COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APPLICATION OF ARTIFICIAL INTELLIGENCE ALGORITHMS TO PILOT A MOTORIZED ROBOT

Autor: Eduardo Moros Barandiarán

Director: Edouard Oyallon Apellido2

Madrid

Julio de 2019

AUTHORIZATION FOR DIGITALIZATION, STORAGE AND DISSEMINATION IN THE NETWORK OF END-OF-DEGREE PROJECTS, MASTER PROJECTS, DISSERTATIONS OR BACHILLERATO REPORTS

1. Declaration of authorship and accreditation thereof.

The author Mr. /Ms. Eduardo Moros Barandiarán

HEREBY DECLARES that he/she owns the intellectual property rights regarding the piece of work: Application of artificial intelligence algorithms to pilot a motorized robot
that this is an original piece of work, and that he/she holds the status of author, in the sense granted by the Intellectual Property Law.

2. Subject matter and purpose of this assignment.

With the aim of disseminating the aforementioned piece of work as widely as possible using the University's Institutional Repository the author hereby **GRANTS** Comillas Pontifical University, on a royalty-free and non-exclusive basis, for the maximum legal term and with universal scope, the digitization, archiving, reproduction, distribution and public communication rights, including the right to make it electronically available, as described in the Intellectual Property Law. Transformation rights are assigned solely for the purposes described in a) of the following section.

3. Transfer and access terms

Without prejudice to the ownership of the work, which remains with its author, the transfer of rights covered by this license enables:

a) Transform it in order to adapt it to any technology suitable for sharing it online, as well as including metadata to register the piece of work and include "watermarks" or any other security or protection system.

b) Reproduce it in any digital medium in order to be included on an electronic database, including the right to reproduce and store the work on servers for the purposes of guaranteeing its security, maintaining it and preserving its format.

c) Communicate it, by default, by means of an institutional open archive, which has open and cost-free online access.

d) Any other way of access (restricted, embargoed, closed) shall be explicitly requested and requires that good cause be demonstrated.

e) Assign these pieces of work a Creative Commons license by default.

f) Assign these pieces of work a HANDLE (*persistent* URL). by default.

4. Copyright.

The author, as the owner of a piece of work, has the right to:

a) Have his/her name clearly identified by the University as the author

b) Communicate and publish the work in the version assigned and in other subsequent versions using any medium.

c) Request that the work be withdrawn from the repository for just cause.

d) Receive reliable communication of any claims third parties may make in relation to the work and, in particular, any claims relating to its intellectual property rights.

5. Duties of the author.

The author agrees to:

a) Guarantee that the commitment undertaken by means of this official document does not infringe any third party rights, regardless of whether they relate to industrial or intellectual property or any other type.

b) Guarantee that the content of the work does not infringe any third party honor, privacy or image rights.
c) Take responsibility for all claims and liability, including compensation for any damages, which may be brought against the University by third parties who believe that their rights and interests have been infringed by the assignment.
d) Take responsibility in the event that the institutions are found guilty of a rights infringement regarding the work subject to assignment.

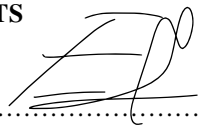
6. Institutional Repository purposes and functioning.

The work shall be made available to the users so that they may use it in a fair and respectful way with regards to the copyright, according to the allowances given in the relevant legislation, and for study or research purposes, or any other legal use. With this aim in mind, the University undertakes the following duties and reserves the following powers:

a) The University shall inform the archive users of the permitted uses; however, it shall not guarantee or take any responsibility for any other subsequent ways the work may be used by users, which are non-compliant with the legislation in force. Any subsequent use, beyond private copying, shall require the source to be cited and authorship to be recognized, as well as the guarantee not to use it to gain commercial profit or carry out any derivative works.
b) The University shall not review the content of the works, which shall at all times fall under the exclusive responsibility of the author and it shall not be obligated to take part in lawsuits on behalf of the author in the event of any infringement of intellectual property rights deriving from storing and archiving the works. The author hereby waives any claim against the University due to any way the users may use the works that is not in keeping with the legislation in force.
c) The University shall adopt the necessary measures to safeguard the work in the future.
d) The University reserves the right to withdraw the work, after notifying the author, in sufficiently justified cases, or in the event of third party claims.

Madrid, on 5…….. of July 2019……..., …….

HEREBY ACCEPTS



Signed……

Reasons for requesting the restricted, closed or embargoed access to the work in the Institution's Repository

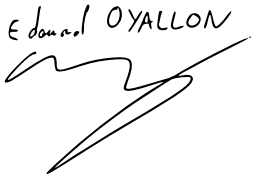
Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título Application of Artificial Intelligence algorithms to pilot a motorized robot en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico …2019……. es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Eduardo Moros Barandiarán Fecha: 4/ 7/ 2019

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Edouard Oyallon Fecha: 4/ 7/ 2019



APLICACIÓN DE ALGORITMOS DE INTELIGENCIA ARTIFICIAL PARA PILOTAR UN ROBOT MOTORIZADO

Autor: Moros Barandiarán, Eduardo

Director: Oyallon, Edouard

Entidad colaboradora: CNRS – Centre National de Research Scientifique français

RESUMEN DEL PROYECTO

Este proyecto es parte de un proyecto más grande cuyo objetivo es acercar la inteligencia artificial a los niños. La inteligencia artificial y el aprendizaje automático (machine learning) son campos en rápido crecimiento que serán clave en el futuro, por lo que preparar a las nuevas generaciones en estos campos es imperativo. Para hacer esto, el CNRS está desarrollando un robot que aprende a hacer diferentes tareas usando algoritmos de machine learning. Encuadrado en este contexto general, este proyecto consiste en desarrollar un algoritmo para pilotar el robot dentro de una arena rectangular con el objetivo de evitar las paredes, usando una cámara frontal como entrada sensorial.

El algoritmo sólo necesita decidir qué acciones debe llevar a cabo el robot, la señal de comanda del robot la calcula el programa del cliente, en el cual este algoritmo está incrustado. Para desarrollar el algoritmo, el cliente suministró un simulador que replica los movimientos del robot en la arena. La mayor parte del trabajo se desarrolló sobre este simulador, haciendo un test final en una arena física y usando un robot AlphaBot de la marca WaveShare, suministrado por el cliente.

La literatura existente separa tres tipos de *machine learning*: aprendizaje supervisado, aprendizaje no supervisado y *reinforcement learning*. De estos tres, *reinforcement learning* es el más adecuado para este proyecto porque es el único interactuando con el entorno en tiempo real, como el robot deberá hacer dentro de la arena.

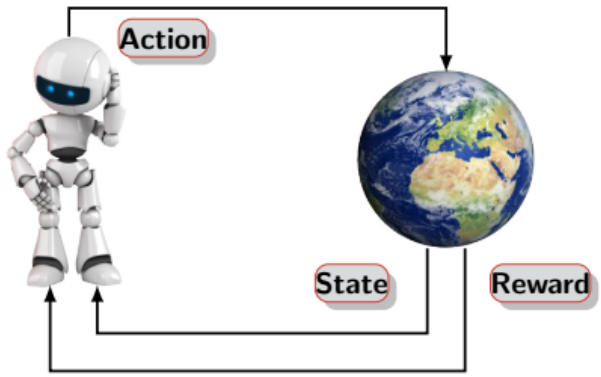


Figura 1: Reinforcment learning

Reinforcement learning consiste en un agente (en este caso el robot) que aprende a relacionar situaciones y acciones maximizando una recompensa numérica

proporcionada por el entorno. Es un método de aprendizaje de ensayo y error, similar a nuestra manera de aprender en un entorno desconocido. *Reinforcement learning* está basado en una interacción en bucle entre el agente y el entorno mostrada en la Figura 1. En un tiempo t, el agente está en un estado s_t, ejecuta una acción a_t siguiendo una política π y observa el nuevo estado del entorno, s_{t+1}, así como una recompensa r_t. La principal dificultad a la que se enfrenta el *reinforcement learning* es la alta dispersión de la información contenida en la recompensa. Las acciones no sólo afectan la recompensa actual, sino todas las futuras recompensas, por lo que interpretar que acciones otorgan un buen resultado, no es sencillo.

Algunos conceptos básicos de *reinforcement learning* son introducidos a continuación:

- **Espacio de estados $\mathcal{S}$:** es el espacio conformado por todos los posibles estados s en los que se puede encontrar el entorno. En nuestro caso el entorno es un vídeo, por lo que el espacio $\mathcal{S}$ es todas las posibles imágenes. $\mathcal{S}$ es enorme en este problema.
- **Espacio de acciones $\mathcal{A}$:** es el espacio conformado por todas las acciones que el agente puede llevar a cabo. En este proyecto $\mathcal{A}$ esta restringido a un espacio discreto de 5 acciones: {Adelante, Adelante Izquierda, Adelante Derecha, Marcha Atrás Izquierda, Marcha Atrás Derecha}
- **Política π:** una política es una función que define el comportamiento del agente, diciéndole que acción tomar en una situación concreta. El objetivo del *reinforcement learning* es llegar a la política óptima.
- **Retorno D:** el impacto de una acción no se puede medir con la recompensa inmediata porque sus repercusiones pueden verse centenas de instantes más tarde. Para medir mejor el impacto de las acciones se usa el retorno (D). El retorno se trata de la suma ponderada de todas las recompensas hasta el final de la experiencia. El retorno depende de la política seguida.

$$D^{\pi}(s) = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s$$

- **Función estado-acción Q:** la función Q(s,a) representa el retorno esperado si el agente en un estado s realiza una acción a. Matemáticamente se define como:

$$Q^{\pi}(s,a) = \mathbb{E}^{\pi}\{D^{\pi}(s)|a_0 = a\} = \mathbb{E}^{\pi}\left\{\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right\}$$

  Q es una función recurrente, porque se puede escribir como la recompensa en el instante t más la nueva función Q para el instante t+1. Esto lleva a la ecuación de Bellman:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a)$$

La mayor parte de algoritmos de *reinforcement learning* calculan explícitamente Q para toda pareja estado-acción y deducen la política a seguir tomando la acción que maximiza Q. Una iteración de este paso unido a una actualización de Q usando la ecuación de Bellman lleva al algoritmo a converger hacia el valor máximo de Q para toda pareja s-a, en consecuencia haciendo converger la política hacia la política óptima. Esto no es viable en este caso porque el espacio de estados es demasiado grande, por lo que no se puede calcular explícitamente una tabla con todos los valores de Q.

En lugar de eso se usa una red neuronal artificial para aproximar Q. Una red neuronal artificial es una red de neuronas interconectadas, cuyas conexiones son regidas por los parámetros θ. Se itera la ecuación de Bellman para entrenar los paramétros θ y, de esta forma, converger hacia el valor máximo de Q. Cuando Q está maximizada, la política óptima se puede deducir siguiendo la política *greedy* (política que elige la acción que maximiza Q). Los parámetros θ se entrenan usando la técnica del descenso de gradientes. Este tipo de algoritmo se llama *Deep Q Learning (DQN)*.

El algoritmo desarrollado en el proyecto es una variación del DQN y su pseudo-código está explicitado a continuación :

Algoritmo DQN adaptado para Learning Robot

Pedir resolución a la cámara #solo para el robot real
Inicializar buffer de memoria $\mathcal{D}$ a una capacidad N
Inicializar γ, n, ϵ y el entorno
Inicializar red neuronal *policy* Q con parámetros aleatorios θ
Inicializar C y red neuronal *target* Q' con parámetros aleatorios θ'
Inicializar optimizador y *learning rate* lr
Inicializar estado $x_1 = \{s_1, m_1\}$
for $t = 1, T$ **do**
    Forzar una acción a_t else
        Con probabilidad ϵ seleccionar una acción aleatoria a_t else
        seleccionar $a_t = \max_a Q^{\theta}(x_t, a)$
    Ejecutar acción a_t en el entorno y esperar respuesta
    while not respuesta
            Optimizar
    Observar estado $x_{t+1} = \{s_{t+1}, m_{t+1}\}$ y recompensa r_t
    Guardar transición (x_t, a_t, r_t, x_{t+1}) en $\mathcal{D}$
    Dibujar curva de recompense y error
    $x_t \leftarrow x_{t+1}$
    Cada C pasos hacer $\theta' \leftarrow \theta$
end for

Función Optimizar

Elegir un grupo aleatorio de n transiciones (x_j, a_j, r_j, x_{j+1}) en $\mathcal{D}$
Calcular predicción $y_j = Q(x_j, a_j; \theta)$
Calcular meta $z_j = \gamma \max_{a'} Q'(x_{j+1}, a'; \theta') + r_j$
$\delta = y_j - z_j$
Calcular error $\mathcal{L}(\delta)$
Ejecutar un paso del optimizador en $\mathcal{L}(\delta)$
Restringir parámetros θ

La principal diferencia con otros algoritmos de *Deep Q Learning* es que se implementa una *experience replay*. El algoritmo guarda cada transición entre el estado t y el estado t+1 en un buffer y utiliza estas transiciones cuando optimiza para conseguir una mejor convergencia.

El error $\mathcal{L}(\delta)$ es el error cuadrático medio de los n valores of δ_j: $\mathcal{L}(\delta) = \frac{1}{n}\sum {\delta_j}^2$

Para que este algoritmo funcione es necesario definir una función de recompensa. La función de recompensa está construida de la siguiente manera: el agente recibe una recomensa igual a su velocidad proyectada hacia delante y se le castigará si está bloqueado o si hace marcha atrás. La función recompensa se define como:

$$r_t = \begin{cases} \boldsymbol{v} \cdot \boldsymbol{u} \;\; if\; \boldsymbol{v} \cdot \boldsymbol{u} > 0 \\ -100 \;\; if\; \boldsymbol{v} \cdot \boldsymbol{u} = 0 \\ -50 \;\; if\; \boldsymbol{v} \cdot \boldsymbol{u} < 0 \end{cases}$$

v es la velocidad y u la dirección perpendicular al eje de las ruedas del robot.

El algoritmo tiene un grupo de parámetros, llamados hiperparámetros, fijados por el programador antes de la experiencia y afinados para adaptar el algoritmo a un entorno en particular. Una parte importante del proyecto fue afinar estos parámetros:

- **Capacidad del buffer N:** representa el número de transiciones que se pueden guardar. Una capacidad pequeña le impide aprender al agente; una capacidad grande malgasta espacio de almacenamiento. Se eligió un valor de N = 1000.
- **Gamma γ:** parámetro con valores entre 0 y 1 que representa la importancia que se le da a las recompensas futuras. Si γ es cercano a 1, el agente tomará mucho en cuenta las acciones futuras y si es cercano a 0 tomará solamente en cuenta las recompensas inmediatas. El valor óptimo se encontró alrededor de 0.8.
- ***Batch size* n:** en la optimización se usa una *batch* aleatoria de n acciones. Cuanto mayor es n, mejor el algoritmo convergerá. Sin embargo, en la práctica n está limitado por la potencia de cálculo del ordenador. Un buen valor de n es n = 100.
- **Exploración:** con una probabilidad de ϵ, se elige una acción aleatoria. A esto se le llama exploración. La exploración impide que el algoritmo converja hacia un mal comportamiento forzándole a recorrer partes del espacio $\mathcal{S}$ x $\mathcal{A}$ que nunca han sido visitadas. Varias curvas de exploración (variación de ϵ a lo largo de la experiencia) se han probado. Finalmente, se eligió un valor constate ϵ = 0.1.
- **Estructura de la red neuronal:** las neuronas de la red neuronal están ordenadas en capas. Hay diferentes tipos de capas: capas densas, capas convolucionales, capas de tipo *pooling*, etc. Se probaron diferentes combinaciones de capas y la estructura elegida fue de tres capas densas consecutivas de 150, 100 and 50 neuronas respectivamente.
- ***Target update* C:** se usan dos redes neuronales para conseguir estabilidad (Q y Q'). La red *target* Q' no aprende. Por lo tanto, sus parámetros θ' tienen que ser actualizados cada C pasos con los valores de θ. Si C es demasiado pequeño, el algoritmo será inestable; si C es demasiado grande, el aprendizaje será errático.

- **Optimizador:** la red neuronal está optimizada usando alguna variación del método del descenso de gradientes. El método utilizado se fija mediante el optimizador. Se probaron dos optimizadores diferentes: SGD y Adam. Se eligió Adam porque daba mejores resultados
- ***Learning rate* lr:** la *learning rate* es un parámetro del optimizador. Representa la velocidad a la que el agente aprende. Cada optimizador tiene un rango de *learnig rate* válidas. Si la *learning rate* es demasiado grande, el algoritmo podría saltarse el máximo global y acabar en un máximo local, aprendiendo un mal comportamiento. Si la *learning rate* es demasiado pequeña el algoritmo tardará demasiado en converger. La *learnig rate* que se eligió para Adam fue 0.01.

El algoritmo se implementó usando Python y la librería para *machine learning* Pytorch. Los parámetros del algoritmo fueron optimizados en el simulador y se hizo un test con el robot real. El robot aprendió correctamente en un tiempo razonable.

Algunos posibles desarrollos futuros se proponen a continuación. Una posibilidad es utilizar un espacio de acciones $\mathcal{A}$ continuo en vez de discreto. Esto permitiría mejores curvas en la arena. Otro posible desarrollo es programar nuevas funciones de recompensa para tareas diferente como, por ejemplo, buscar dianas alrededor de las paredes de la arena o empujar un globo alrededor de la arena.

APPLICATION OF ARTIFICIAL INTELLIGENCE ALGORITHMS TO PILOT A MOTORIZED ROBOT

Author: Moros Barandiarán, Eduardo

Director: Oyallon, Edouard

Collaborating entity: CNRS – Centre National de Research Scientifique français

SUMMARY OF THE PROJECT

This project is part of a larger one that wants to increase understanding of artificial intelligence in children. Artificial intelligence and machine learning are fast growing fields that will be very important in future society, so preparing children in these fields from early on can be very beneficial. To do so, the CNRS is developing a robot who learns to do different tasks using machine learning algorithms. In this frame, the project consists in developing an algorithm who pilots the robot around a rectangular arena with the goal of avoiding the walls, using a frontal camera as sensorial input.

The algorithm only needs to decide what actions to take, the actual command signal for the robot is calculated by the client's program where the algorithm is embedded. To develop this algorithm, the client supplied a simulator replicating the robot's movements. Most of the work was done in this simulator, making a final test in a physical arena using a robot AlphaBot from Waveshare, supplied by the client.

The existing literature points to three main fields within machine learning: supervised learning, unsupervised learning and reinforcement learning. Out of these, reinforcement learning is the most adapted to this project, because it is the only one interacting with an environment in real time, like a robot would do when moving around the arena.

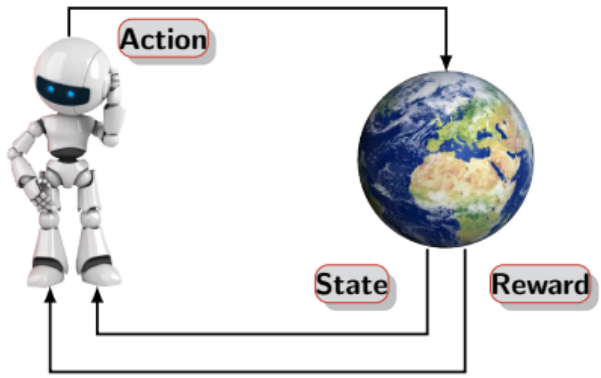


Figure 1: Reinforcment learning framework

In reinforcement learning, an agent (in this case the robot) learns to map situations to actions maximizing a numerical reward given by the environment. It is a trial and error approach of learning similar to our own way of learning when facing an unknown environment. Reinforcement learning is based in an interaction loop between the agent

and the environment shown in Figure 1. The agent is at a state s_t at time t, executes an action a_t following a policy π and observes the new state of the environment s_{t+1} as well as a reward r_t. The main difficulty faced by reinforcement learning is the sparseness of the information given by the reward. Actions affect not only the immediate reward, but also all the subsequent ones, so interpreting which actions render a good reward is not simple.

There are some concepts of reinforcement learning that need to be introduced:

- **Space of states $\mathcal{S}$:** it is the space made of all the possible states s that can be taken by the environment. In our case the environment is a camera feed, so the space of states is all of the possible images shown by the camera. $\mathcal{S}$ is huge in this problem.
- **Space of actions $\mathcal{A}$:** it is the space of all the possible actions the agent can perform. In this project $\mathcal{A}$ is restrained to a discrete space of five actions: {Forward, Forward Left, Forward Right, Backward Left, Backward Right}
- **Policy π:** a policy is the function that defines the behavior of the agent, telling it what action to take in a specific situation. The goal of reinforcement learning is to reach the optimal policy for the problem.
- **Return D:** an action's impact cannot be measured by the immediate reward because its repercussions can show hundreds of steps later. To better measure the impact of actions the variable return D is used. It is the discounted sum of all the rewards from the current time to the end of the experience. It clearly depends on the agent's policy.

$$D^{\pi}(s) = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s$$

- **State-action value function Q:** the function Q(s,a) represents the return value that can be expected if an agent in a state s performs the action a. Mathematically it is defined like:

$$Q^{\pi}(s,a) = \mathbb{E}^{\pi}\{D^{\pi}(s)|a_0 = a\} = \mathbb{E}^{\pi}\left\{\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right\}$$

Q is a recurrent function, because it can be written as the immediate reward at time t plus the new Q at t+1. This leads to the Bellman equation, widely used in reinforcement learning:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a)$$

Most simple reinforcement learning algorithms search to calculate explicitly Q for every pair state-action and deduce the optimal policy by maximizing Q. Repeatedly doing this will lead the algorithm to converge towards the maximum value of Q for every pair s-a and thus making the policy converge towards the optimal policy. This is not feasible in this problem because the state of spaces is too big and Q cannot be calculated explicitly.

Instead a neural network is used to approximate Q. A neural network is a number of interconnected neurons, with the connections being handled by the weights θ. The Bellman equation will be used iteratively to train the weights θ to maximize Q. Once Q is maximized, the optimal policy can be deducted by following the greedy policy. The weights θ are trained using gradient descent. This is called Deep Q Learning.

The algorithm developed in this project is the following one:

Algorithm Adapted DQN for Learning Robot

Ask camera for resolution #only for real robot
Initialize memory buffer $\mathcal{D}$ to capacity N
Initialize γ, n, ϵ and environment
Initialize policy neural network Q with random weights θ
Initialize C and target neural network Q' with random weights θ'
Initialize optimizer and learning rate lr
Initialize state $x_1 = \{s_1, m_1\}$
for $t = 1, T$ **do**
    Force an action a_t else
        With probability ϵ select a random action a_t else select $a_t = \max_a Q^{\theta}(x_t, a)$
    Execute action a_t in the environment and wait for response
    while not response
        Optimize
    Observe state $x_{t+1} = \{s_{t+1}, m_{t+1}\}$ and reward r_t
    Store transition (x_t, a_t, r_t, x_{t+1}) in $\mathcal{D}$
    Plot reward and loss
    $x_t \leftarrow x_{t+1}$
    Every C steps do $\theta' \leftarrow \theta$
end for

Function Optimize

Sample random batch of n transitions (x_j, a_j, r_j, x_{j+1}) in $\mathcal{D}$
Set prediction $y_j = Q(x_j, a_j; \theta)$
Set target $z_j = \gamma \max_{a'} Q'(x_{j+1}, a'; \theta') + r_j$
$\delta = y_j - z_j$
Calculate loss $\mathcal{L}(\delta)$
Perform a step of the optimizer on $\mathcal{L}(\delta)$
Clamp parameters θ

The main addition to normal Deep Q Learning algorithms is that an experience replay is implemented. The algorithm stores each transition from state t to state t+1 in a buffer and uses the transitions when optimizing to achieve a better convergence.

The loss $\mathcal{L}(\delta)$ is the mean squared loss of the n values of δ: $\mathcal{L}(\delta) = \frac{1}{n}\sum {\delta_j}^2$

For this algorithm to work it needs a reward function. The reward function was crafted in the following way: the agent will receive a reward equal to its speed projected in its forward direction and it will be punished if it is blocked or goes backward. The reward is defined like:

$$r_t = \begin{cases} \boldsymbol{v} \cdot \boldsymbol{u} & if\ \boldsymbol{v} \cdot \boldsymbol{u} > 0 \\ -100 & if\ \boldsymbol{v} \cdot \boldsymbol{u} = 0 \\ -50 & if\ \boldsymbol{v} \cdot \boldsymbol{u} < 0 \end{cases}$$

v is the speed and u is the forward direction.

The algorithm has a set of parameters, called hyperparameters, fixed by the programmer before the experience and tuned to adapt the algorithm to a particular environment. A great part of the project was tuning this hyperparameters:

- **Buffer's capacity N:** it represents how many transitions can be stored. A small value can prevent the agent from learning; a big value wastes storage space. A value of N = 1000 was found.
- **Gamma γ:** parameter between 0 and 1 representing the importance given to long-term rewards. If γ is close to 1, the agent will be far-sighted and if it is close to 0 it will be short-sighted. The optimal value was found to be around 0.8.
- **Batch size n:** when optimizing a random batch of n actions is used. The bigger n is, the better the convergence will be. In practice, n is limited by the computing power of the computer. A value of n = 100 was found.
- **Exploration:** with a probability of ϵ, a random action is chosen. This is called exploration. Exploration prevents the algorithm from converging towards a bad behavior by forcing it to explore points of the space $\mathcal{S}$ x $\mathcal{A}$ that had never been visited. Several explorations curves (variation of ϵ throughout the experience) were tried. Finally, a constant value of ϵ = 0.1 was chosen.
- **Neural network's structure:** the neural network's neurons are arranged in layers. There are different types of layers: dense layers, convolutional layers, pooling layers, etc. Different combinations of these layers were tried and the chosen structure was three consecutive dense layers of 150, 100 and 50 neurons respectively.
- **Target update C:** two neural networks were used to achieve stability (Q and Q'). The target network Q' does not learn and its parameters θ' are actualized every C steps with the values of θ. If C is too small, the algorithm will be too unstable; if C is too big, the learning will be erratic.
- **Optimizer:** the neural network is optimized using some variation of the gradient descent method set by the optimizer. Two methods were tried: SGD and Adam. Adam was chosen because it yielded better results.
- **Learning rate lr:** the learning rate is a parameter of the optimizer. It represents how quickly the network learns. Each optimizer has a different range of valid learning rates. If the learning rate is too big, the algorithm can skip the global minimum and settle for a local one, learning a bad behavior. If the learning rate is too small, the convergence will be too long. The chosen learning rate for Adam was 0.01.

The algorithm was implemented using Python and the library for machine learning Pytorch. The algorithm's parameters were optimized in the simulator and a test was set up with the real robot. The robot learned correctly in a reasonable time.

Some future developments are proposed. One possible development is to try with a continuous space $\mathcal{A}$ instead of a discrete one. This would allow for better trajectories in the arena. Another possible development is to code new rewards for different tasks such as looking for targets in the arena's walls or following a balloon around the arena.

To my parents and my friends of the 4T

Acknowledgements

First and foremost I want to thank my parents for giving me the opportunity of studying this career in this university, and especially for the effort they have done to send me to Paris. I also want to thank ICAI for the opportunity of studying in a incredible place like Ecole Centrale Paris.

I want to thank my friend Nicolas for all his help with the integration of the algorithm to the graphic interface, which was a bit tricky.

And finally I want to thank my director Edouard Oyallon and the client Thomas Deneux for their availability at all times and their invaluable counsel. This was a complex field that I did not know and they taught me the bases and had faith on me even when things did not work. Thank you.

Index

Figure Index

1

Introduction

This chapter is an introduction addressing the global project set in motion by the client (CNRS) and the different tools they provided for this branch of the project. There is also an introduction to Machine Learning and its main issues

1.1. The CNRS project

This project was proposed by Mr. Thomas Deneux [1], a member of the CNRS (Centre Nationale de la Research Scientifique). Mr. Deneux is a research engineer who works for the Neuro-PSI (Institut des Neurosciences Paris-Saclay). He is specialized in the field of neurosciences and one of the projects he is conducting is the project "AlphaI", where the project narrated in this memory is included

The goal of the AlphaI project is to familiarize children with the bases of Artificial Intelligence. It consists in a robot (the AlphaBot) that will learn different tasks using Machine Learning methods. There is also a graphic interface that gives a more intuitive vision of the results and allows the children to interact with the robot. The AlphaI project searches to develop this robot, integrating several ways of learning and several tasks to learn. Interacting with it, the children will soon grow comfortable with the concept of an intelligent machine and will acquire a deeper understanding of the process of learning from an early age. This is important because Artificial Intelligence will have an important role to play in future society, and it will be surely needed in a great range of jobs. While the children will learn about Artificial Intelligence and Machine Learning, they will also learn about themselves, thanks to the close relationship between the fields of neuroscience and machine learning. They will better grasp the way the human brain works.

For the moment only one task has been tackled: to navigate a rectangular arena in the best possible way. This can be done in several ways depending on the sensorial inputs allowed to the robot: infrared sensors, wheel speed information, frontal camera and so on. The sensorial inputs determine the complexity of the problem. If only the wheel speed information is available, the robot will learn to move in a straight line and turn once it hits a wall. However, if the frontal camera is available, the robot will be able to "see" the walls and avoid them before crashing into them. Different tasks have

been contemplated as well: a robot race around a circus-type arena and a robot that locates and pursues targets (such a balloon).

The AlphaI project is still in progress. For the moment, the client takes part in events that link children and technology, such as Startup 4 Kids [2], and does shows in schools. However, the goal is to commercialize the robot in the future.

The project treated by this memory is a subproject of AlphaI. It consists in choosing and writing the algorithm that will allow the robot to learn how to navigate a rectangular arena without crashing into the walls using a frontal camera as the sensorial input.

1.2. Introduction to Machine Learning

Let us take an overall look of what Machine Learning is and how it is structured, since we will be talking about it quite a lot. Machine Learning is defined as:

> *"Machine Learning is the scientific study of algorithms and statistical models that computer systems use in order to perform a task effectively without using explicit instructions" [3]*

In other words, ML algorithms learn to perform a task, without a human programmer specifying the instructions to said task. But, how is it different from other programs? In what kind of problem is it used?

ML is used in any of the following three cases [SHAI14]:

- *Problems too complex for humans to solve*: when the data is too large and complex, extracting valuable information becomes very difficult. For example, detecting patterns in large and complex data in areas like biostatistics or weather detection.
- *Problems where humans are incapable of explaining their expertise*: to write a program solving a certain problem, you need to be able to write a set of instructions that follow up to the resolution. However, there are tasks where this is not possible, such as handwriting recognition. How to define what makes up a 2 or a 7, valid for every handwriting in the world?
- *Problems where adaptability is needed*: a classical problem will always do the same thing, unless it is manually changed. But in problems where the "instructions" may change from one experience to another or from one user to another, ML is needed.

In this particular project, the problem faced can be categorized in the second and third cases. Learning to avoid obstacles from vision (camera) is something humans do, but they cannot explain how. Adaptability is also important, because the searched must be universal, so it needs to be independent from the starting position, for example.

Fig. 1 [FRAG18] pictures the difference between the approach of a classical program and that of a ML program. The ML algorithms use the Answers to learn the Rules. Therefore, it stands to reason that ML algorithms should be classified regarding how the Answers are presented to them, because this is what defines what type of learning process will be taking place.

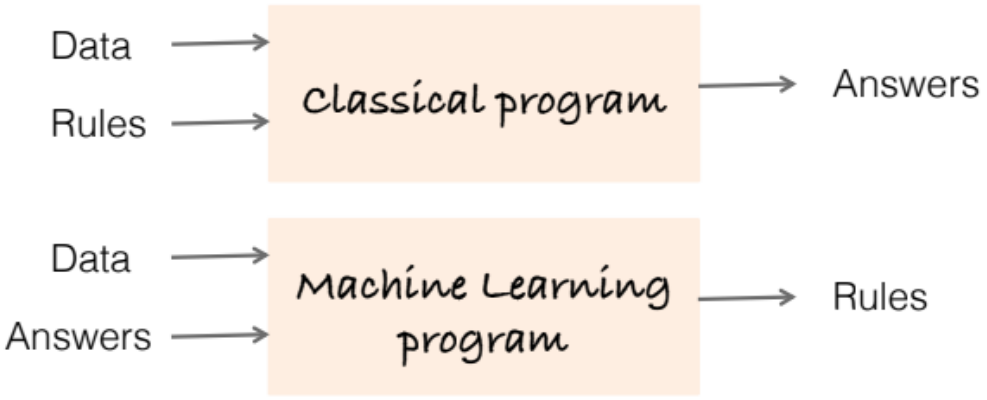


Fig.1 – Difference between a classical program and a ML program

There are three main types of ML algorithms, based on the learning process taking place [FRAG18]:

- *Supervised learning:* the program receives all of the answers. For example, a programs needs to learn how to classify a handwritten number. The program will have a lot of pictures of handwritten numbers already labeled with the answers, and will deduct the rules. This would be the human equivalent to learn through examples.
- *Unsupervised learning:* the program receives no answers. It receives only the data and it will deduct hidden patterns from it.
- *Reinforcement Learning:* the program does not receive all of the answers, but just a part of them, for example, an indicator on how well you are doing. This method of learning can be approached to the human process of trial and error.

2

AlphaI: Learning Robot

The project is part of a larger project, so there are some things that were imposed by the client. The client also supplied some materials that had already been developed. In this chapter all of this will be presented and explained.

The goal of the algorithm is: *"to allow the robot to learn how to navigate a rectangular arena without crashing into the walls, using a frontal camera as the sensorial input"*. There are some things that were fixed beforehand though: the arena, the robot and the camera. On top of that, a graphic interface was provided by the client, as well as a simulator of the robot's behavior to speed up the advance of the project.

2.1 The arena

The client provided the arena used in the project. It consisted in 2 wood planks measuring 2 m and 2 wood planks measuring 4 m. They were attached with hinges that were screwed to the plank. They made 20 cm of height. The planks were painted bright yellow, to have a bigger contrast with background, making them easier to spot.

The robot's wheels had a tendency to slide if the floor was too smooth, so we added a carpet, and we chose it to be blue, to have a bigger contrast with the walls. A diagram representing the arena (in an eagle's eye perspective) is presented in Fig 2.

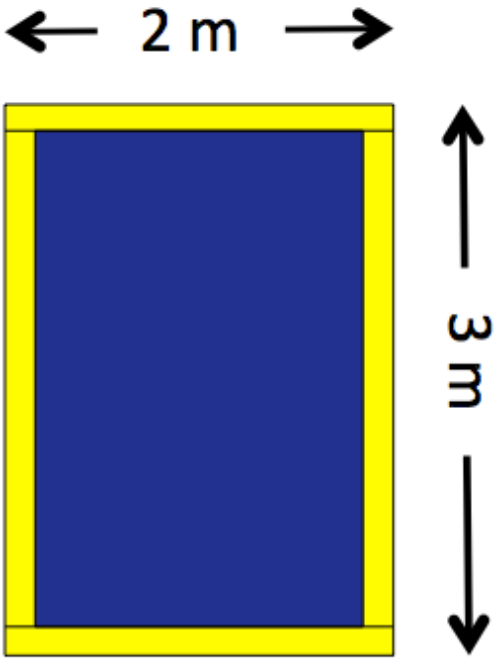


Fig 2 – Eagle's eye diagram of the arena

2.2. The robot

The robot was supplied by the client. It is the model AlphaBot from the electronics store Waveshare. The complete set of specifications can be found in their web page [4]. It is a 2 wheeled robot that can exchange data via a Wi-Fi connection, thanks to the RaspberryPi module. The algorithm runs in the computer and then the commands are sent to the robot via the Wi-Fi connection. This is indispensable because the robot's microprocessor is not powerful enough to run the algorithm itself.

The client also supplied the program that gives the commands to the robot. For example, for the robot to go forward the algorithm supplies the command "Forward" and this program computes the command signal required by the robot (motor's power, brakes, etc.) based on the robot's speed and acceleration. Let us call this program the command program.

The command program's input is one action out of the next 5: Forward, Forward Left, Forward Right, Backward Right, Backward Left. For Forward the two wheels turn at the same speed. For Forward Left the left wheel is blocked and the right runs normally. For Forward Right the left wheel runs normally and the right one is blocked. And so on. This means there are only 5 possible actions at all time.

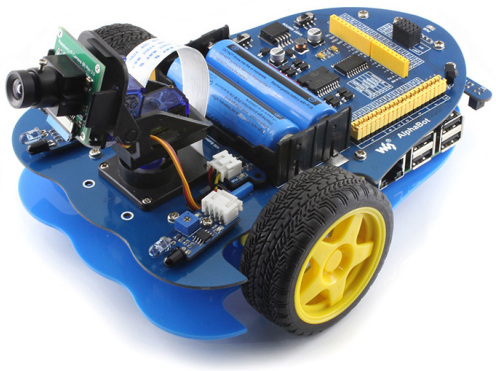


Fig 3 – The AlphaBot robot

2.3. The graphic interface

The client also provided a graphic interface where some information was given in a more esthetic way, and where parameters could be changed more easily. However, it was necessary to introduce the algorithm into the graphic interface's code, and this required a bit of work. In Fig 4 we can see how the graphic interface looks like.

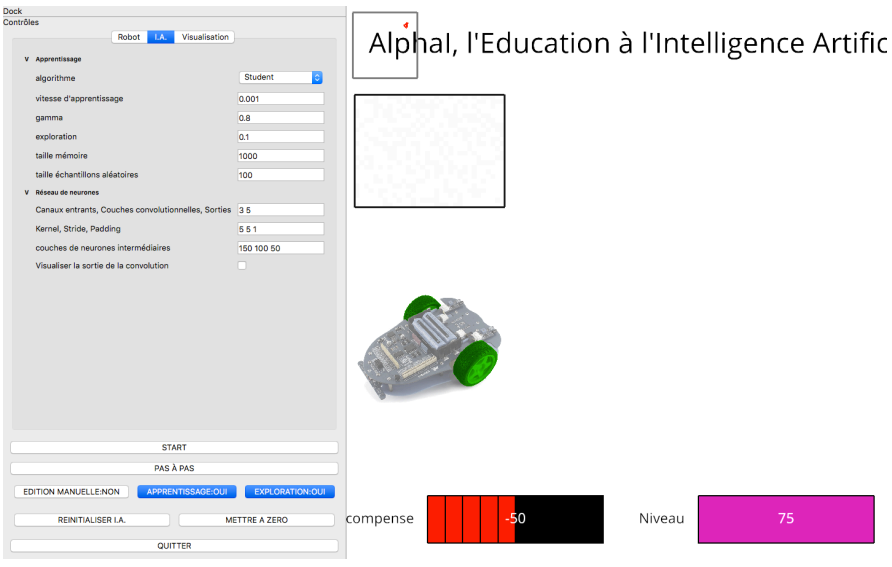


Fig 4 – Graphic interface

2.4 The simulator

The real robot is too complicated and too slow for fine analysis of the algorithm. To be able to develop the algorithm more quickly, the client supplied a simulator that simulated the movement of the robot around the arena. Most of the work done in the project has been with the simulator.

The simulator has different configurations; we will go over them briefly. The configuration that has been used the most throughout the project is TopCameraLocalOri, but the other configurations have been useful for making choices and testing certain aspects of the algorithm.

2.4.1. WheelsBlocked

The first and easier configuration is 'WheelsBlocked'. In this configuration, the simulator's output is a Boolean variable that provides the state of the wheels. If the robot is unable to move and therefore the wheels are blocked, the output will be [True]. If the robot is moving, and therefore the wheels are not blocked, the output will be [False].

2.4.2. RobotState

In this configuration, the simulator's output is an array of 5 real numbers, normalized between -1 and 1, set in the following way: [x position; y position; orientation; left wheel speed; right wheel speed]

An example of output in this configuration is:

```
[array([[ 0.415],
        [ 0.648],
        [ 0.549],
        [-0.25 ],
        [ 0.5  ]])]
```

2.4.3. TopCamera

In this configuration, the simulator's output is an eagle's eye vision of the arena with the robot inside. Normally, images have three channels (RGB), but this simulator provides a black and white image that is easier to manipulate. In this configuration, it is difficult for the computer to see the difference between one instant and the next because the robot is small compared to the rest of the arena. That is why other configurations of the simulator were developed based on the TopCamera configuration.

2.4.4. TopCameraDiff

This configuration is based in TopCamera. With the goal of having a better notion of the robot's movement, the difference between the image in time t-1 and in time t has been calculated. This configuration shows where the robot was in the previous instant in black and where it is now in white.

Fig 5 – Simulator in TopCameraLocal

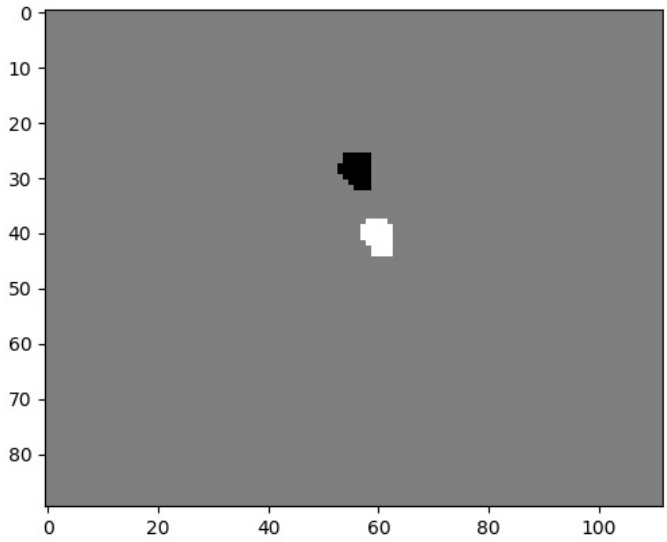


Fig 6 – Simulator in TopCameraDiff

2.4.5. TopCameraLocal

The two configurations showed above are not helpful to learn because there is too much information that is not relevant (all the empty space far from the robot). This is why there is another family of configurations: the TopCameraLocal. In this configuration, the output is still an image, but it is centered in the robot, to get rid of the unhelpful information.

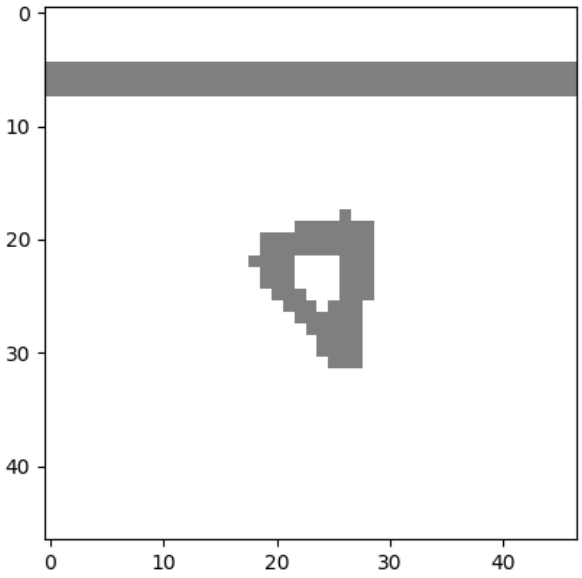


Fig 6 – Simulator in TopCameraLocal

2.4.6. TopCameraLocalOri

This is the final version of the configuration TopCamera. It is a variation of the TopCameraLocal, where the robot is always in the center of the image and in the same orientation. This configuration is the closest to the real situation, because from the point of view of the robot, the robot is fixed and is the rest of the arena who moves. This is also the configurations that makes learning easier, because the robot "sees" the walls approaching.

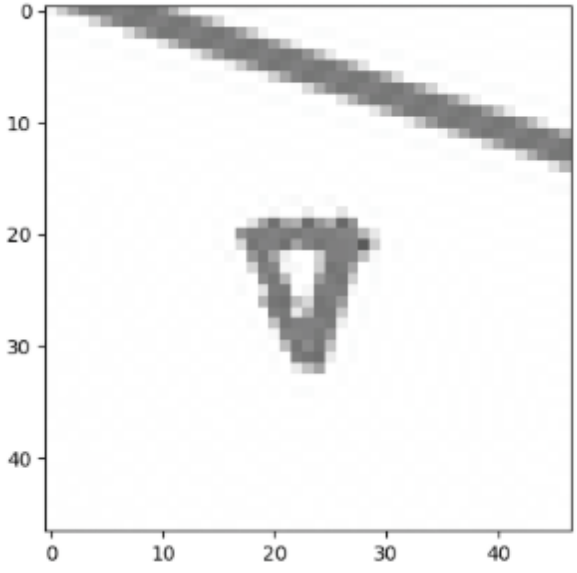


Fig 7 – Simulator in TopCameraLocalOri

3

State of the art

In this chapter we will speak about the different methods of machine learning that could be applied to this problem. This chapter will be a small introduction to the state of the art, but chapters 4 and 5 will also be dedicated in some way to the state of the art

A usual state of the art chapter presents the existing solutions in the industry for problems that are similar to the one the student is facing. In most projects, the goal of the project is to reach an optimal solution for the considered problem, so the state of the art shows the student what solutions (usually optimal) are being developed in the industry. Then, using this information, the student can decide how to solve the problem. The solution is more important than the method. For example, if the project was to measure temperature, it is more important for the project to measure temperature correctly, rather than how it measures it (if it is electronic or mercury based). The "how" (mercury or electronic) is certainly important, but secondary to the "what" (measuring temperature).

However, in this project, the "how" is more important than the "what". There are better ways of piloting a robot through a room than the one that has been developed. For example, a Roomba vacuum cleaner does it more efficiently with a combination of infrared sensors and a pre-fixed algorithm that does not learn [5]. Another example, a Tesla`s self-driving car would use a combination of cameras and machine learning algorithms to recognize the obstacles ahead and turn before hitting them [6]. However, the purpose of this project is first and foremost pedagogical. The Roomba's solution is not interesting because it does not educate children in the field of artificial intelligence. Tesla's solution is not interesting either because it is too complex for children to understand.

This is why this state of the art, rather than talking about solutions developed by other people, will be speaking about the existing machine learning algorithms and whether they are adapted to the task at hand.

Like it was briefly mentioned in the introduction, there are three main type of machine learning algorithms: unsupervised learning, supervised learning and reinforcement learning. Their characteristics and common uses will be discussed, analyzing whether they are appropriate for the task or not.

3.1. Unsupervised Learning

Unsupervised learning algorithms are mainly characterized because they do not access any human expertise in their learning process. Their main objective is to transform data into a more useful type of data. We could divide unsupervised learning algorithms in two types: dimensionality reduction and clustering [FRAG18].

- **Dimensionality reduction:** the goal is to find the redundant information present in some data set, and eliminate these redundancies, reducing the data's dimension. Fig 9 pictures an intuitive description of dimensionality reduction. The data is in 2 dimensions, but when analyzed properly, it is clear that only the position in the diagonal is representative.

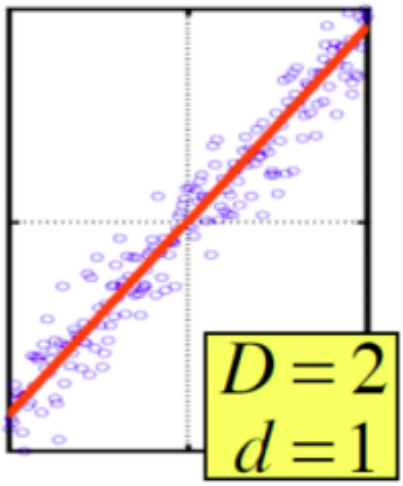


Fig 9 – Example of dimensionality reduction

- **Clustering:** clustering is the most common unsupervised learning application. It takes some data set and finds groups hidden inside. An intuitive definition of clustering can be seen in Fig 10. The algorithms can find relationships between different data points and separate them in clusters. These algorithms are used in applications like: customer segmentation, topic modeling, community detection in networks, and so on.

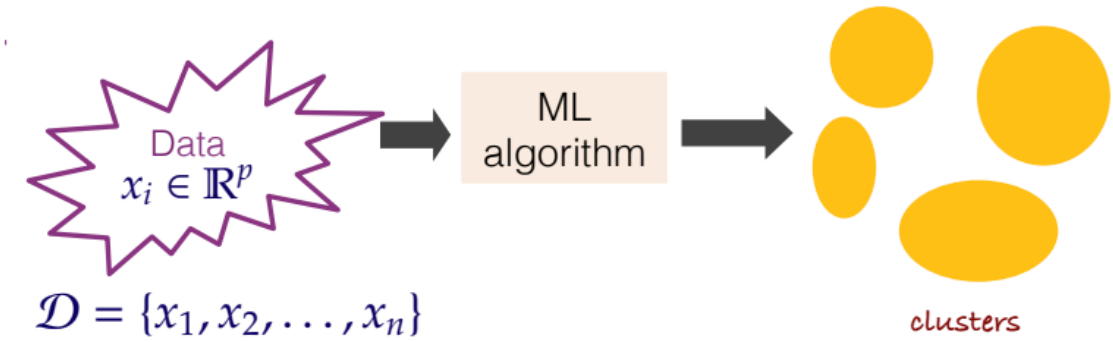


Fig 10 – Example of clustering

Unsupervised learning is very useful, but by itself it is not able to fulfill the requirements of the project. Clustering will point to groups with common characteristics hidden in data, but it will not interpret these groups nor use this information to make decisions. Unsupervised learning is mainly a tool for decision-making, but it needs a human to make the decisions, because there is not human expertise included in the algorithm. Thus, unsupervised learning cannot be used in this problem.

3.2. Supervised Learning

Supervised learning algorithms are the most commonly used in machine learning. Like unsupervised learning, they receive a data set, but unlike unsupervised learning, the data is labeled, introducing human expertise. For example, the algorithm will receive a set of images of cats and dogs and for each image; it will also receive a label saying if the image is a cat or a dog. This label comes from human expertise. Supervised learning algorithms are often used as classifiers. They learn from a training data set how to classify data into some category or another and then they are applied to different data sets, making predictions. A diagram explaining how supervised learning works, is shown in Fig 11.

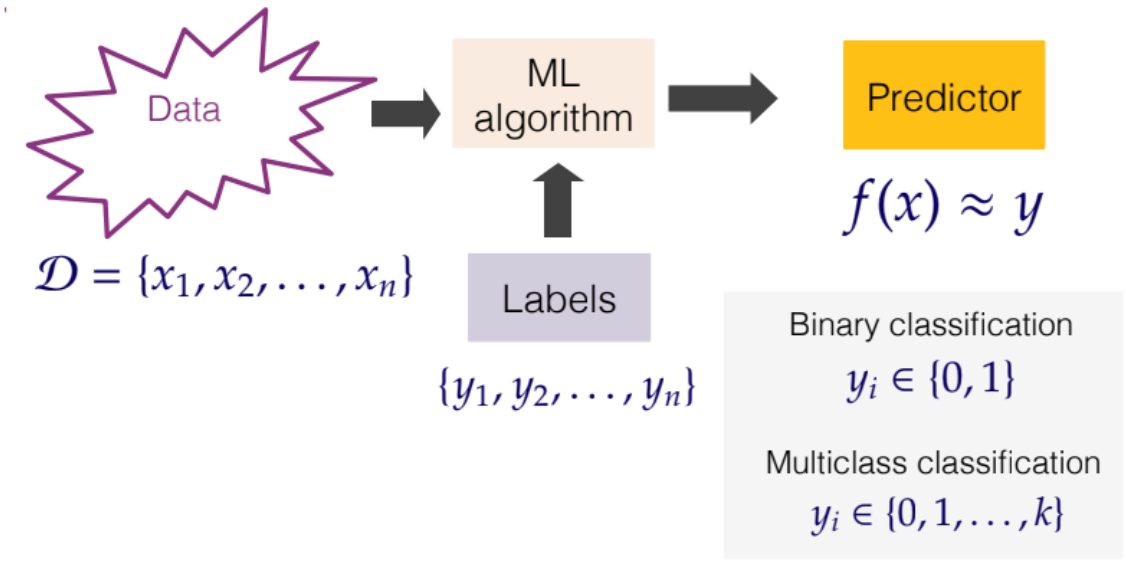


Fig 11 – Diagram of supervised learning

Supervised learning is used in image recognition, speech recognition, and mostly everything that involves any pattern recognition. This includes self-driving cars as well.

A solution for the task at hand using supervised learning could be envisaged. However, it would require a combination of at least two algorithms. For example, a possible solution would be made of:

1) An algorithm of image recognition, capable of differentiating the walls and floor
2) An algorithm linking the elements recognized by the previous one with some risk classes.

However, it is not practical to implement this solution in this case. For most of the supervised learning applications, training databases that are already labeled exist. But in this case, there is not a clear training labeled database to be used, so it would be necessary to build it.

Also, supervised learning is the machine equivalent of learning by examples. A child may learn to read using examples, but a child does not learn to walk seeing examples of people walking. A child learns to walk by a trial and error, and the machine equivalent of this is reinforcement learning. We can see in [RICH17]:

"*Supervised learning is learning from a training set of labeled examples provided by a knowledgable external supervisor. Each example is a description of a situation together with a specification—the label—of the correct action the system should take to that situation, which is often to identify a category to which the situation belongs. The object of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. This is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory—where one would expect learning to be most beneficial—an agent must be able to learn from its own experience."*

3.3. Reinforcement Learning

Reinforcement learning is the third main category of machine learning. The implemented solution is a reinforcement learning one. It is an ample field, however, so a whole chapter will be dedicated to explaining its bases and the different methods that could be applied for the considered problem.

4

Reinforcement Learning

The purpose of this chapter is to explain the basics of reinforcement learning and then choose a solution out of the possible algorithms

4.1. Concept

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them [RICH17]

Using the framework introduced in the diagram depicting the elements of machine learning in Fig 1, the Answers would be the reward, and the Rules, the actions. The process of learning is through trial and error. The learner (called agent) uses the reward to evaluate the performance. However, the information contained in the reward is sparse and delayed, because the actions taken in the present will affect the rewards in the short and middle term futures.

Let us define the term “agent”:.

“*An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.”*[NIKO18]

The agent in this project is the robot. It perceives the environment through the camera and acts upon this environment by moving. When interacting with the environment the agent tries to achieve a specific goal (navigate the arena avoiding the walls); perceives the consequences of its actions (when there is a lot of yellow and I go forward, I get stuck); identifies important behavioral components (when there is yellow there is a wall); modifies its behavior (turn when there is yellow); applies its knowledge in new tasks context (when a different wall is presented it turns).

Fig 12 depicts the classical loop present in all reinforcement learning problems, defining the exchange between the environment and the agent.

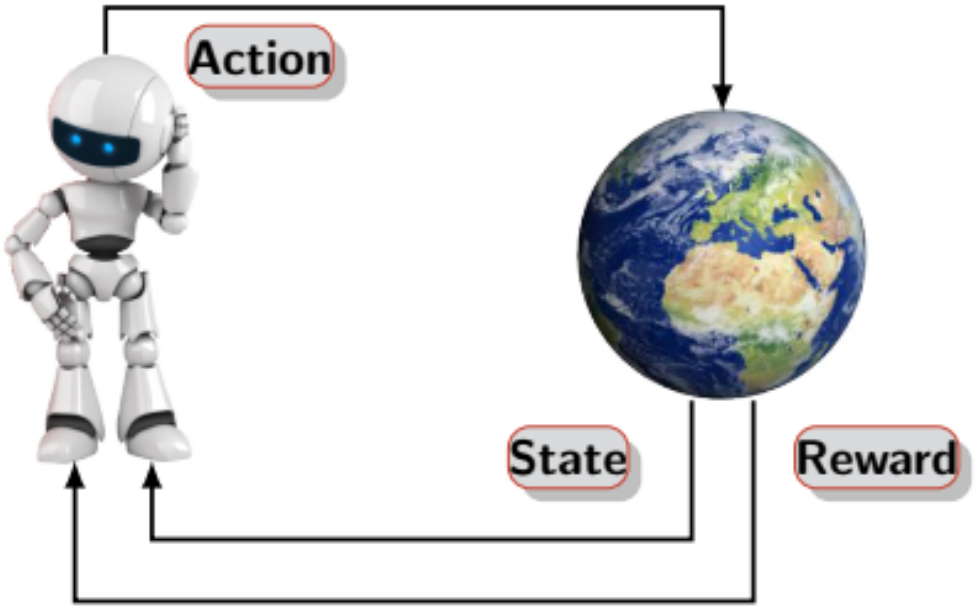


Fig 12 – Reinforcement learning framework

At each time step t, the agent executes an action; making the environment change from state t to state t+1. The environment also supplies an immediate reward to the agent. This is formalized as a MDP (Markov Decision Process).

4.2 Markov Decision Process

A MDP is defined by the following n-tuple:

$$\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma\}$$

where:

- $\mathcal{S}$ is the state space, made of all the possible states of the environment. In our case, one state is an image from the camera. The space of states is all of the possible images that can be shown by the camera. If each image is a RGB image with a resolution of 16x12, and each pixel can range from 0 to 255. There are 3x16x12 = 576 pixels. Therefore, there are 576x255 = 146 880 possible states. The state space $\mathcal{S}$ is huge.
- $\mathcal{A}$ is the action space. $\mathcal{A}$ is a 5-tuple made up of: {Forward, Forward Left, Forward Right, Backward Left, Backward Right}(see 2.2).
- $\mathcal{R}$ is a reward function that will have to be defined.
- $\mathcal{P}$ is the state-transition distribution. It is a mapping from states to the next states via the actions. In other words, $\mathcal{P}_{ss\prime}^{a}$ is 1 if an agent in state s who performs the action a, ends in state s'. Else, it will be 0.
- γ is a discount factor called gamma.

The relationship between the environment μ and the agent at a time step t is defined by:

- Observation of $s_t \in \mathcal{S}$
- Execute action $a_t \in \mathcal{A}$
- Receive reward $r_t \in \mathbb{R}$

4.3. Policy

A policy defines the agent's behavior. It can be either stochastic or deterministic.

- **Deterministic policy**: in a state s, the agent does the action a.

$$\pi : \mathcal{S} \rightarrow \mathcal{A}$$

- **Stochastic policy:** in a state s, the agent does the action a with a probability p.

$$\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$$

What the agent effectively learns is the policy. It can be interpreted as the function that gives the agent the rules to follow in each situation. The policy is changed throughout all the learning experience until it represents a set of rules that accomplish the task correctly. When this is achieved the policy is called the optimal policy π^*.

4.4. Return

Every machine learning problem is ultimately an optimization problem. In reinforcement learning, the goal is to maximize the "return". It is defined as:

$$D^{\pi}(s) = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t \,|\, s_0 = s \qquad (4.1)$$

It is a sum of rewards, prioritizing the immediate rewards over the far-sighted ones, via the discount factor γ. Conceptually, the return can be seen like the impact of a certain action over time. If it is high it means that this action has given a lot of reward over time, therefore it is a good action.

The hyper parameter[1] gamma is important in the reinforcement learning scenario. An action k time steps later will have an impact of γ^{k-1}.So, when γ is close to 0, the agent will prioritize immediate rewards. When γ is close to 1, the agent will have a greater long-term vision. Gamma is inherent to the task and the reward function; there are tasks that require a shorter-term vision than others.

The return is at the base of two important concepts in reinforcement learning: the State-Value function and the State-Action value function.

[1] The term hyper parameter is used to distinguish the parameters that are set by the programmer before the experience (hyper parameters) and the parameters the agent learns by itself during the experience (parameters)

4.5. State-action value function

If the return is the goal of RL algorithms, the State-Value function (V) and the State-Action value function (Q) are its backbone. Every algorithm uses one or the other at some point. Let us define them. In the following equations π represents the agent's policy, s is an state $\in \mathcal{S}$, a is an action $\in \mathcal{A}$, D is the return defined in (4.1), and $\mathbb{E}$ is the mathematical expected value.

$$V^{\pi}(s) = \mathbb{E}^{\pi}\{D^{\pi}(s)\} = \mathbb{E}^{\pi}\left\{\sum_{t=0}^{\infty}\gamma^{t}r_{t}|\, s_{0} = s\right\} \tag{4.2}$$

$$Q^{\pi}(s,a) = \mathbb{E}^{\pi}\{D^{\pi}(s)|a_{0} = a\} = \mathbb{E}^{\pi}\left\{\sum_{t=0}^{\infty}\gamma^{t}r_{t}|\, s_{0} = s, a_{0} = a\right\} \tag{4.3}$$

These two functions are approximately the same concept: a function that calculates what return should be expected if a certain policy π is followed. The difference is that where V represents the return that can be expected when the agent is in a state s, Q represents the return that can be expected when an agent in a state s performs an action a.

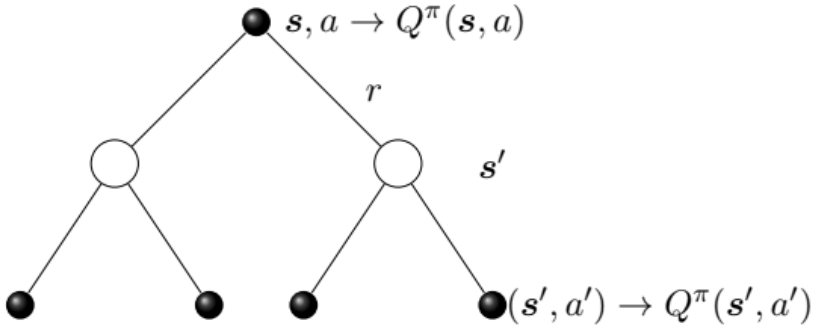
Both of these functions can be written in a recurrent way. Let us take Q:

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}^{\pi}\left\{\gamma^{0}r_{0} + \sum_{t=1}^{\infty}\gamma^{t}r_{t}\,\middle|\, s_{0} = s, a_{0} = a\right\} \\ &= \mathbb{E}^{\pi}[r_{0}\,|s_{0} = s, a_{0} = a] + \mathbb{E}^{\pi}\left\{\sum_{t=0}^{\infty}\gamma^{t+1}r_{t}\,\middle|\, s_{0} = s', a_{0} = a'\right\} \\ &= r_{0} + \gamma Q^{\pi}(s',a') \end{aligned} \tag{4.4}$$

When the equation (4.4) is written for a time step t and following the greedy policy for choosing a' it is called the Bellman equation:

$$Q(s_{t},a_{t}) = r_{t} + \gamma \max_{a} Q(s_{t+1},a) \tag{4.5}$$

The Q function at a time t can be written using the Q function of the time t+1. The recurrence of the Q and V functions is very useful in algorithms, who are based in



recurrent structures. The Bellman equation of Q is shown in Fig 13.

Fig 13 – Recurrence of the Q function

4.6. Problem's characteristics

The theoretic bases in which reinforcement algorithms stand on have been set, but before going into discussion of the algorithms themselves let us discuss about some characteristics important in reinforcement learning algorithms.

4.6.1. Prediction and control problems

Supervised machine learning problems are basically prediction problems. The goal is to use available data to learn how to predict some variable when facing unknown data (see Fig 11). Reinforcement learning problems are not exactly supervised learning, but they are, in a way, prediction problems. RL agents learn how to predict the return (4.1).

However, they are usually considered control problems rather than prediction problems, because their main goal is to control the command signal (the actions) of an agent that interacts with an environment. They resemble in a way classic automatic control problems. They are still prediction problems because they need to predict the return as precisely as possible to efficiently control the agent's actions, but the prediction does not need to be perfect. For example, an algorithm that predicts illnesses from X ray images needs its prediction to be as accurate as possible, usually using lots of statistics to be sure that its predictions are accurate. In RL, the accuracy of the prediction is important but secondary to the control structure.

4.6.2. Model-based vs. Model-free

A RL algorithm can be either model-based or model-free.

Model-based algorithms try to map the internal workings of the environment they interact with. This is called planning. For example, given a state and action, a model-based algorithm might be able to predict the next state and the reward. Doing this for all of the states in $\mathcal{S}$ will lead the agent to know how the environment works and then it will deduct from this the policy that has to be followed to accomplish the task.

Model-free algorithms work by classic trial and error, not trying to understand the environment in any way. They try things and hang on to those that give them good results. At the end they will learn that in a certain state a certain action will give good results in the future, but they do not know why and they cannot predict what the next state will be, nor the reward.

In this project the algorithm searched is model free. The reason for this is that the space of states is too big for a planning method to be feasable. The calculating power needed is too great. However, it is important to take into account that the solution implemented is model-free. Several times during this memory something like "the agent wants to predict the Q function as well as possible" will appear. This is absolutely not planning. The value of Q must be considered more like an intuition of whether one action will be good or not, rather than the prediction of the immediate reward.

4.6.3. Online vs. Offline

Most classic machine learning problems are off-line problems. The agent's learning and the evaluation of the accuracy of the model are done separately. The agent is trained in a training database and then a different test database is used to check that the learned model is accurate. If it is not, the algorithm is changed and so on.

In reinforcement learning algorithms there are both on-line and off-line algorithms. Off-line algorithms don't learn while interacting with the environment. They interact with the environment, and then they stop and they learn from the interaction. This is the same as saying that the policy is not changed while the experience is underway.

On-line algorithms change the policy at each time step. They perform an action, they get information, then they optimize the policy using this information, then they perform another action based in the new policy, they get new information and so on.

In our problem, either kind of algorithm could be deployed, but an on-line algorithm will carry better results. Because of the dimension of the space of states $\mathcal{S}$, there is no way of calculating the value of Q for every state. So, an approximator will need to be used (more on that later). Using an off-line algorithm will be much slower because we are using an approximator rather than calculating directly the Q value, so the value will tend to be further apart from the real one, specially at the start. This means that it will take a lot of experiences to learn. On top of that, off-line algorithms are normally used in episodic kind of environments, which are environments that work by repeating the same experience over and over. Our environment can be made episodic but it is not fundamentally episodic.

Using an on-line algorithm stands to reason, then.

4.6.4. On-policy vs. Off-policy

This difference is subtler. Reinforcement learning algorithms involve generally two different processes: following a policy to act, thus modifying the environment, and optimizing the policy using the information given by the environment. Because of Bellman's equation (4.5) when optimizing the policy π, a Q value needs to be calculated.

If this Q value is calculated using the policy π, the algorithm is called on-policy. Else, if it is calculated using a policy π', that may or may not be equal to π, the algorithm is called off-policy.

In the literature [RICH17], it is recommended to use off-policy rather than on-policy algorithms because they have more flexibility, thanks to the independence between the actions and the optimization. For example, the feature experience replay (which is introduced later) is only possible when using off-policy algorithms. So, this project used an off-policy algorithm.

4.7. Choice of algorithm

Throughout all this chapter and the previous one, a choice of what algorithm to implement has been in progress. For all the reasons that have already been given, the

algorithm that will be implemented needs to be a reinforcement learning algorithm, but which one?

There are three main reinforcement learning methods (each one with their subsequent variations). Let us introduce them briefly:

- **Dynamic programming:** this method consists in setting a random policy, then evaluate this policy (calculate V for every state), then make a policy improvement, and then evaluate the new policy, etc. This method has a major drawback: it requires a complete knowledge of the MDP (see 4.2) for computation of V and for improvement of the policy, which in practice is impossible for a states space so big. This method is model-based (the MDP is needed), and off-line (in fact, there is no need of interaction with the environment, it deduces the optimal policy from the complete knowledge of the MDP).
- **Monte Carlo method:** the Monte Carlo Method does not need knowledge of the MDP. The algorithm will calculate Q for every pair state-action following a policy π and then improve this policy by choosing at each state the action that will maximize Q or a random one to explore every pair state-action (this is called the ϵ-greedy policy). Then it will calculate again Q for every pair state-action and so on. To calculate Q, the algorithm will repeat an experiment several times. Each repetition of the experiment is called episode. At each episode it will sum up the rewards it obtained to compute the return D (4.1). Then, it will estimate Q as the mean of the returns from all the episodes. This method is off-line, and it is hardly scalable because of its high computing complexity.
- **Temporal Difference:** TD does not need knowledge of the MDP either. It follows the same general structure as MC. Calculation of Q for every pair s-a following policy π; improving policy by following the ϵ-greedy policy; repeat. However, to calculate Q it does not require a whole experiment, only a single time step. It starts by a random Q and it actualizes it at each time step using a term called temporal difference error (this term is just the right term of Bellman's equation minus its left term). This is the most adapted type of solution for this project. Two algorithms stand out in this category: the on-policy SARSA and the off-policy Q-Learning. As explained in 4.6.4, an off-policy solution will be implemented for flexibility reasons.

4.8. Q-Learning

The Q-Learning algorithm is online, model-free and off-policy. It starts by assigning a random value of Q for every state-action pair and then it actualizes this value following the following equation:

$$Q^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha\delta_t \tag{4.5}$$

where α is the learning rate and δ is the temporal difference error:

$$\delta_t = r_t + \gamma \max_a Q^{\pi}(s_{t+1}, a) - Q^{\pi}(s_t, a_t) \tag{4.6}$$

The Q-Learning algorithm can be seen in Fig. 14, shown for an episodic problem.

Algorithm 7: Q-Learning control algorithm

Initialize: $Q(s,a)$ arbitrary

repeat

  Initialize s;

  repeat

    choose a for state $\boldsymbol{s}$ using policy derived from Q $(\epsilon - greedy)$;

    Take action a; observe reward r; and next state $\boldsymbol{s}'$;

    $\delta_t = r + \gamma \max_{a' \in \mathcal{A}} Q(\boldsymbol{s}', a') - Q(\boldsymbol{s}, a)$ $Q(\boldsymbol{s}, a) = Q(\boldsymbol{s}, a) + \eta \delta_t$;

    $\boldsymbol{s} \leftarrow \boldsymbol{s}'$;

  until $\boldsymbol{s}$ *is terminal*;

until *for each episode*;

Fig 14 – Q_Learning algorithm

It uses a table, called Q table, of size $\mathcal{S}$ x $\mathcal{A}$, that stores the value of Q for each pair state-action. At the start the Q table is filled in randomly and then it is updated repeatedly using (4.5) for each entry of the table.

At each time step the policy followed by the agent is the ϵ-greedy policy. With a probability ϵ it will perform a random action. This is called exploration, and will be treated further on. Else, the agent follows the greedy policy: $\pi(s) = \max_a Q(s,a)$ deduced from the Q table.

At the end, the Q table's entries will converge towards their optimal value Q*. This means that the greedy policy π will converge towards the optimal policy π*, and the goal of the algorithm has been accomplished.

4.9. Dimension problem

The Q-Learning algorithm suits the problem faced in this project rather well, but there is one huge inconvenience: the size of the spaces $\mathcal{S}$. There are 146880 possible states (see 4.2). This means that the Q table would be of size 146880 x 5 = 734400 entries. $\mathcal{S}$ is too big for the computation of a Q table to be feasible.

The most common solution for this dimensional problem is to use a function approximator that approximates the Q function rather than calculating all the values of the Q table explicitly. The most common approximators used in these cases are neural networks. Using neural networks with Q-Learning is called Deep Q-Learning (DQN). It is important to note that Q-learning in its tabular approach never tries to estimate the Q function, being content with calculating its individual values. DQN tries to estimate the explicit definition of the function depending on s and a, and then apply this definition for every pair s-a during the interaction with the environment.

5

Neural Networks

An introduction to neural networks, their different structures and the method of gradient descent

A typical neural network uses a certain number of neurons (depending on the complexity of the task) that are grouped in layers, with each layer connected to one another. Neurons allow the model to compute data from the input and produce an output, thanks to their activation. Neurons activate based on how the neurons of the previous layers activated and pass their activation on to the neurons of the next layer. The neurons take as input the weighted sum of the neurons from the previous layer and pass it on through their activation function, which results in the neuron's activation. The most popular activation function is the ReLU as it improves stability of the model and allows it to perform non-linear operations on the input.

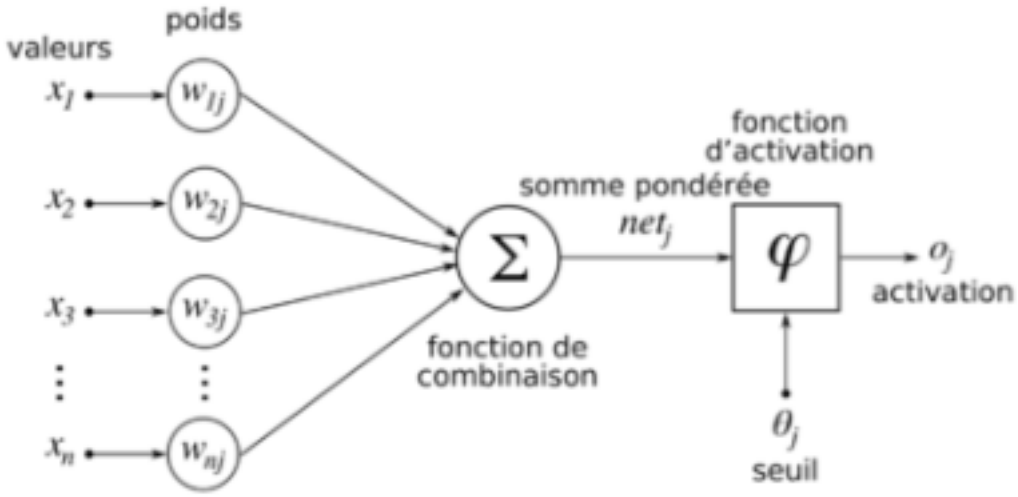


Fig. 15 – Principle of a neural network

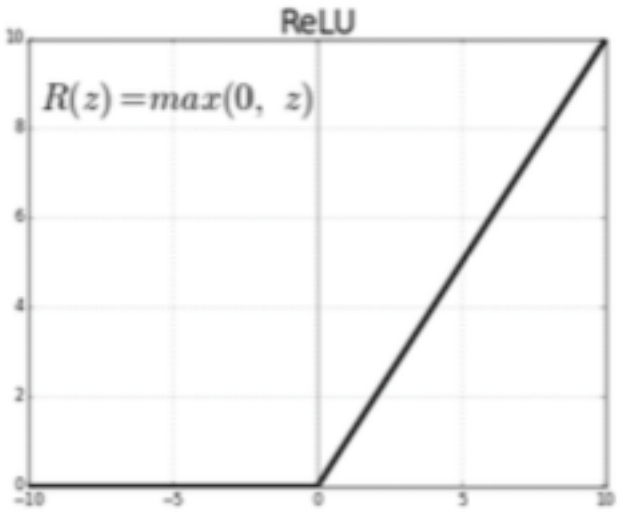


Fig. 16 – ReLu function

Then there are multiple ways to arrange the neurons in layers. The most intuitive one is to put several layers sequentially and connect all the neurons from one layer to all the neurons of the next one. This kind of layer is called a fully-connected layer, or dense layer. As it is a simple architecture it can adapt to several kinds of problems, but it is not always the best solution as it is quite computation hungry.

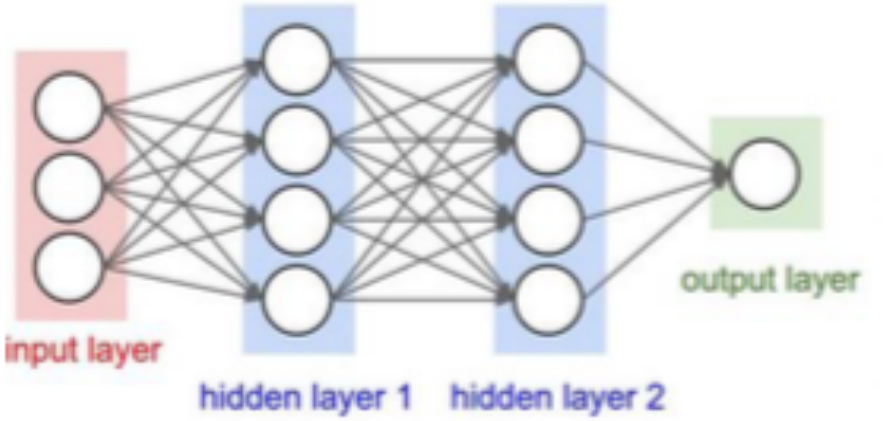


Fig 17 – Fully connected structure

Another way to arrange neurons in layers is to use convolutional layers. These layers use a filter (which is an array of a few neurons) and group the input neurons. Then they apply the filter to each group and it results in an output of one neuron per group. This allows the neural network to use less neurons to compute the same input size and handle much more complex input data, but this kind of layer is harder to tune in order to obtain good results. Also, convolutional layers help the model to detect local features in the image, as it applies the filter on groups of neurons. Convolutional layers can therefore be very useful for image related problems, even though they are quite hard to tune.

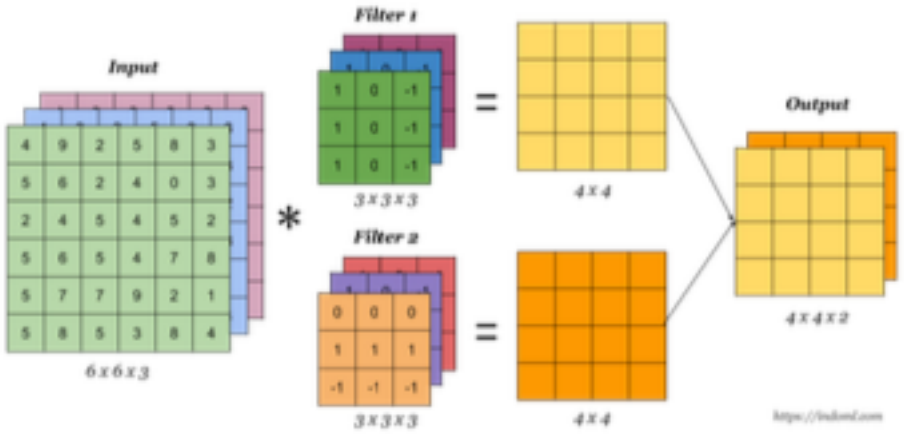


Fig 18 – Diagram of a convolutional layer

Finally, it is sometimes useful to artificially extract the most relevant neurons from the rest. This can be done using a pooling layer, which for instance selects the most activated neuron, or the average activation. This kind of layer works well with convolutional layers.

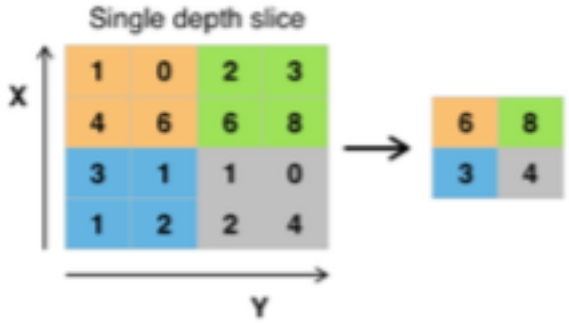


Fig 19 – Pooling layer

In the end, a full deep Q-learning can look like the one in Fig. 20. It has several convolutional and max- pooling layers and a few fully connected layers at the end. This kind of network is often used for image classification problems.

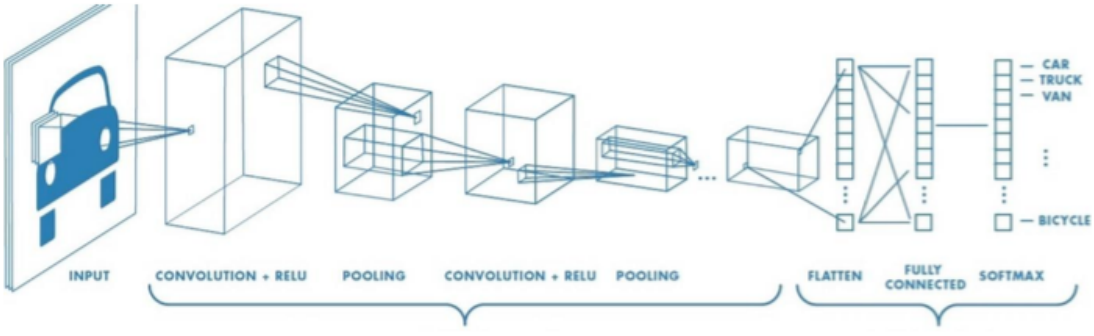


Fig 20 – Example of neural network

Once the architecture of the neural network is set, it is time to train it to achieve the best possible reward. To train the network, the weights on the connection between the neurons (the network's parameters) are updated at each time step. In Q-learning the

algorithm learned by updating the Q table after each action. In DQN the algorithm learns by updating the weights of the neural network. The process by which the weights are updated is called gradient descent.

The general idea of optimizing a function through gradient descent is to set the variables to a random value and then advance in the direction of the gradient to reach the local minimum. To train a neural network this way, the network's parameters are set at random and at each time step the gradient of a function (generally a loss function) will be calculated and each parameter will be updated using the equation (5.1). F is the function that is being optimized.

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial f(\theta)}{\partial \theta_j} \tag{5.1}$$

The parameter α is called the step size. However, it is also usually called learning rate (see 4.8), even if the concept is not exactly the same. In chapter 6, a parameter called learning rate will be discussed. This parameter is none other than this step size, not the learning rate in the Q-learning sense. If the step size is too small the parameter will take a long time to reach the minimum. If the step size is too big the minimum can be passed by and the algorithm risks to diverge.

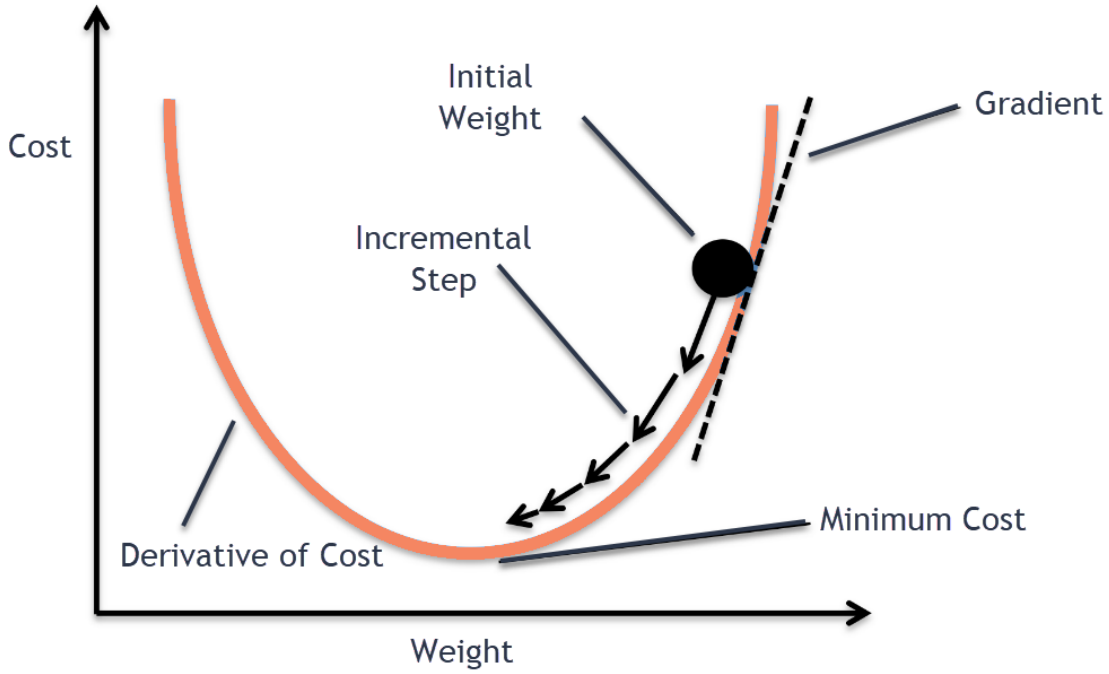


Fig 21 – Diagram of gradient descent

An important concept in gradient descent applied to neural networks is forward-propagation and back-propagation. The gradient of a scalar function that depends on parameters intertwined in layers of a neural network is not easy to calculate. The gradient is calculated using backpropagation. When an operation is performed in the neural network, its inputs and its output are stored. This operations form a sort of pyramid with the parameters at the base and the value of the function F at the top. If N operations O have been made, O_N will have F as output and some variables x, y and z as input, for example. Variable x will come from another operation O_j that will have variables a, b, c, d and e as inputs. And so on until the operations at the base, which have the parameters as input. To calculate the gradient of F in function of the

parameters, the gradient of each operation O_j in function of its inputs will be calculated, starting by the last operations. A sort of chain reaction is set, calculating the gradients of each level of the pyramid of operations until reaching the last level. Then, the gradient of F in function of a parameter θ_i is calculated as the multiplication of all the gradients in the chain that links the parameter to F. This is called backpropragation. A simple example of backprogration can be seen in Fig. 22.

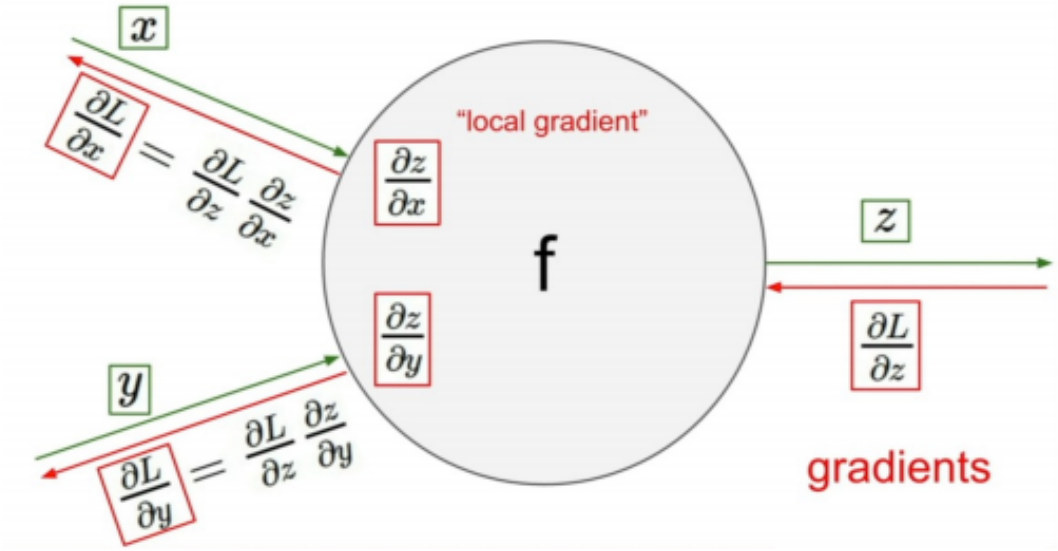


Fig. 22 – Example of backpropagation

6

The algorithm

This chapter presents the algorithms in its final version and explains how it works. The hyperparameters are key parts of the algorithm. There is a section dedicated to the explanation of how each hyperparameter has been chosen and tuned

The theory behind the algorithm has been explained. Like it has been stated before, the chosen algorithm is a Deep Q-Learning algorithm with a continuous state space but a discrete action space, but there are some variations from the algorithms that can be found in the literature [VOLO15] [7]. In this chapter we will present the final version of the algorithm.

But before that, there is something that needs to be defined, that is not rigorously part of the algorithm, but central to the problem nonetheless: the reward.

6.1. Reward function

As explained in chapter 4, the reward is a key element of any reinforcement learning problem. Essentially, in a reinforcement learning algorithm the agent learns to maximize the reward. The objective is to have an agent that learns to do some task, so maximizing the reward should lead to accomplish the task. If this is the case, the reward will be well built. There is normally one rule when building reward functions [RICH17]: the function should reflect **what** the agent has to do, not **how** to do it. For example, if the agent has to play chess, the agent should be rewarded only when winning, not when it takes an opponent's piece, for example. If the agent is rewarded for subgoals like those, it might learn maximize those subgoals, even if the final objective is not met. For example, it might take a lot of the opponent's pieces but lose the game

So, let us consider the task that the robot has to accomplish. The robot's objective is to walk the arena until it sees a wall, and then turn right or left to avoid it. In the reinforcement learning literature it is commonly accepted that for making a robot walk, the robot should be rewarded at each time step by its forward motion [RICH17]. This makes sense because when walking, people don't change sense every other second, but

go straightforward until they reach their destination. So one part of the reward takes into account the forward motion, or in other words, the robot's speed projected into the forward direction. At each time step the robot will be rewarded by the percentage of its speed that is in the forward direction. In the simulator this translates to a reward of 100 when the robot goes forward, a reward of 50 when the robot turns right or left and a reward of 0 when it is blocked or goes backward. In the real robot it is not exactly like that, because the real robot has inertia, so the values are not so regular. But they are close enough.

Unfortunately, this reward function cannot accomplish the task. Let us consider two different scenarios. Scenario 1: the robot crosses the arena going forward, sees the wall a time step in advance and turns. Scenario 2: the robot crosses the arena going forward, crashes into the wall, turns backward and goes on. Let us say that the robot takes 6 time steps from one wall to another.

The reward for the scenario 1 will be: 5x100 + 50 = 550 in 6 time steps. The reward for the scenario 2 will be: 6x100 + 0 + 0 = 700 in 8 time steps. That makes a mean of 91,6 points of reward par action in the scenario 1 and 87,5 point of reward by action in the scenario 2. The reward for the scenario 1 is only 4% bigger than for the scenario 2. However, the scenario 1 describes the perfect behavior, and the scenario 2 a wrong behavior. The difference is too small for the robot to learn the scenario 1, rather than scenario 2, so some modification to the reward function must be made.

This modification consists in punishing the robot when it hits a wall. When the robot is blocked against a wall, its speed will be 0 (and the speed its 0 only when the robot hits a wall). So if the robot's total speed is 0, it receives a reward of -100. To completely punish the behavior of going into a wall, pull backward and keep on going, a punition of -50 is also set for every time the robot goes backward. This can interfere with the process of learning how to clear away from a wall, but not so much, because staying blocked against the wall is clearly more harmful.

Let us redo the calculation of the reward for scenario 1 and 2. Scenario 1: 550 in 6 time steps. Scenario 2: 6x100 – 100 – 50 = 450 in 8 time steps. Scenario 1: 91,6 reward points/action. Scenario 2: 56,25 reward points/action. This means a 63% difference between the two scenarios. With this reward the robot can learn.

Let us name r_t the reward in the time step t, **v** the robot's total speed (vector) and **u** the direction perpendicular to the axe formed by the two wheels. Formally written the reward function is:

$$r_t = \begin{cases} \boldsymbol{v} \cdot \boldsymbol{u} & if\ \boldsymbol{v} \cdot \boldsymbol{u} > 0 \\ -100 & if\ \boldsymbol{v} \cdot \boldsymbol{u} = 0 \\ -50 & if\ \boldsymbol{v} \cdot \boldsymbol{u} < 0 \end{cases} \tag{6.1}$$

5.2. The algorithm

The pseudo-code for the algorithm is the following (the function Optimize has been separated for clarity's sake):

Algorithm Adapted DQN for Learning Robot

Ask camera for resolution #only for real robot
Initialize memory buffer $\mathcal{D}$ to capacity N
Initialize γ, n, ϵ and environment
Initialize policy neural network Q with random weights θ
Initialize C and target neural network Q' with random weights θ'
Initialize optimizer and learning rate lr
Initialize state $x_1 = \{s_1, m_1\}$
for $t = 1, T$ **do**
    Force an action a_t else
        With probability ϵ select a random action a_t else select $a_t = \max_a Q^{\theta}(x_t, a)$
    Execute action a_t in the environment and wait for response
    while not response
        Optimize
    Observe state $x_{t+1} = \{s_{t+1}, m_{t+1}\}$ and reward r_t
    Store transition (x_t, a_t, r_t, x_{t+1}) in $\mathcal{D}$
    Plot reward and loss
    $x_t \leftarrow x_{t+1}$
    Every C steps do $\theta' \leftarrow \theta$
end for

Function Optimize

Sample random batch of n transitions (x_j, a_j, r_j, x_{j+1}) in $\mathcal{D}$
Set prediction $y_j = Q(x_j, a_j; \theta)$
Set target $z_j = \gamma \max_{a'} Q'(x_{j+1}, a'; \theta') + r_j$
$\delta = y_j - z_j$
Calculate loss $\mathcal{L}(\delta)$
Perform a step of the optimizer on $\mathcal{L}(\delta)$
Clamp parameters θ

The algorithm is based in a for loop that goes on until the experience is finished. In this for loop it is easy to recognize the global pattern of action-observation presented in Fig 12.

Let us address first the initializations to introduce all of the elements of the algorithm. The first thing the algorithm needs to know is the resolution of the camera (it can be 4x3, 16x12, 32x24 depending on the camera's setup a). This is a mere question of calculating the size of the input to build consequently the neural network.

A very important feature of this algorithm is that it includes experience replay. This can be assimilated to memory in a human brain. Memory is a very important part of any learning process. When taking a trial and error approach, remembering which actions

worked and which did not is key. For a machine, memory is assimilated by a buffer that stores what happened at each time step (what we call transitions). The transitions stored in the buffer will be used in the optimization phase to improve its efficacy. The buffer $\mathcal{D}$ is a circular buffer with a capacity N. That means that when N transitions are stored in the buffer, the first transition will be replaced by the transition N+1. This saves storage space in the computer.

Then some parameters are initialized. First, gamma (see 4.4). The next parameter is n, the batch size. When optimizing with experience replay, the algorithm samples a random batch of transitions from the buffer and performs the optimization using the whole batch. The parameter n is the size of this batch. Epsilon is the exploration, which will be addressed further on. The environment is also initialized.

The next step is to initialize the neural network that we will be training. The robot learns by changing the weights of the connection between neurons in its neural network (see chapter 5). The neural network is the brain of the robot. The structure of the neural network needs to be built beforehand and it is an important hyperparameter, more about that later. We initialize the weights θ to random values.

In this algorithm a second neural network is needed to increase stability. The algorithm derives the term loss by the parameters θ of the training neural network (see gradient descent in chapter 5). However, the term loss has the Q function in both terms of the difference (because the Bellman equation is recurrent). If the same neural network were used to calculate both terms, a problem would raise up. At each time step, when deriving the loss, there would be a choice between diminishing the prediction term and increasing the target term (it is more complicated than that, but intuitively it comes to that). This clearly disturbs learning, because it can make the parameters go back and force depending on which of the two terms is derived. To avoid this, a second neural network is used. This neural network (called target net) is never trained and it serves only to calculate the target term of the loss. Anyhow, the target network needs to resemble the training network to be able to correctly approximate the Q function. So after C steps it has to be updated with the weights of the training network.

Then two more hyperparameters are initialized: the optimizer and the learning rate. There are a lot of ways of updating the parameters of the neural network to minimize the term loss, usually variations of the method of gradient descent. The optimizer sets the method that will be used for said minimization. The learning rate itself is a parameter of the optimizer, and its meaning can slightly change from one optimizer to another, but it is globally the learning rate that we have introduced in chapter 5.

One last clarification before going into the actual loop is necessary. To improve the robot's efficacy when blocked against a wall an extra term has been added to the state (besides the image from the camera who is represented by the variable s), the variable m. It is a Boolean variable that takes the value 0 when the robot is blocked and 1 when the robot is moving. This input variable is connected directly to the last layer of the neural network, so its value is not drowned in the layers.

The first thing the agent does at each time step t is to choose an action. There is the possibility of forcing the agent to take any specific action via the graphic interface. Forcing the action is interesting because it can speed up the learning process and it allows a bigger interaction between the user and the agent, which is rather useful when demonstrating to kids. However, for the most part the actions will not be forced on the agent. Instead, the agent faces a choice: exploring with a probability ϵ or follow the greedy policy (the policy that maximizes Q).

Exploration is a vital part in any learning process. If the robot was always restrained to do the actions supplied by the neural network, it could learn a bad policy. For example, it could start by turning right and receive a positive reward, which is better than a null reward, so the logical option would be to continue to turn right. This is not the optimal policy for the problem treated, which has as objective to maximize the forward speed. The robot needs to explore, so it will learn new possibilities of the environment and achieve a better performance.

When the choice of the action has been made the algorithm sends this information to the environment and waits. While it waits for the new data (image and reward), it performs as many optimization steps as it can. The real robot has inertia and takes time to perform the actions, so it works in an established loop of two actions per second. At each half-second it asks the computer what action it should do next, then for the next half second it does this action, and right before asking what the next action should be it sends the data. Meanwhile, the algorithm optimizes and when it receives the new data, it decides the new action and so on. This allows a lot more optimizations per time step, which is really beneficial because the learning time is reduced significantly. However, this depends on the computational power of the computer running the algorithm. In a fixed PC, up to 25 optimizations per time step can be done, but my computer can hardly do more than 2.

For each optimization step a batch of states is sampled from the memory buffer. In regular DQN algorithms, the optimization is done only with the following state. The neural network predicts the value of the function Q for state s_{t+1} and chosen action a_t. Then the target network calculates the target term as the reward r_t plus the value of the function Q for the state s_{t+1} if the greedy policy was followed. In that case, $\delta \in \mathbb{R}$ and $\mathcal{L}$: $\mathbb{R} \rightarrow \mathbb{R}$. When using experience replay, both the prediction term y and the target term z are vectors of size n (n is the batch's size). Each coordinate of both vectors is calculated as it was above. Therefore, $\delta \in \mathbb{R}^n$ and $\mathcal{L}$: $\mathbb{R}^n \rightarrow \mathbb{R}^n$.

Let us take an action that happened in a time t. In time t, the agent's network evaluated the state and the possible actions and chose to execute the action a (let us suppose that it did not explore). In a time t' > t, the action is selected randomly for optimization (along with many others). The value of $Q(s_t, a_t)$ is calculated with the state and action of time t, but the parameters θ of the neural network are those of time t'. They could be really different, from those of time t. It is possible for the action a_t to not be the optimal one anymore when calculated with the parameters of time t'. Actions that could be considered optimal at some point are then revisited and a new optimal behavior is set for the same situation. This is the power of the experience replay, the possibility to revisit past situations. For this reason it is useful to optimize several time while waiting for the data, even if a priori there is no new action so there would be nothing to learn. When it optimizes several times the agent revisits past experiences with the information that gives the latest reward, actualizing a larger number of behaviors, thus reducing the learning time steps because the weights are more actualized.

Once δ has been calculated, the term loss $\mathcal{L}(\delta)$ is computed. Normally the mean squared loss is used:

$$\mathcal{L}(\delta) = \frac{1}{n}\sum \delta^2 \tag{5.2}$$

Another possibility is using the Huber loss, which is the mean squared error when δ is small and just the absolute mean of δ when it is high. This can increase stability at the start when the loss is high, preventing it from diverging.

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{n}\sum \delta^2 & if\ |\delta| \leq 1 \\ \frac{1}{n}\sum |\delta| & else \end{cases} \tag{5.3}$$

Then, a step of the optimizer is performed. The optimizer changes the weights θ to minimize the loss function. There are several methods of optimization, but in this project only two have been tested. Both are variants of the gradient descent that has been explained in chapter 5.

Once the backpropagation of gradients has been made in the parameters of the neural network, the parameters are clamped. This means that any parameters with values over 1 (or under -1), are set to 1 (or -1) instead. This helps stability because parameters with too great values diverge more easily.

Once the robot sends the data, the optimizations stops and the algorithm observes the new state s_{t+1} and the reward r_t, it stores it in a transition, plots the reward and loss in a graph for interpreting results, and it sets the state s_{t+1} to s_t to restart the loop.

In this explanation of the algorithm a lot of parameters have been mentioned. It can be confusing, so it is important to remember that the parameters θ that make up the connections between the neurons in the neural network are what we call "parameters" and the network learns by actualizing them. These are different from "hyperparameters" that are parameters of the algorithm set by the programmer, which need to be tuned before the experience for the program to work. Hyperparameters are the tools the programmer has to make work an algorithm that does not work or to increase efficacy. The next section will go on through the tuning of hyperparameters.

6.3. Hyperparameter tuning

First, let us recapitulate briefly the hyperparameters that need tuning:

- **Buffer's capacity N**
- **Gamma γ**
- **Batch size n**
- **Exploration**
- **Neural network's structure**
- **Target update C**
- **Optimizer**
- **Learning rate lr**

All of the hyperparameters will be discussed, and there will be graphs showing the performance of the algorithm with different values of the same hyperparameters. Therefore, some precisions about the graphs need to be done, to be able to conveniently analyze them.

The hyperparameters have been tuned using the simulator explained in 2.4 and more particularly in the configuration TopCameraLocalOri.

Each hyperparameter is treated independently, so all of the other hyperparameters are set to their optimal value. When actually tuning the parameters this cannot be done, because the optimal value of the parameters is rarely known beforehand. In the actual tuning of the hyperparameters what was done was to change the parameter and if the behavior was better, keep the new value, else keep the precedent one. Doing this for all of the parameters is a long process, because judging the behavior is not always easy, so reaching an optimal set of parameters was long and required a lot of complicated reasoning.

In the graphs, three curves are shown. One of the curves represents the instant loss, the term loss at each moment. This allows judging if the algorithm converges or oscillates and if there are any peaks of loss. A second curve represents the average loss over the last 240 actions, which gives a good image of the convergence of the algorithm towards an adequate set of parameters. The third curve shows the mean reward. This is the most important curve, because it shows the level of the agent. We will see if it converges or not and the final level acquired by the agent. All of the experiences last 1500 actions, or 12 minutes and a half, which is time enough for the algorithm to converge in the simulator (it usually converges in about 5 minutes if the hyperparameters are well set).

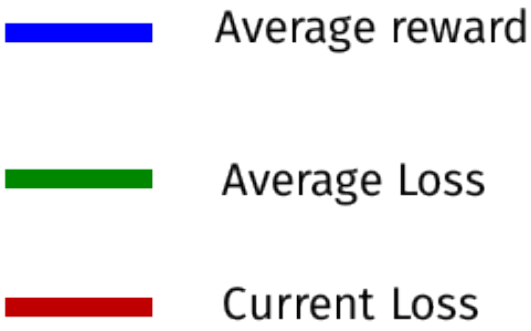


Fig 23 – Graph legend

6.3.1. Buffer'capacity N

The past experiences of the robot are stored in a buffer to be used in the optimization. This buffer is circular to save storage space. However, it is important to choose rightly its capacity N. If N were too small, it would be like teaching a fish. The agent would forget its experiences too fast for the experience replay to help. The agent could still learn but the learning time would be huge and the risk of divergence far greater. If N were too big the agent learns well but the storage space allocated to the buffer would be big, and this is a waste of space, because similar performances can be achieved with a smaller capacity.

Three graphs depicting experiences with values of N = 50, 1000 and 10000000 are shown next.

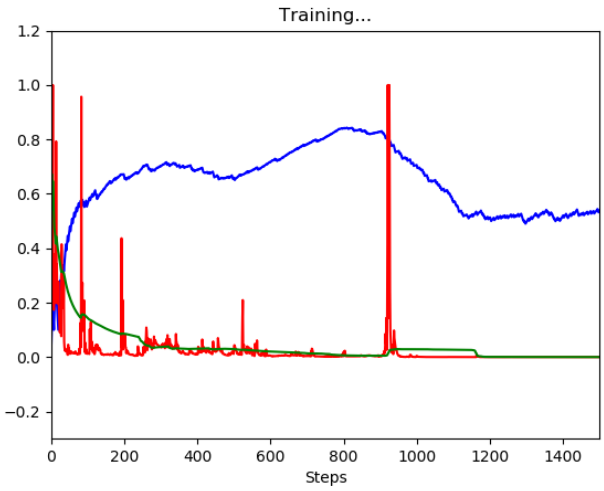


Fig 24 – Hyperparameter tuning: N=50

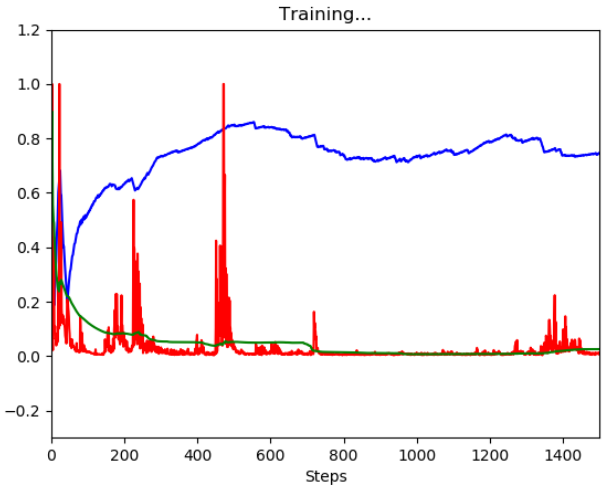


Fig 25 – Hyperparameter tuning: N=1000

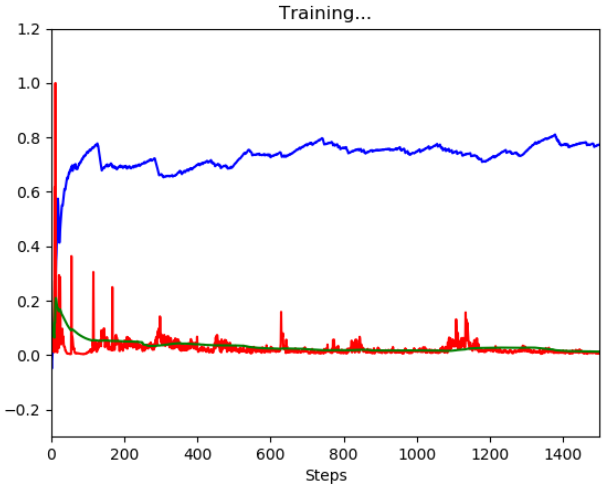


Fig.26 – Hyperparameter tuning: N=10000000

For a small N, the algorithm is stable but is converges towards a small value of reward (level around 0,6). The behavior of the agent in this case is to go forward until hitting a wall and then turning. The memory is too small for the agent to learn to avoid

walls. For greater values of N, the results are good, without much difference between N=1000 and N=10000000, both of them have a nice comportment and a high final level (around 0,8).

The chosen value for N is 1000, to save storage space.

6.3.2. Gamma γ

The hyperparameter gamma, also called the discount factor, has already been introduced. It measures the "farsightedness" of the agent. The return (equation 4.1) includes a factor γ^t. If γ is close to 0, its t power will be smaller and most of the weight in the Bellman equation will be given to the current reward. If γ is close to 1, the weight of futures reward will be higher. However, a too great value of γ can avoid convergence, because the agent will try to maximize long-future rewards from predictions that may not be correct and that may change from one time step to the next, so the algorithm might oscillate.

Let us try to calculate approximately a good value for γ in our problem. In this particular problem, an horizon of five-six actions seems correct, as the agent needs more or less 4 forward actions to go from one point of the arena to the other. Avoiding a wall will diminish the reward in the next 2 actions and will get positive results in the 4th or 5th action. To maintain the 6th action's reward higher than 20% of its real value, a gamma of 0,76 is needed. Therefore, 0,8 seems a good value for gamma.

Let us see graphs for different values of γ:

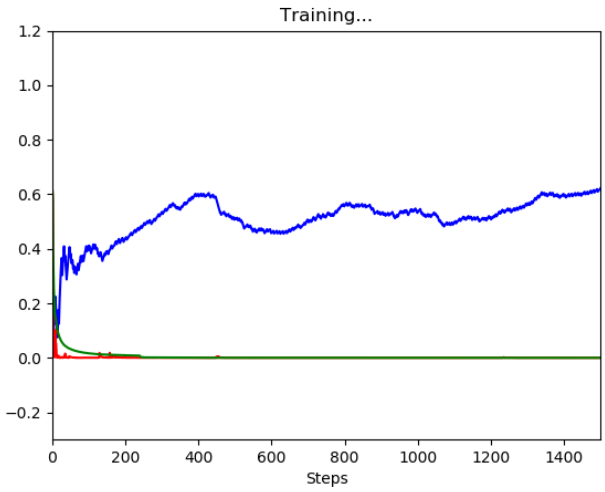


Fig 27 – Hyperparameter tuning: γ = 0.1

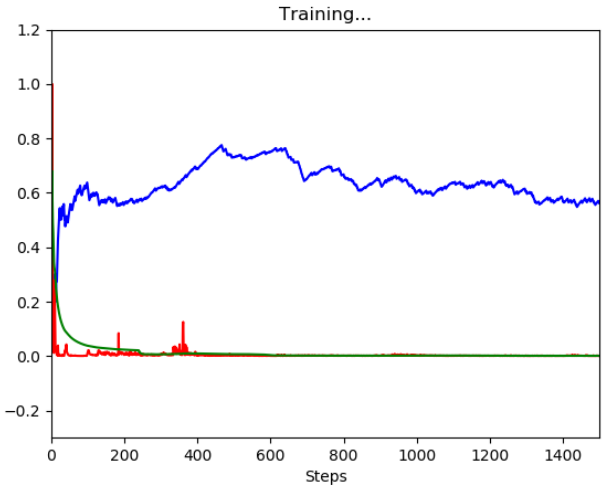


Fig 29 – Hyperparameter tuning: γ = 0.5

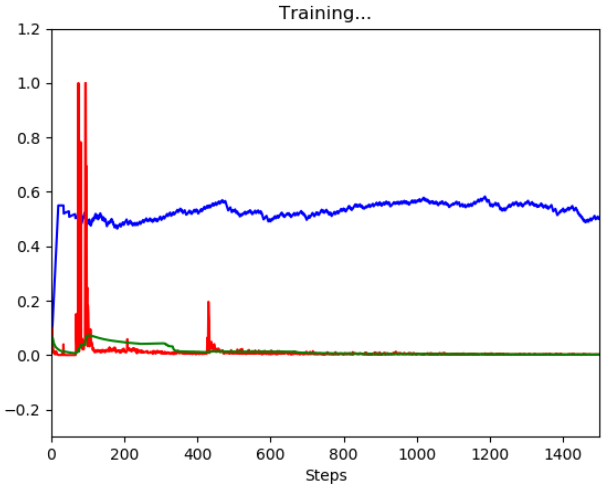


Fig 30 – Hyperparameter tuning: γ = 0.7

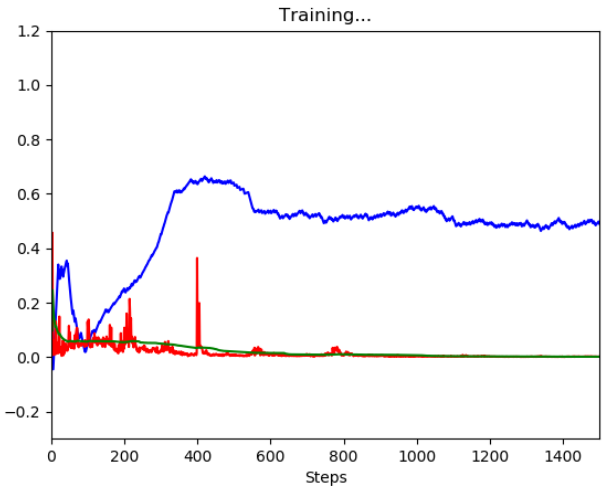


Fig. 31 – Hyperparameter tuning: γ = 0.76

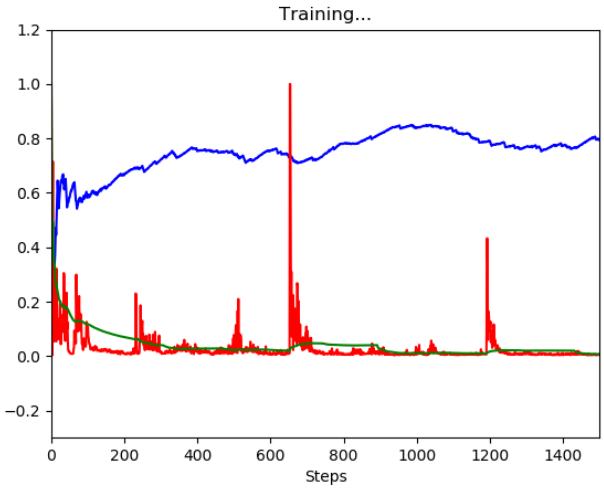


Fig. 32 – Hyperparameter tuning: γ = 0.8

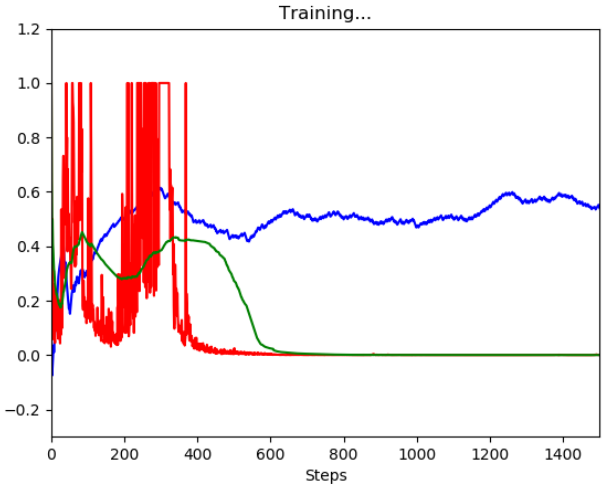


Fig. 33 – Hyperparamter tuning γ = 0.9

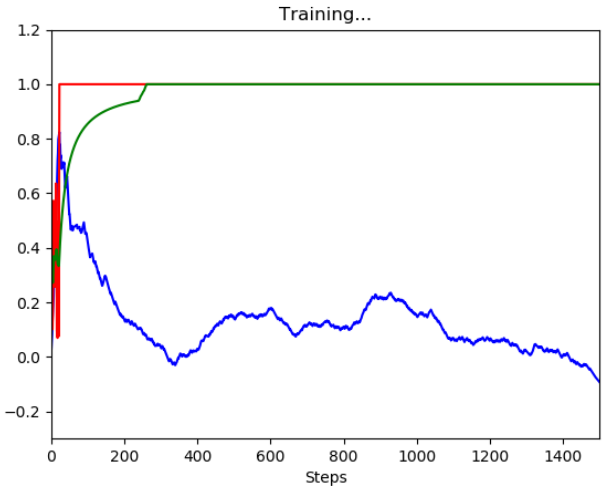


Fig 34 – Hyperparameter tuning γ = 0.99

In small values of γ (0.1, 0.5 and 0.7) the agent is clearly short-sighted and learns quickly to go straightforward but it is never able to learn how to avoid walls. This is shown by an almost immediate convergence of the loss with very little peaks (peaks are

usually formed when the robot tries to avoid a wall and fails, the absence of peaks in this case shows that the robot never tried to avoid the wall). The reward converges towards a level of 0.6, which is not a good level, because a robot that avoids correctly the walls has a level between 0.7-0.8.

The value of γ calculated a priori (0.76) is not good either, but $\gamma = 0.8$ is. In this case the loss converges more slowly and the immediate loss is noisier, but the reward converges towards a level of 0.8, which is really good.

When trying higher values of gamma, we can observe that the loss can easily diverge. In Fig. 33 it threatens to diverge and finally converges around time step 400 but it converges to a reward level of 0.6, which indicates that the agent has tried to learn a better policy than hitting the wall and going backwards but has failed because it has tried to take into account rewards that are too far off. In Fig. 34, the robot does not even learn, it is too difficult to maximize all of the future rewards so the loss diverges instantly.

The optimal value of γ is therefore around 0.8, and it will be taken so for all the other experiences.

6.3.3. Batch-size n

As explained in 6.2, the optimization is actually made over a batch of past experiences rather than from the present experience. This allows the agent to revisit past experiences more often, reaching a better optimal policy. Therefore, the more often the actions are visited, the better the agent will learn. A past experience is revisited with a probability n/N, so the bigger n is, the more often the actions will be revisited (N could be made smaller as well but information would be lost, so there would be no benefit). In the limit where n=N, all the actions will be revisited at each time step. This would mean a perfect convergence.

In practice however, this is not like that. If n is too big, the computing power needed to calculate Q for every past action is far too high. In the simulator (where the robot is compelled to wait for the optimization to finish) the experience will just be very long. But in the real robot, the robot will continue to do the same action for the time it would take to do 3 or 4 actions. Each turning action will turn out to be a tour around itself instead and at each forward action the robot will keep going until it hits a wall.

For too small values of n the robot will not be able to learn correctly, because no experience replay will be done, and the experience replay counteracts essentially the reward's sparseness that is inherent to the problem.

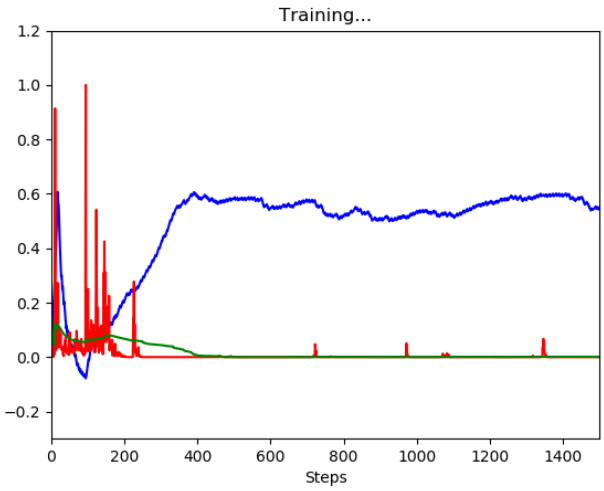


Fig 35 – Hyperparameter tuning: n = 10

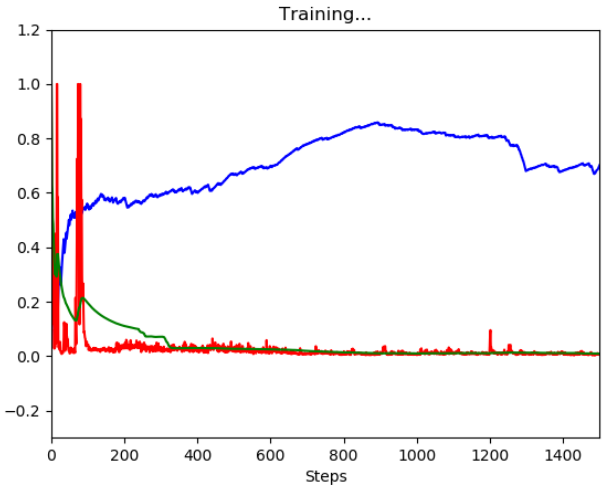


Fig 36 – Hyperparameter tuning: n = 100

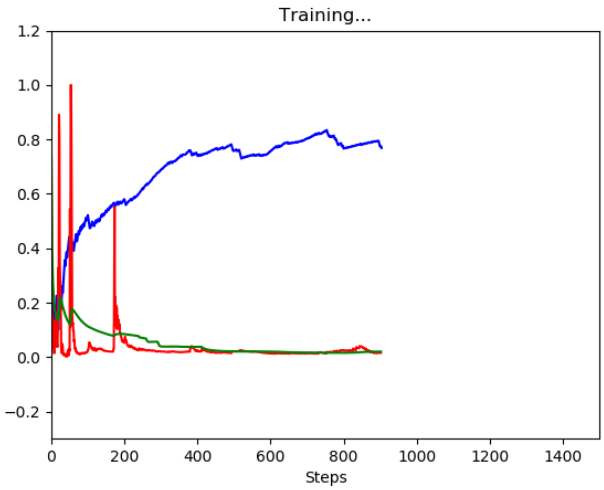


Fig 37 – Hyperparameter tuning: n = 1000

As expected, the agent learns poorly for a small value of n, not being able to learn the best policy and only achieving a reward level of 0.6. For a batch size of around 100 actions the agent learns the correct behavior. For a batch size on the order of the

buffer's capacity the experience becomes too long in the simulator. Even if the behavior is optimal, the program ran thrice as long and executed only a half of the experience.

6.3.4. Exploration ϵ

Following what was said in 6.2, exploration is needed for the robot to achieve the correct behavior, else it would likely not visit all the state-action space. The exploration is set in place with the hyperparameter ϵ. Epsilon is the exploration threshold: a real number between 0 and 1 is randomly chosen and if it is below ϵ, then a random action will be performed. Else, the agent will follow its normal policy.

It is interesting to see what happens if ϵ is changed during the experience, defining a curve of exploration. For example, the idea of having a greater exploration at the start and then reducing it to close to null at the end can be appealing. During the project three exploration curves have been tried: an exponential curve, a piece-wise defined curve and constant values.

- **Exponential curve:** The value of epsilon is set at the start at EPS_START and it decreases exponentially to end at EPS_END with a decay of EPS_DECAY. The value of epsilon is calculated for each action depending on these 3 hyperparameters and the number of actions. This type of curve is widely used in the literature.
- **Piece-wise defined curve:** This is a function defined in three pieces. Two constant values joined by a first order linear function. The time steps at which the linear function starts and ends can be chosen, fixing like that its slope. The concept is to have an exploration that it's reduced gradually but keeping a constant value at the start, setting up a sort of exploration phase. The parameters used are: EPS_START for the first constant value and EPS_END for the last. EPS_LIMIT1 marks the start of the linear function, and EPS_LIMIT2 the start of the final constant value.
- **Constant values:** This is just setting a constant value that will not change during the experience. This solution is simple but has its problems. For example, if the value of epsilon is too big the algorithm will be highly unstable and it may not converge. If it is too small it may converge to a bad behavior for fault of exploring. Also, the maximum possible reward level will never be reached.

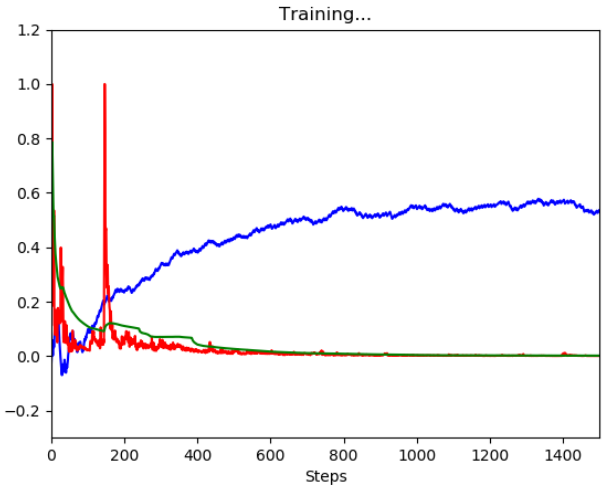


Fig 38 – Hyperparameter tuning: exponential exploration EPS_START = 0.9, EPS_END = 0.05, EPS_DECAY = 200

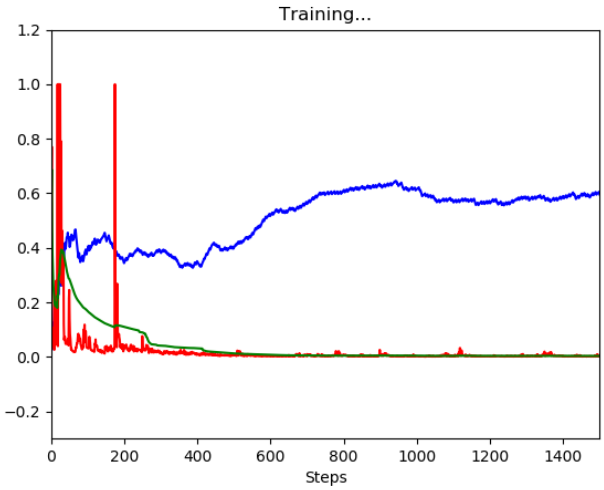


Fig 39 – Hyperparameter turning: linear exploration. EPS_START = 0.4; EPS_END = 0.05; EPS_LIMIT1 = 80; EPS_LIMIT2 = 640

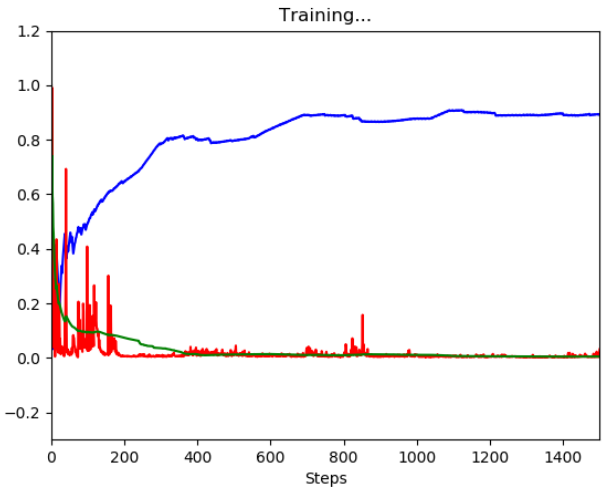


Fig 40 – Hyperparameter tuning: linear exploration. EPS_START = 0.1; EPS_END = 0.01; EPS_LIMIT1 = 200; EPS_LIMIT2 = 600

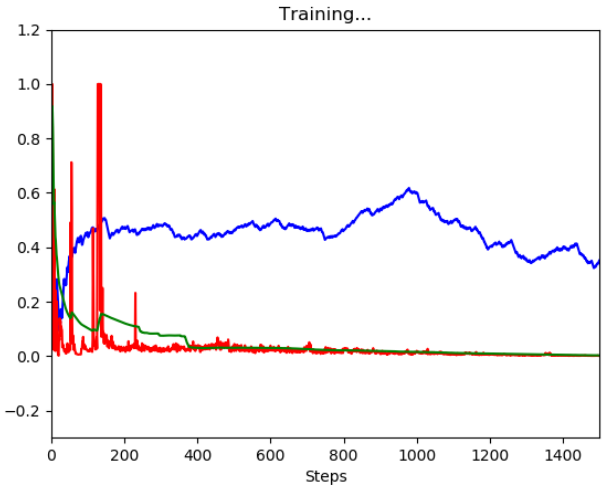


Fig 41 – Hyperparameter tuning: constant exploration. $\epsilon = 0.4$

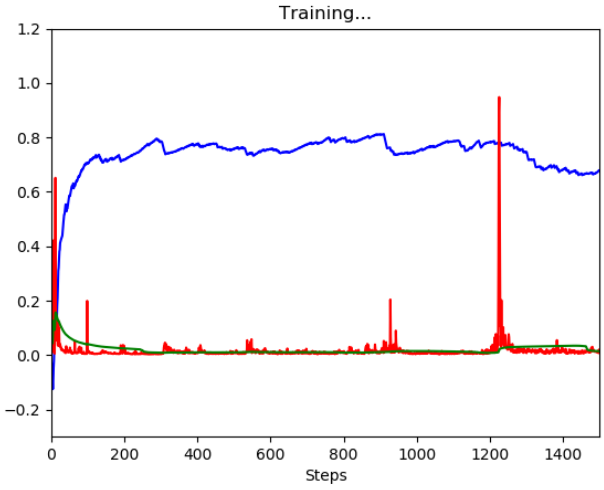


Fig 42 – Hyperparameter tuning: constant exploration. $\epsilon = 0.1$

Let us draw some conclusions. It is clear that too much exploration at the start is harmful to the agent in this environment; all the experiences with high starting values of epsilon fail to learn the correct behavior (figures 38, 39 and 41). However, experiences with smaller values of epsilon from the start converge more quickly and towards higher levels of reward (figures 40 and 42). It is safe to conclude then that an "exploration phase" at the start of the experience is not beneficial. This may be because the exploration disturbs the trial and error method too much for it to reach the optimal policy.

If the environment is reflected upon, three levels of behavior can be distinguished in this problem: 1) completely erratic; 2) going straightforward until hitting a wall and turning around; 3) going straightforward and turning before the wall. Passing from level 1 to level 2 is rather easy, very little exploration is needed; once the agent has gone straight forward once it will do it every time, because there is a high immediate reward for going forward that is easy to analyze. Passing from level 2 to level 3 is a lot harder. Sequences of 6 or more actions need to be analyzed and the reward level by both behaviors is not that different (0.6 for level 2 and 0.8 for level 3). Let us say the agent is analyzing two sequences of six actions, one from behavior 2 and one from behavior 3 to make a choice of action. Let us say behavior 2 has an ideal reward (when no exploration

is done) of 350 and behavior 3 of 550. If there was a lot of exploration, behavior 3's reward could decrease to 400 changing forward actions to turns or backward actions. Because behavior 2 is more erratic, it will be less affected by randomness. So, it would pass maybe to 300. It is a lot harder to tell the two behaviors apart from the point of vie of the agent. It would be easier to assume that they are the same behavior. At the base, exploration is still noise and noise fundamentally perturbs learning.

Another detail that can be noted is that in figure 40, around time step 600 (when exploration reaches 0.01) the reward increases to a level of 0.9 that had never been reached before. Obviously, when the algorithm has converged and it follows an optimal policy, if out of 5 actions one is chosen randomly it will probably not be the better one, so the reward will decrease. When an agent follows an optimal policy and there is no exploration it reaches the optimal value of the reward.

6.3.5. Neural network's structure

In chapter 5 two different types of layers for neural networks were introduced: dense layers and convolutional layers. Every network needs to have at some point a dense layer, so there are two possibilities: either the neural network we are going to use is made up uniquely of dense layers or else it contains at least a convolutional layer. A priori, a neural network made up uniquely of dense layers (let us call it fully-connected network) is more flexible but more computational hungry. Why is this? A fully-connected network can be assimilated to the part of the human brain that controls instincts: it receives an information from a sensor and it passes it through the neurons, then the spine until it reaches the muscles. For example, our finger touches something hot; the neuron of our finger is linked with a lot of other neurons and each connection has a weight; the signal of "hot" is passed through those neurons that have a high weight for temperature until it reaches the spine, who makes the hand move. In this process there is no analyzing, only reactions. It notices the value of an input at a certain place and associates this value to an action. This is computation hungry because it is a glutton approach; no "thinking" is done. In a convolutional layer there is analysis. A filter is passed over the image to extract hidden features. This type of neural network is closer to our reasoned thinking. Convolutional layers are very successful in image recognition, because to interpret an image reasoning is needed.

A priori, the robot will be treating images, so it might seem a convolutional layer can be useful. However, if the task is analyzed more finely it could be concluded that this is not the case. The robot will globally see two things: the floor and the wall. The floor is blue and the wall yellow. When it sees a lot of blue, it means the walls are far and he can go on. When it sees a lot of yellow, the walls are close so it must turn. This process of decision may be done more effectively using instincts; there is no need to analyze the images because no shapes need to be recognized, the colors itself are sufficient.

Anyway, both type of structures have been tested, focusing mainly in fully-connected networks and testing mostly different sizes. Obviously, the bigger the network the better it will be able to sort out information and the better it will perform. However for big fully-connected layers the computational need is very high, so this will be a clear restriction. Convolutional networks have also being tried: a network made of one convolutional layer and then a fully connected layer. In the figures' legends the number of layers will be noted like this: inputs/layer1/layer2/…/outputs.

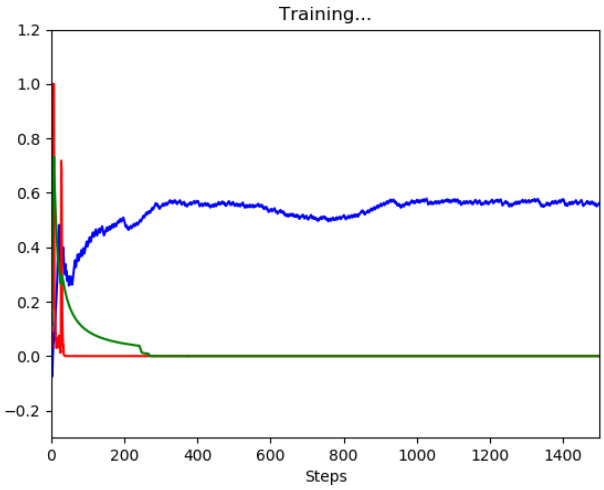


Fig 43 – Hyperparameter tuning: dense NN 3/50/5

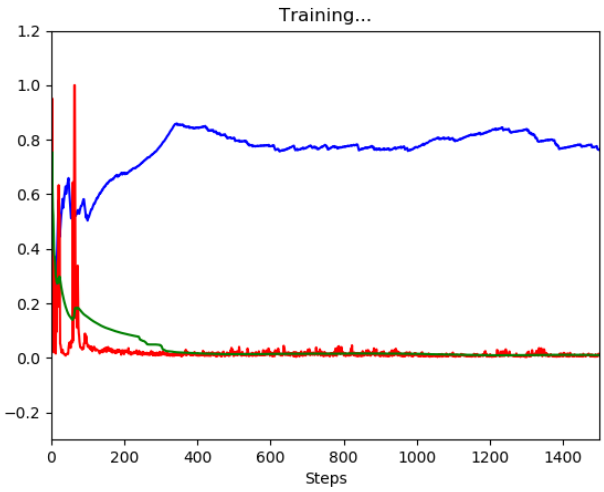


Fig 44 – Hyperparameter tuning: dense NN 3/150/100/50/5

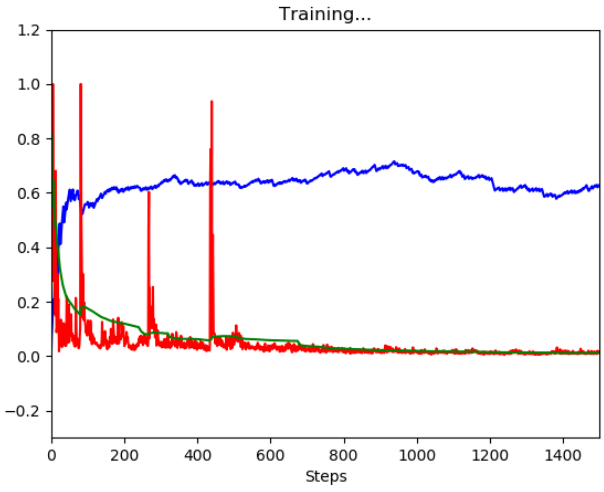


Fig 45 – Hyperparameter tuning: dense NN 3/500/300/200/100/5

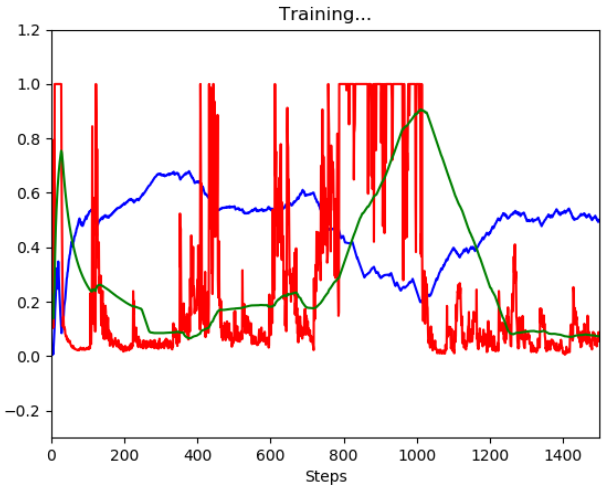


Fig 46 – Hyperparameter tuning: convolutional NN. Kernel = 3; Stride = 2; Padding = 1; Layers: 50

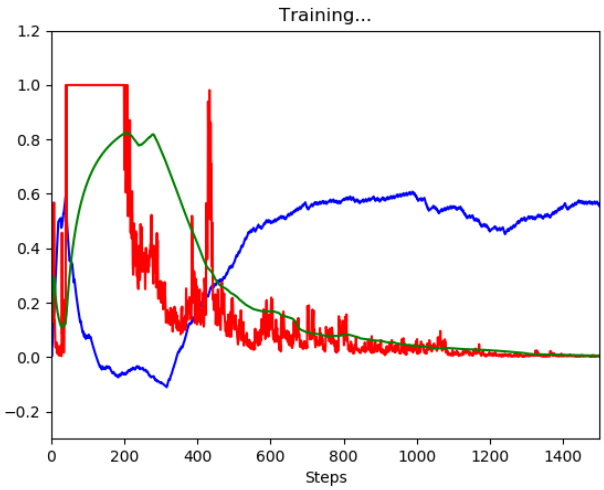


Fig 47 – Hyperparameter tuning: convolutional NN. Kernel = 5; Stride = 2; Padding = 1; Layers: 150/50

There are some conclusions that can be drawn from these results. Fully-connected networks are more stable in this particular problem than convolutional ones. Convolutional networks tends to diverge and even if they converge they are highly unstable. The choice of using fully-connected layers rather than convolutional is well founded then.

However, fully-connected layers do not always reach the optimal behavior. If the network is too small, the problem is too complicated for it to solve it and a simpler behavior is adopted. If the network is too big, its computational need is too high and only one optimization can be done each time step. This reduces the impact of the experience replay, thus reducing the agent's capacity to learn.

6.3.6. Target network's update C

Like it was explained in 6.2, this algorithm needs two different neural networks for it to be capable of correctly optimizing the loss function. There is one network called the policy network (or training network) whose parameters θ are always changing. This

is the network that learns. The second network called the target network, does not learn, but only exists to calculate the target term of the loss function during optimization. Its parameters do not change. But they are supposed to represent the same thing, so the target network needs to be updated frequently for it to keep pace with the policy network. The hyperparameter C regulates the number of steps between actualizations of the target network.

A priori, the smaller the value of C the better, because the two networks will be independent but equal, so they would represent the same thing. However, in practice this is not so. Gradient descent optimization is somewhat noisy, and in its search of the local minimum the parameters being modified can go in one direction, then go backwards a bit, then change directions yet again and so on. The target network is called like that because it calculates the target term of the loss function. As any target, it is easier to reach it if it does not move. If the target network is updated too frequently, the target term will be noisy as well and the learning will be unstable.

The value of C needs to be chosen with care, so that the target network is not completely different from the training network, but keeping a bit of distance for the algorithm to be stable. A too small value of C will compromise stability and a too high value of C can lead to erratic learning.

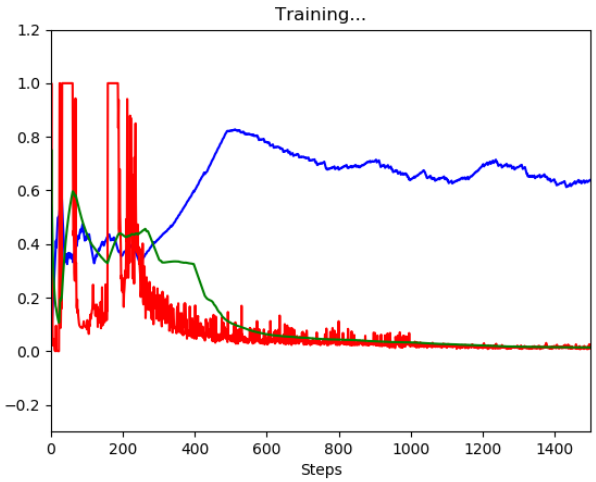


Fig 48 – Hyperparameter tuning: C = 1

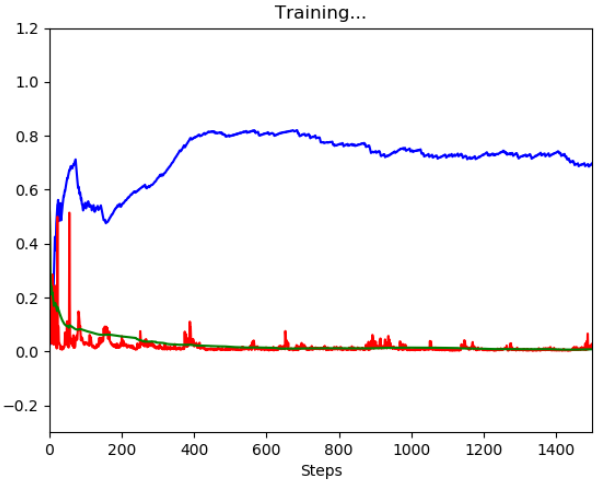


Fig 49 – Hyperparameter tuning: C = 10

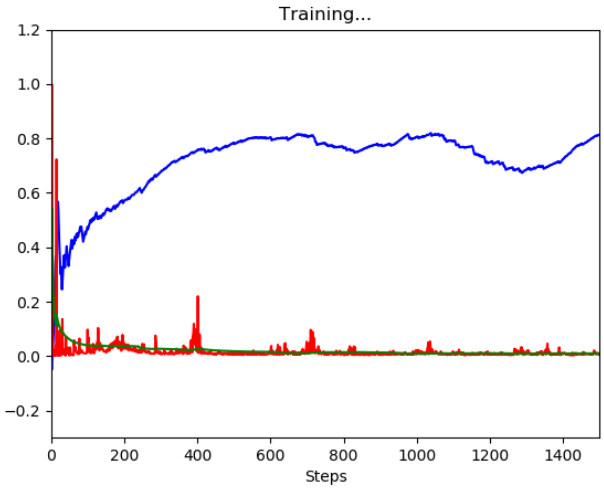


Fig 50 – Hyperparameter tuning: C = 100 (1)

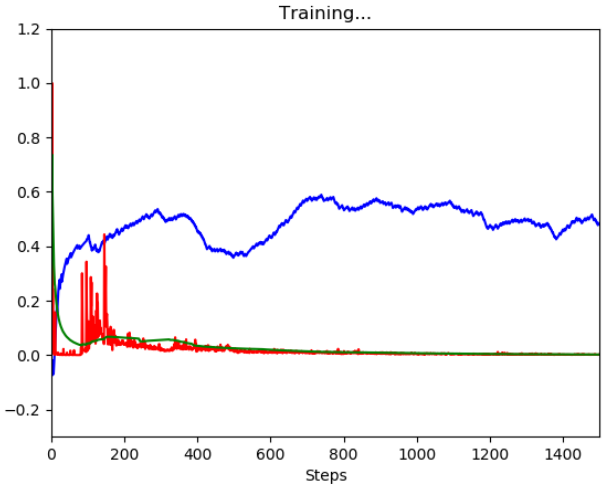


Fig 51 – Hyperparameter tuning: C = 100 (2)

Fig 48 clearly shows that a small value of C is able to learn but it is more unstable, taking longer to converge. The loss function is a lot noisier and the reward level is high but still not perfect. An intermediate of C (10 steps in this case), makes learning a lot smoother and the reward level is the optimal one (0.8).

Harder to analyze is what happens for high values of C. Several experiences where carried out and in half of those the agent acquired an optimal behavior and in the other half it did not. Fig 50 shows a successful example and Fig 51 and unsuccessful one. This could be interpreted as "luck". In this environment there is clearly a component of luck, bounded to the high randomness of actions and the network's parameters at the start of the experience. If the agent has the luck of doing a very good sequence of actions at the start, it will learn fairly easily, and if it does not, it will have to overcome this bad start. Normally, when the parameters are well tuned the agent can overcome a bad start if need be. For C = 100, if it gets a good start the difference between the two networks has not that big of an impact and the agent learns correctly. If the start is not good however, the agent will not be able to overcome the difference between the two networks and will not be able to learn an optimal policy.

6.3.7. Optimizer and Learning rate

These two hyperparameters are highly related, to the point that speaking of one without speaking of the other is not possible. The optimizer represents the variation of gradient descent that will be used. There are several possible methods that can be found in Pytorch's documentation [8]. Anyhow, the learning rate has to be specified for all of them. This learning rate is actually the step size in gradient descent and no matter how the gradient descent is performed, it still consists in taking steps in the direction of the gradient, so a step size is needed.

However, because the methods are not the same, the same learning rate may work for an optimizer but not for another one (in the same problem). Each optimizer has a range of values of the learning rate that work, and within this range an optimal learning rate (that will change depending of the problem, of course).

The standard optimizer to use is SGD. This optimizer applies classic stochastic gradient descent. However, according to the literature [9] there are more efficient optimizers than SGD. Particularly, one that stands out from the others is the optimizer Adam. The variation of gradient descent it uses is rather complex and will not be presented here (see [DIED15]).

Both SGD and Adam were tried using several values of learning rates to better see their span of working learning rates.

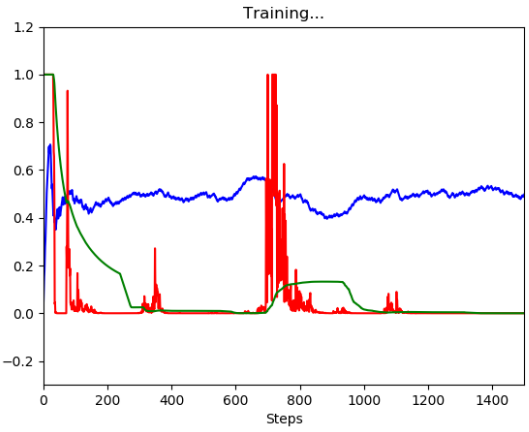


Fig 52: Hyperparameter tuning: Adam, lr = 1

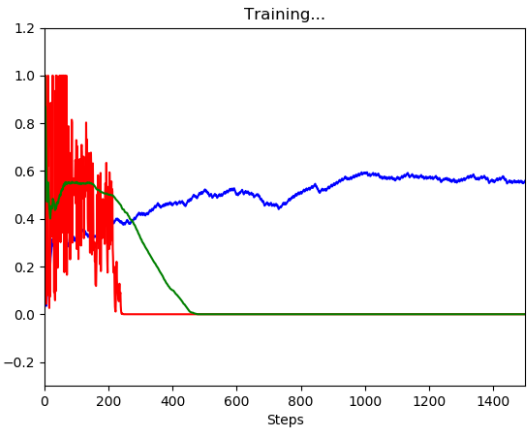


Fig 53: Hyperparameter tuning: SGD, lr = 1

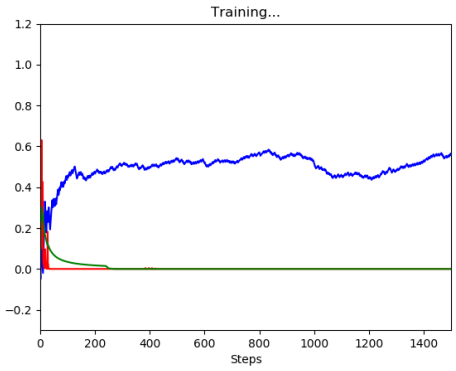


Fig 54: Hyperparameter tuning: Adam, lr = 0.1

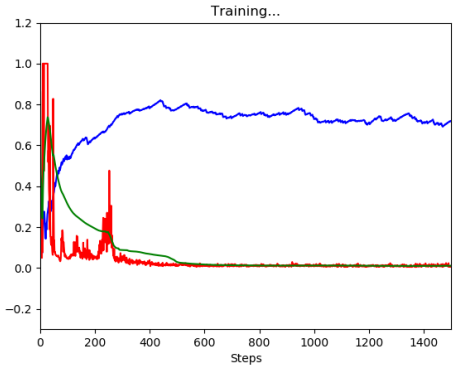


Fig 55: Hyperparameter tunig: Adam, lr = 0.01

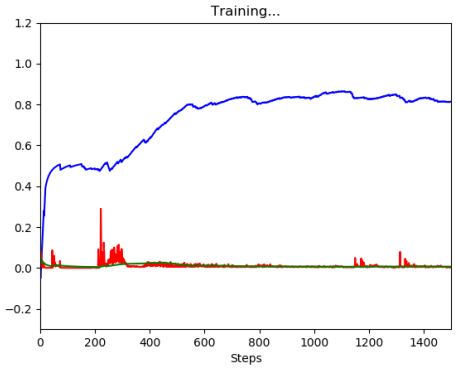


Fig 56: Hyperparameter tuning: Adam, lr = 0.001

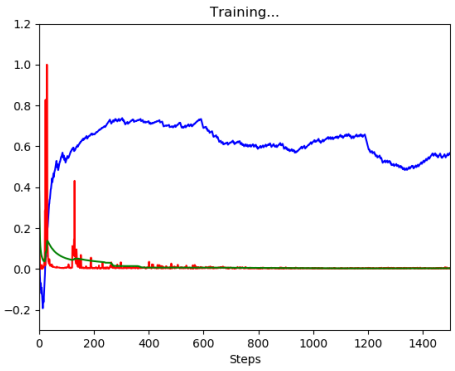


Fig 57: Hyperparameter tuning: SGD, lr = 0.1

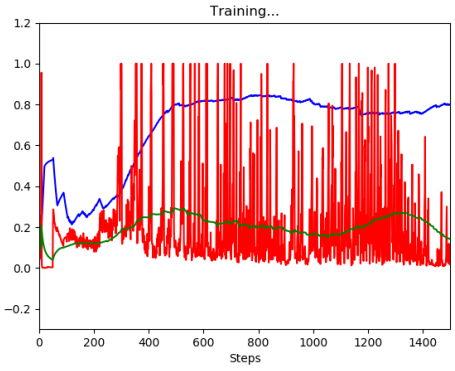


Fig 58: Hyperparameter tuning: SGD, lr = 0.01

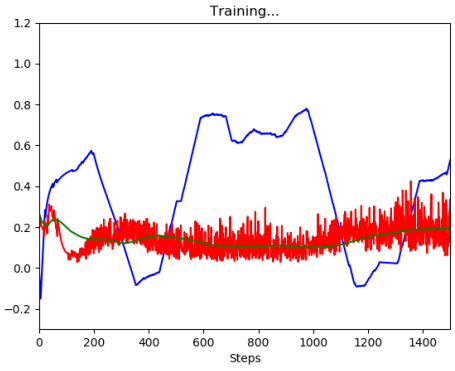


Fig 59: Hyperparameter tuning: SGD, lr = 0.001

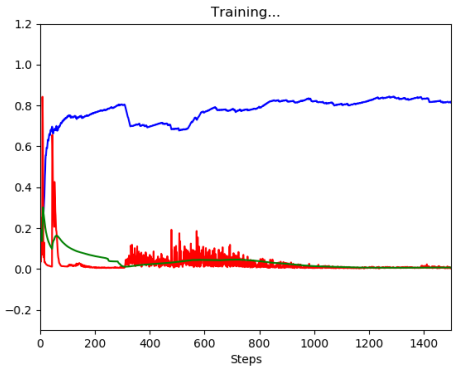


Fig 60: Hyperparameter tuning: Adam, lr = 0.0001

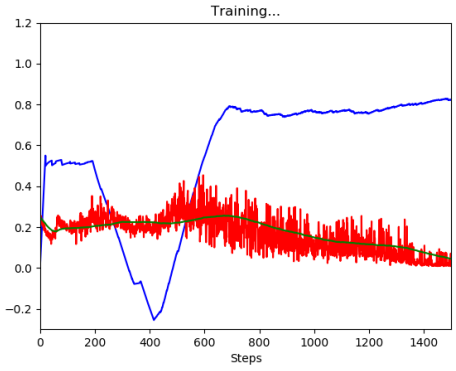


Fig 61: Hyperparameter tuning: Adam, lr = 1e-05

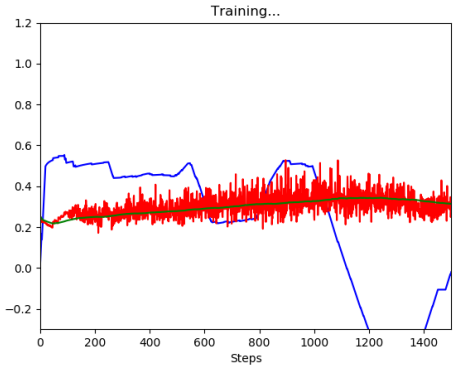


Fig 62: Hyperparameter tuning: Adam, lr = 1e-06

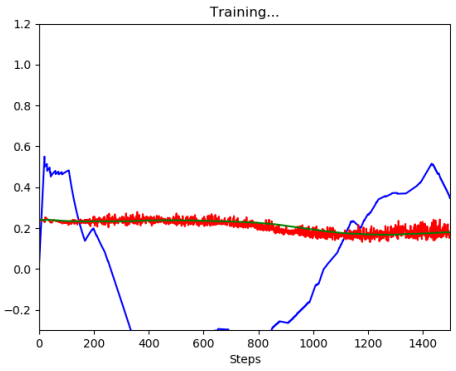


Fig 63: Hyperparameter tuning: SGD, lr = 0.0001

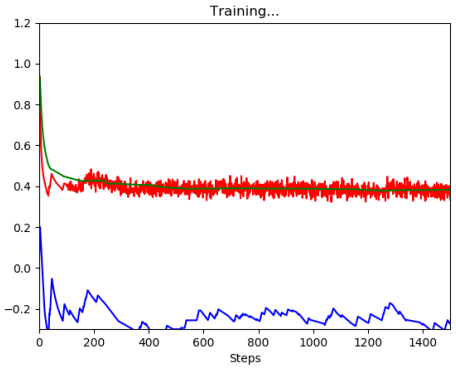


Fig 64 – Hyperparameter tuning: SGD, lr = 1e-05

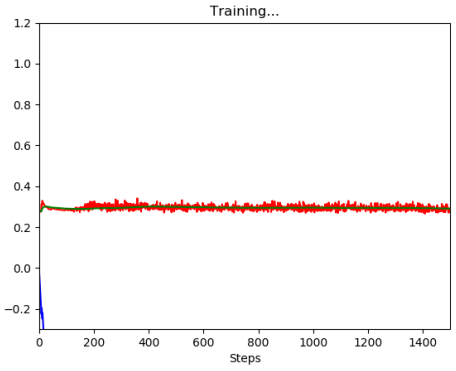


Fig 65 – Hyperparameter tuning SGD, lr = 1e-06

From this comparison between both optimizers some conclusions can be drawn. Firstly, Adam performs globally better than SGD in every way: the range of successful learning rates is bigger and the best learning rates perform better.

Adam has a range of successful learning rates (values of the lr that learn up to a reward level of 0.8) that goes from 0.01 to 0.00001. Over 0.01 the agent learns, but not the optimal policy, and under 1e-05 the agent cannot learn. For SGD however, the only value that can be considered really successful is 0.01.

On top of that, the only successful learning rate for SGD (lr = 0.01) presents a level of noise in the loss curve that means that the algorithm is highly unstable. It could as easily diverge as converge, so it is not reliable. SGD with a lr of 0.1 does not threaten to diverge and achieves a good reward level at the start. However, it fails to maintain this level and settles for an average reward. The optimal learning rate value will likely be somewhere in between those two values.

The only way to determine the learning rate is through trial and error; there is no analytical way of making an approximation. Therefore SGD is highly inconvenient because it is a lot less likely to find a learning rate value that will work with the real problem, when already for the simulated one there are difficulties. In the other hand, Adam is an optimizer that will work with a lot of learning rates so it will not be so important to find the optimal value from the start.

Some conclusions about the hyperparameter "learning rate" and its role in gradient descent can also be drawn. As it has been stated, the learning rate is the step size in the gradient descent. When it is small the agent will take small steps towards the local minimum, and when it is big the steps will be big. It represents in a way how fast the agent tries to learn, because if you want to go quickly somewhere you take big steps.

When the learning rate is too big, the agent tries to learn too quickly and it is easy to pass by the optimal local minimum and reach another local minimum with a poorer result. This can be seen in figures 52 and 54. The algorithm converges without much difficulty but it converges to a different local minimum, the one that sets the behavior of type 2 from 6.3.4.

As the learning rate decreases the convergence is slower. This is shown mainly in figure 61. The mean loss starts at around 0.2 and it rests constant for about 700 actions and then starts to decrease. It looks like the neural network's parameters are set randomly at the start far from the local minimum and the agent updates them slowly, keeping the loss constant because the local minimum is still far apart. When the parameters reach the vicinity of the local minimum, the loss starts to decrease until it converges towards 0. The same could easily happen in Fig 62, but instead of taking 700 actions, it could take 1400. So, for small values of the learning rate the algorithm can yet converge (it is sure to converge even) but it will just take a lot longer because the space of parameters is very big.

7

Results

In this chapter the results of the application of the algorithm to the real robot are presented.

Once the algorithm performed optimally for the simulator, a test was made in the real robot. The real robot and the arena were not always available so there is a lot less data for the real robot than there was for the simulator.

The hyperparameter configuration that was used in this test was:

- Buffer capcity N: 1000
- Gamma: 0.8
- Batch size n: 100
- Exploration: constant exploration of 0.1
- Neural network's structure: network of 3 dense layers of 150, 100 and 50 neurons
- Target update C: 10
- Optimizer and learning rate: Adam with lr=0.01

Some differences between the real environment and the simulator that made the learning process harder were encountered:

- Some small problems that occurred regularly:
  - The robot could disconnect after a shock or could run out of battery,
  - The floor was sometimes too slippery, allowing the robot to be rewarded because its wheels are spinning even if it is hitting a wall.
- The images that the model had as input were much harder to analyze:
  - There is far less contrast between the wall and the ground compared to the black and white of the simulator,
  - The image is noisy and parts of it are useless, due to the camera's placement on the robot, which makes the analysis of the image much harder and longer.
- The robot's movements are trickier and globally worse than in the simulator:

- The walls are not slippery, whereas in the simulator there is a margin of error that is allowed, making a turn against a wall easier at times (not always though).
- The reward is calculated based on the speed of the wheels of the robot. In the simulator it doesn't pose any problem, but in real life the robot has a mass and must accelerate to get to its full speed. Also, it adapts its engine power to its remaining battery, so it might not always go fast enough to get the appropriate reward, even when it is going forward hence should be rewarded a 100%.

8

Conclusions

Some conclusions about the project and about the future possibilities that Machine Learning and more particularly Reinforcement Learning offers

8.1. Conclusions about the project

It is rather obvious when reading chapter 6.3. that the loss function $\mathcal{L}(\theta)$ has two local minima in the space of the parameters $\theta \in \mathbb{R}^d$. The algorithm may converge towards either of them if it is not treated carefully. Each of these minima represents a different behavior for the robot, because the parameters θ define the policy and the minima are found in the space of parameters θ. One of these behaviors is the searched one, where the robot avoids the wall. The other one is the behavior where the robot hits the wall, backs out and then continues forward. This is inherent to this problem, but similar scenarios can be found in other reinforcement learning problems. Therefore, it can be really useful to analyze deeply and carefully the possible behaviors of an agent in an environment before tackling the problem.

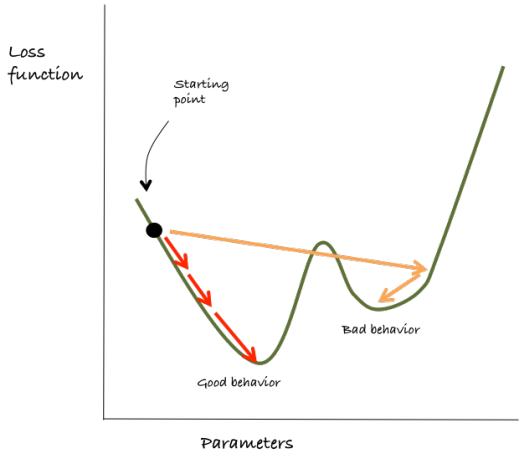


Fig 67 – Diagram showing the problem of two local minimaI

8.2. Possible future advances

There are several ways in which this project could be continued that would prove to be very interesting and educational:

- **Continuous action space $\mathcal{A}$:** in this project, the actions the robot could do were limited to 5 actions. It would be interesting to see what would happen if any direction was allowed to the robot. Would it be able to do perfect oval tours around the arena? However, to accomplish this is rather difficult. We have seen how many problems raised the huge dimension of the space $\mathcal{S}$. If the space $\mathcal{A}$ was continuous another two neural networks would be needed. This method is called DDPG (Deep Deterministic Policy Gradient). DDPG is a really complex method using a lot of computational power, so it would be very difficult to implement.
- **Different type of rewards:** another possibility is to imagine new tasks for the robot to do. The task in this project was rather simple, with only two main different possible scenarios. More difficult tasks like setting targets around the walls of the arena and making the robot go from one to the other could be devised. Something mobile like a balloon (with a different color from the walls and floor) could be set in the middle of the arena, and the robot could try to find it and move it. A barrier could be put in the middle of the arena and the robot could do tours around the barrier like a chariot in a roman circus. This is easier to implement. The most difficult change that would need to be done is the reward function. This involves devising a reward that successfully represents the task we want to do, and then coding it, which would not be easy because some kind of more advance image recognition would need to be done. However, the algorithm itself would not suffer a lot of changes, only some tuning of hyperparameters.

Bibliography

[1] Client's profile: http://neuro-psi.cnrs.fr/spip.php?article921&lang=fr

[2] Web page of the event Startup 4 Kids: http://startupforkids.fr/startups-s4k18/

[3] https://en.wikipedia.org/wiki/Machine_learning

[4] Electronic store where the robot was purchased: https://www.waveshare.com/product/alphabot-pi.htm

[5] Blog article explaining the algorithms in a Roomba vacuum cleaner: https://electronics.howstuffworks.com/gadgets/home/robotic-vacuum2.htm

[6] Blog article explaining the algorithms in self-driving cars: https://www.intellias.com/how-machine-learning-algorithms-make-self-driving-cars-a-reality/

[7] A tutorial from pytorch.org solving a DQN problem: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[8] Pytorch's documentation: https://pytorch.org/docs/stable/index.html

[9] Article studying different optimizers and lr: https://www.freecodecamp.org/news/how-to-pick-the-best-learning-rate-for-your-machine-learning-project-9c28865039a8/

[10]

[SHAI14] Shai Shalev-Schwartz, Shai Ben-David: "*Understanding Machine Learning: From Theory to Algorithms*"

[FRAG18] Fragkiskos Malliaros, Maria Vakalopoulou, Mihir Sahasrabudhe, Enzo Battistella: "*Course en Machine Learning, École Centrale Paris*"

[NIKO18] Nikolaos Tziortziotis: "*Introduction to Reinforcement Learning, Université Paris Sud*"

[RICH17] Richard S. Sutton and Andrew G. Barto; "*Reinforcement Learning; An Introduction*"

[VOLO15] Voldymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller: "*Playing Atari with Deep Reinforcement Learning*"

[DIED15] Diederik P. Kingma, Jimmy Lei Ba: "*Adam: A method for stochastic optimization"*