Machine leaning and transfer learning in robotics : application to real world robots

05/03/20 – 30/09/20

Samuel Beaussant

Industrial tutor : Sebastien Lengagne, Mehdi Mounsif

Academic tutor : Alexis Landrault

5ème année Génie Electrique et M2 PAR

Abstract

The field of machine learning has generated scientific interest for several decades due to its ability to replace an exhaustive and precise list of instructions sometimes difficult to code or tedious. In particular, the field of deep learning has become very popular since current technologies allow to benefit effectively from it (dataset availability, GPUs etc...). However, creating a model that learns from data is a complicated and time consuming task. As such, transfer learning, a research field that study how to effectively transfer or reuse gained knowledge, appear to be a very good solution. But despite scientific breakthrough, transfer learning is still in its early age. This internship was carried out at Pascal Institute's research laboratory on robotics alongside Medhi Mounsif, a phd student. His thesis focuses on the transfer of learned robotics skills between agents with different kinematic models. The goal was to design physical applications of simulated experiments in order to obtain real world data on transfer efficiency.

Keywords : Transfer Learning, Robotics, Reinforcement Learning,

List of Figures

Acknowledgement

First of all, I would like to thank my supervisors Sebastien Lengagne, associate professor and Mehdi Mounsif, Phd student for their trust and involvement allowed me to carry out this end of studies project in the best conditions. Thanks to their support but also to their demands, I have I have been able to improve my skills and learn a lot.

I would also like to express my thanks to Mr Landrault, my academic tutor for his support and availability.

I also thank the administrative staff of the Pascal Institute for their help.

Finally, I would like to express my gratitude to LABEX IMOBS3 which funded this internship.

Glossary

ML : machine learning

RL : Reinforcement Learning

SGD : Stochastic Gradient Descent

MDP : Markov Decision Process

VPG : Vanilla Policy Gradient

PPO : Proximal Policy Optimisation

UNN : Unviversal Notice Network

BAM : Base Abstracted Modeling

GAN : Generative Adversarial Network

WGAN-GP :Wassertein Generative Adversarial Network Gradient Penalty

DCGAN : Deep Convolutionnal Generative Adversarial Network

ROS : Robot Operating System

DoF : Degree of Freedom

Contents

Introduction

While the physical hardware of current robots should make it possible to accomplish increasingly complex handling tasks, control algorithms are struggling to exploit their full potential. Deep learning applied to robotics has shown that it is possible to solve tasks that would have failed with classical control algorithms. However, training a model takes thousands, if not millions of iterations, which in the real world would take a considerable amount of time. Futhermore, as learning may be done through trial and error, repetitive mistakes could damage or even destroy the robot. The solution therefore consists of training a robot in simulation and then transferring the model obtained to the real robot. However, done naively, the transfer may not be effective or even impossible. During his thesis, Mehdi Mounsif developed the Universal Notice Network, a method which allows to transfer skills between agents differently shaped. This method has shown very good efficiency in simulations but has not yet been tested on real robots. The goal of this internship is thus to support Medhi Mounsif's work by implementing the UNN on real world robots. Another transfer method between agent, coachGAN, developped by Mehdi Mounsif is presented but due to a lack of time hasn't been implemented on the real robot.

In the first chapter we present the context of this context as well as Pascal Institute. Then we expose the gantt diagramm use throughout this internship. In the next chapter, we discuss the fundamental concepts of machine learning and deep learning in order to give readers without background in this field, sufficient knowledge to understand the rest of this report. The Chapter 3 consists of a brief summary of the state of the art in the transfer of skills learned in robotics. The aim of part 4 is to present the tools used during the different manipulations. It also gives the first results obtained with them. Finally in Chapter 5, we first present the task that a 6 DoF serial braccio robot has to perform. This robot and its 3 kinematic variations are used to solve a dynamic task where a ball must now be in the center of a gutter. We will present the methodology put in place to implement the UNN, from learning the task in simulation to its transfer to real robots. Each of the experimental results presented will be discussed.

Chapter 1

Context

1.1 Motivations

Being in double degree in electrical engineering and embedded systems / master in artificial perception and robotics, I wanted an instership subject that combine both formations. In addition to fulfilling this condition, this internship also allowed me to work on a cutting-edge and very promising subject, namely machine learning in robotics. Deep learning is a field in rapidly expanding and full of promise which in the near future will disrupt a lot of fields including robotics. In particular, the possibility of transferring knowledge and skills between differently shaped robots, as it has been developped during Mehdi Mounsif's thesis, has many applications in industrial robotics in particular where worn out robots could easily be replaced with different robots with minimal training. Futhermore, wishing to continue my master in thesis, I wanted to do my internship in an environment close to the world of research. This internship in laboratory alongside a phd student was therefore ideal.

1.2 Pascal Institute

The intership was carried out at Pascal Institute of Clermont-Ferrand. It is a research laboratory located on the cezeaux campus at Aubière placed under the triple supervision of the University Clermont Auvergne (UCA), the CNRS (Nationnal Center of Scientific Research) and SIGMA Clermont.

Figure 1: Pascal institute

Pascal Institue is the result of the association of 6 research laboratories in different fields ranging from robotic to health and a member of the FACTOLAB; an association with the Michelin corporation to work on the insdustry of the future (human-machine cooperation,collaborative robots etc..). It is also part of the IMobS3 (Innovative Mobility: Smart and Sustainable Solutions) laboratory of excellence which aims at developping efficient, eco-friendly and innovative mobility techologies and a member of the CNRS EquipEx ROBOTEX, a national network of experimental robotics platforms. Pascal insitute is composed of approximately 370 people from all over the world, including temporary staff such as trainees. Its main fields of application are factory, transport and futuristic hospital divided into 5 main research areas, namely:

Image, Perception Systems, Robotics (ISPR) is specialized in the field of Artificial Perception and Vision for the Control of Robotic Systems.

Process Engineering, Energy and Biosystems (GEPEB). The GePEB axis addresses numerous application areas, such as the production of energy carriers, biomaterials, food processes and many others.

Mechanics, Mechanical Engineering, Civil Engineering, Industrial Engineering (M3G). It is a multidisciplinary axis which aims at meeting the scientific challenges posed by industrial requirements in the fields of mechanics and civil engineering.

Photonics, Waves, Nanomaterials (PHOTON) is particularly interested in nanophotonics, nanostructures, microsystems and nanomaterials and electromagnetic compatibility.

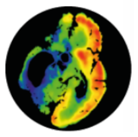


Image Guided Therapies (TGI) brings together 4 research themes: Cardio-Vascular Interventional Therapy and Imaging, Endoscopy and Computer Vision, Image-Guided Clinical Neuroscience and finally Perinatality.

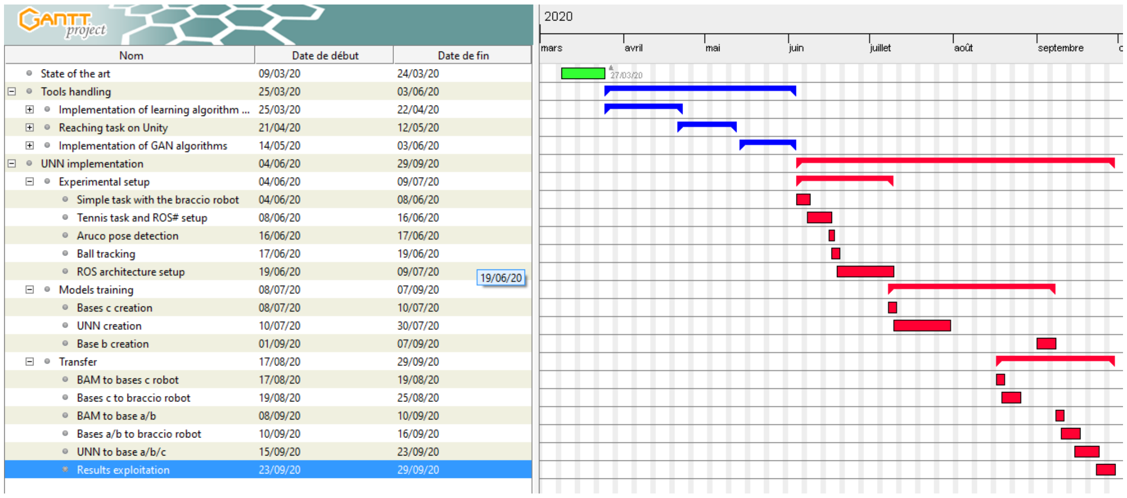
In the case of my intership, I was part of the ISPR axis under the supervision of Sebastien Lengagne and Mehdi Mounsif. This axis is divided in smaller research unit :

- artificial Vision (ComSee)
- modeling, identification and command (MACCS)
- multi-sensory perception systems (PerSyst)
- and hardware and software architecture for perception (DREAM)

As a whole, ISPR is responsible for the research on automatic recognition and real-time monitoring of visual patterns, absolute location of mobile vehicles and modeling and control of robotic systems to name few. More specificly I was working with the MACCS team (Modeling, Autonomy and Control in Complex Systems) whose research focuses mainly on the modeling and control of mobile and manipulative robots, robot vision, active vision, visual control and anticipatory behaviors.

1.3 Gantt chart

The gantt below is the temporal organization of the tasks I performed. As certain tasks were entrusted to me as and when the final task was decided only late, this is a gantt made downstream. However, as of August 17, this is the schedule that I follow until the end of the internship.



Nom	Date de début	Date de fin
State of the art	09/03/20	24/03/20
Tools handling	25/03/20	03/06/20
Implementation of learning algorithm ...	25/03/20	22/04/20
Reaching task on Unity	21/04/20	12/05/20
Implementation of GAN algorithms	14/05/20	03/06/20
UNN implementation	04/06/20	29/09/20
Experimental setup	04/06/20	09/07/20
Simple task with the braccio robot	04/06/20	08/06/20
Tennis task and ROS# setup	08/06/20	16/06/20
Aruco pose detection	16/06/20	17/06/20
Ball tracking	17/06/20	19/06/20
ROS architecture setup	19/06/20	09/07/20
Models training	08/07/20	07/09/20
Bases c creation	08/07/20	10/07/20
UNN creation	10/07/20	30/07/20
Base b creation	01/09/20	07/09/20
Transfer	17/08/20	29/09/20
BAM to bases c robot	17/08/20	19/08/20
Bases c to braccio robot	19/08/20	25/08/20
BAM to base a/b	08/09/20	10/09/20
Bases a/b to braccio robot	10/09/20	16/09/20
UNN to base a/b/c	15/09/20	23/09/20
Results exploitation	23/09/20	29/09/20

Figure 2: Gantt chart

The blue part represents all the learning task I did to handle the useful tools. The red part is the UNN implementation part and all related tasks.

Chapter 2

Machine learning

The goal of this chapter is to provide a brief overview of machine learning (ML) to allow an uninitiated person to understand the rest of the report. We first present basic and general concepts of ML. Then we quickly introduce the deep learning fied and explain how neural networks works. Then, the Reinforcement Learning (RL) paradigm which is at the heart of this intership is further detailled. Finally, in the last section we present the GAN framework used in the coachGAN method.

2.1 Generalities

Machine learning is a sub-domain of artificial intelligence which aim at giving computers the ability to learn via a dataset and a learning algorithm. By learning, we mean the ability of a computer to improve its performance on one or more tasks in total autonomy, that is to say, without having been explicitly programmed for. To do this, the algorithm will build a model from data present in a dataset. This model should allow predictions to be made when new data is presented to it. A daily used example of the use of machine learning is the construction of a mail classification model into two categories: spam or non-spam. In this case, the algorithm's task is to learn to recognize what differentiates spam mail from non-spam mail. Learning the relevant distinguishing features is then done without human intervention.

The usual workflow for a machine learning application is first collecting a dataset large enough (e.g. a collection of several thousand e-mails). If necessary, modify the data in such a way that it is correctly usable (e.g. extracting the face from an image to train a facial recognition model). Separate the dataset into two smaller datasets, the training set which will be used to train the model (represents around 80% of the total dataset) and the testing set which will be used to evaluate the performance of the model on data it has never seen. The next phase is therefore training using one or more learning methods.. Throughout the training, the performance of the model is monitored to ensure that it runs smoothly. Following this, the model is tested on the testing set in order to check its ability to generalize on new examples. If the performances are satisfactory, the model can be deployed in production in order to make predictions from the data provided to it (classification of e-mail i.e. predicting whether it is spam or not).

Depending on the amount of information available and the problem to be solved, three types of learning can be used:

- **Supervised Learning :** In this case, each example present in the dataset has a label indicating how this data should be interpreted by the prediction model. Thus, the model will be able to use this information during training to adjust its parameters and improve its next predictions. In the example of emails, the dataset will therefore contains emails labeled with their category (spam or non-spam). Supervised learning is particularly used in the field of computer vision such as classification or object detection as state-of-art models [1] for image classification on imagenet[2], an image database used for annual competition, now achieve superhuman performance[3] (5.1% of error for a human against 1.3% for the best models).

- **Unsupervised Learning :** In this case, nothing is labeled. The algorithm must succeed alone in finding a structure underlying the data. Learning can be done by grouping data according to their similarities. The algorithm is therefore responsible for dividing the data presented to it into several categories. This technique is called Clustering. This type of learning is very practical when there are no labels for our data. The downside is the unpredictability of the model. Applying unsupervised learning to the problem of emails could, for example, give rise to parasitic categories, ie other than spam and non-spam. Natural language processing (NLP) models trained through unsupervised learning had shown to very be efficient at text generation, sentiment analysis, translation etc... GPT-3 [4] the biggest language model ever made is capable of generating texts that closely resemble texts written by humans.

- **Reinforcement Learning :** Reinforcement learning is at the borderline between supervised and unsupervised learning. Indeed, just like in unsupervised learning, tno labeled set is available. However during training the model still has a signal allowing it to adjust its parameters: the reward function. This function delivers a reward (a scalar) after each prediction. In the context of reinforcement learning, we speak rather of action, the model (called agent) predicts which actions should be taken according to the data provided to it (called observations). Learning is therefore done through interactions with the environment which in return provides it with a reward. Note that some problems in supervised learning can be addressed by reinforcement and vice versa. For example, you could assign a reward based on whether the agent correctly classified an email as spam or non-spam. This type of learning is particularly well suited for robot control and has proven to be very effective in place where classical control algorithm might have failed [5]. As this intership is heavily based on reinforcement learning, this topic will be further detailled in section 2.3.

2.2 Deep learning

Deep learning is a subfield of machine learning that has gained a lot of popularity in the recent years. It is a very powerful function approximator (nonlinear or linear) [6] which makes it a good default choice when it comes to building a model. It consists of several layer of artificial neurones stacked on top of each others to form a network. We feed data as input and a succession of linear and nonlinear transformations are performed on it as the input flow through the network. These transformations are due to three components : *the weights* of the connexion between neurons , *the bias* and the *activation functions* used to introduce nonlinearity. Non-linearity are essentials to build complex model able to model any nonlinear functions given enough training data and time. If we denote by g^* the function we want to approximate, than neural network can be seen as a special kind of parametric function $g(x;\theta)$ with x an input data, that we can train to get as close as possible to g^*. The θ parameter is used to denote the weights and biases of the network. The computation function realized by a single neuron is illustrated below

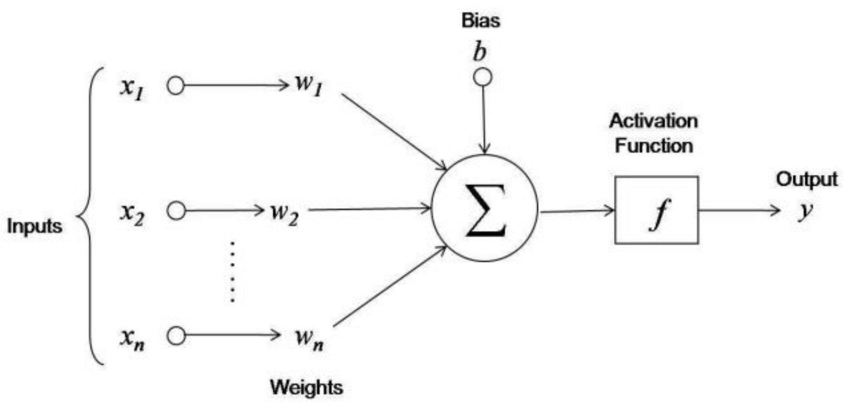


Figure 3: Schematic representation of an artificial neuron

Mathematically, this can be written as

$$y = f(\sum_{i=1}^{n} w_i x_i + b)$$

where w_i is the weigth connecting the x_i input data to the neuron, f is the nonlinear activation function and b the bias adjusting the activation threshold. Considering a layer of multiple neuron, the output vector is

$$\vec{y} = f(W^t.\vec{x} + \vec{b})$$

This output vector will then serve as the input for the next layer and so on until we reach the final layer. This sequential computations on an input to produce an output is called the "forward pass". Once the forward computation is done, the network outputs a single value or a vector of values $\hat{y} = f(x;\theta)$ corresponding to the prediction. To use the example of the mails, this value could be the probability of a particular mail of belonging to the "spam" category. The output is then used to quantify the error of the network by using a *cost function* (sometimes referred as the *loss function*). In the context of machine learning, the error is called the *loss* $L(\hat{y}, y)$ where y is the expected output. Finally, the weights of the neurons are adjusted in proportion of the loss via a optimization technique called *backpropagation*[7] in a direction that minimize $L(\hat{y}, y)$. The backpropagation algorithm compute the gradient of the loss with respect to θ to quantify the contributions of each parameters in the final output. A training step is summarized below :

- First we sample a random batch of data the from dataset. To improve computation speed,the batch size is usualy significantly smaller than the total size of the dataset.
- Then we compute a forward pass on the batch of data to produce $\hat{y} = f(x;\theta)$
- We compute the loss $L(\hat{y}, y)$ and compute the gradient $\nabla_\theta L$

- Finally we update the weights of the network by performing the following update

  $$\theta_{k+1} = \theta_k - \alpha \nabla_\theta L(\hat{y}, y)$$

  where α is a scalar called the learning rate which control the size of the updates.

The optimization algorithm presented above is called the *stochastic gradient descent (SGD).* The training phase is then the repetition of SGD steps. One complete training phase over all data available in the dataset is called an *epoch.* More advanced and efficient optimization technique such as ADAM and RMSprop exists [8],[9]. More generally, training a neural network is an optimization problem where we wish to optimize an objective function (i.e minimizing it or maximizing it).

When designing a neural network, some parameters need to be carefully picked before training:

- the number of neurons per layers
- the number of layers
- the types of layers
- the type of activation functions
- the type of loss function
- the type of weights's initialisation
- the type of optimization technique
- the learning rate
- the type of learning algorithm

Other parameters may be introduced by the choice made for the types of layers, type of loss function, learning algorithm etc... These parameters are decided before training and are called *hyper parameters.* Their value depends on the problem at hand and they are responsible for the training speed and perfomances obtained after training. There is no definitive or theoretical answer to their choice but rather good practices that has proven to give good results. In most cases, they need to be fine tunned in order to find to the values which gives optimal results. Hyper parameters tuning is a very time consumming process especially in large parameters space because this implies that we must train and test the model with different combinations of hyper parameters multiple times. Transfer learning therefore offers a good alternative because it avoids this tedious process by reusing an already trained neural network.

2.3 Reinforcement learning

If supervised learning is seen as learning a ground truth (ex : differentiate dogs from cats in an image), Reinforcement Learning (RL) can be seen as learning a behavior (ex : playing atari games [10]) by trial and error. In the case of a problem like chess where an almost infinte number of combinations exists, treating this in a supervised setting would require a dataset with every move possible and labeled with how to respond to each of them, which is not feasible. Futhermore, the resulting trained model will be, at best, as good as the player which built the dataset, which in turn, makes it impossible to reach superhuman perfomances. This kind of tasks that require solving sequential decision-making problems is particularly well suited for reinforcement learning and has allows computers to beat world champions on a wide range of very complex games [11],[12],[13]. Many robotic tasks such as quadrupedal walking [14] or hand manipulation [15] falls into this category which is why RL is commonly used with robots.

Each RL problems is composed of 2 main components : the agent and the environment. The agent is the entity learning. It has a clear objective or goal such as winning a tetris game or avoiding a maximum of obstacles when driving. In order to fulfill its mission, the agent take actions which he judges to be the best based on the *observations* that the environment provided to him. Each action taken change the environment and the agent is rewarded with a scalar indicating how close he his from reaching his goals or how good this action was. The agent then observe the new state of the environment to take another action and get once again a reward as a feedback. A cycle of (state $\rightarrow$ action $\rightarrow$ reward) is called a time step. Time steps are repeated until the agent achieve his goal, reach a terminal state or the maximum time step (ex : player ran out of time or car crashed). This terminate the *episode*. This all process can be summarized by figure 3 below.

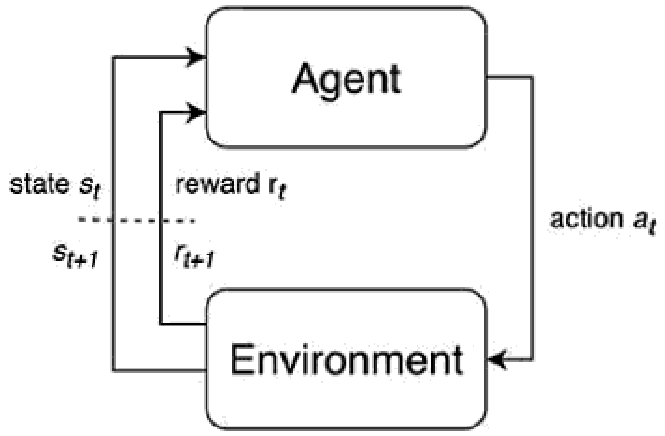


Figure 4: RL control loop

The reward is the signal used by the agent to learn. It depends on the problem we are trying to solve. In most cases, the goal of the agent is to maximize the reward he is getting. As his only clue of how the agent should behave, the reward function must then be carefully craft or inappropriate behavior could emerge.

2.3.1 Markov Decision Process

To describe more formally the process of evolution of an agent in the environment, RL problems uses Markov Decision Process (MDP). The MDP is a mathematical framework used for modeling decision making in a stochastic environment or in situations where outcomes are partly under the control of a decision maker. It is then a really good fit to frame RL problems.

The MDP formulation of a reinforcement learning problem is defined by a 4-tuple :

- S is the set of states (or observations) that the environment can take with $s_t \in S$, the state at step t. It can for example be the joints angular position of a robot arm or the chess board state.
- A is the set of actions the agent can take with $a_t \in A$, the action taken at step t. It can be the new joints angular position for example or the next chess move.
- $R(s_t, a_t, s_{t+1})$ is the reward function which map the action, current state and new state to the reward obtained at each step. It is responsible for providing the agent with a reward at step t as $r_t = R(s_t, a_t, s_{t+1})$.
- $P(s_{t+1}|s_t, a_t)$ is the transition function which map the action and current state to the probability of obtaining new state s_{t+1}. This function account for the stochastic nature that the environment can have. For example, if a robot arm is worn out the new joints state it will land on can be the setpoint plus a small random number. In this case $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ which means that the new state is sampled from the probability distribution $P(s_{t+1}|s_t, a_t)$. If the environment is deterministic (chess game), then $s_{t+1} = f(s_t, a_t)$ All the transistions of an MDP statisfy the Markov property which state that current state s_t and current action taken a_t contains sufficient informations to fully determine the next state s_{t+1}. In other words, $P(s_{t+1}|s_t, a_t) = P(s_{t+1}|(s_0, a_0), (s_1, a_1), ..., (s_t, a_t))$. In the chess example, the chess board state and the chess piece to move at time step t is enough to determine the new chess board state. The transition function and the reward function are specific to the environment and thus not known to the agent
- $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1)...$, a sequence of time step in an episode, is called a *trajectory*.

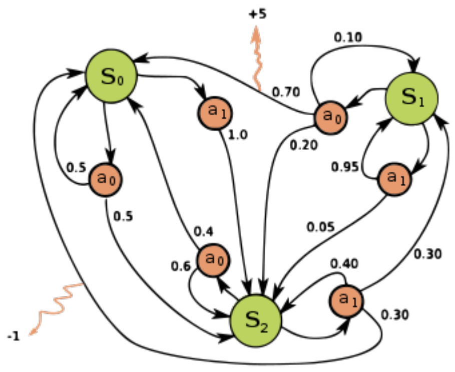


Figure 5: Example of a stochastic MDP with 3 states (green circles)

Figure 4 shows a grapical representation of stochastic MDP. Transition between states (black arrows) are first determined by the action taken (green circle). Then the next state is chosen according to the probability associated with it (i.e the number next to the arrow leading to the new state).

As mentionned earlier, the goal of an agent in an MDP is to maximize the sum reward obtained during an episode. More specifically he is trying to maximize the cumulative discounted reward :

$$G_t = \sum_{k=0}^{T-k-1} \gamma^k r_{t+k+1} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-k-1} r_T$$

also called the *discounted return.* The discount factor $\gamma \in [0, 1]$ is used to adjust the weight given to future rewards. Intuitively, the agent should care more about near future rewards than very far rewards. It is also mathematically convenient when dealing with continuous tasks where $T \to \infty$ as it make the infinite sum G_t converge.

2.3.2 Learnable Function

Deep reinforcement learning make use of the very powerful neural network to learn several function used by RL learning algorithm to solve the MDP. The functions that can learned are the following:

- The agent policy π. It is a function that map s_t to a_t. In other words, it describes how the agent make his decisions. It can be stochastic, in this case $a_t \sim \pi(a_t|s_t)$, or deterministic and $a_t = \pi(s_t)$. The interest of having a stochastic policy is to allow the agent to explore his environment to discover more interesting states.
- The environment model $P(s_{t+1}|s_t, a_t)$ and reward function $R(s_t, a_t, s_{t+1}$ which can then be used by the agent to plan his actions by what would happen for a range of possible choices. This introduce the concept of model-based and model-free algorithm. The former uses a model of the environment to learn whereas the latter doesn't.
- The value function $V^\pi(s)$ and $Q^\pi(s, a)$. The state value $V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[G_t|s_0 = s]$ quantify the reward that the agent can expect to have if it is in state s and follow his policy for the rest of the episode. Its a measure of how good the state is. The action-state value function $Q^\pi(s, a) = \mathbb{E}_{\tau\sim\pi}[G_t|s_0 = s, a_0 = a]$ gives the expected return if the agent start in state s, take an arbitrary action a and then forever after act according to policy π. It is a measure of how good an action is given the current state. However, the value of $Q^\pi(s, a)$ doesn't indicate how much better the action taken was compare to others on average, which is why we need to compute $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$, the advantage.

2.3.3 Learning algorithms

When picking a learning algorithm to solve a RL problems, it is necessary to first determine if the action space of the agent is discrete or continuous. Discrete action space means that the number of actions is finite and countable. It can for example be to move a car to the left or to the right, in this case, 2 actions are available. However, a continuous action space means that the agent must output a real number. The number of possible value is therefore infinite. Continuous action space are heavily used with robot whose motors are controlled by giving a value for the torque. In the case of stochastic policy, the agent rather output a probability distribution, categorical for discrete environment and a gaussian for continuous one. In this context, learning thus mean adjust the parameters to increase or decrease the probabilty of some actions given the state observed as the agent is more and more sure of what actions gave best results. For a gaussian, it shifts the mean and decreases standard deviation.

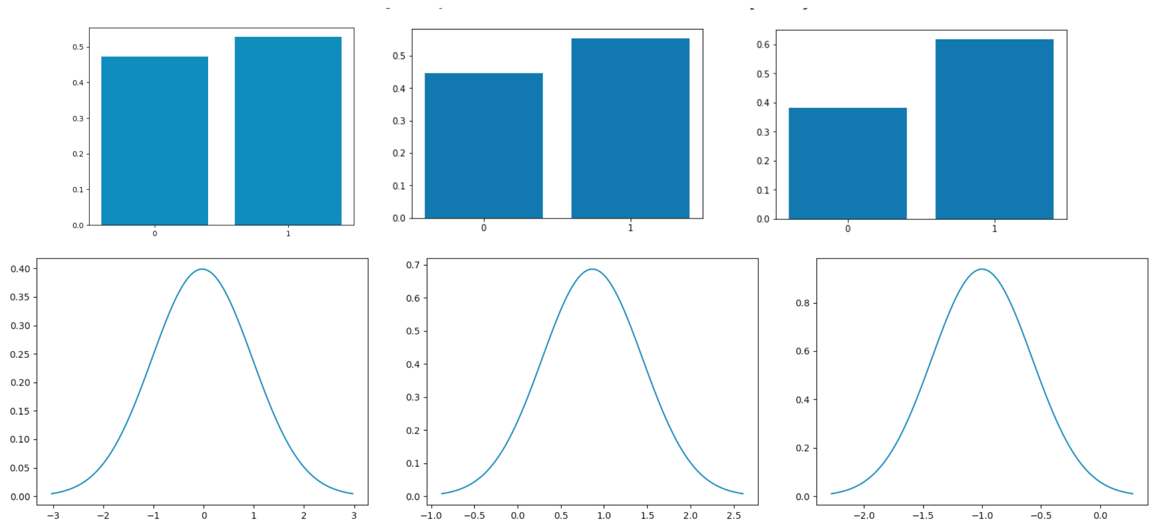


Figure 6: Probability distribution during training
Right column : probability distribution at the begining of the training. Middle column : distribution after 25 epochs. Right column : distribution after 50 epochs.

Some learning algorithm are more efficient with discrete action space and some may even be not usable for a continuous one. Another distinguishing feature is whether the algorithm learn the value functions in order to derive a policy π or learn directly the policy (sometimes with the help of the value functions). The latter category is called *policy optimisation methods* and is model-free. The policy, in this kind of methods, is approximated by a neural network with weights θ updated by SGD or any other optimization methods.We say that the policy is parametrized by θ and we wrote it π_θ. During this intership I worked mainly with policy optimisation methods on continuous action space to learn a stochastic policy $\pi_\theta(a_t|s_t)$.

The next section aims at presenting two very important RL learning algorithm. The first one is the basic policy optimisation algorithm and the last one is currently the most efficient. The implementations and testing of the following learning algorithms are presented in the next section along with the tools used.

Vanilla Policy Gradient

The first algorithm that I implemented is the Vanilla Policy Gradient (VPG) [16]. It the most simple policy optimisation algorithm, it was therefore perfect to familiarize myself with the reinforcement learning paradigm.

When optimizing a policy π_θ, the goal is to maximize an objective $J(\pi_\theta)$. In the case of the Vanilla Policy Gradient method,

$$J_t(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G_t].$$

In other words, we wish to maximize the discounted return we can expect to have at each time step. This optimization problem can be solve using SGD just like any supervised learning problem. However, instead of minimizing the loss $L(\hat{y}, y)$ by gradient descent updates as

$$\theta_{k+1} = \theta_k - \nabla_\theta L(y, y).$$

we replace the loss function by the objective function $J(\pi_\theta)$ and change the direction of the update to maximize instead of minimizing :

$$\theta_{k+1} = \theta_k + \nabla_\theta J(\pi_\theta)$$

with

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)G_t]$$

We can have an estimation of this quantity by sampling a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$ by letting the agent interact with the environment using his policy π_θ. In this case we have

$$\nabla_\theta J(\pi_\theta) \approx \hat{g} = \frac{1}{N}\sum_{\tau\in D}\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)G_t$$

which can easily be computed. This all process will push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return in proportion to G_t. This is why it make sense to discount return as the reward obtained very far in the future after taking an action have little correlation with it and thus should not contribute as much as near reward when doing the update.

The full Vanilla Policy Gradient algorithm is as follow :

```
1 input : Initial parameters $\theta_0$ and learning rate $\alpha$;
2 for  k = 1,2,... do
3     Collect a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$ by running policy $\pi_k = \pi(\theta_k)$ in the
        environment;
4     Compute $G_t$ ;
5     Estimate policy gradient as $\hat{g} = \frac{1}{N}\sum_{\tau\in D}\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)G_t$;
6     Compute policy update, either using standard gradient ascent, $\theta_{k+1} = \theta_k + \nabla_\theta J(\pi_\theta)$ or
        via another gradient ascent algorithm like Adam
7 end
```

Algorithme 1: Vanilla Policy Gradient algorithm

Proximal policy optimization

The proximal policy optimization (PPO) [17] currently achieved state-of-art performance for a wide range of tasks even those with continuous action space. The main idea behind it is to ensure that we can take the biggest possible improvement step on a policy, without accidentally causing performance to collapse. It performs significantly better than Vanilla Policy Gradient without adding too much complexity in the implementation. The objective to maximize this time is

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)A^{\pi_{\theta_k}}(s,a)\right)$$

so the SGD updates will look like

$$\theta_{k+1} = \theta_k + \nabla_\theta \mathrm{L}(s,a,\theta_k,\theta))$$

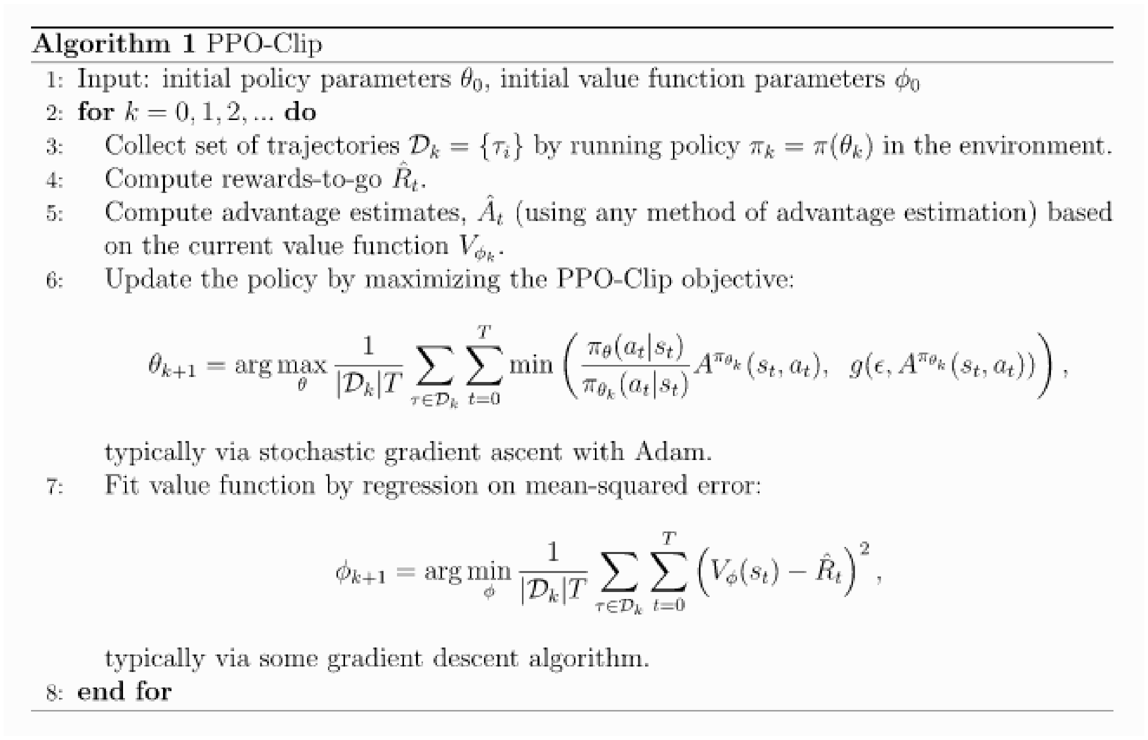
PPO also use the advantage $A^\pi(s,a)$ instead of G_t to improve learning stability and speed. The advantage function $A^\pi(s,a)$ can be estimated using the Generalized Advantage Estimation technique ([18]. This method estimate $A^\pi(s,a)$ with the following equation:

$$A_t^{G\hat{AE}(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V \quad \text{with} \quad \delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

where $V(s_t)$ is the state value function. This recurrence relation can easily be computed. When training with PPO, 2 additionnal hyper-parameters value must then be selected, λ and ϵ To approximate $V(s_t)$ we use a neural network to do a linear regression with mean squared error with G_t as the target. The neural network which will be use to approximate $V(s_t)$ is called a critic as it provide the agent with meaningful feedbacks for his policy updates. The SGD update equation thus look like this

$$\phi_{k+1} = \phi_k + \nabla_\phi \sum_{t=0}^{T}(V_\phi(s_t) - G_t)^2$$

where ϕ are the parameters of the neural network approximating $V^\pi(s)$. PPO therefore make use of 2 neural networks, one for the policy and one for the critic. The whole algorithm is described below ($\hat{R}_t$ is the same as G_t)



Algorithm 1 PPO-Clip

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

2.4 Generative Adversarial Network

GAN's algorithms are part of the unsupervised learning class. They currently achieve state-of-the-art results on a wide range of generating tasks especially realistic image generation. Their goal is to captures the training data distribution in order to generate similar samples. The GAN framework[37] uses two neural network to achieve this.

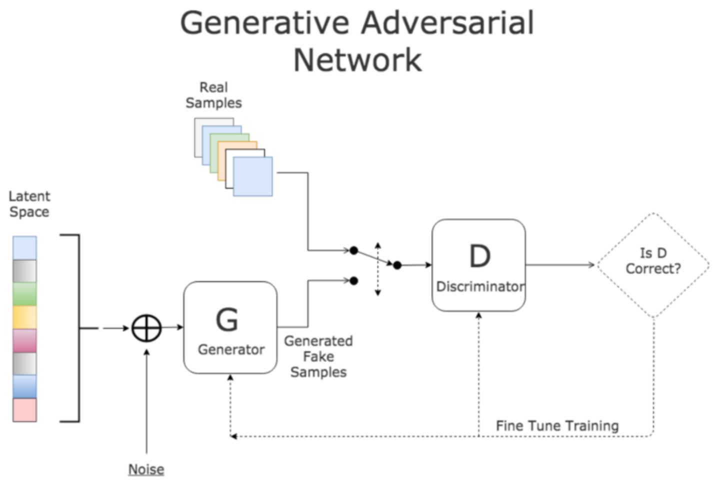


Figure 7: GAN framework

The generator network take a vector of random noise z, sampled from a uniform probability distribution $p_z(z)$ as input and output generated samples $G(z)$. Its goal is to get generated samples $G(z)$ as close as possible to the training data x by modeling the data probability distribution $p_{data}(x)$. The other network is called the discriminator. Its goal is to learn to discriminate real data x from generated data $G(z)$. The processus is as follow : G, the generator, generate a sample $G(z)$ and the discriminator D output $D(G(z))$ which is, according to D, the probability that the $G(z)$ come from the real data distribution. If $D(G(z)) = 1$, then the discriminator is absolutely sure that the generated sample is real, $D(G(z)) = 0$ is the inverse. The discriminator's goal is to maximize the probability of assigning the correct label to both training examples x and samples from G, whereas the G try to minimize it. As D becomes more efficient, G must adapt by generating more convincing samples, which in turn force D to get better. The whole process called a minimax game, encourage both networks to improve as they compete against each other. The objective function to optimize then take the following form

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

G is trying to minimize this objective function by "fooling" the discriminator (i.e get $D(G(z))$ close to 1) while D want to maximize it (i.e get $D(G(z))$ close to 0 and $D(x)$ close to 1). Both G and D must be trained in parallel. The learning algoritm is as follow :

```
for number of training iterations do
  for $k$ steps do
    • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:
        $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$
  end for
  • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:
      $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$
end for
```

However, the classic GAN suffers from unstable training and sometimes lack of variety. The WGAN-GP [38] (Wassertein GAN with gradient penalty) tackle or mitigate these issues by changing the objective function. Another variant DCGAN [39] has demonstrate better sample quality by using convolutionnal layers instead of fully connected layers. I tested my GAN's implementations by generating new samples from two datasets, MNIST and a custom dataset of pokemon. To train the models, I used the google colab notebook, a cloud computer with a GPU available to speed up neural networks training. Below some of the samples I obtained :

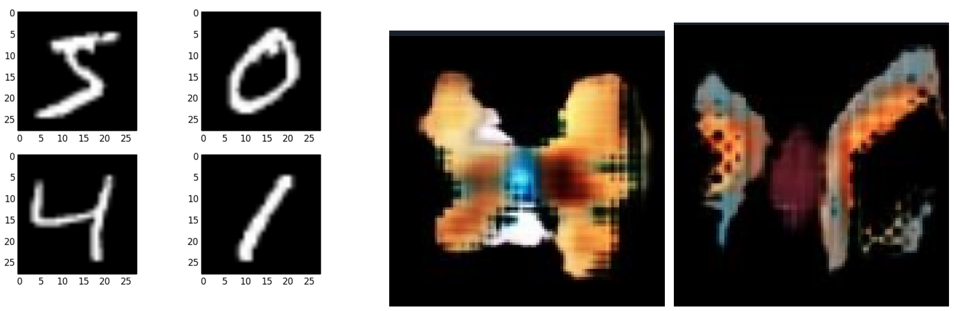


Figure 8: Generated samples after training the generator
On the left side samples from a GAN trained with MNIST. On the right side, pokemon generated by the GAN model.

The pokemon generated are not of a good quality as every pokemon is quite singular and thus the model failed to understand what attributes a pokemon should have.

Chapter 3

State of the Art

This chapter is a brief summary of transfer learning. It presents the different types of transfer possible : one model multiple tasks or one task multiple agents. The former is a popular approach in deep learning where a model trained on one task is re-purposed on a second related task. The latter, currently less explored in the scientific literature, focused on using the knolwedge learned by a task expert model as a starting point to allow another model to efficiently solve the same task with minimum re-training.

Training a model is a very expensive process in terms of time. Very large neural network may require days or even weeks of training [27][28]. Given the very large amount of resources needed to train the most recent models, it is now very common to use a pre-trained model and finetune its parameters on the target task. When dealing with computer vision or natural language processing, segmentation of the knowledge is naturally done as each layer built an increasingly complexe representation of the input as we go deeper in the network [29]. For instance, in the case of digits recognition, the first layers could recognize small edges, the next one borders, the next one basic shapes and so on. In this configuration, detecting shapes and borders is common to every image recognition tasks so the neural network may very well be used for other tasks of computer vision. The only layers that may need to be re-trained are the one specific to the task [30][31]. However, transfer learning in RL is quite challenging particularly with robotics control tasks.

3.1 Transfer learning applied to deep reinforcement learning

Deep Reinforcement Learning methods are known to have an especially poor sample efficiency meaning that a lot of sample are needed to train a policy. Furthermore, in the context of deep reinforcement learning, neural networks are too shallow to allow such hierarchical representation of the knwoledge. As a consequence, task related and agent specific knowledge are mixed which lead to inefficient transfer. As such, knowledge acquired during training by an agent on a task is not re-usable by another agent for the same task. Furthermore, RL agents tend to overfit to their environment. This implicitly means that a robot trained whith a specific domain in simulation will perform poorly in the real world or on another task where the domain will most likely be different. Domain here is meant in the mathematical sense, that is, the set of inputs of a function. In the RL context the domain thus mean the set of states of the policy network. To tackle this issue, multiple techniques focusing on simulation to real world transfer exists based on domain adaptation

[32] or domain randomization[33]. In [32], the authors bridged the gap between real world domain and simulation domain by using a GAN. The generator was trained to translate simulated image into realistic looking image to train the models. Once deployed in the real world, both domains were close enough to obtain satisfactory performances. Domain randomization on the other hand, focuses on randomizing rendering in the simulator. With enough variability in the simulator, the real world may appear to the model as just another variation. Thanks to this approach, the authors in [33] successfuly transferred a model train in simulation to a real world hand robot able to solve a rubiks cube. However, these techniques does not permit segmentation of the knowledge and only make the trained agent robust enough to resists little variations in the states distribution. Consequently, it is not possible to transfer knowledge between differently shaped agent with these methods.

3.2 Universal Notice Networks

The Universal Notice Network (UNN)[34], developped by Mehdi Mounsif, is a novel approach which aims at dividing task and agent specific knowledge during training (cf Figure 9). In the UNN framework, the network is composed of 3 parts : the input base m_i^r and the output base m_o^r which are specific to the robot, and the UNN, m_u^T specific to the task only. The input base map the intrinsic information, $s^{i,r}$, related to the robot constitution, into $s^{i',r} = m_i^r(s^{i,r})$ robot-agnostic and understandable by the UNN. They are the part responsible for interfacing the robot with the UNN. The UNN then receive $s^{i',r}$ as well as s^T, specific task information, as input and output $o^{out} = m_u^T(s^T, s^{i',r})$. The UNN instruction is then re-mapped to the robot space by the following transformation $m_o^r(o^{out}, s^{i,r})$ which yieds a^r the effective action taken by the agent. The UNN focus solely in the resolution of the task. The whole process can be seen as the following pipeline :

$$a^r = m_o^r(m_u^T(s^T, m_i^r(s^{i,r})), s^{i,r})$$

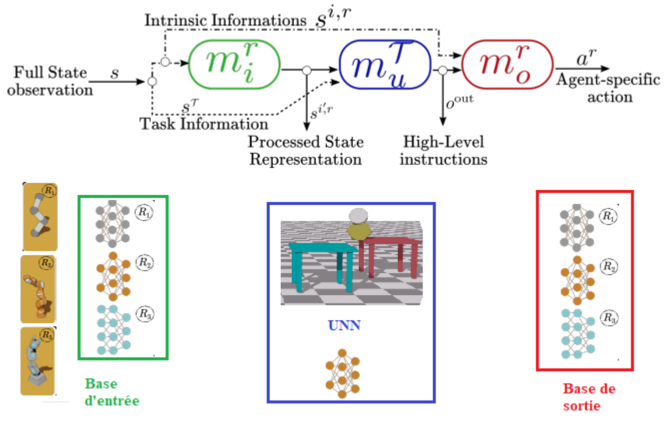


Figure 9: Schematic representation of the UNN

To approximate the three mapping m_i^r, m_o^r and m_u^T it is possible to use a neural network for each and concatenate them at inference time. It is also possible to use analytical methods to obtain

m_i^r and m_o^r. Each of the neural network will need to be trained separately before being put together and fine tuned if necessary. Thanks to this approach we can segment the knowledge necessary to solve a task by decoupling it from robot control

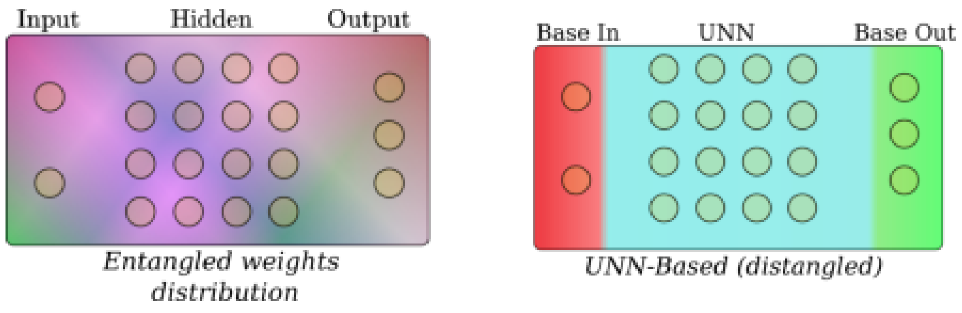


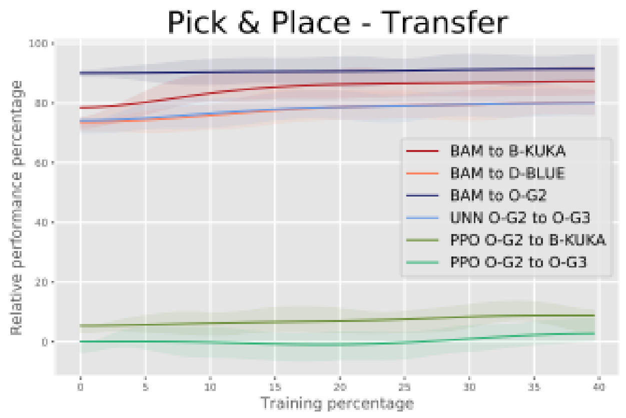
Figure 10: Comparison between a standard training of a shallow neural network in the left (no segmentation) and the UNN framework.

The UNN module can then be transfered to diffently shaped agent provided that their bases are created. This way we obtain reusable skills. This also implies transfer between simulation and real world robot aswell. Indeed, the UNN also tackled this issue by considering the real robot and the simulated as two different robots, each with their own bases. To train the UNN, it is possible to use the Base Abstacted Modeling (BAM)[35] which assimilates the robot to its effector, thus, making no assumption on the robot's constitution and focusing solely on solving the task. This is equivalent to using a robot with infinite DoF. As shown in [37], using BAM allows faster convergence of the policy and a more defined knowledge segmentation, which, in turns, facilitate transfer.

To test the efficiency of the BAM method, Mehdi Mounsif in [37] proposed the following simulation setup :

- Three mobile robots were considered each one with a different mobile base and manipulator type.
- The tasks chosen for the experiments were a simple reaching task, a pick and place task and a stacking task (the goal is to stack a number of cubes).
- Three methods were compare against each others : vanilla transfer (no transfer learning method used), the UNN approach and the BAM approach.

The robots were first trained to perform the reaching task so as to acquire basic motor skills. Then one the robots was trained on the stacking and pick'n place task using PPO, the UNN approach or the BAM method. Finally, the models obtained are passed to the others robots and the performance (mean cumulative reward per episode) is compared. Results are presented in figure 11.



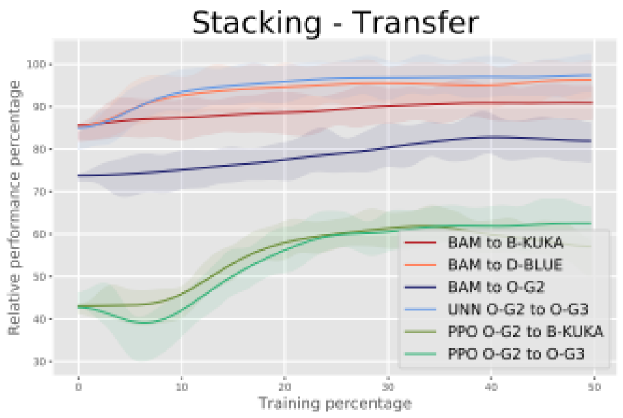


Figure 11: Performances after transfer
The y-axis measure the percentage of the initial mean cumulative reward recover after transfer. The x-axis measure the percentage of the initial training time used to fine tune performance after the transfer (0% means no no fine tuning).

As shown in the figure 11, UNN-based techniques (including the BAM method) recover more than 70% of the initial agent mean cumulative reward instantly after transfer. In the stacking task, full performance is almost recovered after at worst 50% of initial trainng time. In the contrary, vanilla transfer with PPO shows a significant decrease in performance. In the pick'n place case, vanilla PPO transfer is not efficient and even detrimental as the transfer efficiency is nearly 0 %.

3.3 CoachGAN

During his thesis Mehdi Mounsif developped another method to transfer knowledge between agents despite a potentially different kinematic structure, namely coachGAN [36]. The coachGAN method make use of the GAN framework to allow skill transfer between agents. However, unlike generative models, the discriminator network is not a mean to train the generator, it is the other way around. The coachGAN base concept is to train a "teacher", represented by a discriminator network, to distinguish between actions proposed by an expert on the task (i.e an agent already trained) and the teacher generator's. As before this a minmax game where the real data comes from the expert and the generated data come from the teacher's generator. The loss function is then

$$min_{G_T} max_{D_T} = \underset{x_{ISV} \sim P_T}{\mathbb{E}} [log(D_T(x_{ISV})] + \underset{\tilde{x}_{ISV} \sim P_T}{\mathbb{E}} [log(1 - D_T(\tilde{x}_{ISV})]$$

where Pr is the real data distribution (given by the expert), Pg is the generator distribution, defined by $\tilde{x} = G(z)$. Once the teacher is trained to convergence, the discriminator encompass all the expert knwoledge and is thus able to distinguish suitable actions for the task from inappropriate ones. The first step of the transfer processus is thus over and the generator network can be discarded. Next step is to use the teacher acquired knowledge to train a "student", that is, an agent without any prior knowledge on the task. The teacher will evaluates the student proposed actions given what he learned with the expert. The student will then use the teacher's "opinion" to backpropagate his error and improve his performance. This can be seen as a supervised learning problem where the teacher's dicriminator contains the ground truth that the student is trying to learn. In this case, the loss function is

$$L_S = ||D_T(R_{ISV}(G_S(z))) - \alpha_{D_T}||^2$$

This a standard minimization problem where the student, $G_S(z)$, tries to get as close as possible to the expert distribution in order for $D_T(R_{ISV}(G_S(z)))$ to minimize the gap with α_{D_T}, a positive scalar representing the highest teacher discriminator estimation. In the case of a classic GAN discriminator,$D_T(R_{ISV}(G_S(z))) = 1$ means the discriminator is absolutely sure that $G_S(z)$ was produced by the expert, thus $\alpha_{D_T} = 1$. However, as we are trying to transfer knwoledge to a differently shaped agent, we have to take into account the fact that the action space of the teacher and the agent doesn't have the same dimensionality. To account for this issue, Mehdi Mounsif proposed to use intermediate state variable (ISV) to define a robot agnostic learning signal. The teacher doesn't directly evaluate the raw student's proposition of action but rather the transformation $R_{ISV}(G_S(z))$ understandable by the discriminator. A simple case of ISV would be the effector position for a serial robot. In this case, the R_{ISV} operator would simply be a forward kinematic model that map joints state to effector position. As R_{ISV} is used in the learning pipeline of the student it must be differentiable to allow backpropagation. Figure 12 summarized the whole process

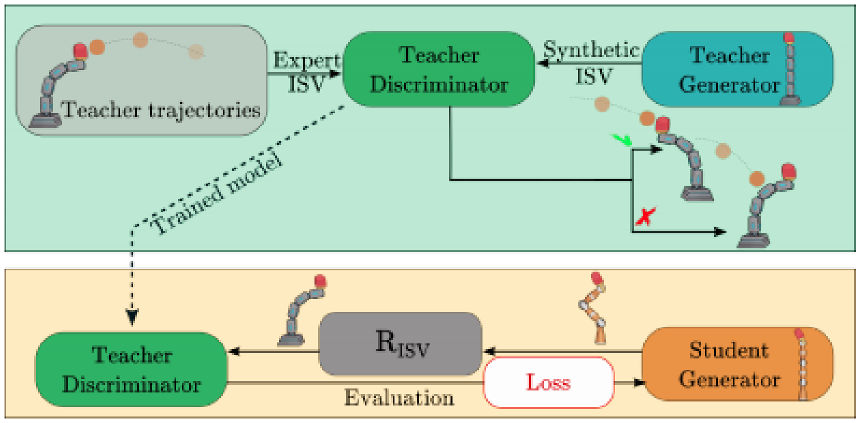


Figure 12: CoachGan's transfer process.
Top green frame represent the teacher training process that is then used for training the student generator (orange frame).

Due to lack of time, I could not implement coachGAN to test it on the real robot with the gutter task.

Chapter 4

Tools used and first results

Throughout this internship I had to use a number of tools to carry out my tasks. The purpose of this part is to briefly present them in order to be able to refer to them later in this report. In section 4.2 and 4.3 I also discuss some of the first results I obtained after training models in different environments.

4.1 Pytorch

Pytorch[19] is a very convenient open source python framework used to build and train machine learning model. It provides a lot of libraries and tools to build all kinds of neural networks. More fundamentaly, pytorch makes it easier to perform the tensor computation necessary for deep learning as it handles a lot of boiler plate code. One of the most interesting feature of this framework is the autograd package which allow us to perform backpropagation effortlessly. Pytorch also comes with all the most common neural networks layers, activation functions, weight initialisations, mathematic functions and many more. The tensor concept (a mutli dimensionnal array) at its core is easily convertible into numpy array to facilitate interface with other libraries or tools. It was developped by facebook and launched in september 2016. It is currently one of the most popular machine learning framework and the main competitor of TensorFlow developped by google. The main reason I used Pytorch over TensorFlow at the begining is because of its python-like syntax, easier to learn and its flexibility. It is overall a better suited framework for reserach purposes where TensorFlow, less flexible but more optimized, is mainly used in a production context. However, due to technical constraints discuss in chapter 5, the models were implemented on the real robot with TensorFlow[44].

The first task I used Pytorch for was to build a neural network able recognize the digits of the MNIST datasets[20] (see figure 13). It is the equivalent of an "Hello World" programm for deep learning. I also performed the same task but this time using only the numpy library, a python library used to perform array computation. Building and training neural network with Pytorch involves a lot of "leaky" abstractions (automatic differentiation, built-in neural networks layers etc..) which hides a lot of the programming complexity that a neural network involves. The goal was thus to obtain a better understanding of deep learning from the computer science/programming point of view. It also gave me an idea of how pytorch is implemented under the hood.

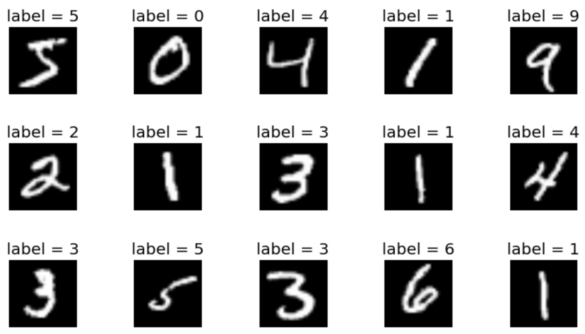


Figure 13: Sample from the MNIST dataset

4.2 OpenAi gym

OpenAi gym [21] is an open source toolkit containing a lot of environment for developping and comparing reinforcement learning algorithms. It was developped by the OpenAI company in order to facilitate the research in the reinforcement learning field. The environment ranged from Atari games to robotics environment. The software architecture of openAI gym is inspired by the figure 4. In this framework the agent is an independant entity that get observations from the environment and rewards from the environment while acting.

To test my implementation of VPG, I first used the environment cartpole. It is a very basic control task where a cart that can be controlled with 2 actions, left or right, must maintain as long as possible a pole vertically.

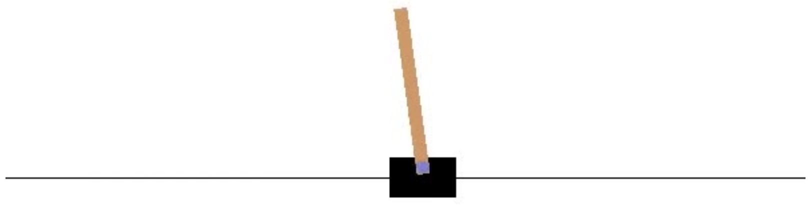


Figure 14: cartpole environment

The action space is therefore discrete. The observations for this task (i.e the input of the policy network) are cart position, cart velocity, pole angle and pole velocity at tip. Then the network output 0 or 1 to chose between left and right. The reward function associated with this environment is +1 for every time step before termination. The episode terminate when the pole fall below a threshold or the total reward for the episode reach 200.

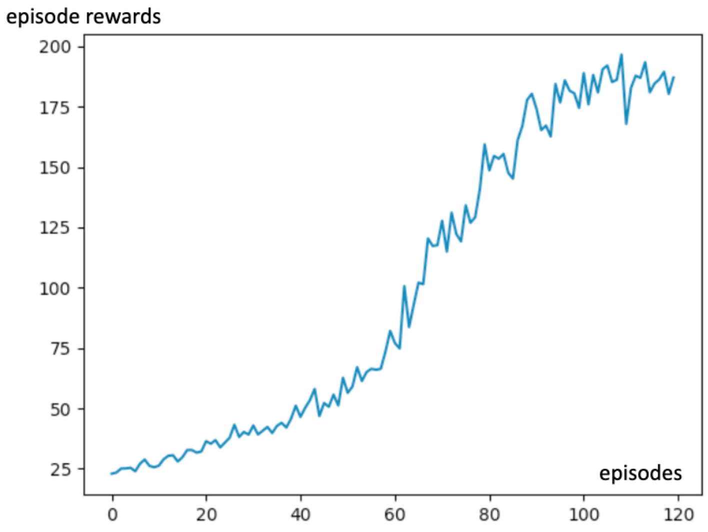


Figure 15: Evolution of the reward obtained on cartpole with VPG agent.

As shown in figure 15, the agent successfully obtained the 2OO total reward on an episode. The VPG was able to solve pretty easily this task. This algorithm works well for this kind of simple task but failed completely with complex task such as BipedalWalker in which a bipedal robot must travel as far as possible by walking.

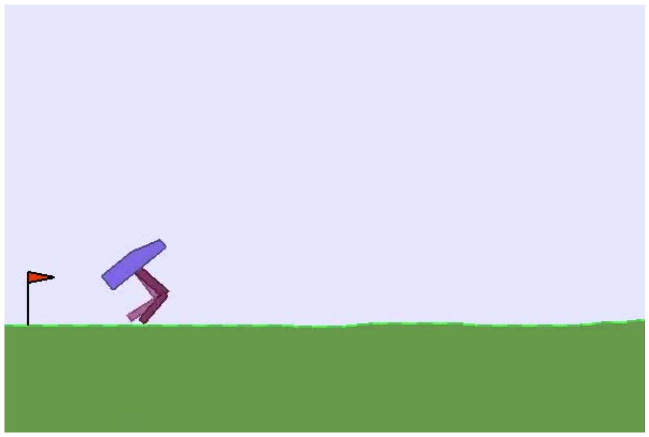
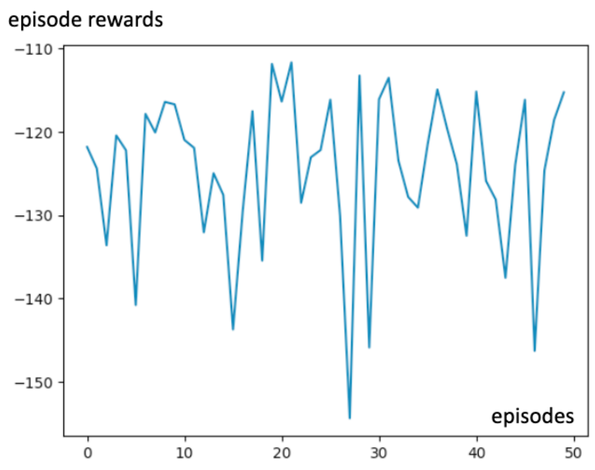


Figure 16: BipedalWalker environment

Here the action space is continuous and the agent output four value corresponding to the torque/velocity of both hips and knees. For this task, the policy network get the speed and the angular position of the 4 joints of the robot. The agent receive a positive reward for every step forward and -100 for falling. Applying motor torque also costs a small amount of points to reward cost effective walking strategy.



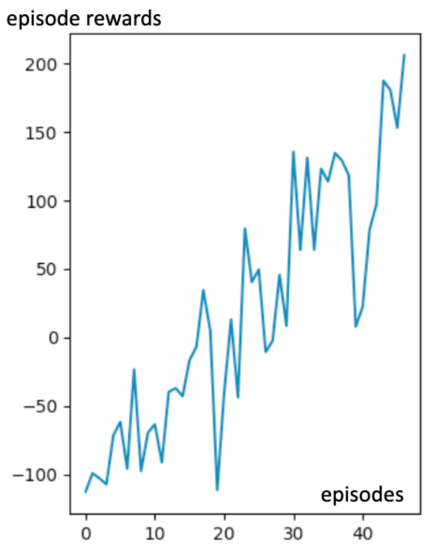


Figure 17: Rewards obtained per episode for BipedalWalker
On the left is plotted the reward obtained by the VPG agent and on the right the one obtained by PPO agent.

As we can see in the above figure, the agent performance doesn't improve, even after 50 epochs. The main drawbacks of this algorithm is the fact that policy update are not guaranteed to improve performance. As a matter of fact, the Vanilla Policy Gradient algorithm tend to give unstable learning which can cause the performance to collapse without being able to recover from it.

However, as mention earlier, PPO is a much more reliable and effective learning algorithm than VPG. As such it was able to solve the BipeddalWalker environment in 50 epochs where VPG failed completely. As we can see in figure 17, the agent is effectively learning and increasing the reward he is getting over. The learning curve, without being monotonic, shows a constant growth trend.

4.3 Unity3d and ml-agents

Latter during this intership, I needed to build my own environment in order to train the model that will be used to control the real world robots. Unity[22] is a very popular game engine. Thanks to the physic engine that it possesses, Unity can aslo be used to make fast and accurate simulations. The ml-agents package [22] is an initiative of Unity which aims to offer the possibility of doing reinforcement learning on their 3d physics engine. The simulation environment is created in a *Unity scene* and the behavior is defined inside a C# script, an object oriented programming language very close to C++. Other technologies such as MuJoCo, another physic engine, offers the same service as Unity. From a technical continuity standpoint, it appeared more suitable to rely on this software for my experiments as the previous simulations realized by Mehdi Mounsif were also created under Unity. It also allowed me to benefit from his expertize.

The first tasks I carried out using Unity were reaching tasks where a robot arm must learn to place his effector at a target position. The first thing to do was to create the unity scene where the simulation would take place :

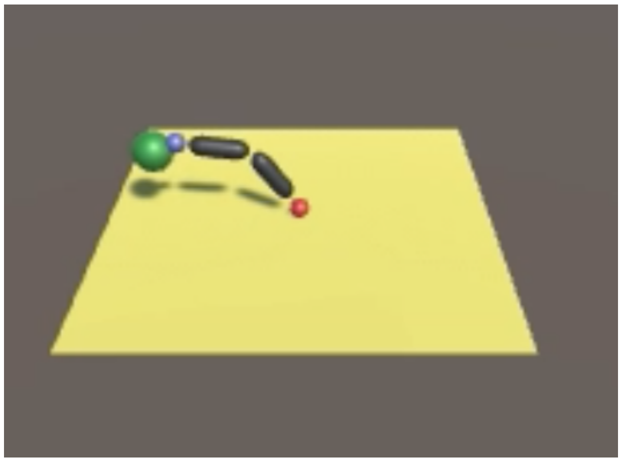


Figure 18: Reacher task.
The target position is the green sphere and the effector is the blue sphere. The robot has 2 joints : base (red sphere) and middle (the intersection of the 2 black cylinders) .

Several variations of this task were performed depending on whether the robot could move only in a 2D plane or in 3D space. Initially, the robot couldn't rotate its base allowing only movements in the plane to make the task easier (2D tasks). The robot thus had only 2 degrees of freedom (rotate base along z axis and middle joint along z axis). Then I added base rotation along y axis to allow 3D movements. The other variations depended on whether we gave the robot a torque or an angular setpoint to make it move. These simple tasks were intended to help me familiarize myself with Unity. The observations that the agent was getting for this task are the joint states and the target position (relative to the robot base). Thus the state space is $S \in \mathrm{R}^4$ for 2D tasks and $S \in \mathrm{R}^5$ for 3D tasks. The target was moving around in the scene randomly to allow the agent to train with as many target positions as possible. The reward the agent was getting each time step is

$$R(s,a) = \begin{cases} r & \text{if } \ d_{t,e} < \delta \\ -\alpha d_{t,e} & \text{else} \end{cases}$$

where $d_{t,e}$ is the distance between the target and the effector, δ define a small range under which r, the reward, is a small positive constant (0.1 in this case) and α is a scaling factor.

For the learning part, I used the PPO implementation of ml-agents which is very reliable and complete. The hyper-parameters such as λ, ϵ, γ, the number of neurones per hidden layers (hidden_units) and the learning rate were specified in a config.yaml file (cf Annexe A I decided to stick with the recommended values for these hyper-parameters More details for the meaning and choice of the other parameters can be found on the ml-agents documentation [24].

The behavior of the agent is handled inside an Unity script(cf Annexe B). It is handled by 3 functions. Inside the collectObservations we define the information that the agent is getting for his task by passing it to the VectorSensor. Neuralnetworks performs better when its input are inside $[-1;1]$ range so it is very important to normalize all the input data. It also allows the model to be used in an environment where the states have different ranges than the ones he was trained on (the real world for instance). OnActionReceived and OnEpisodeBegins are both callbacks respectively used when the agent made a decision (and thus outputed the vectorAction), or when a new episode begin. OnActionReceived must contains the code that make use of the agent decisions (for example applied the torque).

4.4 OpenCV

All the tools mentioned so far are used to create and train a models in simulation. In this section and the next one, we present a set of tools used to implement the agents in the real robot. OpenCV (Open Computer vision)[25] is a software library specialized in computer vision. It brings together several useful feature for image processing (smoothing, filtering, thresholding ...), video processing (reading webcam, detecting object...) and even classical machine learning algorithm such as K-means or Random Forests. In my case, I used OpenCV to retrieve useful observations data with the webcam.

4.5 ROS

The Robot Operating System (ROS)[26] is an open source software development platform for robots. It is the grouping of several features that facilitate the development of flexible and modular applications for robotics and provides a inter-process and inter-machine communication architecture. ROS is compatible with a lot of programming language such as C++, Python, JAVA etc.. ROS processes are called nodes and each node can communicate with others via topics. Topics can be thought as pipelines where messages travel from one node to another. Each topic is associated with one kind of message either standard (provided by ROS) or custom. Messages are language agnostic which means that a Python node and a C++ node can esealy communicate. Nodes are seperated in two kinds : Publisher and Subscriber. A publisher node send data over a topic whereas a subscriber receive data. A node can subscribe to a topic while publishing data to another topic. I used ROS with python to implement on the real robot the models obtained in simulation in order to benefit from available features (serial-communications, language agnostic interface). In the context of this intership, ROS is used as software architecture to facilitate data echange between the different parts (models, observations retrieved from the webcam via OpenCV, arduino code of the robot etc ...) The ROS process is summarized in the figure below

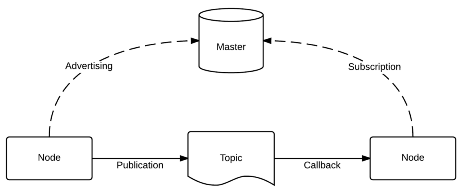


Figure 19: Operating diagram of ROS

Chapter 5

Real world manipulation

The final goal of this intership was to implement and test the UNN on real world robot to obtain data on the efficiency of UNN transfer other than in simulation. It was thus necessary to chose a well suited task. Multiple tasks have been considered but for technical reasons that will be discussed further, avorted. There is 2 kind of transfer to consider : simulation to real robot transfer and robot to robot transfer. For the first one, transfer efficiency can be improved with two techniques mention earlier, domain randomization and domain adaptation. For the sake of simplicity and in order to determine the benefits of the UNN only, none of this methods were used. Two settings were considered :

- The agent is trained in simulation with Unity along its bases and then transferred to the real robots. Its performance are then measured according to task-specific metrics on the different robots.
- Agent is trained from scratch in the real world by interacting with the environment. Performance obtained in this setting will serve as a baseline to show the UNN benefits.

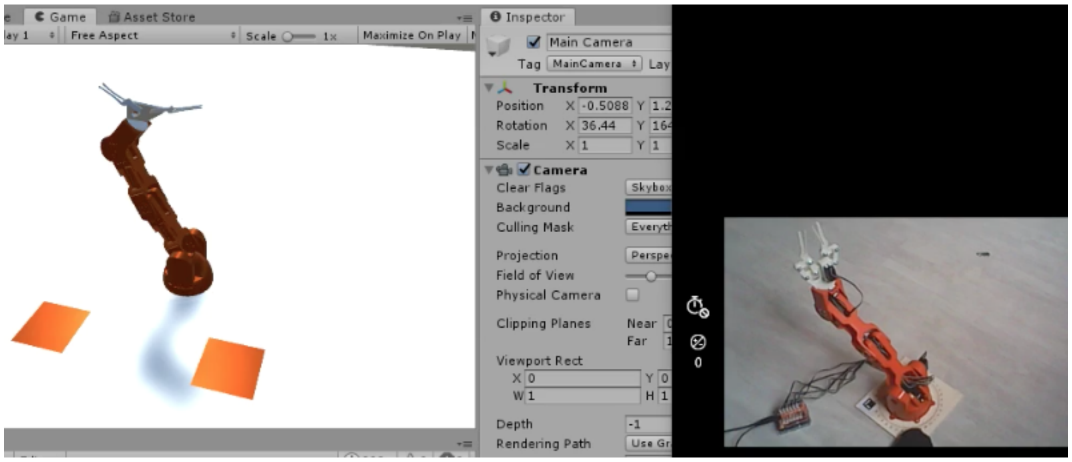
As mentionned earlier, the UNN is used to transfer knowledge between robots. However, for technical reason, only one robot was available. In order to emulate multiple different robot, 3 bases configurations have been considered :

- **a:** The first one uses analytical methods to create the bases (i.e forward kinematic for the input base and backward kinematic for the output base).
- **b:** The second one uses deep learning to approximate the analytical models with a neural netwok by using an approriate reward function which encourage the agent to take action close to the analytical models.
- **c:** The last approximate m_i^r, m_o^r by learning it from scratch (i.e without "imitating" the analytical models).

These 3 bases configurations namely fully analytical, analytical approximated and fully learned will be denoted respectively a,b and c latter in the document.

5.1 Tennis task

Firstly, It was decided that the task was going to be a tennis task. The concept was the following : a robot was trained in simulation to play tennis against another virtual robot. Then, the resulting model was transferred to the real robot which could then play tennis on Unity via his virtual avatar. The appeal of this task was two fold. First, this task was complex enough to justify the need to use the UNN method, as training from scratch each robots would take a fair amount of time. Secondly, this task didn't required much equipement, only the braccio robot. For this to work, it was therefore necessary for the real robot to be able to communicate its movements to its avatar. The first step was thus to make Unity and ROS communicate. Fortunately, a github project called ROS# [43] had what I needed. It brings together several software libraries and tools in C# for allowing ROS and .NET applications, including Unity, to exchange data. Thanks to this package, I was able to control a robot in unity and the real robot from a ROS node simultaneously.



Position X -0.5088
Rotation X 36.44
Scale X 1 Y 1
Clear Flags Skybox
Projection Perspe
Physical Camera
Viewport Rect X 0 Y 0 W 1 H 1
Depth -1

Figure 20: Both robots executing the same commands

Both robots were receiving the same commands and thus, moving in the same way. However, It quickly appeared that in order for the real robot to control its Unity counterpart, access to the joints states was needed. The braccio robot uses servomotors which means that the angular position control is handled internaly by them. In other words, I can't directly have access to the joints positions. After mentionning the matter to my tutors, it was decided that any workaround was too complicated and thus not worth it so we decided to postpone this task.

5.2 Gutter task

The accepted task was a manipulation task where the braccio robot needs to keep a ball at the center of a gutter. The gutter is fixed at one end and held at the other end by the robot which therefore decides of its orientation and, as a consequence, of the position of the ball.

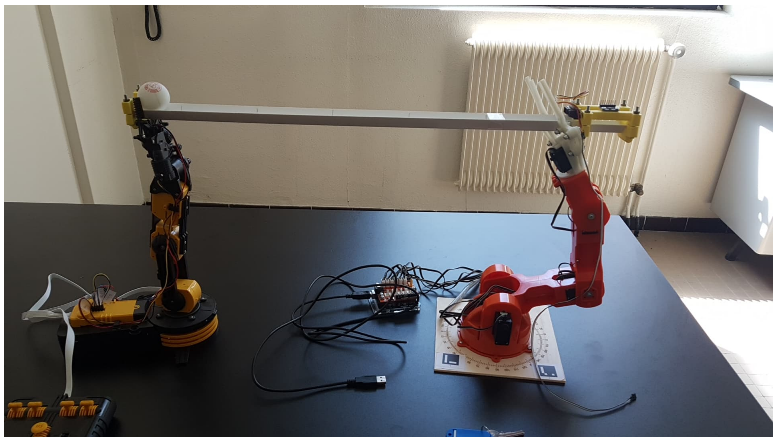


Figure 21: Task settings.
The left robot can't communicate with the PC and is only used to held one the gutter ends.

As we can see on figure 21, the task can be assimilated to a 2D problem. In this setting, the intrinsic information for the braccio robot, $s^{i,r} \in \mathrm{R}^3$ (base, wrist and grip rotation are not needed for this task), is a vector of the joints positions. The task related observation vector $s^T \in \mathrm{R}^4$ will be the ball position and velocity on the gutter and the effector position in the plane given by $m_i^r(s^{i,r})$, the input base output. This pipeline can then be written as

- $a^r = m_o^r(m_u^T(s^T, m_i^r(s^{i,r})), s^{i,r})$ where the superscript r is either a, b or c. The input base m_i^r maps $s^{i,r}$ to the braccio effector position and m_o^r maps o^{out}, the UNN instructions which indicates the position in which the effector should be and $s^{i,r}$ to joints position $a^r \in \mathrm{R}^3$.

The UNN module is then common for all of the robots. Only the bases differ as they are intrinsically dependent on the robot configuration.

5.3 Experimental setup

The robot used to carry out the tasks was a braccio robot (cf figure 22) with 6 DoF and controlled in position (using servo-motors).

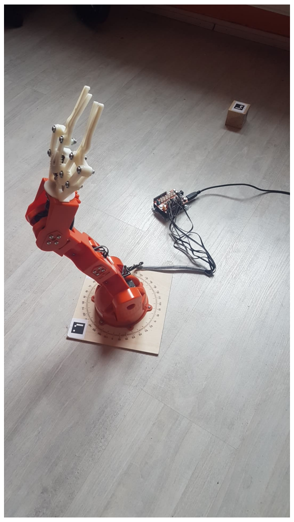


Figure 22: Braccio robot

The robot is controlled by an arduino connected to the computer. A ROS node then communicate with the arduino/robot via the serial port. this node can then subscribe to different nodes depending on the task to be performed. The *Braccio-Arduino-ROS-RViz* project on github [40] contains all the necessary code and libraries to interface the braccio robot with ROS. I just needed to modidfy some part of the arduino code to suit my need. The system is schematized in figure 23.

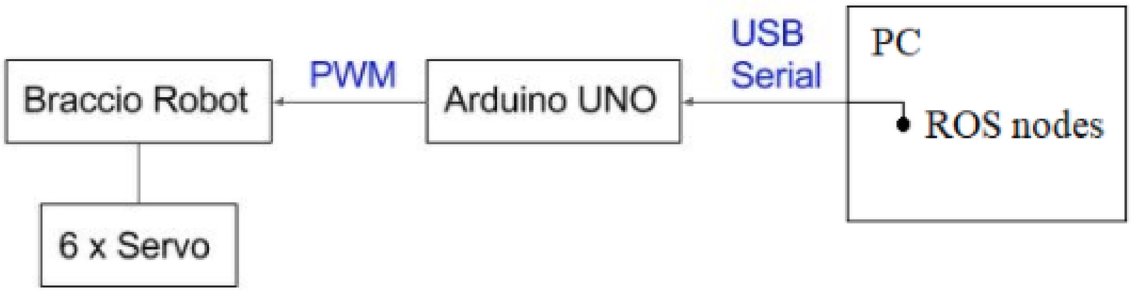


Figure 23: Hardware system

In order to obtain the positions that the models implemented on the robot will need, I used aruco markers[41]. The robot also doesn't have embedded position sensor and may have offsets so aruco markers were used to keep track of the effector position when needed. These marker are used along with OpenCV to obtain pose estimations. They are easily detected and reliable thanks to their black borders and contains an inner binary matrix which determines their ids and allows for error detection and correction. Figure 24 shows a marker detection and its pose estimation. We can retrieve the marker position with a simple monocular camera which makes aruco marker a good choice when working with limited material ressources. It is also very easy to obtain these markers as they can simply be printed.

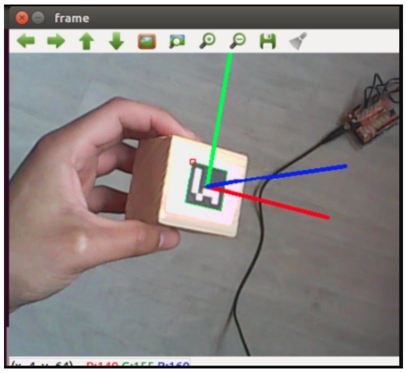
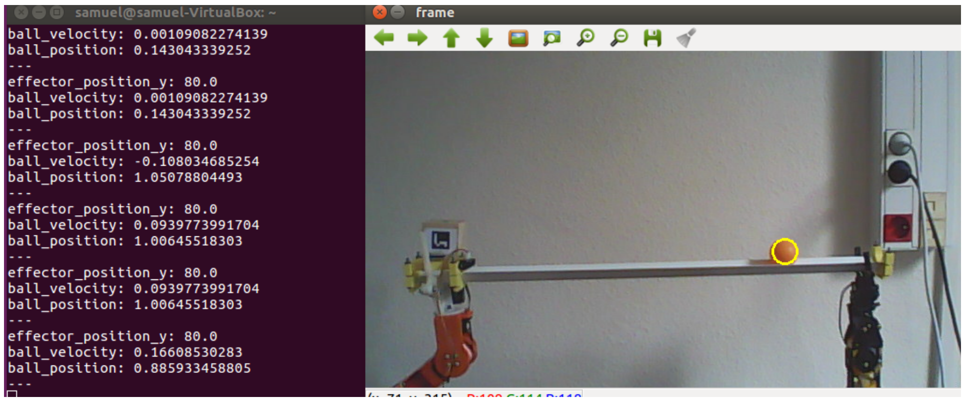


Figure 24: Aruco maker detection and pose estimation

As mention earlier, aruco marker detection is done with a monocular camera. In my case, I used a c170 logitech webcam. Pose estimation of an aruco marker require the camera to be calibrated and the image to be undistorted. This is achieved by finding the intrinsic and extrinsic parameters of the camera.The OpenCV documentation contains a script that computes and stores these parameters in a text file for latter use [42].

To operate, the UNN module require to have access to the ball velocity and local position. These 2 information can be obtained by tracking the ball position with the OpenCV library. The ball tracking is done by thresholding the image according to range of interest. Here the goal is to obtain a binary mask (an image where every thing is black except the ball) in order to use it with the *findContours* method of OpenCV. To perform the thresholding, we first need to determine the appropriate HSV color range that will be used as upper and lower bounds. The HSV color space (hue, saturation, value) is commonly used in image processing task. The hue channel encode the color type so it is easier to find a satisfying bounding ranges for segmenting objects based on its color.

For a better detection, a orange ball was used with a white background. However, the braccio robot is also orange and appear in the image. To avoid false positives, I needed to have a very precise bounding range. I used a script to determine valid thresholding range that will gave the best binary mask. Example of binary masks obtained are available in annexe C and D. The findContour method will try to find the biggest enclosing contour on the mask. Contour detected with a radius too small are treated as noise and dismissed, thus leaving only the ball contour. We can then get the coordinate of the contour center. Doing this process every frame allows us to track efficiently the ball. However, this gave access to the pixel coordinate in the image not the local position of the ball in the gutter. The local position can be retrieved by computing the distance between the ball and one end of the gutter (marked by an aruco marker) and then normalized by dividing the gutter length in the image. The velocity of the ball is simply obtained by getting the position of the ball at time $t1$ and $t2$, computing the distance traveled between these two instants and dividing it by $t2 - t1$. Then the velocity is normalized by dividing with the maximum velocity measured. The results is shown in the figure below



```
ball_velocity: 0.00109082274139
ball_position: 0.143043339252
---
effector_position_y: 80.0
ball_velocity: 0.00109082274139
ball_position: 0.143043339252
---
effector_position_y: 80.0
ball_velocity: -0.108034685254
ball_position: 1.05078804493
---
effector_position_y: 80.0
ball_velocity: 0.0939773991704
ball_position: 1.00645518303
---
effector_position_y: 80.0
ball_velocity: 0.0939773991704
ball_position: 1.00645518303
---
effector_position_y: 80.0
ball_velocity: 0.16608530283
ball_position: 0.885933458805
---
```

Figure 25: Tracking of the ball velocity and position

The observations are then passed to the environment.py on the /observations topic. None of the standard messages suited my need so I created my own. It is very simple with two float32, one for the velocity and the other one for the position.

5.3.1 Bases creations

The first step was to create the 3 bases configurations. I started with bases c which are the one learned from scratch. The output base m_o^c can be thought as a 2D reaching task so I re-used some of the work presented in section 4.3 .However the braccio robot has 6 degrees of freedom so I needed to reshape the virtual robot. To speed up the training, I used several instances of the same robot

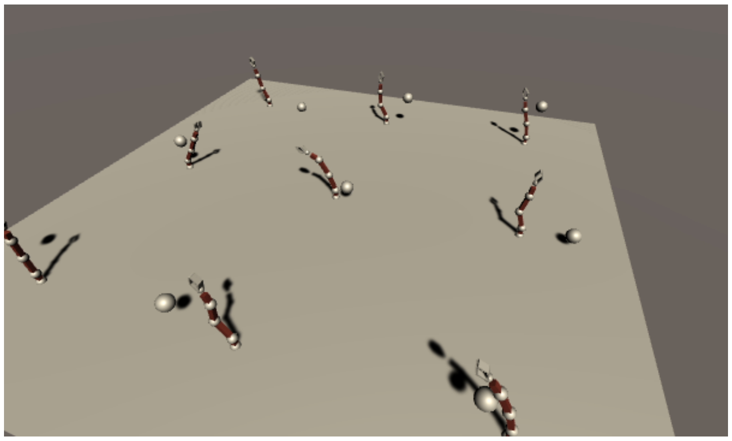


Figure 26: Braccio robots on unity

To make the virtual robot as close as possible to the real one, I used the same proportions, that is middle segments 2 times bigger than both extreme segments. Unlike before, there is an additionnal constraint with this task : the effector must remain perpendicular to the horizontal plane to hold the gutter properly. This behavior can be obtained by adding a penalty to the reward function for each time the effector deviate.

$$R(s,a) = \begin{cases} r - \beta\theta & \text{if } \ d_{t,e} < \delta \\ -\alpha d_{t,e} - \beta\theta & \text{else} \end{cases}$$

Here θ is the angle between the y axis and the effector and $d_{t,e}$ is the distance between the target and the effector. By trying to maximize the reward he is getting, the agent will have to minimize θ. The constant β is used to balance the penalty. Too small and the agent will not grant it enough importance, too large and it will overshadow the agent's reaching task real goal so it is yet another hyper-parameter to fine tune. However, this extra term in the reward function imply that the base obtained is task specific and will most likely not be repurpose for other tasks.

The input base m_i^c, in the oter hand, is much simpler. Its only goal is to map the robot joints angular positions to the effector position. The robot is moving randomly and each time step the agent try to predict the effector position. Unlike the reaching task where the agent controlled the robot to put his effector as close as possible to the target sphere, the agent this time ,control the position of the target sphere and try to put it as close as possible to the effector. The Unity scene is the same, only the C# script describing the behavior of the agent need to change. The reward signal the agent gets is the following

$$R(s,a) = \begin{cases} r & \text{if } \ d_{e,\hat{e}} < \delta \\ -\alpha d_{e,\hat{e}} & \text{else} \end{cases}$$

where $d_{e,\hat{e}}$ is the distance between the effector position and effector position predicted by the agent. The robot will then try to minimize $d_{e,\hat{e}}$ to maximize his reward.

The bases a are analytical models. The input base m_i^a is a simple forward kinematic model obtained with basic trigonometry rules.

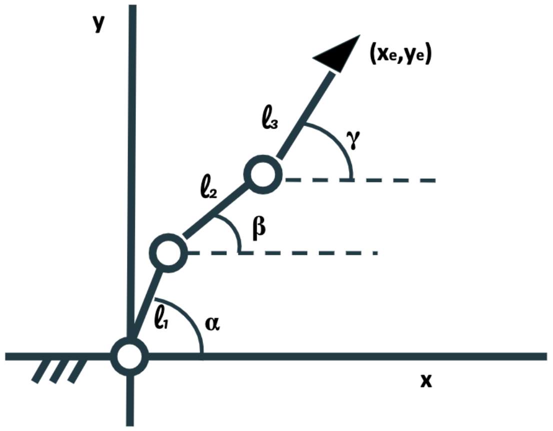


Figure 27: Braccio robot on 2D plane i.e 3 degrees of freedom (wrist and grip rotation neglected)

The effector coordinate (x_e, y_e) can be computed with :

$$x_e = l_1 cos(\alpha) + l_2 cos(\beta) + l_3 cos(\gamma)$$

$$y_e = l_1 sin(\alpha) + l_2 sin(\beta) + l_3 sin(\gamma)$$

The output base m_o^a , in the other hand, is obtained with a backward kinematic model.

Bases b was obtained by learning the analytical models above. As such, it could be treated as a supervised learning problem where m_i^a and m_o^a are respectively the ground truth for learning m_i^b and m_o^b. In this case the loss function is

$$L(m_i^b, m_i^a) = ||m_i^b - m_i^a||^2 \text{ for the input base}$$

$$L(m_o^b, m_o^a) = ||m_o^b - m_o^a||^2 \text{ for the output base}$$

In the RL settings, we would simply have $R(s, a) = -L$.

5.3.2 UNN module creation

Once the bases trained, I focused on creating the UNN module. In the case of this task, creating the UNN module means training an agent to balance the gutter and keep the ball at the center. The robot is assimilated to its effector (BAM settings) and the UNN output $o^{out} \in$ R (continuous action space), a single value indicating at what height below or above the horizontal reference position of the gutter the effector should be (see figure 28).

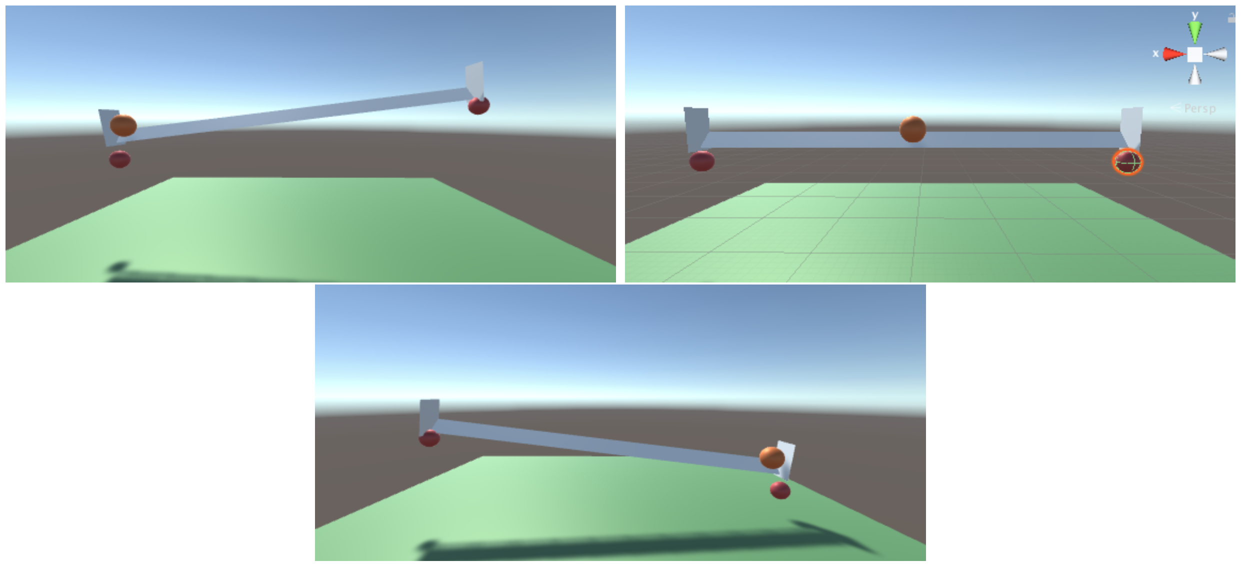


Figure 28: Gutter configurations.
Top left is at maximum height, top right is at horizontal reference position and bottom is at minimum height. The right red sphere represent the effector of the robot (in BAM setting the robot is assimilated to its effector).

Initially, the Unity physic engine was giving unrealistic behavior. With the default settings, the ball fell very slowly, making it too easy for the agent in simulation compare to real world conditions. So to make the simulations more realistic, I measured in the real world the time (approximately 2 sec) that the ball needed to reach the end of the gutter starting from the other end with the gutter tilted by an angle of 30 degrees with horizontal plane. Then I increased the ball falling speed until I obtained the same time in the same conditions.

To carry out his task, the agent need to know the position of the effector as well as the ball position and velocity. In the UNN framework the effector position is provided by the input base m_i^c, but when training the UNN in BAM settings, we have to provide directly this information to the module. At first, the reward function associated with the task was once again

$$R(s,a) = \begin{cases} r & \text{if } \ d_{c,b} < \delta \\ -\alpha d_{c,b} & \text{else} \end{cases}$$

where $r = 0.1$, $\delta = 0.5$ and $d_{c,b}$ is the distance between the ball and the center of the gutter. However, this reward function was too simple and gave rise to unwanted behavior : the agent was keeping the ball still by alternating between maximum height and minimum height as quick as possible. This behavior may maximize the reward but is not optimal because the robot makes a lot of unnecessary movement. To tackle this issue, I added an extra term to the reward in order to penalize movements. The new reward function is

$$R(s,a) = \begin{cases} r - \beta||y_{t+1} - y_t|| & \text{if } \ d_{c,b} < \delta \\ -\alpha d_{c,b} & \text{else} \end{cases}$$

where y is the y coordinate of the effector and β a scaling constant. This highlights the fact that the choice of the reward function plays an important role in the final perfomance of the agent.

5.3.3 Putting all the pieces together

Once the bases c and the UNN module were created, I was able to assemble the different parts to construct the UNN pipeline as in figure 9. Little change needed to be made as the effector position was now given by the the input base and the output base's target was now the UNN output.

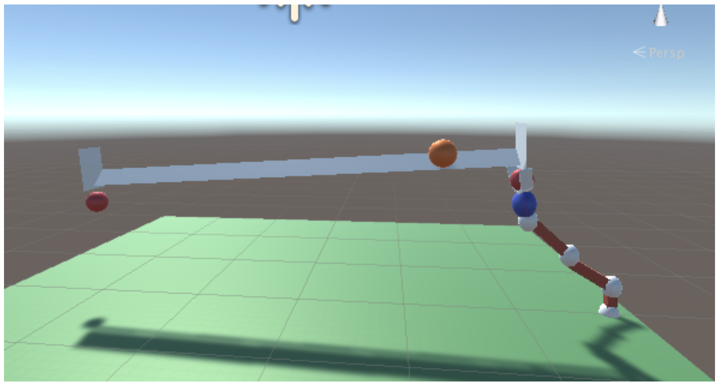


Figure 29: Gutter task.
The blue sphere represent the UNN instruction (or target) where the output base must place the robot effector.

Results

The cumulative reward obtained per episode was used as the metric to compare the UNN module (with the robots assimilate to its effector) against the same module transferred to the robot with bases c. The reward function with the extra penalty term was used, the normalized positve reward range chosen was $\delta = 0.1$ (i.e a radius of 10% of the gutter length), the positive reward constant r was 0.3. The scaling constant used were $\alpha = 0.2$ and $\beta = -15$ The test lasted 400 episodes of 200 steps. Figure 30 display the result obtained

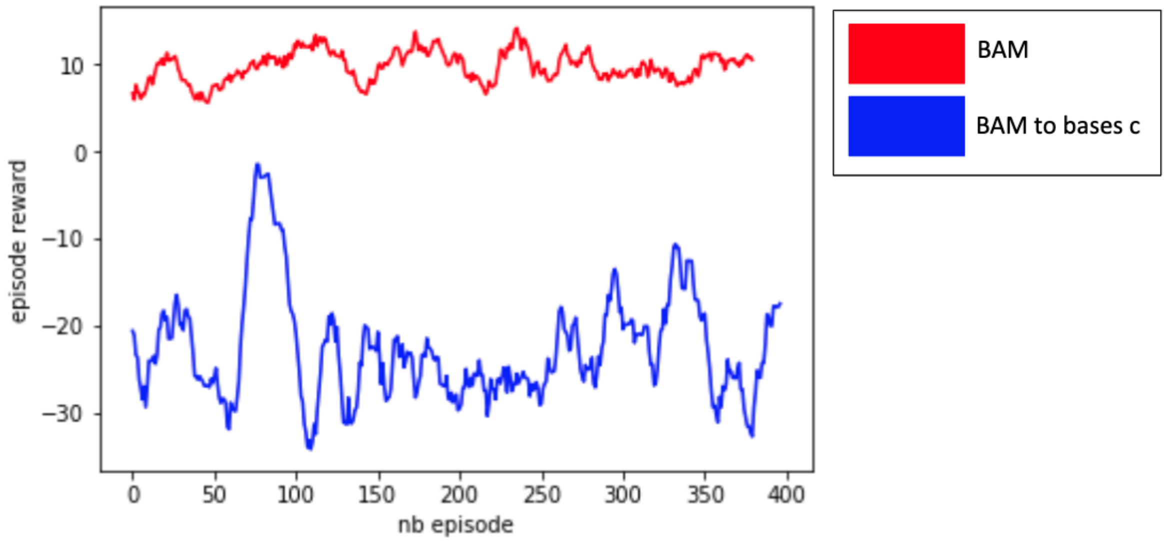


Figure 30: Gutter task.

As shown in the figure above, the BAM only agent is performing way better then the transferred one. In fact, it manage to obtain almost only positive cumulative reward meaning that he is putting the ball on the center pretty fast and keeping it still with minimum movement as the penalty would overwhelmed the postive reward r otherwise. In the case of the transferred agent the negative reward obtained is mainly due to the output base used. Indeed, during training the output base wasn't penalized for extra movements done. As a consequence, when coupled with the UNN agent, while managing to put the ball in the center easily, the effector of the robot performed a lot of useless movements, receiving therefore many penalties. This highlight the fact that in the UNN methods, the base quality plays in an important role and thus, must be carefully crafted. It is also worth noting that, for max performance, the output base used was trained on very specific positions used by the UNN output, i.e only on a target moving vertically.

5.3.4 Setting up the manipulations

To set up the real world manipulations, the first step was to create the software architecture that will be use to make all the differents parts communicate efficiently. In order to replicate the reliable openAI software structure, I decided to have a python agent and a environment class. The agent class handle the loading of the model and its use to get the agent actions and then pass it to environment. The environment handle all the outside interactions (webcam and real robot) and compute the reward obtained by the agent. It is a ROS node that subscribes to camera.py to get all relevant observations and publish the action of the agent to the braccio.ino node. Inside the camera.py node, OpenCv is used to first get the camera frame and then all the needed positions (effector, ball) or velocity depending on the task.

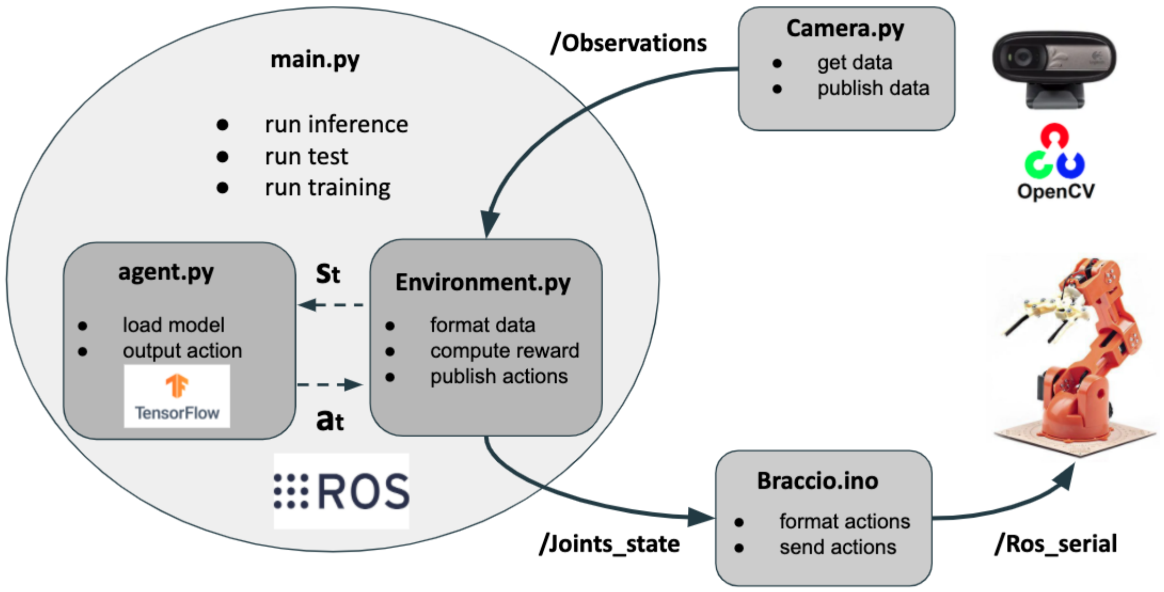


Figure 31: ROS software architecture

In figure 31, the solid arrows represents ros topics and the dotted arrows represents simple data exchange inside the main.py file where an agent and a environment object is instantiated. The agent.py contains the TensorFlow model used for inference.To deal with task specific attribute such as reward computing and the state or action dimensions, the Environment class was designed to be a class form which specific environment like ReachingEnvironment or GutterEnvironment can inherit. Thus, only the task specific functions needs to be overload. An extra class robot.py used in environment.py, not visible in the figure 31 handle all the robot specific attributes such as the number of joints, segment lenghts etc... The main.py file contains 3 functions :

- the inference function use the model to control the robot according to its policy.
- the training function is use to train the model from scratch with Pytorch. It is used to get the baselines performance for comparison with transfered models with the UNN.
- the test function is used to quantify the transfer efficiency.

The ml-agents package is using Tensorflow as framework to train the agents which means that the models obtained after training are in a tensorflow format. Initially,I tried to convert the tensorflow models to pytorch models as it was the framework I was used to. Unfortunately, I wasn't able to find a tool to convert a tensorflow model into a pytorch one. Therefore I decided to take a look at how the model was stored to try and convert it myself.The first step was to parse the weigths parameters of the model into numpy's arrays. Then construct an identical pytorch model (same type of layer, same number of neurons per layer etc...) and initialize the weights using the numpy arrays aforementionned. However, there was still missing pieces such as the activation functions used and I wasn't experienced enough with this framework to know where to find them. After several failures, I finally decided to use Tensorflow as framework for implementing the models with the robot.

5.3.5 Bases transfer

Once the c bases models were available, it was possible to perform a first simulation to real world transfer. First, the ouput base c was transferred. To obtain the target position I used two aruco markers, one was playing the target role and the other one was used as the origin of the coordinate system, place at the base of the robot, to get the target local position. This data was then normalized by dividing it with the robot arm range as in previous reaching tasks with Unity. As mentionned earlier, the aruco marker detection is handled by camera.py. The joint state of the robot were still not accessible so the agent was simply getting as input the joints angular position the robot was aiming for (i.e the agent output).

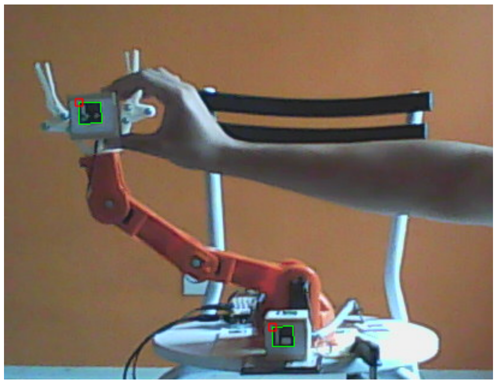


Figure 32: Reaching task with braccio robot

The transfer seems to be efficicient as the agent successfully place the braccio's effector close to the target multiple times. The input base c was also transferred using the same system setting.

In order to dertermine how efficient the transfers were, I needed to compare the real robot against the virtual robot. For a fair comparison, the metric was measured with the same target positions for the output base (reaching task) and with the joints position for the input base. I added in the Unity agent's script a function to write the successive target positions/joints position used during the test as well as relevant parameters such as the reward obtained by the virtual agent each episode and the normalized positve reward range. Then the text file was read by the test function of main.py to retrieve these information and feed them to the model of the real robot. Another thing to take into account is the execution speed. Simulation in Unity are a lot faster than the processing loop of the real system. In consequences , the agent in simulation will accumulate rewards quicker The comparison protocol for the reaching task is the the following : the agent receive a target position and output the desired joint states. If the robot effector land inside a certain range close to the target it get a postive reward, 0 else. For the input base the protocol is nearly the same. The robot move according to the test settings and the agent output the effector position. In both experiment the effector position is given by an Aauco marker placed inside the robot grip This way we get rid of the execution speed issue.

Results

The experimental constants used were : $\alpha = -0.2$ for scaling factor and $\delta = 0.16$ for the normalized positive reward range. The test were run for 50 episodes and the results were plotted with a 10 points moving average. As expected, the real robot is not performing as well as the simulated robot.

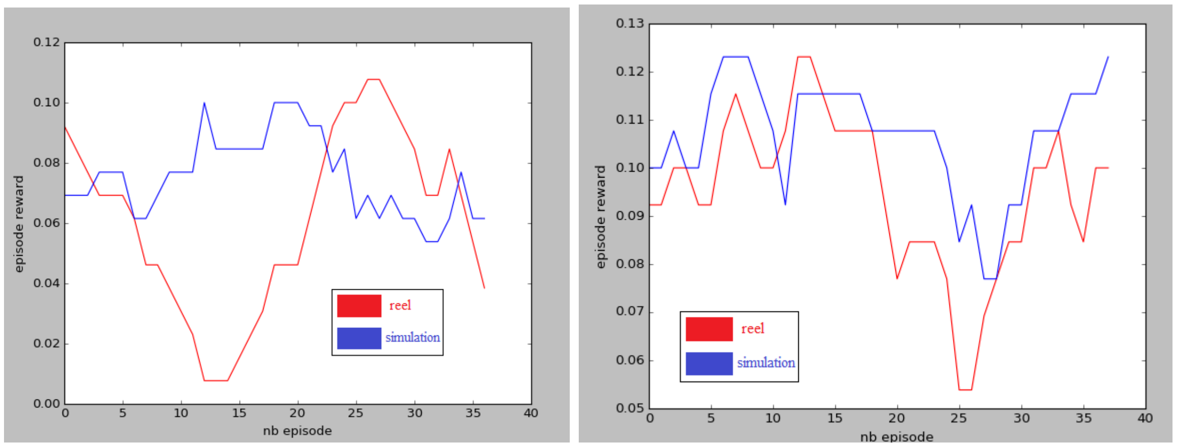


Figure 33: Cumulative rewards obtained (reaching task/output base on the left and input base on the right)

This can be explained by the fact the real robot is not "perfect", its joints have offsets because of the way it was assembled. Therefore the robot failed to precisely follow the agent instructions. The offsets also lead to imprecise observations as the joint state receive by the model are not exactly the ones the robot is in. Overall, the simulation to real world transfer efficiency for the reaching task is 85.71% and 87.03% for the input base.

Conclusion

This internship at the Pascal Institute was a very important accelerator of my skills in robotics and deep learning. The tasks carried out as well as the scientific articles read allowed me to develop my scientific mind while developing my technical skills. I had the chance to work on a cutting-edge subject, supervised by people who were experts in their field.

However, I regret the fact that I did not have more experimental results to present at the time of writing this report. I still have to create the bases a and b, transfer them to the real robot and carry out the transfer of the UNN module on each of them. In spite of this, the remaining month of internship should allow me to finalize and complete the manipulations, almost already setup.

Finally, I also feel extremely lucky to be able to deepen the subject of transfer learning in robotics in the thesis that I am starting in October.

References

[1]”Fixing the train-test resolution discrepancy: FixEfficientNet”,Hugo Touvron, Andrea Vedaldi, Matthijs Douze, Hervé Jégou

[2] http://www.image-net.org/

[3]”ImageNet Large Scale Visual Recognition Challenge” Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei

[4]”Language Models are Few-Shot Learners”, OpenAI

[5]OpenAI, ”Solving rubiks cube with a robot hand”.

[6] ”Multilayer feedforward networks are universal approximators”, KurtHornik et al.

[7] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (9 October 1986). ”Learning representations by back-propagating errors”. Nature. 323 (6088): 533–536.

[8]”Adam: A method for stochastic optimization”, Diederik P. Kingma, Jimmy Lei Ba

[9] ”An overview of gradient descent optimization algorithms”, Sebastian Ruder

[10]M. Volodymyr, et al., Play atari with deep reinforcement learning, Tech. rep. (2013)

[11] D. Silver, et al., Mastering the game of go with deep neural networks and tree search, The Journal of Nature.

[12] ”Dota 2 with Large Scale Deep Reinforcement Learning”,OpenAI

[13] ”Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”, David Silver et al.

[14]”DeepGait: Planning and Control of Quadrupedal Gaits using Deep Reinforcement Learning”

[15]OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. W. Pachocki, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, “Learning dexterous in-hand manipulation,” CoRR, vol. abs/1808.00177, 2018. [Online]. Available: http://arxiv.org/abs/1808.00177

[16] ”Policy Gradient Methods for Reinforcement Learning with Function Approximation”, Richard S. Sutton et al.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” CoRR, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[18] ”High-Dimensional Continuous Control Using Generalized advantage estimation” Schulman et al.

[19] https://pytorch.org/docs/stable/index.html //changer

[20] http://yann.lecun.com/exdb/mnist/ // changer

[21]G. Brockman, et al., Openai gym (2016). arXiv:arXiv:1606.01540.

[22] https://docs.unity3d.com/ScriptReference/index.html

[23] A. Juliani, et al., Unity: A general platform for intelligent agents, CoRR abs/1809.02627. arXiv:1809.02627. 560 URL http://arxiv.org/abs/1809.02627

[24] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md

[25]https://opencv.org/

[26] https://www.ros.org/

[27] ”A Style-Based Generator Architecture for Generative Adversarial Networks” at arXiv.org

[28] ”Deep Residual Learning for Image Recognition”, Kaiming He,Xiangyu Zhang, Shaoqing Ren, Microsoft Research

[29] ”Going deeper with convolutions”, Christian Szegedy et al.

[30] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” CoRR, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv. org/abs/1810.04805

[31]Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer,

and V. Stoyanov, “Roberta: A robustly optimized BERT pretraining approach,” CoRR, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.1169

[32] ”Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping” Konstantinos Bousmalis et al.

[33]J. Tobin, et al., ”Domain Randomization for Transferring Deep Neural Net585 works from Simulation to the Real World”, ArXiv e-printsarXiv:1703. 06907.

[34] Mehdi Mounsif et al., ”Universal Notice Networks: A broad panel of applications”.

[35] ”BAM ! Base Abstracted Modeling with Universal Notice Network : Fast Skill Transfer Between Mobile Manipulators” Mehdi Mounsif,Sébastien Lengagne, Benoit Thuilot and Lounis Adouane

[36]”CoachGAN: Fast Adversarial Transfer Learning between differently shaped entities”, Mehdi Mounsif,Sébastien Lengagne, Benoit Thuilot and Lounis Adouane

[37] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. WardeFarley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Networks,” ArXiv e-prints, Jun. 2014.

[38]T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans,” in Proceedings of the 30th International Conference on Neural Information Processing Systems, ser. NIPS’16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 2234–2242.

[39]”Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” Alec Radford, Luke Metz, Soumith Chintala

[40] https://github.com/ohlr/braccio_arduino_ros_rviz

[41] https://docs.opencv.org/trunk/d5/dae/tutorial_aruco_detection.html

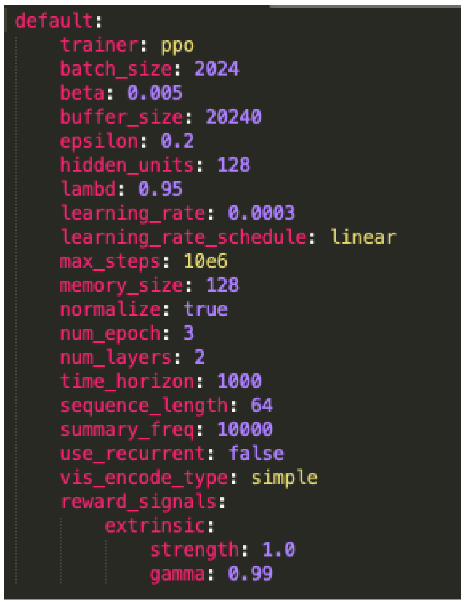
[42] https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html

[43] https://github.com/siemens/ros-sharp

[44] https://www.tensorflow.org

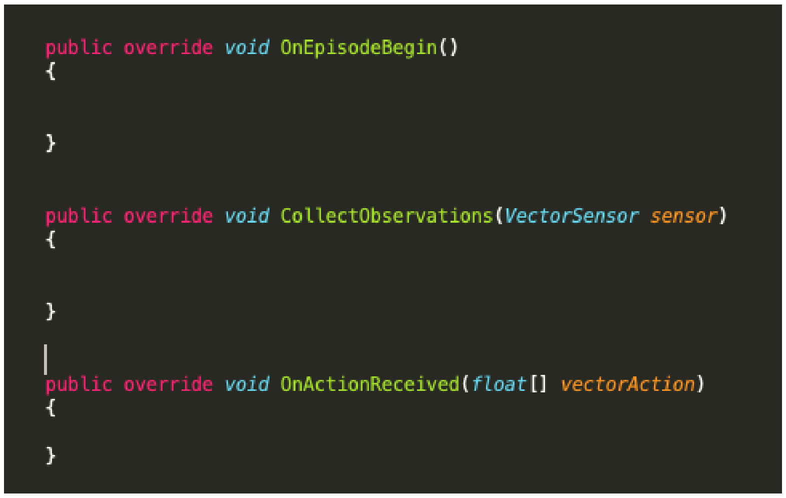
Annexe

Unity



```yaml
default:
    trainer: ppo
    batch_size: 2024
    beta: 0.005
    buffer_size: 20240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 0.0003
    learning_rate_schedule: linear
    max_steps: 10e6
    memory_size: 128
    normalize: true
    num_epoch: 3
    num_layers: 2
    time_horizon: 1000
    sequence_length: 64
    summary_freq: 10000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
```

Annexe A : confi.yaml file containing hyper-parameters values

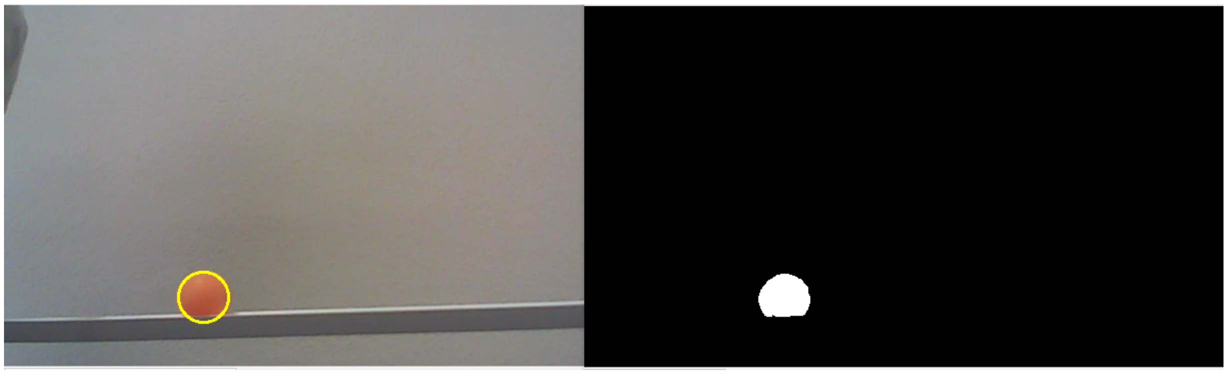


```csharp
public override void OnEpisodeBegin()
{


}


public override void CollectObservations(VectorSensor sensor)
{


}


public override void OnActionReceived(float[] vectorAction)
{

}
```

Annexe B : Agent unity script

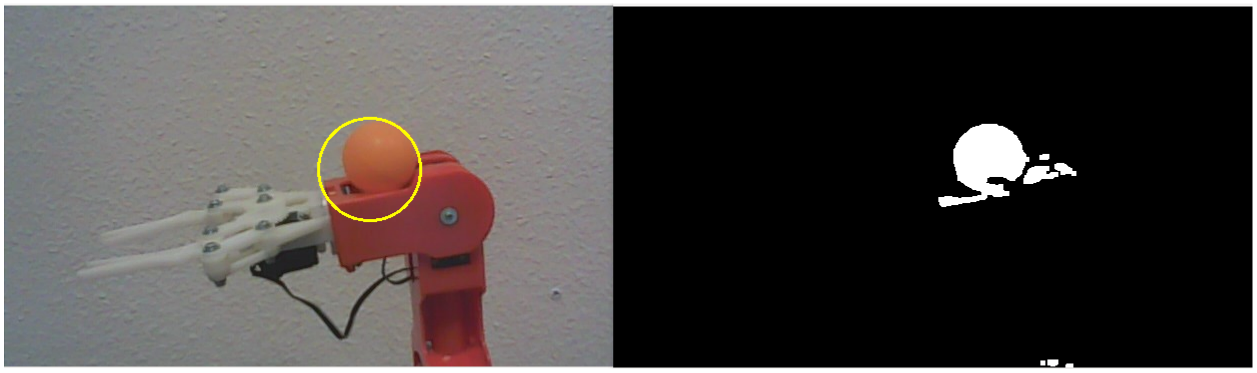
Color range

First configuration is the simplest : the ball is in the gutter with a white back ground.



Annexe C : thresholding with white background

As we can see, the thresholding allow for an accurate detection of the ball in the image. The next configurations was designed to see if the bounding ranges chosen were robust to disruptive elements such as the color of the robot.



Annexe D : Thresholding with braccio appearing in the image