policy in a totebox from simulation, 2020 [online]. URL: https://www.tandfonline.com/doi/full/10.1080/01691864.2020.1757504 [cited 07th November 2020]. xii, 10, 11, 21, 22, 25

[39] libfreenect2 installation tutorial [online]. URL: https://github.com/OpenKinect/libfreenect2 [cited 22th December 2020]. 75

[40] Lstm explained [online]. URL: http://d2l.ai/chapter_recurrent-modern/lstm.html?highlight=long%20short%20term%20memory [cited 26th December 2020]. xiii, 80, 81

[41] Lstm hidden state explained [online]. URL: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 [cited 26th December 2020]. 81

[42] Make sense - website to create labels for images [online]. URL: https://www.makesense.ai/ [cited 23rd April 2021]. 78

[43] Market for professional and domestic service robots booms in 2018 [online]. URL: https://ifr.org/post/market-for-professional-and-domestic-service-robots-booms-in-2018 [cited 08th December 2020]. 1

[44] Mean average precision [online]. URL: https://arxiv.org/pdf/1607.03476.pdf [cited 12th June 2021]. 12

[45] Open-ai gym [online]. URL: https://gym.openai.com/ [cited 09th January 2021]. 77

[46] Open-ai gym atari tutorial, phil tabor [online]. URL: https://www.youtube.com/watch?v=RfNxXlO6BiA [cited 09th January 2021]. 77

[47] Open3d - modern 3d data processing library. [online]. URL: http://www.open3d.org/docs/0.11.0/python_api/open3d.visualization.html [cited 13th May 2021]. 55

[48] Paper cup - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/free-paper-cup-3d-model/460068 [cited 25th May 2021]. 54

[49] Path aggregation networks. [online]. URL: https://arxiv.org/pdf/1803.01534.pdf [cited 14th May 2021]. xi, xiii, xiv, 82, 83

[50] Photoscape editor [online]. URL: https://www.photoscapeonline.com/ [cited 10th June 2021]. xii, xiii, 20, 28, 32, 43, 44, 48

[51] Pointnet++ grasping approach available on github [online]. URL: https://github.com/pyni/PointNet2_Grasping_Data_Part [cited 12th February 2021]. 25

[52] Pointnet++ grasping: Learning an end-to-end spatial grasp generation algorithm from sparse point clouds, 2020 [online]. URL: https://arxiv.org/pdf/2003.09644.pdf [cited 15th January 2021]. 10, 19, 25

[53] Pointnetgpd approach available on github [online]. URL: https://github.com/lianghongzhuo/PointNetGPD [cited 12th February 2021]. 25

[54] Pointnetgpd: Detecting grasp configurations from point sets, 2019 [online]. URL: https://arxiv.org/abs/1809.06267 [cited 25th November 2020]. 17, 19, 25

[55] Preparations for nvidia cuda [online]. URL: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html [cited 7th October 2020]. 70, 73

[56] pytorch installation on ubuntu [online]. URL: https://pytorch.org/ [cited 7th October 2020]. xi, 73, 74

[57] Q-learning in reinforcement learning [online]. URL: https://en.wikipedia.org/wiki/Q-learning#Algorithm [cited 26th December 2020]. 81

[58] Richard s. sutton and andrew g. barto. 2018. reinforcement learning [online]. URL: http://incompleteideas.net/book/the-book-2nd.html [cited 03rd June 2021]. 7, 8, 9, 10

[59] Robot simulator coppeliasim [online]. URL: https://www.coppeliarobotics.com/downloads [cited 9th October 2020]. 36, 45, 46, 47, 49, 52, 53, 54, 72, 74

[60] Ros - installation tutorial [online]. URL: https://jderobot.github.io/BehaviorStudio/install/ros_noetic#noetic [cited 8th October 2020]. 36, 73

[61] Shot cup - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/free-obj-mode-shot-glass/581454 [cited 25th May 2021]. 54

[62] Spatial pooling pyramid layer. [online]. URL: https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c [cited 14th May 2021]. xiv, 83, 84

[63] A survey in space robotics [online]. URL: https://ntrs.nasa.gov/citations/20030054507 [cited 10th December 2020]. 2

[64] Survey on human-robot collaboration [online]. URL: https://www.sciencedirect.com/science/article/pii/S0957415818300321?via%3Dihub [cited 08th December 2020]. 1

[65] Torch api - machine learning library. [online]. URL: https://pytorch.org/ [cited 13th May 2021]. 55

[66] Torchvision api - torch pachage for 3d data transformations. [online]. URL: https://pytorch.org/vision/stable/index.html [cited 13th May 2021]. 55

[67] Towards data science - mean average precision explained. [online]. URL: https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52 [cited 23rd April 2021]. 12

[68] Turbosquid website for 3d models. [online]. URL: https://www.turbosquid.com/ [cited 21st May 2021]. xiii, xv, 38, 39, 48, 50, 51, 52, 54, 55, 57, 58, 60, 61, 62, 63

[69] U-net - another type of convolutional neural network [online]. URL: https://en.wikipedia.org/wiki/U-Net [cited 26th December 2020]. xiii, 81

[70] Ubuntu 18.04.5 bionic beaver [online]. URL: https://releases.ubuntu.com/18.04/ [cited 7th October 2020]. 54, 72, 73, 77

[71] Unsupervised learning for physical interaction through video prediction, 2016 [online]. URL: https://arxiv.org/pdf/1605.07157.pdf [cited 29th January 2021]. 16, 17, 25

[72] Unsupervised learning for physical interaction through video prediction webpage [online]. URL: https://sites.google.com/site/robotprediction/ [cited 12th February 2021]. 25

[73] Ur3 robotic arm [online]. URL: UR3:https://www.universal-robots.com/pt/produtos/ur3-robot/ [cited 11th December 2020]. 4, 36, 47

[74] Using geometry to detect grasps in 3d point clouds [online]. URL: https://arxiv.org/pdf/1501.03100.pdf [cited 08th June 2021]. 17

[75] Visual pushing-for-grasping model available at github [online]. URL: https://github.com/andyzeng/visual-pushing-grasping [cited 12th February 2021]. xii, 25, 27, 34, 36, 37, 38, 46

[76] Vodka shot - 3d model cup from turbosquid [online]. URL: https://www.turbosquid.com/3d-models/3d-shot-vodka/911923 [cited 25th May 2021]. 54

[77] Whiskey cup - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/whiskey-glass-3ds-free/879008 [cited 25th May 2021]. 54

[78] Whiskey cup (2) - 3d model from turbosquid. [online]. URL: https://www.turbosquid.com/3d-models/runner-whiskey-glass-3ds-free/879240 [cited 25th May 2021]. 54

[79] Ycb dataset - dataset containing cutlery objects. [online]. URL: http://ycb-benchmarks.s3-website-us-east-1.amazonaws.com/ [cited 24th February 2021]. xv, 37, 50, 51, 52, 54, 55, 56, 59, 60, 62, 63

[80] Ycb dataset - kitchen objects. [online]. URL: https://www.ri.cmu.edu/pub_files/2015/8/ycb_journal_v26.pdf [cited 19th March 2021]. xii, 38

[81] Yolov4 structure [online]. URL: https://arxiv.org/pdf/2004.10934.pdf [cited 12th June 2021]. xii, 11

[82] Yolov5 - object detector model. [online]. URL: https://github.com/ultralytics/yolov5 [cited 02nd April 2021]. xii, xiii, 12, 35, 48, 82