BACHELOR'S DEGREE THESIS

Degree in Mathematics

Title: Machine Learning for Robotic Manipulation in Cluttered Environments

Author: Ferran Alet Puig

Advisor: Alberto Rodriguez (MIT) and Maria Alberich (UPC)

Department: Mechincal Engineering Department at MIT and Mathematics Department at UPC

Academic year: 2015-2016

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Mathematics and Engineering Physics Bachelors Thesis

MACHINE LEARNING FOR ROBOTIC MANIPULATION IN CLUTTERED ENVIRONMENTS

Author: Ferran Alet Puig
Supervisors: Alberto Rodríguez (MIT)
Maria Alberich Carramiñana (UPC)

Facultat de Matemàtiques i Estadística
Universitat Politècnica de Catalunya
Barcelona, May 2016

Contents

List of Figures

Abbreviations

AI	Artificial Intelligence
APC	Amazon Picking Challenge
CNN	Convolutional Neural Network
DL	Deep Learning
LSTM	Long-Short Term Memory (type of NN)
ML	Machine Learning
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
NN	Neural Network
PCA	Principal Component Analysis
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning
RRT	Rapidly exploring Random Trees
SL	Supervised Learning
UL	Unsupervised Learning

Abstract

Machine Learning and Robotics have experienced an enormous growth in the recent years; they are set to dramatically change our lives and are already starting to do so. Despite their growth and success, robots are still completely incapable of performing manipulation at a human-level. Achieving this would have major implications for the retail and logistic industries and would be a key step towards the human dream of having robots in our homes. This thesis proposes a solution for two major issues that robots have trouble facing: clutter and lack of structure: in a warehouse objects are stuffed in tight spaces and one cannot pick an object without taking others into account.

In this thesis we focus on a particular application: designing the planner for MIT's entry in the Amazon Picking Challenge, a robotic competition aiming at pushing the frontiers of manipulation until robots can substitute human pickers in warehouses. Given a set of manipulation primitives (such as grasping, suction, scooping, placing or pushing) we designed a system capable of learning a planner from a set of manipulation experiments. After learning, given any configuration of objects, the planner can come up with the optimal sequence of primitives applied to any object on the scene so as to maximize the probability of successfully picking the goal object.

In doing this research we have analyzed Reinforcement Learning, Deep Learning and Planning approaches. For each one, we first describe the background theory, characterizing it for our application to robotics. Then we describe a prototype done in the area and the lessons learned from it. Finally, we combine the strengths of all the areas to create the final design of our system.

We are confident this work will lead to great improvements in the performance of our system in the Amazon Picking Challenge and can give insights into the difficulties and advantages of applying Machine Learning to Robotics.

Preface

I have been able to write this memoir thanks to a collaboration between the Center of Interdisciplinary Education (CFIS), the Polytechnic University of Catalonia (UPC) and the Massachusetts Institute of Technology (MIT). I have been conducting my research from October 2015 to July 2016 at the MCube Lab, led by Professor Alberto Rodriguez. Professor Maria Alberich has been my contact at UPC and has revised all my work.

My research has been divided in three parts. First, designing a Machine Learning intelligence that will control our program in the Amazon Picking Challenge, in which this thesis focuses. Second, implementing several manipulation primitives for the aforementioned contest. Finally, improving the state-of-the-art in Heteroscedastic Gaussian Processes.

The research presented in this work will have a direct application on MIT's participation in the Amazon Picking Challenge (in which they placed 2nd last year), being the main program that will control which actions are performed. Moreover, we are confident that, after all learning experiments are finished, we will be able to publish a paper summarizing this thesis for ICRA 2017.

Given the broad spectrum of my research problem, and the liberty my advisor gave me; the hardest challenge was deciding which paths were worth pursuing and, more importantly, which were not. Moreover, having to combine theoretical research with an engineering focus has given me a new view of Machine Learning and research in general. All in all, I believe this work has been an excellent prelude for my future PhD, which will be conducted at the Computer Science department of MIT.

Acknowledgements

I feel very lucky of having worked with Professor Alberto Rodriguez and would like to thank him for our weekly meetings, providing thoughtful advice while giving me space and confidence to develop my own ideas and making me feel at home in the lab.

I would also like to thank Professor Maria Alberich for carefully going through the manuscript, providing great suggestions and discussing it through Skype.

This thesis has also benefited greatly from insightful conversations with MIT professors Tomas Lozano-Perez and Leslie Kaelbling. Moreover, my learning curve would have been much flatter had it not been for the help of my friends at the MCube lab.

Without CFIS, the people behind the organization and the donors, I wouldn't have had this opportunity; I will forever feel indebted to this community.

Finally, because good results can't and shouldn't come at the expense of happiness, I would like to thank my girlfriend, my family and my flatmates for providing plenty of it.

1 | Introduction

1.1 Motivations

1.1.1 Technologies with great impact

At present, *Machine Learning* and *Robotics* are two of the areas with the highest potential impact on society in the few decades to come. Both AI and Robotics have received a high amount of expectations since several decades ago, but it hasn't been until very recently that those expectations have been within our reach. In the last few years, several improvements in neighbour technologies have solved some of the basic needs for ML and Robotics to go from isolated successes to mainstream technologies.

We are already experiencing them: Google is using Machine Learning in most of its core products (from predicting the next YouTube video a user wants to see, to selecting the best ad), Facebook uses ML to suggest possible tags of your friends in the photos you upload and Amazon recommends products you could buy based on people with similar interests. Similarly, in robotics, in a conference at MIT Chris Urmson (director of Google self-driving cars) suggested they are less than 5-10 years away from being on the market. Amazon is already using Kiva robots in its warehouses and Boston Dynamics recently showed great progress in humanoid robots.

In the case of Machine Learning (ML), wide spreading of sensors and moving our lives online have massively increased the amount of data that we generate: *The Economist* predicts an increase of 20-40% year over year in the amount of data produced, now creating 10^9 Gigabytes of information every year (*30*). Similarly, storing costs have plummeted (38% every year according to a study made by BSA (*29*), now at 3 cents per Gigabytes. Moreover, exponential improvements in processing power (following the famous *Moore's Law*) allow us to process this increasing amount of data.

As mentioned by the current director of Google Research, Peter Norvig, in a famous conference (*The Unreasonable Effectiveness of Data*, figure 1.1) we can now work with data on orders of magnitude big enough to create proper models. And that makes all the difference. The basic Deep Learning algorithms were created in the 80s, but it wasn't until the last 10 years that we have had enough computing power and data to make them much better than standard non-learning algorithms.

Robotics has also profited immensely from the drop and reduced size of computing power, as well as great reduction costs in batteries and actuators (electric motors). This drop in actuators' cost is the same basis for the wide spreading of 3D printers, that have gone from very rare to common in the last 5 years. Moreover, there's also been an improvement in sensing technology such as 3D cameras (XBox' Kinect being the most famous example) that have a direct impact on robot's performance.

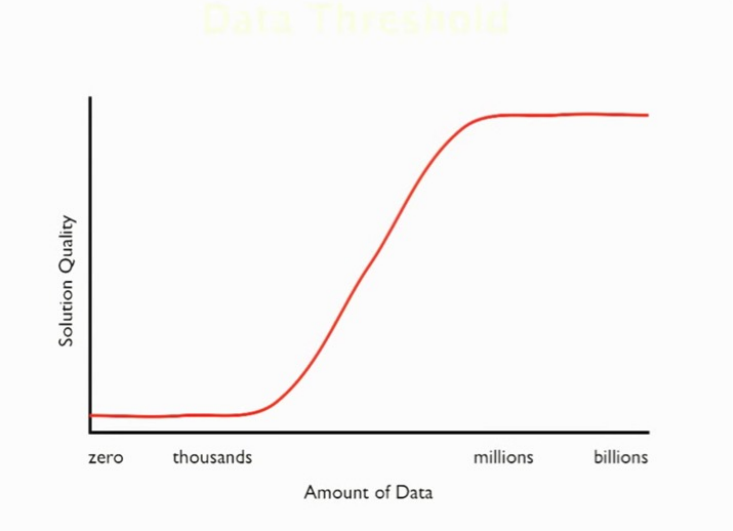

Figure 1.1: Performance vs. data (from *The Unreasonable effectiveness of Data*)

1.1.2 Personal motivations

Beyond their impact, the author had multiple reasons for pursuing his thesis in ML and Robotics, from which we will highlight three:

- **Multidisciplinary**: Both ML and Robotics are very interdisciplinary projects, the former combining Mathematics and Computer Science and the latter a combination of those two with Physics. Having done two degrees, in Mathematics and Engineering Physics, while doing Algorithmic competitions in college; combining the three areas was a promising idea.
- **Increase in overall efficiency**: ML and Robotics promise to deliver immediate impact. Even more, going further, they affect the efficiency of humanity as a whole, which makes them orders of magnitude more important. Making an improvement in car's fuel efficiency can save millions of dollars and have a big impact on Climate Change. However, creating intelligent beings (both virtual and physical) could potentially have impact on all the challenges humanity faces. Increasing efficiency instead of making a one-time improvement is analogous to the difference between increasing the value of a function and the value of its derivative: even if the change of derivative is smaller, it will eventually be much more important.
- **Understanding ourselves**: one of the most important philosophical and scientific questions that remain unsolved is how intelligence arises. Several scientific communities believe that understanding the brain and making intelligent machines are two goals that should go hand in hand because progress, and failures, in one of these areas can help better understand the other.

1.2 Structure

The primary goal of this research was to find a good application using both Machine Learning(ML) and Robotics. To do this, we spent most of the time doing background

research and then focusing on an application towards the end. We will follow this background research explaining the concepts, while describing the learned lessons of trying to apply them to Robotics and to our particular application.

There are a lot of ML and Artificial Intelligence(AI) techniques, and, after trying to apply several of them, some of them are applicable to our case, but some aren't. Differentiating between those and understanding the reason behind the success of several methods has been the bulk of this work.

The written thesis will follow the same structure as the research itself: each chapter explains an area of AI we explored, first going through the theory, while making additional comments for its application to robotic manipulation, then explaining the prototype we did in that area and what things we learned by doing that prototype. The chapters following this structure are: Reinforcement Learning, Deep Learning and Planning. In the final chapter we explain how we combined the ideas from all the previous chapters in our final system.

To do the background research we relied on papers and books, but also profited from the opportunity of talking with leading researchers in the field. Therefore, in some cases, we will only cite the name of the researcher. We also want to apology for the presentation of a few figures; the computer in which we had the data and codes from a few experiments crashed; thus, a couple graphs couldn't be rebuilt to improve their appearance.

1.3 Scope of this thesis

One of the main intentions when writing this memoir was that it was readable and pertinent to a Final Degree Project. There's no point in trying to write 200 pages if nobody will enjoy reading them. To achieve this goal we made a few decisions:

This memoir only contains the *basic* theoretical ideas on each topic, not the actual acquired knowledge. A lot of information has been skipped in several ways:

- **Omitting several concepts** that were not fundamental to understand the final prototype or that area of research. Some examples are the geometric algorithms to compute the features, algorithms for scene generation, research done in recursive Neural Networks, LSTMs and Neural Turing Machines, Gaussian Processes for 1-object scenes, other pure planning algorithms such as Rapidly exploring Random Trees (RRTs), multiple general Machine Learning concepts and several others.
- **Simplifying and summarizing some concepts** worth weeks of work such as: the final planning algorithm, the exploration-exploitation system of the Reinforcement Learning prototype, Guided Policy Search and others.

Omitting the descriptions of the code implementations, which represent 50% **of the work done.** A significant part of the research consisted in coding prototypes, physics simulations, robotic manipulation primitives, ML systems, etc. Most of this work is not covered, or even mentioned at times, but this doesn't mean it wasn't useful for our research. We decided not to cover it in the written thesis because it could get in the way of explaining the theory behind the implementations.

We assume the reader has a good mathematical background. This thesis required using topics from Computer Science(CS), Mathematics and Physics. To narrow its scope we will assume the reader already has a good mathematical background (Bachelor's or equivalent) and notions of CS and Physics.

Not describing the technologies involved. In this thesis a diverse set of tools, most of which the author didn't know beforehand and thus, significant effort went into learning them. Here's a short-list:

- *Robot Operating System (ROS)*: system/language to handle the real robot.
- *MuJoCo*: physics simulator used for the Deep Reinforcement Learning prototype.
- Programming Languages:
  - *Python*: to code prototypes and everything involving the real robot.
  - *C++*: to code simulations using MuJoCo.
  - *Matlab*: to code the RL prototype and MuJoCo.
- *Google Tensorflow*: Python API to build Neural Networks and other Machine Learning systems.

Not mentioning other research done in the lab: we focused the written thesis on the main topic of the author's research stay, thus omitting research done in Heteroscedastic Gaussian Processes, particle filters and other tasks related to the Amazon Picking Challenge.

2 | Preliminaries

2.1 Amazon Picking Challenge

The Amazon Picking Challenge is a manipulation competition which started in 2015 (*36*). Its goal is to advance the automation of warehouse tasks in semi-structured environments. There are currently robots that can pick and place objects with high speed and accuracy, but they are limited to well-structured environments such as a pharmacy, where most packages are boxes of similar size of which the robot knows their exact location. The APC contest aims to bring robots into a more unstructured environment such as Amazon.com's warehouses.

As of 2016, Amazon has Kiva robots deployed in several warehouses moving shelves to a human that picks the desired object from the shelve and puts it in a box without having to move. Using the Kiva mover robots substantially decreases errors, time, complexity and ultimately costs. Replacing the human pickers for robot pickers could bring a similar revolution, bringing down costs substantially.

In the contest, the robot receives a json file containing a list of all the objects in every bin of a normal Amazon shelf (specific positions and orientations for the objects have to be determined by the robot) and a goal object to pick from every shelf. The robot has 10 minutes to autonomously pick as many goal objects as it can.

In its first edition, the APC asked participants to retrieve objects in a very uncluttered environment, meaning that one didn't need to care about colliding with other objects when trying to pick the goal object. However, real environments are cluttered, and so are real Amazon shelves. Following this logic, a major difficulty added for 2016 has been clutter. Figure 2.1 is a comparison of a test shelf on 2015 and 2016.

Moreover, table 2.1 shows quantitatively the increase in clutter for 2016.

	2015	2016
1-obj bins	4	2
2-obj bins	4	3
3-obj bins	4	0
4-obj bins	0	3
6-obj bins	0	2
8-obj bins	0	1
10-obj bins	0	1
Avg. # of obj	2	4.2

Table 2.1: Numerical clutter comparison

(a) 2015 **(b)** 2016

Figure 2.1: Visual comparison of tasks

Notice that bins with ≥ 4 objects will inevitably present non-isolated objects, while bins with ≥ 6 objects present bigger challenges. In those bins, objects will certainly be on top of each other, in front or behind each other, etc. Those bins, which represent 58% and 33% of the cases respectively, will require more careful and intelligent manipulation.

Team MIT, which placed second in the competition, developed a strategy comprising of several *manipulation primitives*: scooping, top and side suction, grasping and topple. To decide which primitive to use, the team had done a few experiments for every pair (primitive,object) and then, in the competition, the robot attempted each primitive in decreasing order of success probability. (*31*)

Following this paradigm, we decided that a great project to apply ML to manipulation was to implement the intelligence for 2016, with much higher requirements than 2015. In 2016, we can no longer solely decide on the object type, but we have to look at its position and surroundings. To do this, we can no longer rely on a simple precomputed list, but have to design an algorithm that looks at the present scenario. Moreover, this intelligence will have to deal with cases in which the goal is partially occluded or behind another object. In those cases, we will have to design a sequence of primitives that ends with a successful pick of the goal object.

2.2 Machine Learning: an overview

Machine Learning (*25*) is a sub-field of Computer Science and Artificial Intelligence whose main goal is to draw knowledge from data. More concretely, it aims to build algorithms that can learn from data and make accurate predictions on unseen data without the help of a human.

2.2.1 Types of Machine Learning

Although there are tons of ML algorithms, we usually divide ML into three main types, depending on the goal of the algorithm and the structure of the problem. These are: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

Supervised Learning

Supervised Learning (SL) is the most common and canonical type of ML: each datapoint consists of a pair (input, output). The algorithm is fed a set of such pairs and then the task consists on receiving only inputs and giving outputs that are appropriate for such inputs. The most well-known example is Linear Regression where the algorithm just learns an affine transformation from the input space to the output space that minimizes the quadratic error on the training set (see 2.2.3), hoping to minimize it for unknown inputs. For instance, one could try to see the relation between hours a student studies and their grade on the exam.

A different popular example in the current ML community is image recognition: given the raw pixels of an image the algorithm gives you the category of the object that appears on the image (such as 'dog' or 'car').

Those two examples are also different in nature. The first example does *regression*: estimating a continuous variable (be it just a number or a vector), while the second does *classification*: identifying a discrete class. In making this difference we should note that classification problems are often turned to classification problems by trying to regress the probability of every class, usually what is called the *cross-entropy error*. Let p_i be the probability of class i, and let q_i be our estimation for p_i. Then the cross-entropy is:

$$H(p,q) = -\sum_{i=1}^{N} p_i \cdot log(q_i).$$

It is interesting to note that this is closely related to the entropy often mentioned in statistical physics and the compression on the Huffman Algorithm. Moreover, when p is fixed this is equivalent to the *Kullback-Lebler divergence*, often shortened to KL divergence, which describes the amount of information lost when a probability distribution Q is used to approximated the true distribution P:

$$D_{KL}(P||Q) = \sum_i P_i \cdot log(\frac{P_i}{Q_i}).$$

Although it is often treated as a difference it cannot be considered a true metric, since it is not symmetric nor does it satisfy the triangle inequality.

A final and very important property of cross-entropy is that minimizing it with respect to q_i given the constraint $\sum_i q_i = 1$ gives the true probability distribution $q_i = p_i$. Let's prove it using Lagrange multipliers.

We want to minimize $F(q_i, \lambda) = -\sum_i p_i \cdot \log \frac{p_i}{q_i} + \lambda(\sum q_i - 1)$. Then,

$$\frac{\partial}{\partial q_i} F(q_i, \lambda) = \frac{p_i}{q_i} + \lambda = 0 \Rightarrow q_i = -\frac{p_i}{\lambda},$$
$$\frac{\partial}{\partial \lambda} f(q_i, \lambda) = \sum q_i - 1 = 0.$$

The second equation ensures the constraint is satisfied. Adding the first equation for every i we get:

$$\sum_i q_i = \sum_i -\frac{p_i}{\lambda} = -\frac{1}{\lambda}\left(\sum_i p_i\right).$$

Since p_i also describes a probability:

$$-\frac{1}{\lambda}(\sum_i p_i) = -\frac{1}{\lambda} \cdot 1 = 1 \Rightarrow \lambda = -1$$

Substituting we get $q_i = -\frac{p_i}{\lambda} = p_i$. This is one of the reasons why cross-entropy is heavily used in classification. However, some talks at NIPS '15 mentioned the interest in looking for other objective functions with nice properties that could complement cross-entropy. This is likely to be a challenge for the next few years.

Unsupervised Learning

Unsupervised Learning (UL) deals with problems where you are only given raw data and you have to find some structure in it. In contrast to Supervised and Reinforcement Learning, we don't have a reward or feedback signal for our signal. However, it still englobes a lot of interesting problems. Some examples are clustering (in which we divide the data set into a few subsets where individuals resemble themselves more than they resemble individuals from other subsets), compression or anomaly detection.

Probably the most famous example of UL was the experiment run by Google where they trained a Neural Network (see (*23*) for more details) on random images from YouTube videos. They did so by creating an *autoencoder* (figure 2.2):

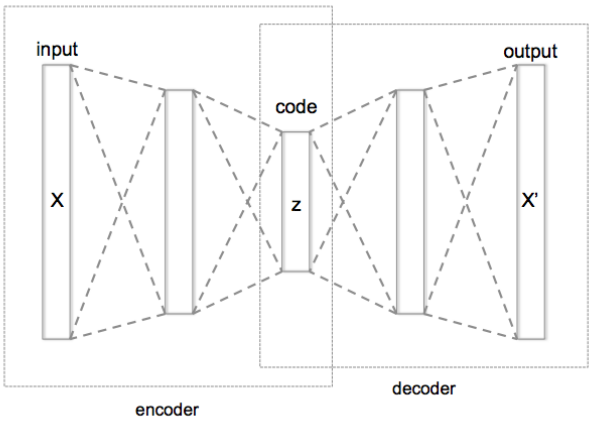

Figure 2.2: Autoencoder

The network is trained such that X' resembles X after passing through a compression (encoder) and decompression (decoder). When the size of the code is much smaller than the input, the network is enforced to come up with efficient representations. Without any

feedback the neurons in the code learned to detect useful complex shapes such as a human or a cat face(*23*).

Lacking a feedback signal, UL is much tougher than SL; however, asd this example proves, it has a much bigger potential for disruption. Labeled data has become commonplace, but is usually expensive to create, whereas we have huge amounts of unlabeled data from which we could learn. For instance, humans learn mostly from unlabeled data (especially in an early age where we don't have language) and then complement it with a small amount of labeled data. This strategy for learning was also underlined at NIPS '15 as one of the goals for the ML community.

Reinforcement Learning

Reinforcement Learning (RL) is the most extreme of all three types of learning. There's essentially no data and the agent only learns by interacting with an environment. A real world example for RL is training a dog by giving it cookies only when it performs the desired action: the dog won't know what its goal is, nor how is the best way to achieve it, but by trying to maximize the number of cookies it receives it will eventually figure out both things.

We will talk about RL and its relation to robotics in greater detail in the next chapter.

2.2.2 Errors, fittings and regularization

When implement a ML algorithm, we are generally introducing an implicit *bias* because we are restricting ourselves to a specific type of function, such as polynomials of a certain degree, Neural Networks of a specific structure,etc. Since the real data will most probably not follow one of the functions of our selected group, approximating it by this subset of functions introduces a *bias* error (*24*). The *capacity* of a set of functions is a measure of its expressive power, how big and flexible the set is.

A good measure for the capacity is the *VC-dimension*: the cardinality of the largest set of points for which the algorithm can get the correct result for every possible labeling we could assign to those points. For instance, the class of polynomials of degree d has VC-dimension of $d + 1$.

When this bias error is large and the function cannot capture the complexity of the data, we say we are *underfitting* the dataset. Two examples of *underfitting* are the left images in figures 2.3 and 2.4. In this case the solution is generally to increase the model's *capacity*.

We can also be on the other side of the spectrum, when the capacity is too large; sucumbing to a *variance* error. Since we only have a minor subset of all the possible datapoints; depending on the subset we choose, we will have a different fitting function. If the model capacity is too large we may not only capture the data, but also the noise. In this case, the chosen subset of the data will heavily influence the fitting function, which is not desired. Two examples of *overfitting* are the right images in figures 2.3 and 2.4. In this case, there are several solutions: get more data, reduce the capacity of the model or introduce *regularization*.

Regularization is introducing a prior knowledge about the form of the functions within the possible set. A very common type of *regularization* involves penalizing functions with large parameters, following the philosophy of *Occam's Razor* that states that between two

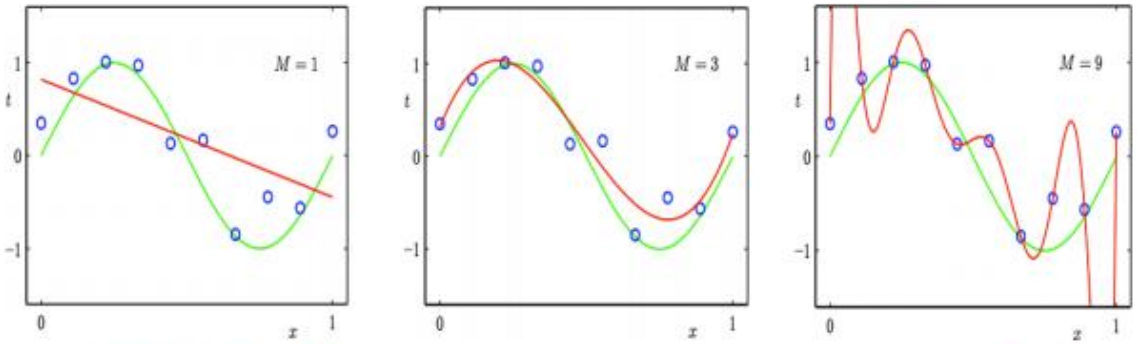

Figure 2.3: Underfitting and overfitting: Regression (*26*)

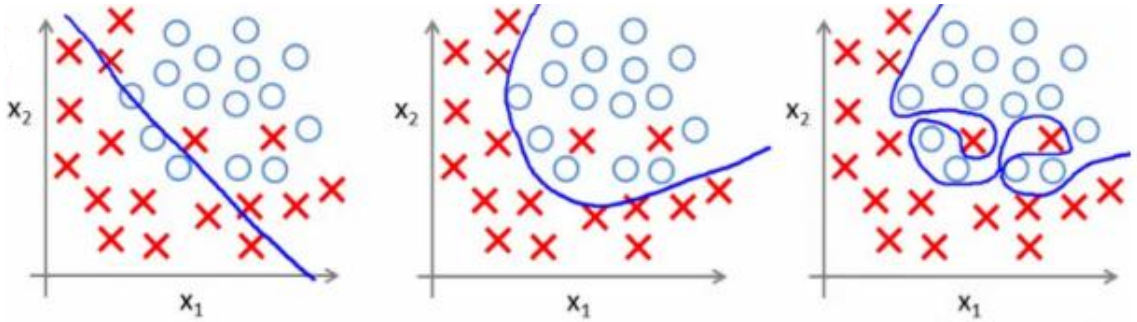

Figure 2.4: Underfitting and overfitting: Classification (*26*)

possible explanations for the same phenomenon, the simplest one is generally true. Two other popular ways of doing reglurazition are early training stopping (for ML algorithms that consist of iterative training) and Dropout (*27*) in the case of Neural Networks.

In general, there's a trade-off between bias and variance error with which the data scientist has to play. As can be seen in figure 2.5, as the capacity of the model increases, the bias error (usually associated to error in the Training Set) decreases while the variance error (distance between Training Set error and Validation Set error) increases. In general we care about both errors, thus settling for a middle point in terms of capacity. This ideal capacity is not constant, being an increasing function with respect to the number of data-points.

Finally, apart from the *bias* and *variance error* there's also the *noise error*; therefore, even the best possible models (such as the central images in the figures) will not be able to get all the answers correctly.

2.2.3 The importance of data

In classical Algorithmics we mainly dealt with two resources: time and memory. Therefore, our goal was to design algorithms that could give a good result using the minimum amount of those resources. In Machine Learning, data is just another resource, and, due to the huge progress of computer hardware both in terms of speed and storage, it is usually ,not always, the scarcest one. Even in the cases when it is not, we can usually work around the other difficulties: in case of memory scarcity, we can move to online algorithms that only process each data point once without storing and, in case of time scarcity, it has become standard to run algorithms in parallel on the cloud using

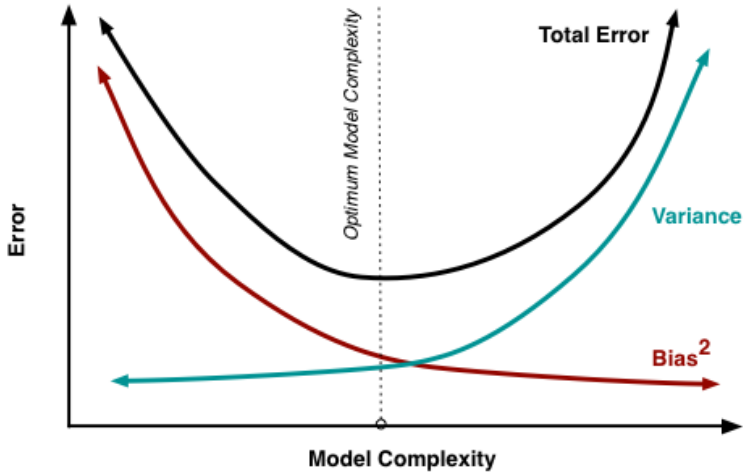

Figure 2.5: Bias-variance trade-off (*20*)

general parallelization algorithms such as MapReduce (*21*) and renting computer capacity to companies.

However, several comments have to be added to the previous statement. Notice that there are two time quantities that matter: prediction time and training time. The first implies running the modal forward whereas the second usually consists of running an optimization algorithm. The former generally takes on the order of microseconds to a few seconds, whereas the latter can take from several seconds to several months. When dealing with huge nonlinear functions (such as Neural Networks), optimizing their parameters can take a lot of time, even when doing parallelization. This is why, there have been recent enormous efforts to make optimization more efficient and why sometimes it is also a limiting factor.

Cutting this training time may seem unimportant (because it doesn't affect the end user), but usually the scientist iterates over several models trying to find what works best. When iterating to find a good model, it is of critical importance that the training time is cut down to a few hours or at most a few days so that the human can effectively improve the model after seeing how it performs after training.

Moreover, although a lot of current ML methods are usually data hungry, this doesn't need to be the case and there is currently some efforts to significantly decrease the amount of data required by such methods. An interesting example of such efforts are the ones that try to mimic how humans learn such as (*22*).

Data Scarcity in Robotics

Data scarcity is specially important in Robotics. First, with robots we usually have very few data sources: compare for instance learning Youtube clicks with billions of users to the usual amount of robots that a scientist has: one. Even with the scale of Google the amount of robots working to collect data doesn't go up to more than 20 (*28*). This is one of the reasons why MIT Technology Review has underscored the importance of robots that learn from each other (even if not owned by the same entity), by naming it as one of the 10 most important technologies of 2016 (*19*).

Another reason of this scarcity is that obtaining data in the real world is usually much

more expensive both in time and money than in the virtual world. For instance, in 2014 *Deep Mind* was able to accomplish human level performance in several Atari games (*16*, *35*) by being able to simulate those games extremely fast. In comparison, obtaining a datapoint in a robot usually takes on the order of 5-50 seconds. Moreover, a robot and its maintainance (necessary when running tons of experiments) usually cost a lot of money. Even more, some experiments (like the one that concerns us for APC) must be supervised by a human to ensure that the robot doesn't collide or cause any damage.

One solution is to use a simulator to pretrain the system and then run it in the real world, but this rarely works out of the box. What is usually done, is to practice and try different approaches in the simulator and then, if they work there, try them with the real robot, which is a much more complicated system. This is one of the approaches we took.

Division of the data set

In the majority of applications of ML, we learn a function from input to output; and then we have to test whether it is a good model or not. For this we need to split the dataset into three subsets: training, validation and test set.

Training set. The training set is the data from which the algorithm learns. In some cases it is the whole dataset, in some others we only devote 70% of the data to training, for the reasons explained below. The algorithm can look at the training data as many times as it requires: although algorithms like linear regression have a closed form expression, some other trainings are iterative and thus require multiple passes over the training set.

Test set. Remember that we want our data model to perform well on unseen data, not seen data. Therefore, to measure how good our model is, we need to keep some data that our model has not had access to: this is what we call the *test set*. Therefore, we have to take out part of our data and reserve it for a final evaluation: on the one hand we want the test set to be small, so that we can train on more data, on the other hand we want the test set to be large, so that we have enough statistical confidence of our model's performance. In many cases the test set is a random subset of the total data set, of around $10 - 20\%$ of the total number of points.

Sometimes, that may not be the case. For instance, when our dataset is increasing, we may not want to change the dataset with it, so that we can compare older models with current ones. Moreover, sometimes a part of the data may be more representative of the real world than other data points. For instance, when training a handwriting recognition system for Spanish letters we can also add to the dataset letters from the English language (because they are a subset of Spanish), but the distribution of English letters is not the same to the Spanish one, so it would be better to just test on Spanish.

Ideally we only want to evaluate on the test set once, and then throw this data out. If the scientists looks at the result of its trained system and uses this information to build his or her next model, the assumption of the test data being representative of unseen data is no longer true. Even if the scientist nor the program hasn't had access to the specific data points, just the sole information of the model performance on the test set is an information leakage that should be prevented. In practice, we can use the test set multiple times (on the order of $O(n)$ where n is the size of the test set) without a significant overfitting on the test set, but doing so has become one of the biggest problems in Machine Learning.

To attack this problem, a recent paper came in 2015 describing the algorithm *Thresholdout* (32), that selectively adds a small amount of random noise to the test set result to allow $O(n^2)$ evaluations instead of $O(n)$ (notice that for typical values of $n \sim 1000$ it makes all the difference). Proof of its importnce is the fact that it was published in the prestigious (and generalist) journal Nature.

Validation set. It is commonly the case that there are several *global* parameters that have to be tweaked affecting the whole structure of the learning model. Some examples could be the iteration step of the optimization algorithm (called *learning step* in ML), the number of layers in a Neural Network or the amount of regularization. Notice how this parameters aren't learned as a function of the data in the same way the coefficients α_1, α_2 of a linear model $y \approx \alpha_1 x_1 + \alpha_2$ or the weights of a Neural Network would.

Even if they aren't directly computed as a function of the data, we certainly cannot use the *test set* to optimize them (because we don't want to look at the test set until we've finished the model) nor do we want to optimize using the *training set* because the training set is most certainly overfitted (because, by definition, we've optimized our model for those specific data points). Therefore, we need a third set for those parameters, which is usually around $10 - 15\%$ of the data, resulting in a classical $70 - 15 - 15$ distribution in training, validation and testing.

3 | Reinforcement Learning

Of all three major approaches of Machine Learning (Supervised, Unsupervised or Reinforcement Learning), Reinforcement Learning (*11*) is likely the most suited for robotics (*48*). As we will see, the philosophy behind RL is a perfect match for most robotics goals. At the same time, the *fixed dataset* paradigm of both SL and UL, where the learning agent doesn't choose the inputs of the dataset, is a limitation that we don't generally have in the case of robotics, since the agent usually has a say in the states it faces.

We will first make an introduction to RL to then explain a couple of prototypes we did, describing their potential and what they lacked.

3.1 Introduction to Reinforcement Learning

Reinforcement Learning (*17*) is concerned with learning agents that have to perform actions in an environment so as to maximize some sort of reward. In contrast to SL the agent doesn't receive any feedback such as which was the best action to perform given a state, nor a specific correction for those actions that were wrongly selected.

The basic RL setting consists of the following (see also figure 3.1):

1. A set of environment states S,
2. A set of actions A,
3. A model of transitions between states,
4. A model that determines the reward r of the transition,
5. A model for the observations of the agent given the state.

Although it may not be the case, all models are in general treated as stochastic; thus viewing the RL as either a MDP (*Markov Decision Process*) or a POMDP (*Partially Observable Markov Decision Process*). This is a very good match with a paradigm that has proved very successful in robotics in recent years: *Probabilistic Robotics* (*15*), which claims that the best way of treating robots is with stochastic models, instead of trying to define either perfect models or hard-coded rules.

In general the agent is not interested in maximizing the immediate reward, but some sort of *cumulative reward*. For finite time (*episodic*) problems one can simply use

$$R = \sum_{t=0}^{N-1} r_{t+1},$$

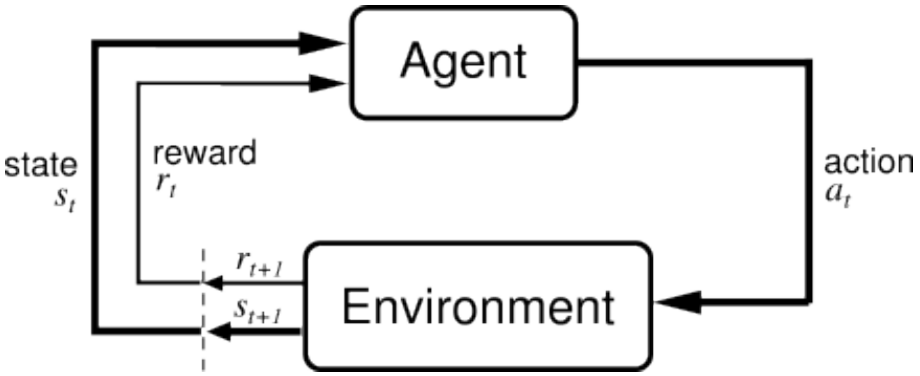

Figure 3.1: Reinforcement Learning Setting

where r_{t+1} is the reward for the t-th transition. For *non-episodic settings*, the most common one is the discounted cumulative reward:

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1},$$

where $0 \leq \gamma \leq 1$ is the *discount factor*, which regulates how much the agent cares about the future.

3.1.1 Specific challenges

Credit assignment. Since the rewards and actions are correlated by the state transitions, one cannot assume that the immediate reward is only caused by the last action applied. In the SL setting one receives a measure of success directly for its answer, but in RL the action that had the most impact in some reward is, a priori, unknown. Take, for instance, a chess match where the reward is $r_t = 0$ if the match hasn't ended at step t, $r_t = 1$ if it ended with a win and $r_t = -1$ if it ended with a loss. The decisive action is likely not the last one, as in general the winning player has already gained an advantage through prior good actions. Determining which of all the actions caused the reward is called *credit assignment*, and can be extremely challenging, especially if the reward action is *sparse* (as in our example) and doesn't provide much information.

Exploration vs. exploitation. Another major challenge of RL is the tradeoff between exploration and exploitation. We can see the agent as an optimizer for cumulative reward, which aims to find the global maximum in the space of functions from state to action. As in optimization there's a trade-off between trying to find small changes to an already good solution or adventure into unexplored spaces to find better solutions and avoid local maximums. Moreover, in RL it is usually the case where one wants the agent to perform well while it's learning, which makes the trade-off even more challenging.

An algorithm that, although it's simple, is common in state-of-the-art research is epsilon greedy: with probability $1 - \epsilon$ do the action which reports the highest expected reward and with probability ϵ do a random action.

3.1.2 How our paradigm helps RL approaches

Due to its generality, RL is a very powerful paradigm, but it is also almost impossible to solve, except if one makes key assumptions or improvements. In our case, using *manipulation primitives* instead of low-level control actions helps in two major ways:

It introduces a model. RL is model-free, in the sense it doesn't need to be told a model for the environment in order to learn a policy. This has huge advantages, since there is no need to hard-code it into the system and the model will be automatically tailored for the type of actions and rewards the robot cares about. On the other hand, if the environment is complex (and in most robot applications, it is) it is almost impossible to learn a physics model from just the very small information provided through the reward function and a Partially Observable state.

Using primitives can dramatically simplify model creation: in general each primitive serves a very specific purpose which is expressible in a simple form in terms of general state variables (such as positions of the objects). Knowing the effect of small robot movements in contact with objects is a very challenging problem in itself (*18*) and one cannot aim to solve it while learning a high-level policy at the same time.

It dramatically simplifies credit assignment. Had we used low-level actions we would have around 300 actions: performing credit assignment with that many actions is exponentially more difficult than doing it with the average 3 manipulation primitives. A good intuition behind this is that, unless one makes more assumptions the possible combinations of actions in T time-steps grows exponentially with T and to create a good model of the reward we have to explore a reasonable part of this space.

3.1.3 Q-learning

There are two major ways to approach RL, policy search (an example of which is given in 4.2.2) and value function approaches, of which Q-learning is the prime example. Policy search is finding a function from state to action, while value function is finding a function from state s to expected reward R under policy π:

$$V^{\pi}(s) = E[R|s, \pi].$$

This are called state-values. It is usually more useful to define action-values. Given a state s, action a and policy π, the action-value function is defined as:

$$Q^{\pi}(s, a) = E[R|s, a, \pi].$$

Given an optimal policy and its action-value function Q we can construct the optimal action-value function $Q*$ by doing $\pi(s) = argmax_a(Q(s, a))$. In other words, knowing the optimal action-value function tells us how to act optimally.

A very intuitive way to learn such action-value function is repeatedly updating the Q value of the observed pairs with the rewards observed. If we assume the discounted reward mentioned above we arrive at the equation seen in figure 3.2.

Notice how this function $Q(s, a)$ may be a look-up table (as section 3.3, but it can also be whatever type of function such as a linear regression, a local regression (as section 3.2) or even a Neural Network (see section 4.2).

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

Figure 3.2: Q-learning equation

3.2 Prototype of RL with Local Regression

3.2.1 Description

Our first prototype was building a RL system using Q-learning and a local regression function using local kernels, inspired in (*10*). To test the idea we designed a problem which we deemed comparable, and at the same time simpler, than our original problem. The scenario consisted in a 2-D square with circles, of three colors: green, blue and black, each of 1cm of diameter. The blue and the black circles were solid and could collide with each other and could be moved. The green one was penetrable and couldn't be moved. In one step one could move a single circle 1cm in one of four Cartesian directions: the circle then moved in that direction until contacting with a wall or another circle or moving 1cm. The goal was to put the center of the blue circle inside the green circle. This setup enabled us to build a simple physics model being able to code the collisions with simple geometrical formulas, making the physics system very fast (and thus allowing more experiments).

An example of this setup can be seen in figure 3.3. Notice how, to move the blue circle inside the green one it is necessary to move the black circles first, giving the sense of clutter as our main problem. To ease learning, we used 3 tricks:

- The agent received all the information (the 2-D coordinates of all objects). From the pair $(action, state)$ we implemented a set of 10 hand-crafted features and learned in this feature space instead of the original space.
- Using *Curriculum Learning*: first providing examples with no black circles, then with 1, then with 2. Having no clutter eased the problem, allowing the system to solve the basic instances before trying to solve the strictly more complex cases.
- The reward function wasn't sparse: apart from a high reward for finishing the task, we also gave a finite time for the task of 15 steps, so that the system wouldn't get stuck. Moreover, we penalized objectively incorrect movements such as the ones that the circle didn't move absolutely anything. It wouldn't have been a good idea to penalize small movements before collisions because in cluttered space, as when we park a car in a tight space, the system had to do a lot of small movements to accomplish its goal.

In feature space (10 dimensional), for each of the $4 \cdot (1 + b)$ possible actions, where b is the number of black objects, we counted the number of points within a ball of radius ϵ that we had tried before and the mean obtained reward of those points. The number of points in the ball is a measure of how much that part of the space had been explored, while the mean reward serves as the estimation for $Q(s, a)$.

To decide which action to take, there was a weighting between how unexplored that point was and $Q(s, a)$ and then the action with the highest value was selected. The

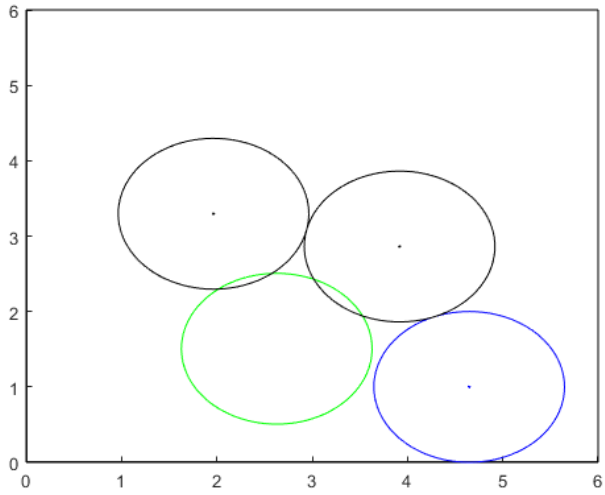

Figure 3.3: Example of the Circles problem

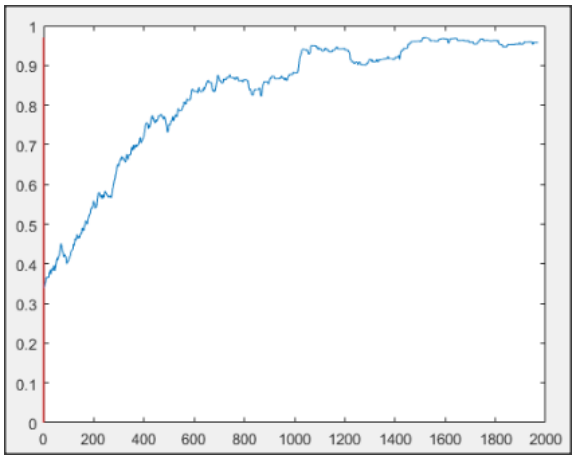

Figure 3.4: Evolution of % of success (of last 50 attempts) as a function of training runs

algorithm from the original paper was improved by carefully redesigning the exploration-exploitation. function.

Doing all this the system was able to learn a great policy on the problem, achieving over 90% success.

3.2.2 Why we finally didn't go in this direction

Despite its percentage of success, the system performed poorly in some situations, failing to produce not only suboptimal actions but producing very bad actions. Trying to improve the system we deeply understood its problems. There were two important ones:

The problem was much tougher than we thought . Although we had designed it as a basic version of our problem, the fact that the average number of steps was higher than 10 or 15 in many cases made it significantly harder to perform credit assignment, as

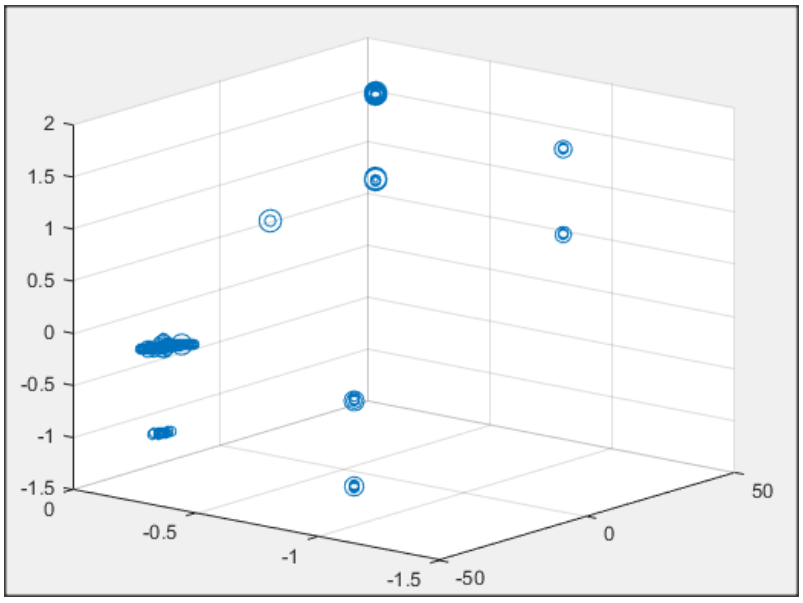

Figure 3.5: PCA of the points in feature space

mentioned in a previous section.

Local kernels suffer from the *curse of dimensionality*. In general, as the number of dimensions of the input space grows, the size of the space grows exponentially, which makes a lot of ML algorithms fail. Local kernels are specially prone to failure because, by definition, they cover only a small part of the space and we thus need an exponential number of them to fill the whole space.

What resulted was that, although many states where, in many sense, analogous, they were not neighbours in feature space. Moreover, since the space was huge, there inevitably were some points that had no neighbours inside their ϵ-radius ball, which made the algorithm select them to explore that region. Increasing ϵ to decrease this problem could result in both local minima and joining parts of the space which were not that similar. Moreover, regardless of the ϵ, the points concentrated in very few regions of the space. We experimentally proved this hypothesis by performing a Principal Component Analysis (PCA) to reduce the 10-dimensional feature space to a plotable 3-D space, which revealed some very big clusters and a few very small ones (see figure 3.5).

Seeing how the system suffered from the curse of dimensionality we decided to switch to Deep Learning, which is less prone to suffer from it.

3.3 Prototype of classical Reinforcement Learning

3.3.1 Description

Classic Reinforcement Learning simply consists on using discrete tables with the statistics of both the transitions and the rewards for each action pair. Let $p(s, a, s')$ be the probability of transitioning from state s to s' after applying action a and let $r_{s,a}$ be the expected reward after applying action $r_{s,a}$. Then we can do an approximate dynamic

Error	Experiments	MuJoCo	APC
20%	15k	5min	2.2 days
10%	60k	21min	8.7 days
5%	250k	1h23min	35 days
2%	1.5m	8.7hours	217 days
1%	6.2m	1.4days	870 days

Table 3.1: Estimation of number of experiments depending on desired error. Those estimations were probably too optimistic, compare to the more accurate estimations in table 6.2

.

programming algorithm to calculate $Q(s, a)$ as follows:

$$Q(s,a) = r_{s,a} + \gamma \cdot (\sum_{s'} p(s,a,s')V(s')),$$

$$V(s) = max_a Q(s,a).$$

To be able to apply this logic we considered discretising the state space our main problem: for every possible set of poses of all the objects of a bin and an indication of which one is the goal object, we had to come up with a discrete number that could reasonably represent the state.

First, we estimated how big we could make this discrete state space, so that we could train the system. We know that the standard deviation of the estimator for the mean of a binomial is:

$$\sqrt{\frac{p(1-p)}{n}} \leq \frac{0.25}{\sqrt{n}}.$$

Approximating the possible error in the estimations as the standard deviation and making an estimation for the average implementation time of an experiment in both simulation (MuJoCo) and the real system (APC) we were able to make an estimation for the total training time required. Given S states and A actions, the fraction spent estimating variables $Q(s, a)$ is $\frac{1}{S \cdot A}$, thus to get to a specific percentage of error e the number of total experiments is:

$$N = \frac{S \cdot A \cdot 0.25}{e^2}$$

The computed times for 5 actions and 300 states are presented in table 3.1.

Getting to a 5% error in the real system seemed like a reasonable goal, but those estimations relied on accurate estimations on the time per experiment which was a very uncertain estimation.

To use around 300 states we decided for a discretisation using a set of features seen in table 3.2.

The meaning of each primitive seen in table 3.2 is the following:

Feature	# of options
x-axis	4
y-axis	$3-4$
z-axis	$3-4$
Suctionable	2
Goal	2
Graspable	2

Table 3.2: Features with respective number of options

- **x-axis**: whether the goal object has something (wall or obstacle) close to its left (2 options) or right (2 options).
- **y-axis**: whether the goal object has something in front or behind. Supposing there are no rows of 3 objects we only have 3 of the 4 options.
- **z-axis**: whether the goal object has something below or above. Supposing there are no towers of 3 objects we only have 3 of the 4 options.
- **Suctionable**: whether suction would work in this object in an uncluttered environment.
- **Goal**: whether the object considered is the final goal or just an intermediate goal. Thus, when trying to remove obstacles one only needs to shift its field of view and consider the obstacle its new goal.
- **Graspable**: whether grasping (and scooping) works for this object in an uncluttered environment.

Moreover the set of designed actions were:

- Pick,
- Scoop,
- Top suction,
- Front suction,
- Remove.

Where the remove action was simplified as an intelligent primitive following a greedy algorithm: first removing the objects in front of the goal, then on top, then on the sides, then on its back.

3.3.2 Why we finally didn't go in this direction

Although this was a promising direction, suggested by prof. Lozano-Pérez, we were afraid the time estimations for APC were much too optimistic and thus that the discretization would be too big to work. At the same time we were losing a lot of detail from the discretization, with very different states going to the same discretized state.

Thoroughly analyzing the system we saw that there were a lot of symmetries that could be exploited, such as that the properties regarding the goal object do not change

with any of the primitives (and thus there is a major fraction of the state-space that cannot possibly follow). Moreover the primitives followed a very clear purpose: picking an object in different forms; and that should be exploited. Finally, the fact that the biggest length of an optimal sequence of primitives was 3 or 4 was a big assumption that was not included in tis setting.

Although this wouldn't go on as our final approach, these reflections, together with ideas from Deep Learning and Planning became our final implementation.

4 | Deep Learning

Deep Learning (DL) is the area of Machine Learning that has seen the most success in the last few years in many diverse applications, including robotics. Because of this, we considered DL as a possible solution, and we implemented a prototype to test strengths and weaknesses. In this chapter we will describe the basics of DL, why it has been so successful, and why sometimes it is not. We will also describe the motivation for attempting to solve the problem with Deep Learning and why we finally decided not to continue this line of research.

4.1 Introduction to Deep Learning

4.1.1 Basics of Deep Learning

Note: most of the introduction will assume that we are doing Supervised Learning; we later explain the adaptations to do Reinforcement Learning.

Neural Networks (NN) are a system of interconnected Neurons (individual nodes that do a couple of basic operations), *loosely* inspired in animal brains. They are used for function approximation, optimizing the real-valued weights in the connections between those neurons so as to minimize the objective function.

A single neuron in a NN generally computes an affine transformation of a subset of neurons followed by a non-linearity, called *activation function*. More formally, if neuron j receives values $v_{n_{j,1}}, \ldots, v_{n_{j,|n_j|}}$ are the values of the neurons that connect to a particular neuron its value is $v_j = \sigma(\sum_i^{|n_j|} w_{n_{j,i},j} \cdot v_{n_{j,i}} + b)$, where σ represents a non-linearity function and b a bias.

Activation Functions

Putting a non-linearity is necessary, otherwise affine transformations of affine transformations remain affine transformations, and NN wouldn't have much representational power. However, the exact type of non-linearity used is less important, and a lot of non-linearities have been successfully used. It is usually the case that the non-linearity used depends on the type of output we desire. Figure 4.1 shows a plot of several common non-linearities.

For instance, for classification purposes, where the output has to be within 0 and 1, the *sigmoid* is frequently used: $\sigma(x) = \frac{1}{1+e^{-t}}$. It is a good transition from hard logic to *fuzzy logic*, where things aren't either 0 or 1, but something in between: the sigmoid tends to 0 and 1 at the extremes being $\frac{1}{2}$ at 0. Linearly related to the sigmoid is the hyperbolic

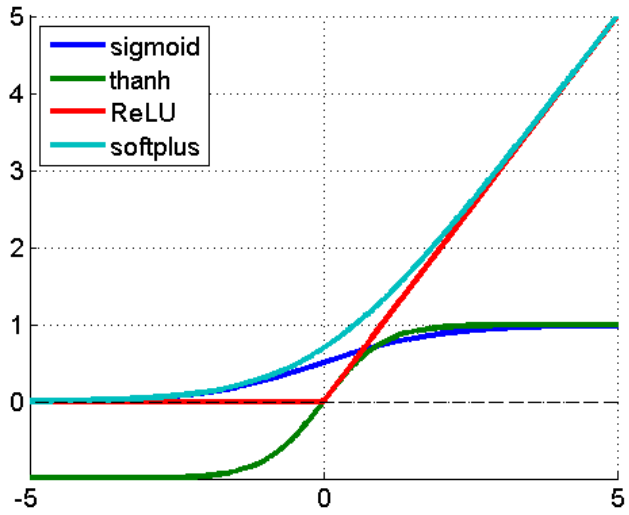

Figure 4.1: Examples of activation functions.

tangent: $tanh(x) = 2 \cdot \sigma(x) - 1$, with an also common range between -1 and 1.

In probabilities it is also very reasonable to enforce that the sum of all probabilities adds up to one. To do that a very common solution is to use a softmax:

$$p(i) = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}}.$$

When using softmax it is also common to add a parameter T of *temperature* that regulates the competition between the different options. This makes an analogy to physics, where the probability of different states as a function of their energy and the system's temperature follows a similar function. Notice how for $T \to 0$ the system tends to a delta at the state with the highest x_j, while $T \to \infty$ makes the system tend to a uniform distribution.

Finally, a few years ago it was shown that the particular activation function used wasn't of much importance, thus it is now common to apply the one with the simplest computations: the *Rectified Linear Unit* (ReLU) $f(x) = max(x, 0)$. Before that, the community was concerned about possible troubles using functions that were not C^1 and thus used a similar function, the softplus: $f(x) = ln(1 + e^x)$, whose derivative is the sigmoid function.

Deep Feedforward Neural Networks and Convolutional Neural Networks

The most simple type of NN are feedforward neural networks which consist of several layers: the input layer, multiple hidden layers and an output layer. The hidden layers are responsible for intermediate computations between the initial and output layer. Information then goes from one layer to the next, until arriving to the output. Figure 4.2 provides an example with 1 hidden layer.

Once we have computed the output, we can compare it to the target values and compute the error using the cost function. In the case of NNs this cost function could

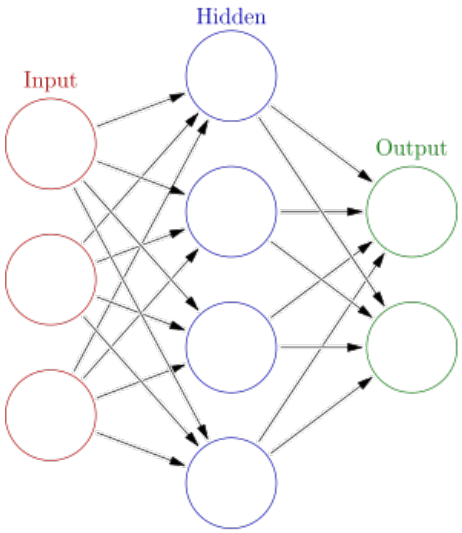

Figure 4.2: Feedforward Neural Network (*1*)

include a regularization term such as the sum of the weights squared. We will see in section 4.1.2 how to optimize this cost function.

Those classical Feedforward NN are also called *fully connected* or *dense* NN; and, although they are still very much used, they are avoided when possible because of the large number of parameters that have to be trained. In general it is preferable to find some structure to exploit in the data. The most common of those exploits is the Convolutional Neural Network (CNN), without which the paper that established the beginning of the dominance of NN in Computer Vision wouldn't have worked (the other ingredient being the Dropout regularization mentioned in the introduction) (*2*).

CNNs exploit two things of natural 2D images:

1. There are local features that can be identified without looking at the entire picture, such as edge detection, shapes, even objects that don't occupy the whole image.
2. We can analyze several parts of the image with the same filters: the filters that work in the top-left part of the image can be the same that those that analyze the bottom-left or the center.

With those ideas in mind we can make convolutional filters that only look at small patches of the image (or small patches of the previous layer) and, with very few parameters, compute high dimensional outputs. In *Backpropagation* (see 4.1.2) the same parameters are trained in all positions of the image. For instance these could be some simple edge detection filters:

$$\underset{\text{Horizontal edge}}{\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}} \quad \underset{\text{Central edge}}{\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}} \quad \underset{\text{Vertical edge}}{\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}}$$

Although these are 'ideal' filters, after training the NN, its first layers are indeed very similar to filters used in classical Computer Vision. An example of applying those filters could be the following:

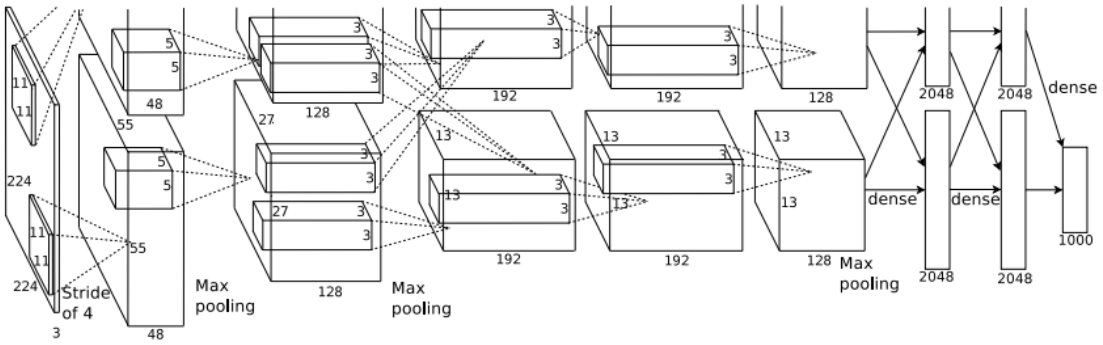

Figure 4.3: AlexNet Neural Network

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \end{bmatrix}$$

To reduce dimensionality, those convolutional layers are often combined with maxpool layers, which take the value of the highest activated node in some region. Thus, an n-by-m maxpool layer would reduce the dimensionality by a factor $n \cdot m$ (which is why maxpooling don't usually go bigger than 4x4, which already makes a compression of factor 16). For instance applying a maxpooling to the previous matrix yields:

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 3 & 0 \\ 0 & 3 & 0 \end{bmatrix}$$

CNNs, maxpooling and fully connected layers are usually combined together, such as in figure 4.3 depicting AlexNet (*2*), the current most popular NN in vision.

An important point has to be made in the case of robotic vision: maxpooling works well for reducing dimensionality, but also prevents us from knowing the exact origin of the features, and thus localizing interesting objects. This is fine when we want to do image recognition, saying that a picture is a dog, but not when doing localization (saying where the dog is). A popular method to overcome this, called *sliding window*, consists in using classical CNNs and then take several patches of the original image and see in which one the object is. However, this process is not differentiable, and thus not trainable end-to-end with Backpropagation which is useful when we want to train another NN on top of the vision NN that does something else, such as executing motor actions (see 4.2.2).

Network structure: deep vs. shallow

Notice that neurons, in general, receive inputs from a subset of the total of neurons. For instance, in the simplest feedforward case, neurons only receive inputs from the neurons

of the previous layer. Connections could be added between layers and, in fact, any Directed Acyclic Graph could potentially be used as a feedforward Neural Network.

Neural Networks can also be cyclic, and are frequently used in the literature, especially in problems involving sequences of varying length. However, Recurrent Neural Networks are much harder to train in practice due to the problem of exploding and vanishing gradients when doing Backpropagation (*3*) (see subsection 4.1.2). Several solutions, which we won't cover, have been proposed; most notably Long-Short Term Memory Networks (LSTMs) with gradient clipping.

Lately other interesting structures have been proposed such as attentional models (*4*), pointer networks (*5*) or Neural Turing Machines (*6*), the latter emulating the actions of a classic Turing Machine, but acting in a differentiable (and thus trainable) way.

Although the terms Neural Networks and Deep Learning are frequently interchanged, the later comes from the success of Neural Networks that have multiple hidden layers. Deep networks of several (≥ 4, but some going up to more than 100) hidden layers, are generally able to compute interesting functions and work very well compared to shallow networks that have the same number of nodes or other types of function approximation with the same number of fitting parameters.

One of the most important questions yet to be solved in ML is why deep Neural Networks work so remarkably well. An intuition behind it is the following: we are not interested in approximating *any* function. We are only interested in approximating functions that come up in the real world. We can therefore assume certain characteristics of those functions. For instance, linear regression assumes that a function is linear, which works well in many cases. Other methods, such as Gaussian Process and other kernel methods are based in the notion of continuity: $f(x) \approx f(x + \epsilon)$ for small ϵ. As such, Deep Learning is also making an important assumption: compositionality, the idea that concepts are formed as a compound of several other, simpler, concepts.

It turns out that this idea is very appropriate for our world's functions: for example, faces are made out of a mouth, a couple of eyes, a nose, etc; at the same time, each of those parts of faces can be represented as a mix of simple filters such as edge detection filters. Those simple filters can then be expressed directly out of raw pixel values. In practice, intermediate values (hidden layer activations) of Neural Networks are still not completely understood; doing so is an active area of research. Figure 4.4 shows a visualization of the computation done in several layers.

4.1.2 Backpropagation Algorithm and Stochastic Gradient Descent

Although the Backpropagation Algorithm was originally invented in the 1970s, it became well-known with a paper by Rumelhart, Hinton et al. and has ever since become the standard algorithm that powers most NN. For a mathematician, it can be viewed as combining the idea of Gauss-Newton method together with the chain rule of derivatives. The algorithm consists on the following:

1. **Initialize the network's weights to small random values**. This is necessary in order to break possible symmetries; otherwise, if all connections were the same, they would receive the same corrections, would stay the same and thus would be superfluous.
2. **Make predictions** running the network forward.

Figure 4.4: Activation of Neural Networks in several layers (7).
Next to them patches of images that highly activate such neurons. Notice how lower layers consist of geometric local filters, while higher layers have more invariant representations.

3. **Compute error** on the output units: $\Delta = (predicted - actual)$. This step may be more complex such as for a non-quadratic cost function (which leads to a different error than linear) and adding regularization terms to the function that has to be optimized.
4. Now, going from output layer to input layer compute the gradient with respect to the weights going into the specific layer. Notice that, having the errors on the output layer we can calculate the gradient of this error with respect to the weights of the incoming connections and also the gradient with respect to the values of the nodes of such connections. Passing this error back to the previous layer, using the chain rule, allows to quickly calculate the errors of the weights of the previous layer. This is repeated, in a FFNN, until reaching the input layer.
5. Once all the partial derivatives with respect to every weight have been calculated, then weights are updated: $w := w - \alpha \nabla Q(w)$, where Q is the cost function.

Stochastic Gradient Descent is the most popular implementation of Backpropagation. Approximating the true gradient with only a small subset of the data one can compute it much faster, speeding up the optimization significantly. This typically results in calculating gradients in 'batches' of 32 or 64 data points and then updating the weights before calculating another gradient.

To speed things up even more several improvements have been suggested; one of those is to compute several gradients in parallel in multiple computers and then update the weights asynchronously. Although this makes even stronger assumptions on convergence than traditional SGD, it has been successfully used in practice, specially in companies with large computer cluster resources.

Another common improvement is to include momentum in the optimization:

$$\Delta w := \alpha \nabla Q(w),$$
$$w := w - \Delta w.$$

This makes an analogy to a particle traveling in parameter space with some momentum while going down on the hyper-surface while the gradient acts as a force.

Back-propagation doesn't require strict C^1 differentiability, it just has to be differentiable in practice such as trainining NN with Rectified Linear Units. Interestingly, we can also build NN structures that are non-differentiable or even discontinuous; in this type of problems one cannot apply back-propagation. However, recent papers attack this problem by using Reinforcement Learning.

Notice that back-propagation is not guaranteed to find the global optimum (and most likely will not) because of the non-convexity of the optimized functions. However, it is believed that this is not a major issue in most practical cases.

4.2 Deep Reinforcement Learning

After carefully examining the literature on Deep Reinforcement Learning applied to problems similar to manipulation we ended up considering two approaches: Deep Q-learning and Guided Policy Search. We will explain their strengths and origins before explaining how we built our Deep Reinforcement Learning prototype.

4.2.1 Deep Q-learning

Deep Q-learning consists in implementing the same Q-learning explained in subsection 3.1.3 and using a NN as a function approximator for the Q-function. Using NN instead of the non-parametric functions based on local kernels is that we can leverage their power to represent high-dimensional spaces.

Deep Q-learning was popularized by the company *Deep Mind* with a couple of papers published between 2013 and 2015 (*16*, *35*). In the latest paper, published in *Nature*, they built a Convolutional Neural Network system capable of learning to play most popular Atari games with a human-level or better performance. The most impressive of this result is that *the same* system, without any extra fine-tuning or human help, was able to autonomously learn very different games, suggesting a high level of generality.

4.2.2 Guided Policy Search

Guided Policy Search is an idea popularized by Sergey Levine and Pieter Abbeel, whose aim is to reduce the complexity of typical Reinforcement Learning by dividing the problem in two. This allowed them to train end-to-end robotic controllers performing tasks such as putting blocks in an arbitrarily positioned hole or placing a clothes hanger in a bar (*42*, *44*). Notice that, unlike our task, this is low-level control with relatively simple vision versus high-level control (and shorter sequences), but with much more complicated environment and vision.

More concretely, pure Reinforcement Learning has to deal with complex dynamics(environment) and a complex policy(control); dealing with the two at the same time can be very challenging. The idea is to create two systems:

1. **Optimal control**: given a fixed problem with a sole start and finish it is much easier to train: for instance, one can use Dynamic Programming methods relying on the dynamics and goal being constant or just try multiple strategies until finding a good one. This provides training samples for system 2.
2. **Supervised Learning**: picking the samples from system 1 we now have a standard Supervised Learning setting from state to action which can be easily trained. This training is necessary to be able to generalize to unseen states. As we have previously mentioned, most breakthroughs in Deep Learning have been in Supervised Learning settings; now system 2 can leverage this progress.

There are a couple of important ideas to be added:

- System 1 and 2 can be trained with different inputs. For instance, to determine the perfect control we can train with full observability (such as telling the system the correct position of objects without having to detect it from vision). Then system 2 has to learn from a harder input, with only partial observability. This allows system 1 optimization to be easier, while system 2 will learn a input to action function that doesn't rely on full observability, which doesn't happen in uncontrolled environments.
- A couple more concepts have to be curated to ensure good performance of this system. For instance system 1 may generate data sets that system 2 has a tough time mimicking (both because of partial observability and the intrinsic difficulty of properly approximating a function from a finite set of examples). Thus, in system 1

policy search (RL)	complex dynamics	complex policy	HARD
supervised learning	~~complex dynamics~~	complex policy	easy
optimal control	complex dynamics	~~complex policy~~	easy

Table 4.1: Scheme Guided Policy Search (*8*)

optimization we have to add a term to increase preference for outputs that system 2 can predict.

4.3 Prototype

4.3.1 From local kernels to Neural Networks

To explore the potential for Deep Reinforcement Learning in our manipulation task we decided to implement a Deep Q-learning algorithm because of its simplicity, with a couple of adaptations for our case. We started in the same setting as the RL with kernels prototype of section 3.2. Now, however, we introduced a Neural Network controller.

The Neural Network was coded with Google's recently opensourced TensorFlow system (*37*) which simplifies the task of working with NN. We thus had to change from Matlab to Python in order to use its API. With it, we implemented a three layer ReLu Neural Network that received the positions of the circles and had to predict the Q-value.

Unlike in an Atari game, here we could profit from full observability and also try to predict the future positions of the circles after the manipulation primitive. Notice that approximate full observability will also be the case in the general manipulation task, assuming we have a good vision system. We thus could try to predict both the future positions of the circles and the Q-value. The latter being much more noisy, training with the future state provides a much more reliable feedback.

The system was able to properly train and get to only 5% error in future state (circle positions) prediction, but poor Q-value predictions. However, although we didn't pursue it, we are confident that either training for more time (training time was cut at 10 hours) or hand-coding features as we did with the first RL prototype would have been sufficient to solve the problem.

4.3.2 From a simple scheme to Physics simulations

The second improvement over the first type consisted in using a physics simulator instead of the initial simplified model with 2-D circles. We devoted more than one month getting acquainted with the physics simulator MuJoCo (Multi Joint dynamics with Contact) developed by Emo Todorov of the University of Washington (*38*) and then coding the scenario. MuJoCo has the advantage of being very fast and reliable for doing physics simulations with collisions, contact and friction; while having the disadvantage of being poorly documented because its a very recent product.

In MuJoCo we implemented a scene with 3-D cylinders, 3 walls and a gripper that could perform several of the manipulation primitives that we wanted to do in real life such

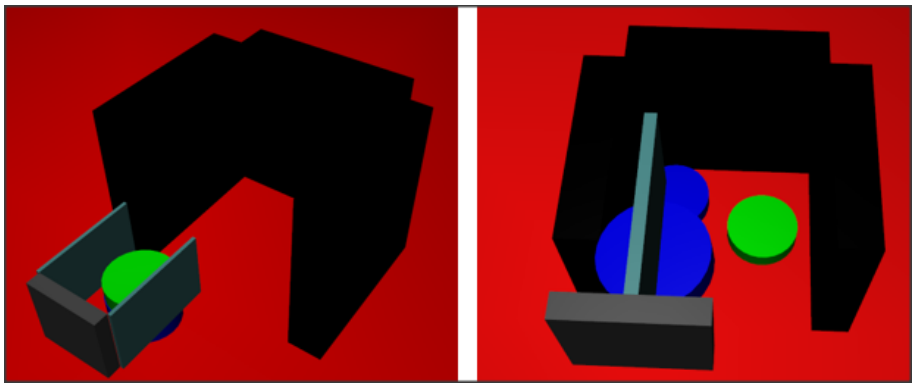

Figure 4.5: Grasping and Suction primitives implemented in MuJoCo.

as pushing front and side, grasping and placing, and suction. Images of this system can be seen figures 4.5. Unfortunately our trial period for MuJoCo finished in the middle of the research, and since its subscription costs more than 1000$ we decided not to renew its subscription. Thus, although we had the NN controller and the scenario and manipulation primitives coded in MuJoCo, we couldn't evaluate the performance of the ML system in the simulator, deciding to move directly to the real system.

4.3.3 Design for the real manipulation planner

Although it wasn't our final version, a lot of thought was devoted to the design of the real world Deep Learning system for the APC. Parting from the initial design for the circles simulation in which the Neural Network had to predict both the next positions for the circles and the Q-value, we designed a similar system for the real world experiment, whose scheme can be seen in figure 4.6.

The inputs for this system would be the manipulation primitive to execute, the position of all objects in the scene and the specific type of object of the goal object (such as 'book', 'vase', 'kleenex', etc). With the manipulation primitive and the position of all objects in the scene we can design a series of hand-coded features such as distances to the nearest objects and walls (less distance reducing the probability of success of some action), whether the goal object had some object on top or in front, etc.

With the manipulation primitive and the type of object (and its orientation and position) we can also create several features. This is important because some objects, or some orientations of an object are much tougher or just impossible (for instance an object with holes cannot be picked with suction). However, unlike the geometrical-spatial features captures from the positions of the objects, those features are much tougher to deduce and are probably better obtained through experimentation, a problem which we solve with the scheme as explained below.

The system then introduces such parameters or features into the Neural Network that has to predict both the Q-value and the Object new positions. As mentioned before, both things can be measured from real-world: the Q-value depends on the success of this particular primitive and whether and how fast the goal object is later picked; and the new object positions with the vision system (always on at training time). The predicted values are compared to the real values and the errors are back-propagated through the NN, updating its weights. Moreover, although in a conventional Neural Network the

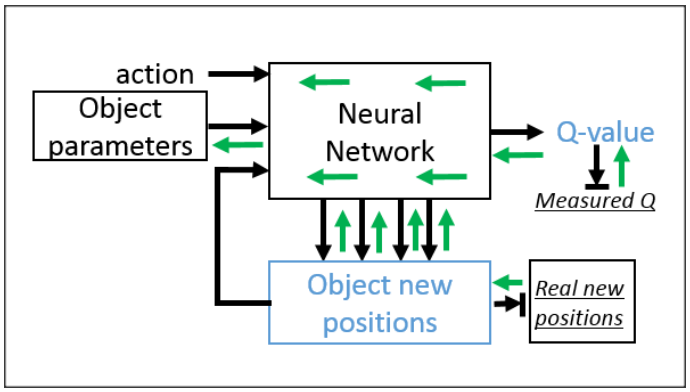

Figure 4.6: Scheme of the Deep RL for the APC system. The blue lines refer to inference computations while the green lines refer to training information that is back-propagated to update parameters.

errors aren't back-propagated to the input values (because those are considered fixed), we can consider the (primitive, object) parameters mentioned above as variables that can be fine-tuned and passing the error also there. Although not common, this type of idea has been successfully implemented before; albeit it always has to be done with care: in the beginning the Neural Network weights are *very* bad and thus the back-propagation error is also extremely noisy and thus passing this information to the parameters could corrupt them.

At test time, the competition, calling vision takes time and thus perception is a manipulation primitive in itself. Thus, if the system is confident enough, it could potentially decide not to call perception and then just pass the predicted new positions as measured positions. Notice that, in this setting, the perception primitive has a very elegant form: passing the real new positions instead of the predicted positions.

Here, both the input and the output object positions carry an uncertainty value: as with any other real world system, the perception primitive has an intrinsic measurement error; moreover, doing manipulation also increases variance. It is thus as important to correctly predict the variance of a prediction as it is to make a good one. Calling perception after doing manipulation will obviously reduce the variance of the estimations.

A proper question could be: how do we update the variance if we only know the real positions? A proper way could be to maximize the probability of prediction of the real-world value, assuming a normal distribution. If y is the correct value, $\hat{y}$ the predicted value and $\hat{\sigma}$ the predicted variance:

$$p(y, \hat{y}, \hat{\sigma}) = \frac{1}{\sqrt{2\pi}\hat{\sigma}} e^{-\frac{(y-\hat{y})^2}{2\sigma^2}}.$$

Doing the partial derivatives we have:

$$\frac{\partial p(y, \hat{y}, \hat{\sigma})}{\partial \hat{y}} = \frac{e^{-\frac{(y-\hat{y})^2}{2\sigma^2}}}{\sqrt{2\pi}\hat{\sigma}^3}(y - \hat{y}) = p(y, \hat{y}, \hat{\sigma})\frac{y - \hat{y}}{\hat{\sigma}},$$

$$\frac{\partial p(y, \hat{y}, \hat{\sigma})}{\partial \hat{\sigma}} = \frac{e^{-\frac{(y-\hat{y})^2}{2\sigma^2}}}{\sqrt{2\pi}\hat{\sigma}^4}((y - \hat{y})^2 - \hat{\sigma}^2) = p(y, \hat{y}, \hat{\sigma})\frac{(y - \hat{y})^2 - \hat{\sigma}^2}{\hat{\sigma}^2}.$$

This theoretical calculations follow our intuition: $\hat{y}$ should increase if and only if the actual value is higher than the prediction, while something equivalent passes with the standard deviation when the estimation of the variance is compared to the measured variance.

Although this model seemed promising (complex enough to capture the real world, simple enough to analyze it and improve it), as we explain below and later in the thesis; we didn't pursue its implementation, going for another model.

4.4 Why Deep Reinforcement Learning isn't (currently) the best approach

One of the most important steps in this thesis was stopping our approach based on Deep Reinforcement Learning and focusing in another approach, explained in chapter 6. Some of these reasons were discussed in a couple of meetings with MIT robotics professors Tomas Lozano-Perez and Leslie Kaelbling along with my advisor at MIT Alberto Rodriguez. There are multiple reasons, here are some of them:

- Deep Learning still requires huge amounts of data that we do not have. This is the reason why *DeepMind* is exploring games (where data can be obtained fast and cheaply). Not until recently has it moved to robotics, but in simulators, not in real life.
- RL was a dangerous setup for our implementation: in the robotic experiments with RL failure to choose a reasonable action had basically no consequences. In our case, choosing the wrong primitive implies crashing objects, destroying the gripper, etc. These limitations have to be taken into account.
- Although once created the system learns *autonomously*, the specific system isn't that easy to create and fine-tune to our specific problem. For instance, the size of the convolutional layers used in the Atari Q-learning was specifically tuned to capture most interesting objects in a typical Atari game. Typical characters in Atari games (spacecrafts, monsters, etc) have a relatively common size of a few pixels; adjusting the size of the convolutions allowed the system to learn to detect such objects with ease. Something similar would have to be done in our case, with no ability to do experiments and few time resources.
- The Deep Q-learning experiments in Atari games showed the potential of this approach, but also its shortcomings. The biggest of them was the ability to do high level, long term, approaches. The system was very good at reactive games (ping pong, boxing, shooting...), but very bad at games that with long term horizons such as those involving mazes like Pac-Man. Dealing with long sequences is still one of the challenges of NN. We initially thought our approach of learning sequences of primitives instead of low-level control would allow us to by-pass this problem, but there was still the problem of predicting the effect of very complex and long-term actions in a complex environment.

This doesn't mean Deep Learning won't be a viable solution for robotics. Although most probably we will have to combine it with some kind of planning (see section 5.5.2) in the future. Attention methods and similar approaches may reduce the need for data,

progress in Unsupervised and Transfer Learning may allow us to exploit learning in one domain in another. Thus, we concluded that Deep Reinforcement Learning may be a good direction for the field of autonomous robotics, but not a path that can provide immediate results.

5 | Planning

A lot of research in robotic decision-making has been, and still is, in the field of planning. In contrast to ML, planning is more predictable, can provide mathematical guarantees, doesn't require data (again, a great constraint in robotics) and can easily be understood and debugged by humans. Planning can be done both at low-level geometrical constraints (*geometric planners*) and at the high-level decision-making. We explored both types of planners and also how to combine planning with learning.

5.1 How our approach on manipulation simplifies planning

Roughly speaking, most planning algorithms can be seen as a tree search in some state space, be it robot joint space, physical space, constraint space, etc. The running time is, in general proportional to the number of states considered, thus, to the number of states of the tree. Let us assume this planning tree has a constant ramification degree d (also called *branching factor*), a height (the planning depth) h and a probability of pruning the state constant p. Then the number of states follows the following recurrence:

$$T[d, h, p] = 1 + (1 - p) \cdot d \cdot T[d, h - 1, p]; T[0] = 1,$$

which is well known and results in:

$$T[d, h, p] = \frac{((1 - p) \cdot d)^{h+1} - d}{d - 1} = \Theta(((1 - p) \cdot d)^h) = \Theta(d_{ef}^h),$$

where d_{ef} is the *effective* branching factor.

Using manipulation primitives instead of low-level control allows to significantly decrease d since the number of manipulation primitives (around 10) is much smaller than the potential number of movements of a 7-degree-of-freedom arm (which, we could say is on the order of $3^7 \approx 2000$ if for every degree it can do nothing or move in any one direction).

More importantly, though, is significantly decreasing the exponential factor h, going from taking a decision every $0.1s$ to $10s$ the number of states will get the order of magnitude of the problem decreased by a factor of 100: which makes an intractable problem, now solvable.

5.2 Constraint-based Planning

Our initial first prototype of planning for sequential decision-making was based, coincidentally, on a paper recently written by MIT professors Tomas Lozano-Perez and

Leslie Kaelbling: A constraint-based method for solving sequential manipulation planning problems (9). It describes how to do a task similar to ours: integrating task and motion planning by performing a symbolic search for a sequence of manipulation primitives involving geometric constraints.

We can see every manipulation primitive as requiring a set of geometrical constraints such as:

- *Disjoint*(o_1, p_1, o_2, p_2): object o_1 in pose p_1 and object o_2 in pose p_2 don't collide.
- *Contained*(o,p,R): object o in pose p is in region R.
- $\exists$ *Valid Path*(H,K,o,G,$\mathcal{O}$): there exists a valid path from robot's home to configuration (H,K) while holding object o in grasp G and avoiding obstacles from set $\mathcal{O}$.

With these and other constraints we can specify the requirements for a successful execution of a specific manipulation primitive such as moving, grasping or placing, which, in its turn, will provide some geometric guarantees. Knowing all this geometric constraints and implications one can devise a plan starting from our goal (which can also be expressed as a set of constraints) down to a state in which its geometric constraints are satisfied by the current configuration. If we find a sequence of primitives $s_{1...n}$ in which s_0 constraints are satisfied and the execution of $s_{1...i}$ guarantee the constraints for s_{i+1} for any i and $s_{1...n}$ guarantee the constraints specified for the goal, then we found a valid solution.

A more visual example for this algorithm is shown in the next section.

5.3 Planning prototype

In the same manner as the *Deep Reinforcement Learning* prototype we implemented a small prototype implementing a basic version of (9) using MuJoCo. The prototype was of a specific cluttered scenario we believed was a good benchmark: a bookshelf, which can be seen in figure 5.1.

Using constraint-based planning the manipulation primitives and their constraints were:

- **Push front**: space in front and on the back of the object.
- **Grasp and move to the side**: space near the object on both sides (to grasp it) and from the object to its final position.
- **Grasp and move front (finish)**: space both in front and to the sides.

The implementation was long and tedious, but successfully dealt with geometric constraints. However, although generally the geometric solver is very complex, in this scenario it could be hard-coded working with boxes whose edges were parallel to the world axis. The conclusion from this prototype was that, although the planning was reasonable and simple, implementing the geometry could be a much tougher problem.

For instance, to pick the pink book the planner gave:

1. Pick the left black obstacle and move it left.

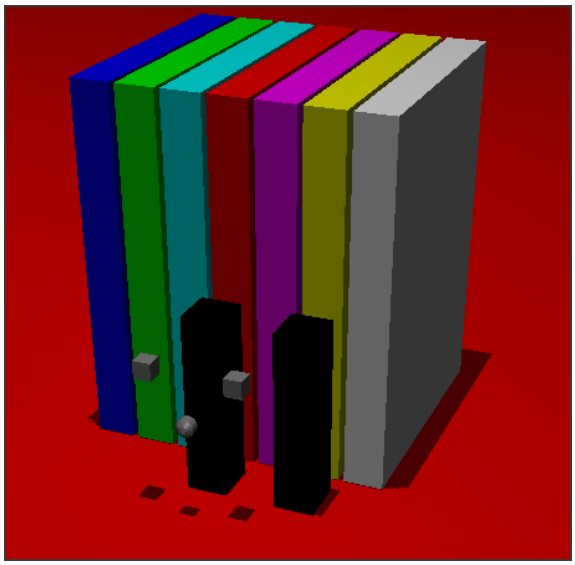

Figure 5.1: Implemented bookshelf scenario in *MuJoCo*. The small cubes are *graspers*, the small one in the center is for pushing. The big colored blocks represent books and the black ones are extra objects for requiring more varied manipulation sequences.

2. Pick the right black obstacle and move it right.
3. Push the red book back.
4. Push the yellow book back.
5. Pick the pink book: end.

5.4 Why pure planning wasn't a good option

Doing the prototype and comparing planning to learning we found several reasons to discard pure planning as our best approach. Some are:

1. Geometric constraints are complex and difficult to state and we need to deal with geometric solvers.
2. In our specific application there are two types of geometric constraints:
   - **Hard constraints** (walls): the implementation of manipulation primitives deal with those.
   - **Soft constraints** (objects): there's an advantage on allowing collisions and moving objects as long as we accomplish our goal, since the gripper was designed for it. Decisions should take into account those obstacles but in a more complex way than just as constraints that can't be touched.
3. Pure planning requires precise models. This isn't always the case for us:
   - Vision isn't perfect and has some (small) error.
   - Some objects are soft and their geometry is hard to model.
   - Even when objects are hard, some have complex forms which are hard to model.

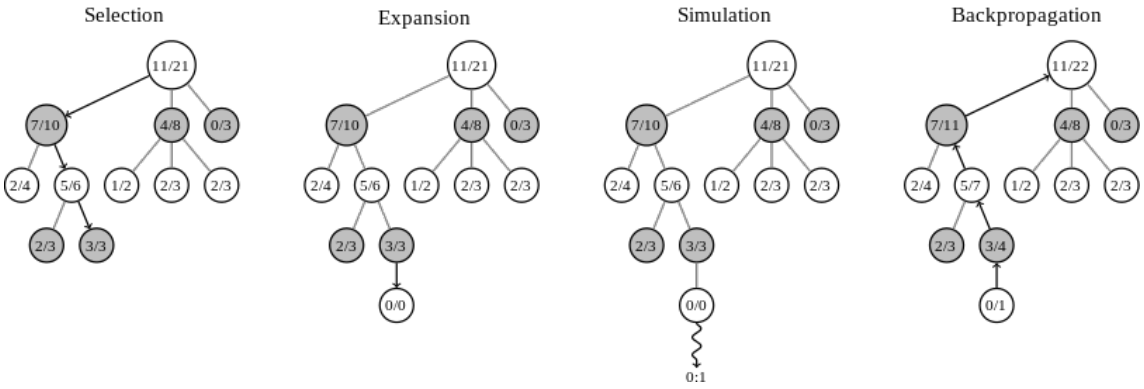

Figure 5.2: Monte Carlo Tree Search (*34*). The change of colors correspond to the different players.

4. Doesn't deal with probabilities of success, which could be informative to plan the strategy.

5.5 Combining planning and learning

5.5.1 Monte Carlo Tree Search

To understand the algorithm described in the next subsection we first need to describe the *Monte Carlo Tree Search* (MCTS) algorithm (*13*, *14*). MCTS is a planning algorithm used in imperfect information games such as Scrabble and Bridge and with a lot of success in Go. Compare this to our manipulation problem, where we aren't in a game (in the sense we don't have an adversary), but there's also a sense of imperfect information due to unpredictable dynamics. MCTS does the following 4 steps (see fig 5.2):

1. *Selection*: starting from the root node successively go down to one of its child nodes until arriving to a leaf. This child selection is random, but favors promising moves, which is the essence of MCTS.
2. *Expansion*: if L doesn't end the game create one or more child nodes C.
3. *Simulation*: play a random game from each node C and compute the fraction of games won.
4. *Backpropagation*: add this fraction to every node in the path from the root to L.

As an aside note, it is interesting to compare the probabilistic nature of this algorithm in contrast to the minimax algorithm with $\alpha - \beta$ pruning algorithm, used by Deep Blue in chess, which is deterministic. This merges well with the contrasting approaches between pure backtracking (in chess) and ML (in Go) discussed next.

5.5.2 AlphaGo: the first computer system to win humans at Go

In the end of 2015 DeepMind published another influential paper (*34*) which combined planning and Deep Learning to solve the most challenging classical game for AI: Go. In

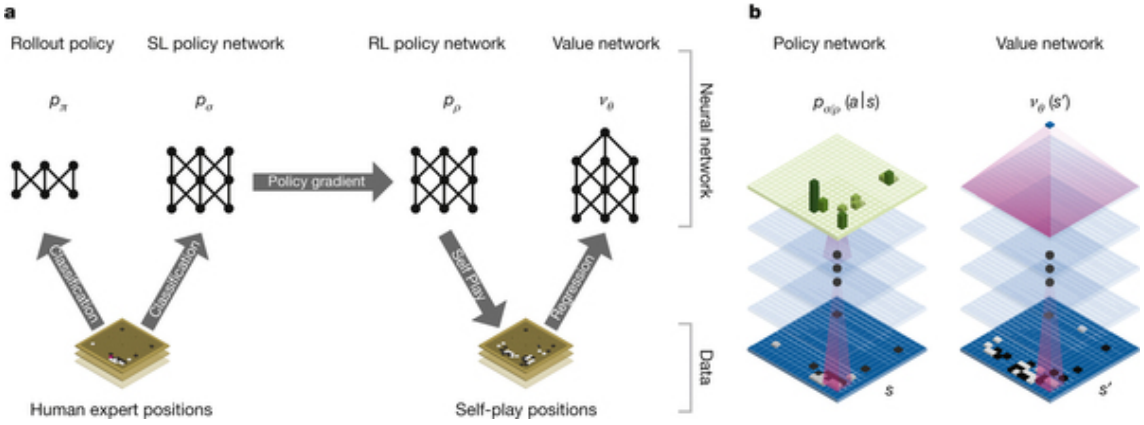

Figure 5.3: AlphaGo's Neural Networks (*34*)

this section we (briefly) explain how their system works, which will serve explaining how it inspired our final system design.

In the planning tree paradigm we explained before, Go can be regarded as a much more complex problem than chess in several ways:

- The average game is 3-5 times longer, increasing the depth of the tree.
- The average branching factor is an order of magnitude higher: from 20 possible moves to 200.
- Hand-crafting an evaluation function (for pruning) is much tougher in Go than in chess (in fact, it still hasn't been done).

To solve these problems they implemented two Neural Network systems:

1. *Policy network*: for every possible move output a probability, indicating the probability that this is a good move.
2. *Value network*: given a board state gives the probability that player *white* wins.

Notice how the *policy network* aims to reduce the branching factor (from 19^2 to 2-5) and the *value network* aims at reducing planning depth. At every node, the probability of winning was estimated doing a Monte Carlo Tree Search, but this calculation is very expensive (specially when the tree is very deep); the value network was aimed at reducing those needs of computation by doing another estimate which was combined with the estimation from the MCTS.

To train them they made the following steps:

1. **Training a Supervised Learning Neural Network** on strong amateur Go games database. Given the game situation the NN was trained to predict human's next move, accomplishing $\approx 50\%$ success. We call this network the *SL policy network*.
2. Now the NN knows how to estimate humans fairly well **we start doing RL**.
3. **Playing the program against itself** we get a new huge data-set of games, from which we can now **train two networks**: the *RL policy network* which suggests

moves (now not intending to imitate humans, since we want to be better than them) and a the *value network*: predicting the probability of winning (trained with the game state and the final winner of the game from which that specific game state came from).

Although the final implementation of our system had major differences to AlphaGo; it inspired the final idea of combining planning and learning, explained in chapter 6.

6 | Final Implementation

6.1 Description of the system

Although it was unintentional, our final system combined ideas from all the analyzed topics: Deep Learning, Planning and Reinforcement Learning. It is based in a very simple idea: each primitive serves a very concrete goal. Thus, if it succeeds the state is very predictable; if it fails, it is quite unpredictable and we are better off calling perception and rethinking again. Moreover, except the minor primitive of pushing, all primitives serve the same purpose: taking out an object.

6.1.1 Planner

Let us assume we have some way of calculating the probability of success for all primitives in any scenario. Now, we construct a planner that looks at all possible sequences of pairs (primitive, object) and for every sequence $(p_{1:n}, o_{1:n})$ meaning it estimates its probability of success. We can make several assumptions and observations about such primitives:

- An object is at most called once (since it will be removed).
- The final object must be the goal object.
- The maximum length of such sequence will be, in most cases, 3. Since all objects can be seen from in front of the shelf there can't be any severe obstructions. Moreover, this assumption will almost certainly be the case for scenarios with less than 6 objects. For bigger scenarios it is still quite likely to be the case. Even if it's not the case we will see that the planner will come up with a reasonable plan, from which it will be able to replan and improve its original plan.
- If a primitive is successful we assume the scenario is exactly the same except for the object, which is removed from the scene.
- The probability of such plan to succeed is the product of the probabilities of each primitive to succeed. We can also add a penalty of $(1 - \epsilon)$ for every extra step, to account for mistakes due to successful primitives modifying the scene, not calling perception and making further primitives fail. This also makes shorter sequences preferable, an important point since there's also a time constraint.

Given such assumptions we can do a backtracking algorithm:

1. If it's the third step, select goal object. Otherwise choose an object o from the list of available objects (which can also be the goal object).

```
num_objects: 2
objects:
  -
    ObjId: crayola_24_ct
    binId: 4
    position: [0.1, 0.15, 0.06, 0, 0, 0, 1]
  -
    ObjId: soft_white_lightbulb
    binId: 4
    position: [0.2, 0.05, 0.06, 0.5, -0.5, 0.5, 0.5]
goal_object: 1
```

Figure 6.1: Example of an input for an APC scene. The position of each object include its xyz position and quaternion for the orientation.

2. Compute the probabilities for every primitive of being successful with that particular scenario. Pick the one with highest probability of success.
3. Assume success and remove the object from the scene.
4. Call the function for the new scene, increasing the step.

There are a couple of optimizations which can be done to the basic backtracking:

- As mentioned, instead of searching the set of sequences (primitive, objects) we only search in the set of objects and then greedily select the best primitive for that particular scenario. Notice how different primitives can be better for picking the same objects depending on which other objects have been picked before. Thus, the best primitive has to be computed for every scenario.
- The scenario after picking object A and then B is the same as the one after picking B and then A. Thus, we can turn the backtracking algorithm into a DP algorithm to avoid such recalculations.
- The final probability is the product of all probabilities; thus, if the product of the probabilities of the beginning of the sequence is already worse than the best sequence obtained so far, there's no need to continue the backtracking, since the probability can only get smaller.
- Try promising objects first so that there are more pruned states using the previous strategies.

6.1.2 Logistic Regression

To compute the probability of success we opted for a combination of Gaussian Processes and Logistic Regression.

We separate the probability of success as the product of two factors: the 1-object factor and the clutter factor. The first one is the probability of grasping the object if there's no clutter, while the second is the probability that the primitive fails because of clutter. Assuming independence we can assume the result is the multiplication of both probabilities.

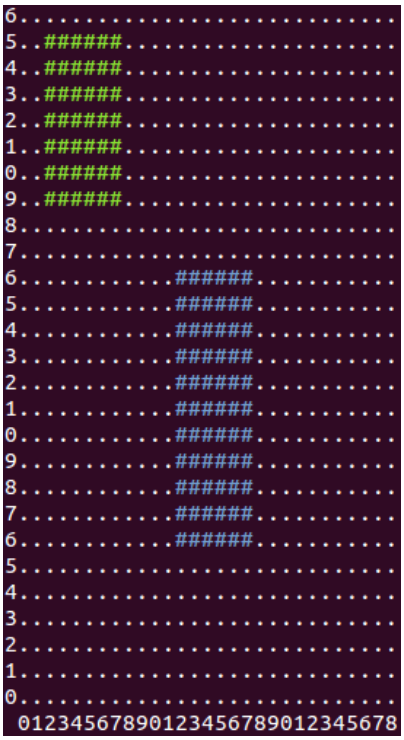

Figure 6.2: Visualization of the APC scene from figure 6.1; seen from above. Approximating all objects by boxes (orientations need not be parallel to the edges); green represents the goal, blue the obstacles.

The 1-object probability prediction is computed with a Gaussian Process. Although we've made significant amounts of research on the topic, we won't go into details because it is pretty detached to the core topic of this research. Very succintly, a Gaussian Process is a way of estimating a continuous function by some evaluations distributed across its range. Assuming random gaussian noise and that values from points that are near are correlated (using a kernel), one can obtain a closed expression for an estimation of such function. In our case, we would have as input variables the pose and position of the object and output whether it succeeded or not. Doing the Gaussian Process would give us the probability of success for points that haven't been seen.

This problem can be simplified assuming that objects can be in positions that are physically stable, which most likely implies that one of the axis of the object is parallel to the z-axis. Thus, another simpler option would be to discretize the set of possible orientations and positions.

For the multiple object term we compute a set of hard-coded features and learn a Logistic Regression on top of them, one for each primitive. A Logistic Regression is essentially a 1-layer Neural Network with a sigmoid non-linearity. It is one of the simplest models for classification, analogous to linear regression for regression problems. Given a primitive p, state s and features $f(s,p)$ the probability is:

$$\sigma(\vec{f}(s,p)) = \frac{1}{1+e^{\vec{w}\cdot\vec{f}(s,p)}}.$$

Our initial set of primitives can be seen on table 6.1. Most primitives either refer to

the goal object or to the relation between the goal object and its surroundings.

Feature	GTFSP
Human probability evaluation	YYYYY
Height front objects/height goal object	YYYYY
Has object in front?	YYYYY
On top of other object?	YNMYY
Distances to closest y-obstacles	YNNMY
x-coordinate	MMMMM
Distance to nearest wall	YMMYY
Distance to nearest obstacle in different angles	MMMMM
Total Volume of Objects in the bin	YNNMY
Height	NYNMM
Linear clutter in y-direction	YNNNY
Object behind?	NNNYM
Object weight	MMMMM
Object max-diagonal	MMMMM

Table 6.1: Set of initial primitives. For each feature, whether it will be included (Y), not included (N) or possibly included (M) in the features for each primitive: Grasp (G), Top Suction (T), Front Suction(F), Scooping (S), Push (P).

We can learn the weights of the logistic regression quite easily with Gradient Descent since the problem is convex. Moreover, since there are less parameters than in a bigger Neural Network we need less training examples to train it.

Finally, at training time, to choose which of the primitives we try we just use ϵ-greedy, explained in chapter .

6.2 Human in the loop

To speed things up, we can leverage human knowledge in multiple ways. In contrast to many ML applications, in this case it is preferable to use domain expert humans (in this case the students who coded the manipulation primitives) as they will have more understanding of what the primitives can and can't do.

Interactive way of creating features. An effort has gone to create a useful visualization tool (examples of which is figure 6.2) that shows the user the scene from above. Not only that; most features are also computed using this discretized table. This allows a human to see the cases where the prediction failed (*caution*: the human also can only look at the training set, otherwise there's risk of over-fitting), understand why and redesign the current features or create new ones. Notice that the weights for the logistic regression

are still calculated automatically. Finally, evaluating the new model on a validation set is used to prove whether the new primitive design is better than the previous one.

Human-labeling of scenes. As previously mentioned there are many drawbacks in using a robot for data gathering, especially if it's the only robot you have to compete in the challenge. Although we have replacements for all gripper parts, there's a high time cost in changing the hardware. Moreover, the robot should be sent with a couple of weeks in advance to the venue of the competition. Finally, human labeling from an image is much faster than robot labeling (which has to physically execute the action, the object has to be put again in the bin, etc).

However, although such labeling is useful it is not entirely representative of the quantity we care about, robot performance. As such, as mentioned in section , human-labeled scenes cannot be in the test set. Finally, since the primitives are pretty complex by themselves this task cannot be outsourced using systems like Amazon Mechanical Turk, because we need domain expertise.

A small amount of hard-coding. Although as a scientist it isn't the cleanest way to solve the problem, as an engineer it would be foolish not to use several hard-coding rules when dealing with obvious certainties. For instance, there are several combinations of object-actions that cannot work: porous objects cannot be suctioned, objects bigger than the gripper cannot be grasped. In this case it is probably better to just put the probability to 0 instead of trying to deal with it probabilistically.

Conditions within primitives. Each primitive has certain conditions specified within its code that are checked before execution, to ensure that the primitive will run safely. All of them assume there are no obstacles, but there are cases where the primitive is bound to fail. Some of them are: grasping a small object near the edge of the bin makes the gripper collide with its tip, trying to use top-suction in an object that is too tall (giving no space to put the gripper between the object and the top of the bin). Since the primitive has these limitations already in its code, we do not need to predict the failure of these preconditions because there's no real impact on those behind negligible computation time (it won't be executed).

Moreover, most of these preconditions are expressed as inequalities. Then, for every inequality of the form $f(s) \leq \epsilon$ we can just make the feature $f(s)$. These features are potentially very useful since being close to a precondition being unmet could be a symptom of potential failure.

6.3 Estimation of number of experiments

An effort was made into ensuring that there was enough time to properly train the whole system. To ensure this, we needed good estimates both for a sufficient number of training examples and for the number of training samples that could be obtained in a 4-week period.

6.3.1 Estimating the throughput of training samples

There are multiple factors that influence the production of training samples. First, we assume that primitives are already optimized and no time is consumed in improving them: pure planner training. The obtention of data follows these steps:

1. A human boots the computer and sets up the robot environment.
2. A human, reading a computer generated list, puts the objects in the shelf, in the configurations he or she desires. All 12 bins are now furbished with several items per bin.
3. The robot now picks *all* objects from the bin. This is in contrast to the challenge, where we only pick 1 object per bin. Notice that to pick an object in general more than one primitive has to be executed (because of obstacles and failures).

Knowing this, those are the following quantities to be estimated:

- **Human shelving per object**: how much time it takes for a human to put an object in the bin. This was evaluated in a 42 object test moving at average pace.
- **Execution time of a primitive**: this was estimated by the coders of the primitives.
- **Constant time per experiment**: time to set up the system.
- **Primitives/obj**: average number of primitives per object, due to failures and removal of obstacles.
- **Effective time/day**: time a human is working with the robot with its full attention.
- **Working days/week**: how many days we do data collection per week.
- **Pessimistic coefficient**: slack for unpredictable events.

Then, we get to table 6.2.

$$\frac{Time}{Primitive} = \frac{ExecutionTime}{Primitive} + \frac{Primitives}{Object}\left(\frac{HumanShelving}{Obj} + \frac{ConstTime}{Experiment}\frac{Experiment}{Objects}\right)$$

$$\frac{Primitives}{Day} = \frac{1}{7}\frac{Workingdays}{week}\frac{EffectiveTime}{Day}\left(\frac{Time}{Primitive}\right)^{-1}\frac{1}{PessimisticCoefficient}$$

Although there's a factor of difference of 13 between the most pessimistic and most optimistic estimations, those ranges are still useful to see what we can achieve. Moreover, we are confident the actual result will be near the Medium estimation.

6.3.2 Estimating the needed number of training samples

It is estimated that for a logistic regression, similar to a linear regression, the number of needed training examples grows linearly with the number of parameters. For the system to be determined we need more training examples than parameters, but to avoid overfitting it is recommended to use $40 \cdot n$ where n is the number of parameters to train.

Factor	Optimistic	Medium	Pessimistic
Human shelving / obj	10s	15s	20s
Execution time / prim	10s	15s	20s
Constant time / exp	60s	90s	120s
Primitives / obj	2.25	1.75	1.25
Time / primitive	17s	27s	41s
Effective Time / day	8h	6h	4h
Working days / week	6	5	4
Pessimistic coefficient	1.1	1.5	2
Primitives per day	1.300	500	100
Primitives per 2-week	18.000	7.000	1.400

Table 6.2: Estimations for the number of primitives per day.

Colors for all three possible systems with their respective needed training samples:
Simple Logistic Regreesion Big Logistic Regression Small Neural Network

4k 8k 20k

Time	Optimistic	Medium	Pessimistic
1 day	1.3k	0.5k	0.1k
3 days	4k	2k	0.4k
1 week	9k	3.5k	7k
2 weeks	18k	7k	1.4k
3 weeks	27k	10k	2.1k
4 weeks	36k	14k	2.8k
5 weeks	45k	17k	3.5k
6 weeks	54k	21k	4.2k

Table 6.3: Estimations for the number of days needed. The colored cells show the first deadline when there will be enough samples to build the system represented by the color.

For 5 primitives, we estimate the number of features between 20 and 40, thus between 100 and 200 parameters in total. This leads to a range between 4000 and 8000 examples. We could also create a 2-layer simple Neural Network: 20 hidden layers with 20 parameters each (to 20 common features) and then 5 output layers, one per primitive, with 20 parameters each, thus $20 \cdot 20 + 20 \cot 5 = 500$ parameters and $500 \cdot 40 = 20,000$ training examples.

All this leads to table 6.3.

6.4 Intermediate experimental results

Although the plan was to start training experiments in early April (and thus have some results by May), delays in the design and construction of the gripper made us postpone training to mid May and thus, no results of this training can be seen at the moment.

Nevertheless, the planning and learning agents have already been implemented. We hard-coded a few reasonable parameters to the Logistic Regressions to estimate the prob-

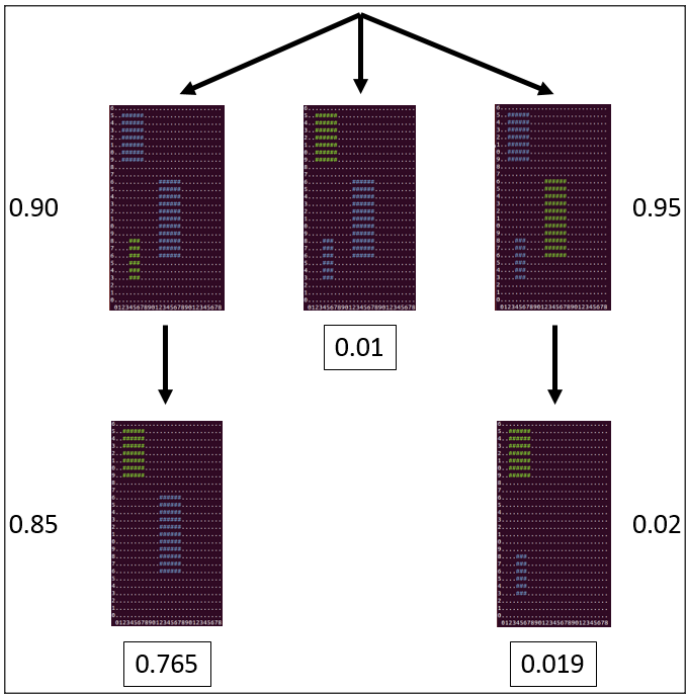

Figure 6.3: Visualization of the planning tree for an APC scene. Simplified to a maximum of 2 primitives. Looking from the top of the scene; in blue we have the obstacles while in green we have the (temporal) goal objects. We want to pick the top-left object. From each height-map the ML algorithm has predicted its corresponding probability, then the planner outputs the best path.

ability of success of (untrained) primitives. With these fake-primitives the planner was already capable of some smart planning capabilities as can be shown in figure 6.3.

7 | Conclusions

7.1 Evaluation of results

Our initial goal of designing and implementing a planner to choose the optimal sequence of manipulation primitives has been achieved. The experiments have not been yet completed due to delays external to the author; however, we are confident that they will be finished in May so that we can train the system before the contest in July. Given that the final system has been designed to perform well in situations of data scarcity, of a few thousand experiments, we believe it will still provide great results in the APC.

On the theoretical sight, we have analyzed a very broad set of techniques and their application to Robotics and we were able to determine, with confidence, their advantages and disadvantages thanks to both theoretical arguments and experiments with prototypes.

Finally, handling the diversity of all approaches and deciding on what to focus and implement has probably been the greatest challenge of this thesis. Given that we have successfully integrated a system that combines ideas of all domains (Deep Learning, Reinforcement Learning and Planning) we are very satisfied with our multidisciplinary approach.

7.2 Immediate continuation of this work

Based on this work, there are several things we intend to do before the Amazon Picking Challenge (July 2016):

Run the experiments: now that the system has been theoretically designed and implemented, we only need to wait for the primitives to be ready, in mid May, to start running the experiments until mid June.

Higher level planning: this work explained the high level planner deciding the actions of a robot. Even higher, we need to implement a much simpler system that decides which bins to attempt to maximize the number of points awarded in the challenge (some objects and bins are awarded with more points). This will likely be done with Dynamic Programming.

Writing a paper: our initial intention was to do a poster in June (at RSS) and submitting a paper in September (for ICRA). Although we dropped the poster because of lack of experiments before the submission deadline for RSS, we have the intention of submitting the paper to ICRA.

7.3 What could go next

Looking at APC 2017 and further work in manipulation, there are several ideas that could be implemented:

Dealing with even more complex configurations in 2017: just recently the APC organization decided to restrict the possible bin configurations: there cannot be objects on top of other objects nor completely occluded objects. This system was designed to support objects on top of others, which gives us a competitive advantage going to 2017, but it cannot yet deal with unseen objects. Some thought has already been given to the issue, but this challenge should be addressed in the future.

Dropping hand-crafted features: from a philosophical and scientific point of view it would be preferable that the system can learn everything without any need for human help; in particular hand-crafted features. From an engineering and practical point of view it is very challenging to obtain comparable results without them. The most approachable way to try it could be to use a Convolutional Neural Network on the height-map from which most primitives are computed. CNNs are very efficient in the number of parameters, which makes it potentially feasible to build such a network.

7.4 A look into the future

In the broad scheme of things, looking at the next few decades for the AI, ML and Robotics communities, there are several important and interesting ideas, in the view of the author, that should be tackled and of which this memoir tries to give a taste.

Successful combination of ML and Planning in Robotics. Robotics, and in particular the field of manipulation, has always dealt with physics models as a way of dealing with reality. However, there's no model that can deal with the full complexity of the real world. Thus, Machine Learning and planning that takes into account this probabilistic nature of robotics have the potential of bringing robotics to the next level.

The role of Reinforcement Learning in Robotics. Out of all the possible ML paradigms, Reinforcement Learning seems by far the most suited to robotics, since they essentially share the same setting. However, Reinforcement Learning has a lot of intrinsic challenges and so does Robotics. Turning this potential of RL applied to Robotics to reality, possibly leveraging planning and physics models, could give robots the autonomy they now lack.

Hierarchical Planning: combining high and low level control. As we have seen in this thesis, we cannot plan nor learn solely low-level actions. This would lead to huge uncertainties, an exponential set of possibilities and a lack of high-level detail to appreciate the general picture. On the other hand one cannot plan only in the high-level idealistic domain, completely disregarding the complexities of the real-world low-level implementation. A great and fascinating challenge will be figuring out how to combine

those high level decisions with low-level implementations in a way that is both learnable and practical in terms of planning time.

Exciting times await us in Artificial Intelligence. We hope this thesis has contributed its grain of salt to the field and given a good view of what's ahead.

Bibliography

(*1*) Wikipedia Feedforward NN image from Wikipedia., 2016.

(*2*) Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, 1–9.

(*3*) Pascanu, R., Mikolov, T., and Bengio, Y. (2012). On the difficulty of training recurrent neural networks. *Proceedings of The 30th International Conference on Machine Learning*, 1310–1318.

(*4*) Mnih, V., Heess, N., Graves, A., and koray Kavukcuoglu (2014). Recurrent Models of Visual Attention. *Advances in Neural Information Processing Systems 27 27*, 1–9.

(*5*) Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer Networks. *NIPS*, 1–9.

(*6*) Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing Machines. *Arxiv*, 1–26.

(*7*) Zeiler, M. D., and Fergus, R. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014; Vol. 8689 LNCS, pp 818–833.

(*8*) Levine, S., and Koltun, V. (2013). Guided Policy Search. *Proceedings of the 30th International Conference on Machine Learning 28*, 1–9.

(*9*) Lozano-Perez, T., and Kaelbling, L. P. In *IEEE International Conference on Intelligent Robots and Systems*, 2014, pp 3684–3691.

(*10*) Boularias, A., Bagnell, J. A., and Stentz, A. (2015). Learning to Manipulate Unknown Objects in Clutter by Reinforcement. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence Learning*, 1336–1342.

(*11*) Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research 4*, 237–285.

(*12*) Andrew Bagnell, J. (2014). Reinforcement Learning in Robotics: A Survey. *Springer Tracts in Advanced Robotics 97*, 9–67.

(*13*) Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008). Monte-Carlo Tree Search: A New Framework for Game AI. *Aiide*, 216–217.

(*14*) Browne, C. B., Powley, E, Whitehouse, D, Lucas, S. M., Cowling, P. I., Rohlfshagen, P, Tavener, S, Perez, D, Samothrakis, S, and Colton, S (2012). A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games, IEEE Transactions on 4*, 1–43.

(*15*) Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic robotics (intelligent robotics and autonomous agents series). *Intelligent robotics and autonomous agents, The MIT ... 45*, 52.

(*16*) Mnih, V. et al. (2015). Human-level control through deep reinforcement learning. *Nature 518*, 529–533.

(*17*) Sutton, R. S., and Barto, A. G. (2012). Reinforcement learning. *Learning 3*, 322.

(*18*) Bauzà, M. Probabilistic Data-Driven Models for the Pushing Problem., Undergrad thesis, Universitat Politècnica de Catalunya, MIT, 2016.

(*19*) MIT Technology Review 10 Breakthrough Technologies 2016., 2016.

(*20*) Fortmann, S. Bias-Variance tradeoff image.

(*21*) Dean, J., and Ghemawat, S. MapReduce., 2010.

(*22*) Lake, B. M., Salakhutdinov, R., and Tnenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science 350*, 1332–1338.

(*23*) Le, Q. V., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G. S., Dean, J., and Ng, A. Y. (2011). Building high-level features using large scale unsupervised learning. *International Conference in Machine Learning*, 38115.

(*24*) Goodfellow, I., Bengio, Y., and Courville, A., *Deep Learning*; MIT Press: 2016.

(*25*) Bishop, C. M., *Pattern Recognition and Machine Learning*; 4, 2006; Vol. 4, p 738.

(*26*) Ng, A. (2012). 1. Supervised learning. *Machine Learning*, 1–30.

(*27*) Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout : A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (JMLR) 15*, 1929–1958.

(*28*) Levine, S., Pastor, P., Krizhevsky, A., and Quillen, D. (2016). Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection.

(*29*) BSA What's the Big Deal With Data?, 2015.

(*30*) Economist, T. The data deluge: Five years on., 2014.

(*31*) Yu, K.-T., Fazeli, N., Chavan-Dafle, N., Taylor, O., Donlon, E., Lankenau, G. D., and Rodriguez, A. (2016). A Summary of Team MIT's Approach to the Amazon Picking Challenge 2015., 8.

(*32*) Dwork, C., Feldman, V., Hardt, M., Pitassi, T., Reingold, O., and Roth, A. (2015). The reusable holdout: Preserving validity in adaptive data analysis. *Science 349*, 636–638.

(*33*) LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature 521*, 436–444.

(*34*) Silver, D. et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 484–489.

(*35*) Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.

(*36*) Correll, N., Bekris, K. E., Berenson, D., Brock, O., Causo, A., Hauser, K., Okada, K., Rodriguez, A., Romano, J. M., and Wurman, P. R. (2016). Lessons from the Amazon Picking Challenge.

(*37*) Abadi, M. et al. (2015). TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems.

(*38*) Todorov, E., Erez, T., and Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. *IEEE International Conference on Intelligent Robots and Systems*, 5026–5033.

(*39*) Simeon, T. (2004). Manipulation Planning with Probabilistic Roadmaps. *The International Journal of Robotics Research 23*, 729–746.

(*40*) Pastor, P., Kalakrishnan, M., Chitta, S., Theodorou, E., and Schaal, S. (2011). Skill learning and task outcome prediction for manipulation. *2011 IEEE International Conference on Robotics and Automation*, 3828–3834.

(*41*) Kaelbling, L. P. A constraint-based method for solving sequential manipulation planning problems., DOI: 10.1109/IROS.2014.6943079.

(*42*) Zhang, M., McCarthy, Z., Finn, C., Levine, S., and Abbeel, P. (2015). Learning Deep Neural Network Policies with Continuous Memory States.

(*43*) Nehmzow, U., Gatsoulis, Y., Kerr, E., Condell, J., Siddique, N., and Mcginnity, T. M., *Intrinsically Motivated Learning in Natural and Artificial Systems*, 2013, pp 185–207.

(*44*) Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2015). End-to-End Training of Deep Visuomotor Policies. *Arxiv*, 6922.

(*45*) Kober, J., Oztop, E., and Peters, J. (2011). Reinforcement learning to adjust robot movements to new situations. *IJCAI International Joint Conference on Artificial Intelligence*, 2650–2655.

(*46*) Stulp, F., Theodorou, E. a., and Schaal, S. (2012). Reinforcement learning with sequences of motion primitives for robust manipulation. *IEEE Transactions on Robotics 28*, 1360–1370.

(*47*) Dogar, M. R., and Srinivasa, S. S. (2011). A Framework for Push-Grasping in Clutter. *Transit*, 1–6.

(*48*) Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement Learning in Robotics : A Survey. *The International Journal of Robotics Research 32*, 1238–1274.