Master Thesis

Neural Networks applied to Path Planning of Mobile Robots

(ニューラルネットワークを用いた移動ロボットの軌道計画)

Thesis submitted to the
Department of Engineering and Management
Master degree in **Mechatronic Engineering**

School of Engineering
University of Padova

Master Candidate:
Nicola Fiorato

Supervisor: **Prof. Yasutaka Fujimoto**
Supervisor: **Prof. Roberto Oboe**

April 2018

Abstract

Mobile robots are spreading in more and more working areas, not only in environments where human access is not allowed, but the trend is now specially rising for utilization in industrial plants sharing or not space with workers and in domestic or public places where operating among people is normal.
The first category would include mobile robots used to carry a tool to perform a special task in a particular position of the map where almost any way of contact with the special conditions of the environment is considered dangerous for human health: bomb defusing, highly polluted areas by toxic wastes and extraterrestrial planets exploration (e.g. Curiosity rover on Mars planet) are clear examples.
Meanwhile, in industrial plants, mobile robots can be utilized to ease avoid workers from executing simple but stressing duties like carrying heavy products and tools, moreover we are aware of mobile robots currently employed in full-automated warehouses sharing or not space with human workers.
Domestic or public places applications include mobile robots built to get in contact with people or operate among them, there we would include robots developed for rehabilitation like automated wheelchairs, elder people care assistance (e.g. IBM MERA), public patrolling and street sweepers.
The path planner is a fundamental component of the mobile robot's navigation system. Planning a path consists on finding the shortest pathway to the goal point while avoiding obstacles.
The conventional methods needs to know the entire map from starting to goal position before starting to search. However, it is common to consider a map where obstacle can change their initial position or new obstructions can be discovered while the robot is navigating. For this reason, an auxiliary tool that update and store the known map in the robot memory at a certain frequency can be considered too costly.
The purpose of this thesis is to present a new method based on neural networks designed to operate in unknown environments, meaning that the robot has no access to prior informations regarding the navigation space. However, it will rely just on real time sensing of the environment in the local scope of the mobile robot and relative informations between current and goal position.
Supervised learning is the approach selected to train the agent proposed. Given a set of path solutions provided by a supervisor, the neural network is subjected to a training process in order to learn to reproduce similar solutions by repetitively making mistakes and updating its internal parameters.
Experiments results are showed in a simulation environment built on purpose for such kind of problems: a performance analysis and comparison with a widely applied method called A* algorithm. Subsequently, an implementation with a real wheeled mobile robot platform is introduced and tests in an indoor environment are analyzed.

Acknowledgements

First of all, I am profoundly grateful to both my supervisors as the ones who made possible this important opportunity for me: Professor Y. Fujimoto has been a great lead during the entire work here in its laboratory and Professor R. Oboe started and realized my aspiration of having a study experience abroad and moreover in such a good university and beautiful country.
I would like to thank our local industry guild in Italy - Confindustria Vicenza - for the important scholarship I had the chance to be awarded.
I am really thankful to all my mates in Fujimoto laboratory for the nice environment they made me part of and for being there whenever I needed help, specially the Cognition & Plannig group members who I shared opinions and thoughts about our researches, and to my tutor Yusuke Aoki for his kind and always ready support to my common needs in Japan.
Then, a big thank goes to all my professors of Mechatronic Engineering at the Department of Management and Engineering in Vicenza who I am proud to have had.
Eventually, I would like to remind my best friends who have been always close to me through the years, making me feel better every time we were together.

Dedication

This work I realized is dedicated to my family: my mother Beatrice, my father Francesco and my brother Sebastiano.
They always have been an unconditioned and important support to me in every situation, whatever the purpose may have been.
I will always be indebted for their unfading faith in me.

Nicola Fiorato
Vicenza, Italy
April 13, 2018

Table of Contents

List of Figures

List of Tables

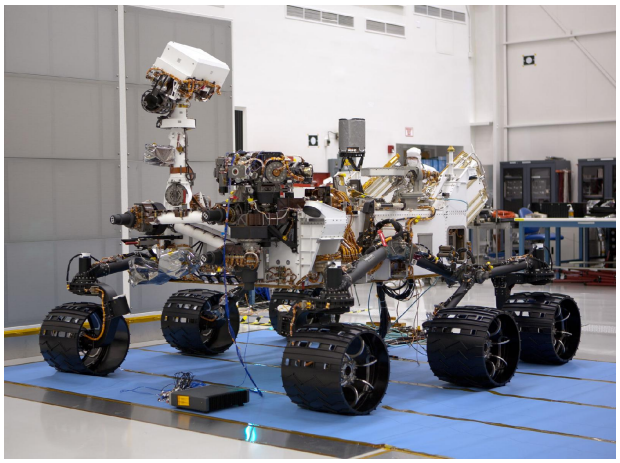
Introduction

A mobile robot can be defined as an automatic system capable of locomotion. In addition, if it is designed to accomplish any task without human intervention it can be considered fully autonomous. Meanwhile, any operation performed by a remote assistance, or a generic external tool, lower the degree of autonomy of the mobile robot.
These interventions range from teleoperating to giving access to the complete environment informations or just a partial part which source is not on board of the mobile robot.
The three main problems of mobile robot navigation are:

- simultaneous localization and mapping (SLAM)
- path planning
- motion control

the first consists in building a virtual version of the environment using the readings of one or multiple sensors the robot carries on itself, the second has to provide the plan to reach a goal point in less time possible and the third is meant to effectively realize the velocity commands by controlling the moving joints.
The listing order is made on purpose because the planner needs a map, or environment informations in general, to search for a solution which is a sort of reference signal to be best follow by the motion controller.
Among the many classifications that can be recognized, I would like to differentiate mobile robots in terms of their target environment, because planning a new path faces different challenges if an area is supposed to be static or have presence of moving obstacles, or if there is no prior informations of the environment or even shallow ones can be taken into account.
The following three main categories are introduced and real examples showed in Figure 1:

1. human intervention not allowed
2. industrial plants related applications
3. important presence of people, domestic and public places



(a) Curiosity rover for Mars exploration [1]

(b) Mobile robots for industrial applications [2]

(c) Robots for eldercare assistance [3]

Figure 1: Examples of mobile robot general applications

I would relate the first one to high-end application, the hardest in terms of unknown environment, the only ones where, in worst case scenarios, replacing autonomous driving systems with teleoperating has reasons to exist. The term unknown is in general associated to environments having no informations about, so the robot has to perceive the navigation space while travelling.
The second simply include every deployment in industries, where the particularity is sharing the workspace with other mobile robots or human workers, the latter may be in a minimal measure.
Meanwhile, considering the work among people as the first design property is what the robots related to the last category share with each other.
The path planning problem internally divides into two stages: the global path planning which main features are low resolution of the environment high level solution taken as a reference signal by the local planning which deals with an high resolution version of environment and dynamic constraints of the robot.
The target of this thesis work was to search for a new solution of global planning based on neural networks taking into account the hypothesis of unknown environment. First of all, because it is hard to guarantee that working area will always be known or however a on-board map building component that updates and store the map during travelling could be too much costly in terms of energy consuming, unless global informations are always provided by external sensors, like a camera, installed in the working area. Hence, the proposed solution aims to solve a path planning problem in static maps with reduced resource usage, meaning that it can only rely on real-time local sorrounding map awareness produced by a local range finder (LRF) sensor and relative informations between the current and goal position such as euclidean distance and relative angle.
The neural networks are a machine learning technique to map inputs to outputs. The term "learning" means they require a training process in order to minimize a certain loss function associated to how good they can relate input-output pairs. In particular, the approach utilized aims to train a neural network to best imitate the behaviour of one of the mostly applied heuristic method for path planning, called A* (pronounced A-star) algorithm.
Chapter 1 introduces the path planning problem, starting with highlighting which elements of the general mobile robot navigation system are dedicated to, the important distinction between global and local path planning is explained and the most applied global planners are described, since the proposed model is meant to compete with them.
Chapter 2 deals about neural networks from their basic idea to the feed-forward and recurrent typologies and then how these models are used as path planning agents.
In Chapter 3 the simulation environment realized is presented, along with the procedure followed to design the method proposed, then a performance comparison with the conventional A* algorithm is evaluated. The last one, Chapter 4, talks about the method implemented on the IRobot Create 2 in combination with the Hokuyo UTM-30LX-EW LRF sensor, same platform as the popular vacuum cleaner, and tests taken as demonstrations are analyzed.

Chapter 1

The Path Planning problem

The purpose of this chapter is to properly define the problem of path planning that the proposed methods aims to solve, starting from explaining what global and local planning are meant to be, subsequently the former is described in details along with the most popular and applied algorithms of this category.

1.1 Global and Local Path Planning

The act of planning a path means to decide where the robot has to go, to generate the next trajectory in order to:

- avoid collisions with obstacles
- reach the goal point as fast as possible

in a general mobile robot navigation system, the planning element fits as showed in Figure 1.1.

Thus, from environment informations that could be metric, like a grid-based map, or topological, and informations about the goal point, the planner generates the velocity commands addressed to the robot structure as a whole.
Focusing on the path planner block, it internally divides into the global and local planner, as showed in Figure 1.2.

The *global path planning*, also known as discrete path planning, generates a low-resolution high-level path from start to goal, which means that the dynamic model or even the base structure of the robot are not taken into account. Instead, the robot is reduced to just an entity operating in a large scale environment, where only large obstacles are avoided.
To the global planner is addressed the task of taking decisions that in this problem are limited to which direction is better to take among the ones available. It is common to call *agent* an entity that takes decisions, in the next section this concept is developed. So, I would conclude that the property of being autonomous starts with the global planner.

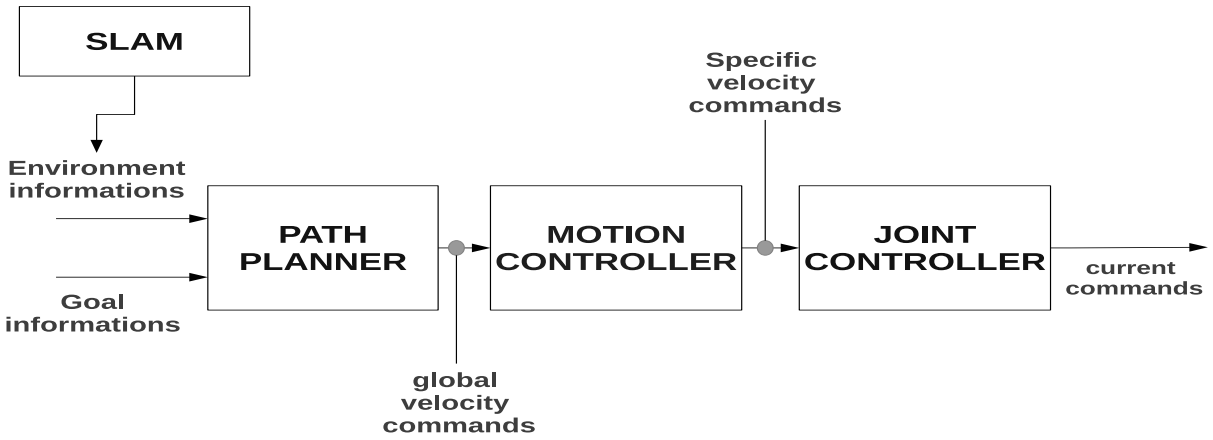


Figure 1.1: A very general diagram block representation of the navigation system

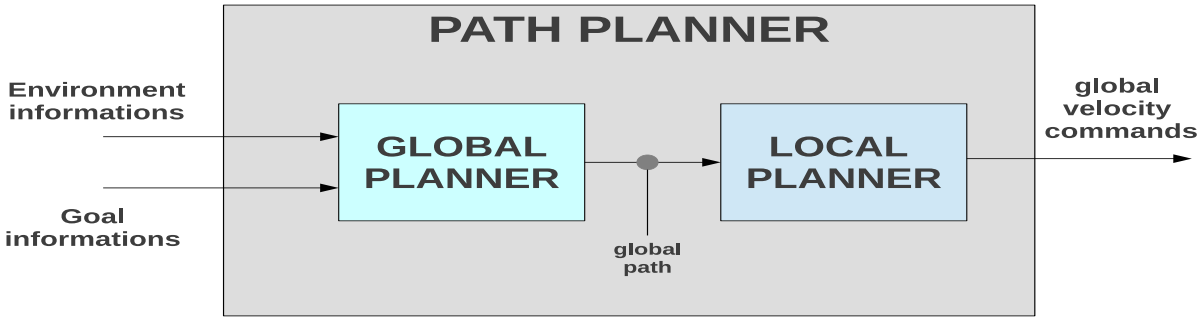


Figure 1.2: How global and local planner are placed inside the general path planner block

On the other side, the *local path planning*, also called continuous path planning, outputs a high-resolution low-level path in a near segment of the global path that is taken as a reference. This is the component that deals with the motion planning, so the dynamic model constraints are taken into account in order to generate the velocity commands referred to the whole robot structure [4].
In Figure 1.3, an example to differentiate global and local plan is showed. The orange line refers to the global plan, it is clear that there is no consistent relationship with the robot structure and it is calculated on a farther horizon with respect to the local plan,

represented by the blue line. Furthermore, the latter stands in a local scope of the robot, in this simple example it is evident how it is calculated to best follow the global path and it is linked to the robot structure as it starts from the central point of the robot base.

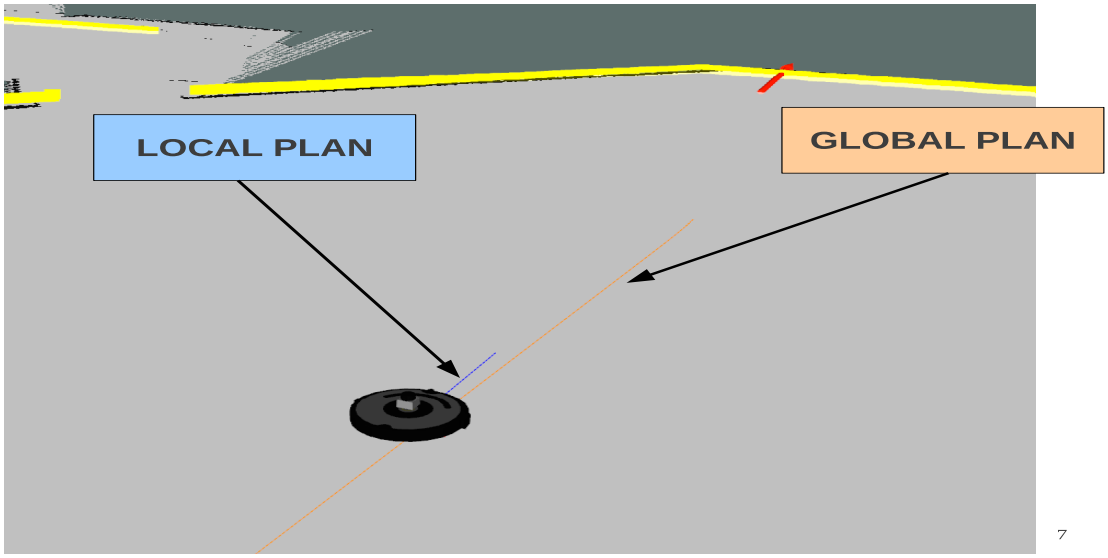


Figure 1.3: Global and local plan generated by a non-holonomic robot

1.2 The Global Path Planning Agent

First of all, a series of important definitions used throughout the document are listed below.

1.2.1 Definitions

Definition 1.2.1. *(Agent)* An agent is an entity able to perceive the environment through dedicated sensors, it process these data in order to perform actions thanks to the actuators.

Definition 1.2.2. *(State)* A series of informations that consistently identify a condition inside an environment. For example, the coordinates of a point or pixel in a map.

Definition 1.2.3. *(Action)* Operation that when performed gets the agent from the current state to a next one.

Definition 1.2.4. *(Deterministic)* If the next state can be completely built by the agent standing in the current state, there is no uncertainty regarding the next decision.

Definition 1.2.5. *(Stochastic)* The outcome calculated by the agent is affected by uncertainty, meaning it is given in terms of probability.

Environment properties

Definition 1.2.6. *(Fully observable)* An environment is fully observable if the agent has complete access to each point at each timestep. In other words, the sensors can give detect all aspects relevant to the next decision performed by the agent.

Definition 1.2.7. *(Partially observable)* Partial observability refers to the cases when sensors noise or similar downgrading feature like low accuracy, are not neglectable in the calculation of the choice of action. This means that the state univocity property is not guaranteed.

Definition 1.2.8. *(Known)* The agent has prior knowledge of the environment given by a third entity, like a sensor not initially associated to it. In addition, it has a complete cognition of the outcome given by every available actions.

Definition 1.2.9. *(Unknown)* The agent can extract environment features by only using its sensors. It does not know about any outcome regarding the actions it can take, so the agent has to learn what behaviour they generate.

Definition 1.2.10. *(Static)* The environment does not change while the agent is processing the calculation of the next decision.

Definition 1.2.11. *(Dynamic)* The opposite of static, environment modifications are possible after the agent has got complete knowledge of the current state and before it outputs the next choice.

Generally speaking, the agent concerning path planning is a *goal-based agent*, it needs some sort of goal information, more precisely, the goal is represented by the final coordinates to be achieved.
Meanwhile, the process of solving a path planning problem produce a result in terms of a fixed sequence of actions. In literature, this is called also *problem-solving agent* - or simply search argent - if it consider a state made of atomic entities, otherwise if the state is composed by logical values I would talk about *planning agents* [5].
In this thesis, just the former is considered, as the method proposed has a state consisting on a series of measures and also it is more straightforward compared to the other category.
A minimal and well formulated definition of a search agent for path planning has to include its own description of the following characteristics:

- **Initial state**: the point of the map where the agent starts searching for the goal point
- **Actions**: a description of what is the behaviour associated to each action
- **Transition Model**: returns the state that results from the last choice of action performed, easily called successor state
- **Goal Test**: a consistent condition to check whether a successor state is the goal point

The solution found by the agent is a sequence of states connected to a sequence of actions, defined in one word as *path*. Every agent associates to the path found a path cost calculated following a predefined rule that generally in mobile robots path planning is the travelled distance along the path found: since the purpose will be to find the path with minimal cost, the shortest one means minimal time of travelling required.
Another operating property that identifies a global path planning agent is whether it is designed to perform an *online search* or an *offline search*, it is really crucial to be clear on these last two definitions because the final method proposed based on neural networks is an online agent that will be compared with an offline agent, the A* algorithm. Sections below are dedicated to their general meaning and description of advantages and disadvantages.

1.2.2 Offline Search

An offline search algorithm has to complete the calculation of the whole path found as the solution before the local planner actually generates the real commands for the mobile robot actuators in order to start moving.
First of all, searched distance is at least equal or higher with respect to the final travelled distance and a characteristic behaviour is that it is allowed to expand nodes farther two or more steps away from the current position.
An important hypothesis arise, this kind of agents needs, in the worst case, to know the entire map before starting the path calculation [6]. Therefore, offline search is more

suitable for known and static environment, so that generally the former implies the latter and viceversa.
In simulation environments, the map is usually given because it consists of an image or any other form of grid-shaped map, for example in our case of study an ASCII character occupies a pixel of the map and it is labelled following a predefined rule. So, in this kind of simulation environments there is no computation time dedicated to build the map.
Meanwhile, in real world applications, this is a critical step because the goodness of the solution calculated by the path planning algorithm directly depends on that. Figure 1.4 can help to understand better this passage, after getting goal informations like distance and relative angle, external sensors or on-board sensors are responsible to build a discretized map by perceiving the real world, pale yellow coloured pixels stands as the minimal size of the map portion stored in agent's memory referred to the case analyzed, thus offline search is computed and ultimately the agent's starts to physically move.

Considering this line of work, it should be easy to guess that if an application sets

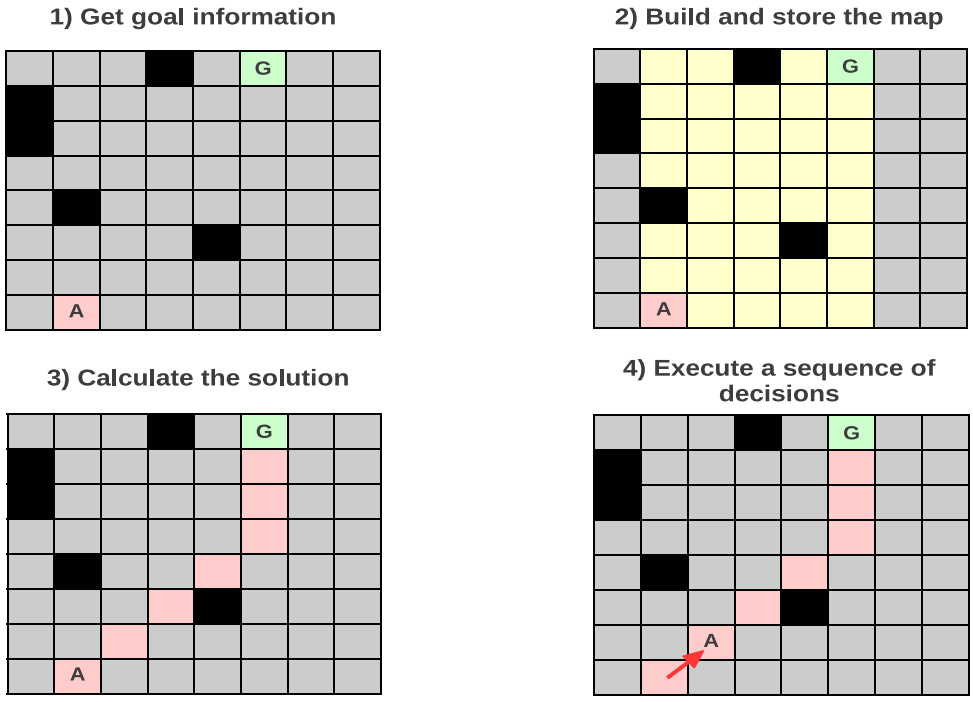


Figure 1.4: Schematical representation of offline agents operations

computation time as an operating limit, it could be a constraint too hard to meet for this kind of agent. On the upside, it is suitable in every situation where the optimality of the solution - shortest travelled distance found - is the most important solution performance parameter, moreover failures given by getting in a blind spot does not have reason to occur if the map built is large enough to include a safe way to the goal.

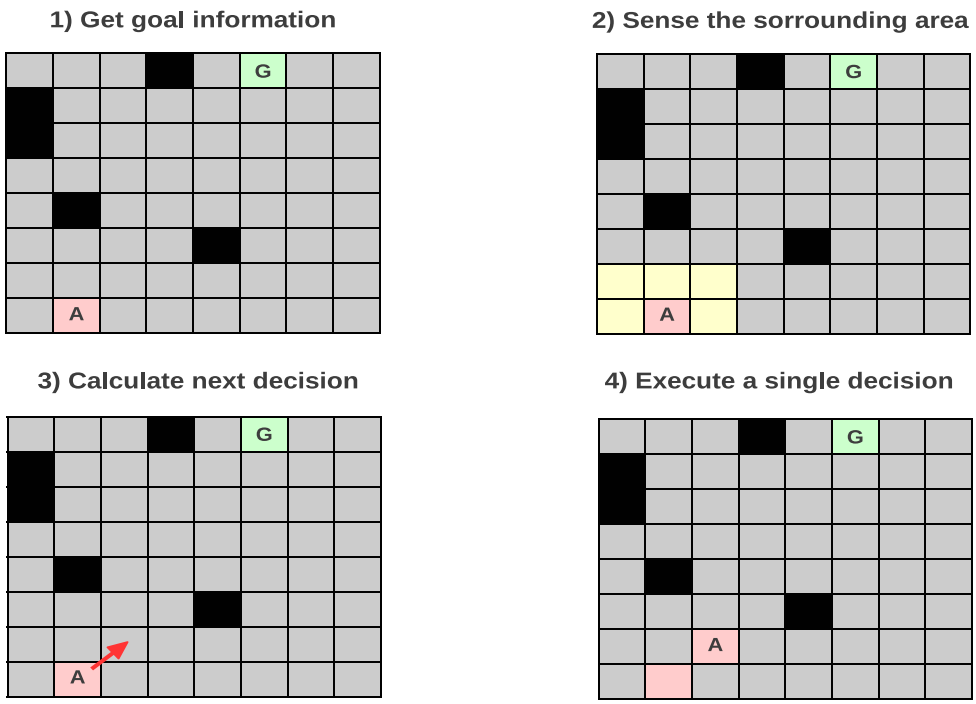


Figure 1.5: Schematical representation of online agents operations

1.2.3 Online Search

Online search consists on executing the last decision calculated at the previous step, observe the local environment and compute just the next decision that will be immediately executed [7].
The nature of this search type presents a more severe hypothesis than offline agents: searched distance is equal to the travelled distance, every expanded node is included in the solution found.
The combination of this property and the act of performing a new observation at each step, makes this category of agents suitable for unknown environments but specially when dynamics are present. Furthermore, computation time is restricted to just one physical step, which makes it easier to meet real-time applications demand, compared to offline agents search including the time to build the map.
In Figure 1.5 are showed the main steps concerning an online search cycle supposing the last decision is already taken and executed.
Every time an online agent is deployed in a new environment and does not know what is the consequence of its actions we talk about it is in a sort of state of ignorance, so it has to face the *exploration problem* consisting on using the available actions to build a background of results from which it can benefit to make next choices better than they could have been when the exploration did not started.
A widely known problem related to online search is represented by getting stuck in blind spots, for example, suppose the agent during its exploration has developed a concept of

getting closer to the goal point is better than any other result and an initial case scenario showed on the left of Figure 1.6, the consequence depicted on the right is the one that will most probably occur. Escaping from such situation that in a maze could easily happen is a tough challenge for an online agent.

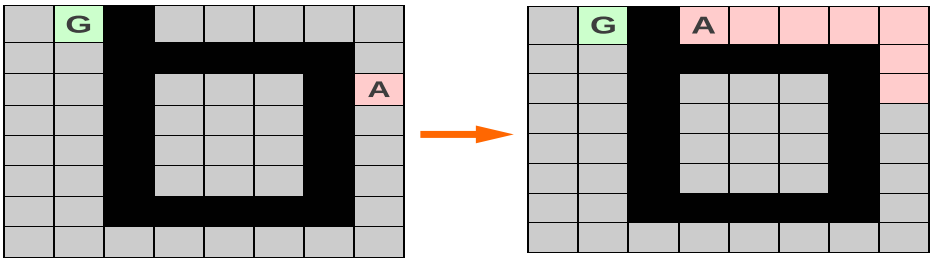


Figure 1.6: Scenario of getting stuck in a blind spot

Another weakness is that in almost every case the travelled distance found to achieve the goal position is far from the optimal, a practical meaning is solution found cost is generally higher compared to an offline agent solution.
Summarizing, the online agents takes into account more severe hypothesis with respect to offline agents which, leaving out computation time, are in general expected to have better performance results primarily represented by travelled distance found.

1.3 Path Planning Algorithms

1.3.1 The A* algorithm

Pronounced "A-star", this algorithm belongs to the offline agents category and represents one of the most applied extension to the Edsger Dijikstra (1959) algorithm [8].
It follows the *best-first* approach, which means associating to each applicable action at node n a value calculated with an evaluation function $f(n)$ and selecting the next step corresponding to the best value.
In this case, the evaluation function is directly proportional to the cost estimation, so the node with the minimal value should be the next expanded.
The function $f(n)$ is defined as follows:

$$f(n) = g(n) + h(n) \tag{1.1}$$

where $g(n)$ represents the cost from starting point to node n and $h(n)$ is the estimation of the future cost from node n to the goal point.

The former is easily calculated following the backpointers until the starting node is reached, while the latter represents an heuristic distance estimation from node n to the goal point and it represents the future cost evaluation.
Since these kind of algorithm operates in grid-shaped maps, a digression is needed regarding which heuristic function gives the best search behaviour. The grid map is a discretization of the real world, where every pixel can assume a predefined value referred to either obstacle or free space.
The most important thing to be verified is that the backward cost and future estimation cost must match in terms of how they are calculated and this is mainly related to the allowed directions the agent can choose. Considering square grid maps, two variants are possible:

1. cartesian directions, 4 allowed moves
2. cartesian plus diagonal directions, 8 allowed moves

A cost parameter c is associated to cartesian directions, while d is associated to diagonal moves. If the first variant is selected, the most reasonable heuristic function is called *Manhattan* and defined as follows:

$$d_x = |n_x - g_x| \qquad d_y = |n_y - g_y| \tag{1.2}$$

$$Manhattan(n) = c \cdot (d_x + d_y) \tag{1.3}$$

where (n_x, n_y) are the coordinates of the node currently analyzed and (g_x, g_y) are the coordinates of the goal point, an example of this variant is showed in Figure 1.7. The cost parameter c can be easily set to 1, however just a simple rule needs to be met: moving one step closer to the goal should increase $g(n)$ by c and decrease $h(n)$ by c, so when you add the two in $f(n) = g(n) + h(n)$, the result will stay the same; this is a sign that the heuristic and cost function scale parameters match.

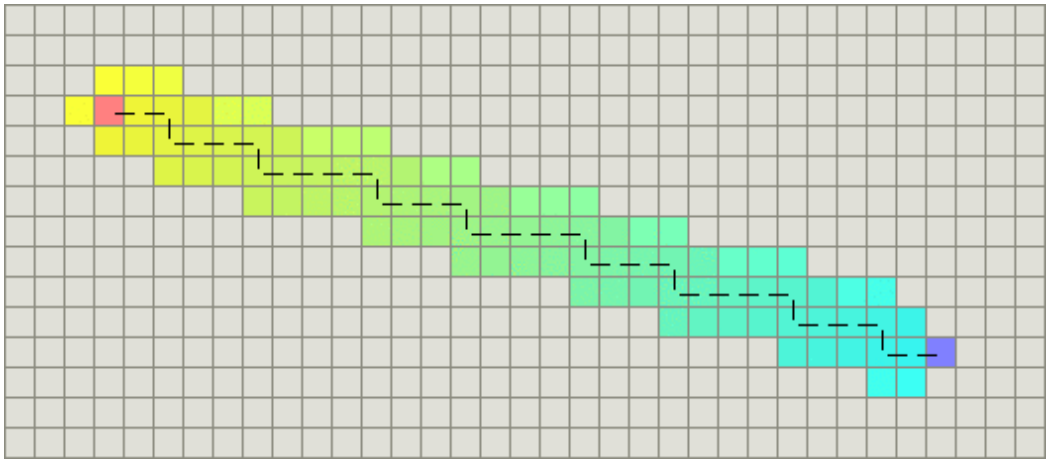


Figure 1.7: Example of first variant search using Manhattan heuristic function [9]

Following the same principle of the Manhattan function, the general formula for the second variant computes the number of steps you take if you cannot take a diagonal move, then subtract the steps you save by using the diagonal moves. In particular, there are $min(d_x, d_y)$ diagonal steps, and each one costs d but saves you $2 \cdot c$ non-diagonal steps:

$$diagonal(n) = c \cdot (d_x + d_y) + (d - 2 \cdot c)\, min(d_x, d_y) \tag{1.4}$$

if $c = 1$ and $d = 1$ this function is known as *Chebyshev distance*, meanwhile if $c = 1$ and $d = \sqrt{2}$ it is called *octile distance*, an example is showed in Figure 1.8 [9].

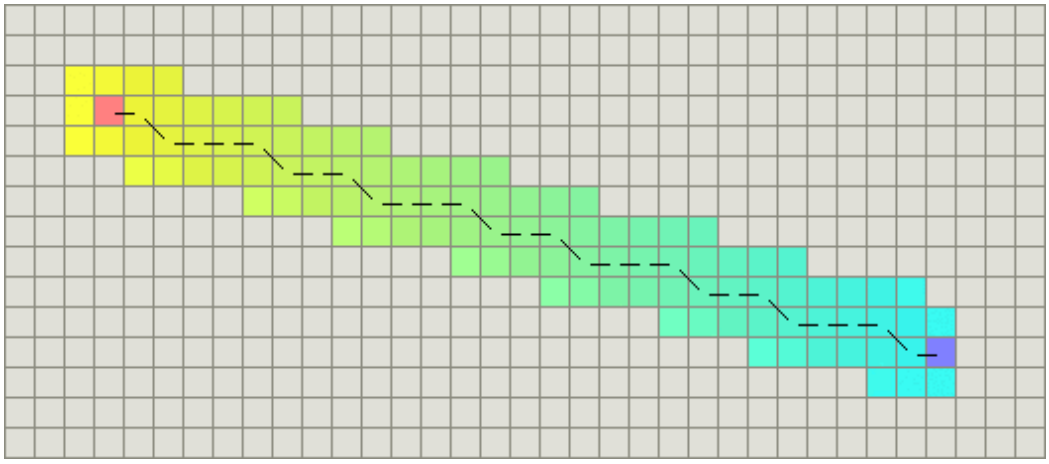


Figure 1.8: Example of second variant search using Octile distance [9]

An alternative to *diagonal* formula could be represented by the euclidean distance, the absolute minimal estimation, but the value returned is way too lower than the backward cost $g(n)$ and a mismatch would occur. Underestimating the forward cost $h(n)$ still returns the optimal path, but computation time of A* increases and easily overtake an acceptance threshold.
This phenomena was experienced during my research work dedicated to find the best A* configuration, however it was found that adding a scale parameter to the euclidean formula matches the backward cost so the overall search works better and computation time is at least in range with the *diagonal* distance case:

$$euclidean_{scaled}(n) = s \cdot \sqrt{d_x^2 + d_y^2} \tag{1.5}$$

of course there is no magic numbers regarding the parameter s which has to be tuned on field for each different working environment adopted. In Chapter 3, a value particular for the simulation environment realized is showed.
Assuming that starting point node is known, as well as the goal node, respectively labelled with S and G, at this point the evaluation function $f(n)$ is already defined as mentioned above, the conventional A* star algorithm is showed in Algorithm 1 [10] where the definition below is taken into account:

Definition 1.3.1. *(Adjacent node)* Given the current state node, the distance calculated in terms of steps between the last and one of its adjacent nodes is exactly equal to 1, in other words the adjacent nodes of the node n equal to its *successor nodes*.

Algorithm 1 A* algorithm

```
1:  procedure A-STAR(S,G)                          ▷ starting node, goal node
2:      OpenSet ← S                                ▷ node s "marked" as open
3:      f(S) = g(S) + h(S)
4:      t ← 0                                      ▷ t as timestep
5:      while OpenSet != ∅ do
6:          n_next = n_min ∈ OpenSet s.t. f^(t)(n_min) < f^(t)(n_j) ∀j = 1, ..., N
7:          if n_next = G then
8:              Terminate()                        ▷ Goal reached
9:          else
10:             ClosedSet ← n_next                 ▷ node n_next "marked" as closed
11:             start execute expand(n_next)
12:             for all m ∈ ADJ − NODES(n_next) do  ▷ [1]
13:                 f^(t)(m) = g^(t)(m) + h^(t)(m)
14:                 if m ∉ ClosedSet then
15:                     OpenSet ← m
16:                 else if f^(t)(m) < f^(t_i)(m) then   ▷ [2]
17:                     OpenSet ← m
18:                 end if
19:             end for
20:             end execute expand(n_next)
21:         end if
22:         t ← t + 1
23:     end while
24: end procedure
```

[1] Here $ADJ - NODES(x)$ represents the fixed set of adjacent (successor) nodes of x

[2] $t_i :=$ timestep when m was marked open

1.3.2 The D* Algorithm

As introduced previously, if the environment is unknown, offline planners like A* has to be supported by a tool to build the map from sensors informations, at least the portion where the search is performed.
So, after the path solution is calculated, if the robot encounter a discrepancy in the map, it has to stop, update the map and call the optimal planner for a new solution. This is the reason why algorithms like A* are more suitable for static maps.
The main purpose of the "D-star" algorithm is to manage better these situations allowed to occur in dynamic maps: a new obstacle gets in the way of the optimal path during travelling, this means a discrepancy with respect to the map built before starting to move.
The connection between two states is called *directional arc*, or simply arc, the cost of traversing an arc from a state x to a state y is a positive number given by the arc cost function $c(x, y)$, hence, this method assume this parameter can change during the problem solving process. Thank to its on board sensor, the robot has a real time perception of the arc cost related to its adjacent states, there a map error can be identified within the sensor scope.
Even D* can be considered a derivation of the Dijikstra's algorithm [8], the basic search strategy is pretty similar to A* algorithm. Given the state x, an operator $t(x)$ associates the tag *OPEN* if it is currently on the *OPEN* list, the tag *CLOSED* if it is no longer on the *OPEN* list and the tag *NEW* if it has never been in the *OPEN* list.
For every state x on the *OPEN* list, an heuristic estimation $h(x)$ of the path cost from x to the goal point G is maintained. Furthermore, a *key* function $k(x)$ is defined to be equal to to be equal to the minimum of $h(x)$ before modification and all values assumed by $h(x)$ since x was placed on the *OPEN* list.
The key function classifies a state x on the *OPEN* list into one of two types: a *RAISE* state if $k(x) < h(x)$, and a *LOWER* state if $k(x) < h(x)$. The *RAISE* states on the *OPEN* list to propagate information about path cost increases (e.g. due to a new obstacle in the pathway) and *LOWER* states to propagate information about path cost reductions (e.g. a new shorter path to the goal).
An important parameter is k_{min}, defined as $min(k(x))\ \forall x$ s.t. $t(x) = OPEN$. If the path costs less than or equal to k_{min} are optimal, and those greater than k_{min} may not be optimal. Instead, the parameter k_{old} is defined to be equal to k min prior to most recent removal of a state from the *OPEN* list. If no states have been removed, k_{old} is undefined.
The fundamental functions of the D* algorithm are called $PROCESS - STATE$ and $MODIFY - COST$ [11]. The former is included in the main routine described in Algorithm 2, it is repeatedly called until the state x is removed from the *OPEN* list or this list results empty, meanwhile the latter (Algorithm 3) is called every time an error in the arc cost parameter arise.
The function $PROCESS - STATE$ is described in Algorithm 6, the other algorithms showed below to this are dedicated to all the subroutines mentioned.

Algorithm 2 D* main routine

```
1: procedure D*(s,g)                                  ▷ starting state, goal state
2:     for all states x do
3:         t(x) = NEW
4:     end for
5:     h(g) = 0
6:     INSERT(g, h(g))
7:     repeat
8:         res = PROCESS − STATE()
9:     until res! = −1
10: end procedure
```

Algorithm 3

```
1: procedure MODIFY-COST(x,y,new-cost)                ▷ starting state, goal state
2:     c(x, y) = new-cost
3:     if t(x) = CLOSED then
4:         INSERT(x, h(x))
5:     end if
6:     return GET − KMIN()
7: end procedure
```

Algorithm 4

```
1: procedure MIN-STATE
2:     if OPEN − list = ∅ then
3:         return NULL
4:     else
5:         return x s.t. k(x) = k_min
6:     end if
7: end procedure
```

Algorithm 5

```
1: procedure GET-KMIN
2:     if OPEN − list = ∅ then
3:         return NULL
4:     else
5:         return k_min
6:     end if
7: end procedure
```

Algorithm 6

1: **procedure** PROCESS-STATE
2: $x = MIN - STATE()$
3: **if** $x = NULL$ **then**
4: return -1
5: **end if**
6: $k_{old} = GET - MIN()$
7: $DELETE(x)$
8: **if** $k_{old} < h(x)$ **then**
9: **for all** $y \in ADJ - NODES(x)$ **do** ▷ [1]
10: **if** $h(y) \leq k_{old} \ \wedge \ h(x) > h(y) + c(y, x)$ **then**
11: $b(x) = y$
12: $h(x) = h(y) + c(y, x)$
13: **end if**
14: **end for**
15: **end if**
16: **if** $k_{old} = h(x)$ **then**
17: **for all** $y \in ADJ - NODES(x)$ **do**
18: **if** $\left(\begin{array}{l} t(y) = NEW \vee \\ (b(y) = x \wedge h(y) \neq h(x) + c(x, y)) \ \vee \\ (b(y) \neq x \wedge h(y) > h(x) + c(x, y)) \end{array} \right)$ **then**
19: $b(y) = x$
20: $INSERT(y, h(x) + c(x, y))$
21: **end if**
22: **end for**
23: **else**
24: **for all** $y \in ADJ - NODES(x)$ **do**
25: **if** $\left(\begin{array}{l} t(y) = NEW \vee \\ (b(y) = x \wedge h(y) \neq h(x) + c(x, y)) \end{array} \right)$ **then**
26: $b(y) = x$
27: $INSERT(y, h(x) + c(x, y))$
28: **else if** $(b(y) \neq x \wedge h(y) > h(x) + c(x, y))$ **then**
29: $INSERT(x, h(x))$
30: **else if** $\left(\begin{array}{l} b(y) \neq x \wedge h(y) > h(x) + c(x, y) \wedge \\ t(y) = CLOSED \wedge h(y) > k_{old} \end{array} \right)$ **then**
31: $INSERT(y, h(y))$
32: **end if**
33: **end for**
34: **end if**
35: return $GET - MIN()$
36: **end procedure**

[1] Here $ADJ - NODES(x)$ represents the fixed set of adjacent (successor) nodes of x

Algorithm 7

1: **procedure** INSERT(x,h_{new})
2: **if** $t(x) = NEW$ **then**
3: $k(x) = h_{new}$
4: **else if** $t(x) = NEW$ **then**
5: $k(x) = min(k(x), h_{new})$
6: **else if** $t(x) = CLOSED$ **then**
7: $k(x) = min(h(x), h_{new})$
8: **end if**
9: $t(x) = OPEN$
10: $h(x) = h_{new}$
11: **end procedure**

Algorithm 8

1: **procedure** DELETE(x)
2: $OPEN - list \setminus \{x\}$
3: $t(x) = CLOSED$
4: **end procedure**

Chapter 2

Neural Networks

The term "neural network" comes from the early studies on information processing in biological systems, in particular how the nervous system could be represented with a mathematical model. The first mention would date back in 1943 [12], but important advances start in the 1960s when the structure known as Feed-Forward neural network or Perceptron was defined by Frank Rosenblatt [13] and authors started talking about the process of "training a neural network" [14].
Since those years there have been tremendous achievements by this mathematical model, now it can be considered one of the most performant tool for *pattern recognition* problems, along with Support Vector Machines and Kernel Machines [15] [16].
The neural networks set "state of the art" performance in important information technology related applications, I would here mention object recognition, speech recognition and machine translation [17] [18] [19] [20] [21].
When it gets to decide about how to train a neural network for any application, three approaches can be considered: supervised learning, reinforcement learning and unsupervised learning.
Supervised learning is the method most applied and easiest to set up, the characteristic property is that a supervisor entity provides the ground truth data for every sample included in the training set, so that the loss would be in terms of "distance" between the output calculated by the neural network and the ground truth.
In **Reinforcement learning** there is no concept of ground truth, updating the learnable parameters is driven by maximizing a reward function, an easy example is the Q-learning agent who fills a Q-table where very single value is a score associated to the pair state-action. Meaning that given a state, the action corresponding to the highest Q-value is selected.
Unsupervised learning involves observing the samples as input data and attempts to learn the "features" they could present.
Through the years many different kind of structures have been proposed, in Sections 2.1 and 2.2 are dedicated to the theoretical basics regarding the ones tested and working for path planning of mobile robots. Section 2.3 concludes this chapter introducing how neural networks are applied to solve the path planning problem.

2.1 Feed-Forward Neural Networks

Also known with the name *perceptron*, it is the most traditional structure. It consist of many connected units called neurons, each one produces a real value as result of applying a non linear activation function to a linear combination of weights multiplying the values of other neurons.
Consider the simple structure showed in Figure 2.1 where an arrow represents a single connection, the term feed-forward is given because every single connection is associated to a weight that multiplies a neuron closer to the input layer and the result will be included in the value of a neuron closer to the output layer, in other words just connections from left to right exist.
In general, a layer of this kind is also called *full connection* (FC) because every unit receives a connection from each unit of the previous layer plus the bias term, the layers of neurons between input and output are called *hidden* layers because their values are not given in the problem data.

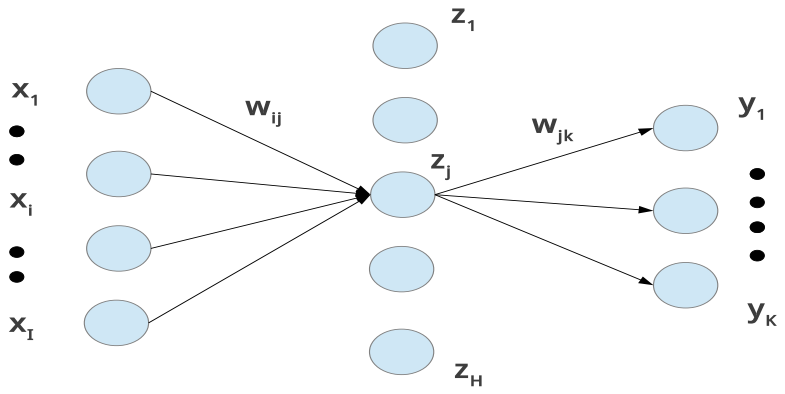


Figure 2.1: Schematic representation of a feed-forward neural network

At this point, the basic mathematical model can be introduced. Suppose the input layer is composed by I values and H is the dimension of the hidden layer, also known as its *width*. Firstly, H different linear combinations of the input neurons values are built:

$$a_j = \sum_{i=1}^{H} w_{ij}\, x_i + w_{j0} \tag{2.1}$$

where $j = 1, ..., H$ and w_{ij} are the learnable parameters known as weights, in particular w_{j0} stands for the bias.
Thus, the value of a neuron z_j is given by the application of a non linear activation function $h(\cdot)$ to the quantity a_j, also called activation of the neuron:

$$z_j = h(a_j) \tag{2.2}$$

the most widely known activation functions are listed below and their trend is showed in Figure 2.2:

1. sigmoid function $\sigma(a) = \frac{1}{1+e^{-a}}$
2. hyperbolic tangent $tanh = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
3. linear rectifier $ReLU(a) = max(0, a)$

Proceeding from Eq. 2.2 and still referring to the structure of Figure 2.1 and defining K as the output layer size, the activation and value of the k-th neuron in the output layer is calculated as follows:

$$a_k = \sum_{j=1}^{H} w_{jk}\, z_j + w_{k0} \tag{2.3}$$

$$y_k = h(a_k) \tag{2.4}$$

given a series of input $x_i \ \forall \ i = 1, ..., I$, calculating every $y_k \ \forall \ k = 1, ..., K$ is known as executing the *forward pass*.
The last equations are related to a K-dimensional classification problem, meanwhile if a regression problem would have been taken into account, the size of the output would have been set to $K = 1$.
The basic operation of training a neural network is called *backward pass*, here it is showed in the context of supervised learning because of its simplicity and it is the approach selected for the final model proposed.
The backward pass for a feed-forward neural network follows a gradient descent method called *back propagation* to minimize the loss function $E(\mathbf{w})$ selected [22], where $\mathbf{w}$ is the vector containing all the weights of the neural network. The simplest weight update rule follows the gradient descent method and updates the learnable parameters in order to produce a small step in the direction of the negative gradient, for a generic weight w_{ji} the expression would be:

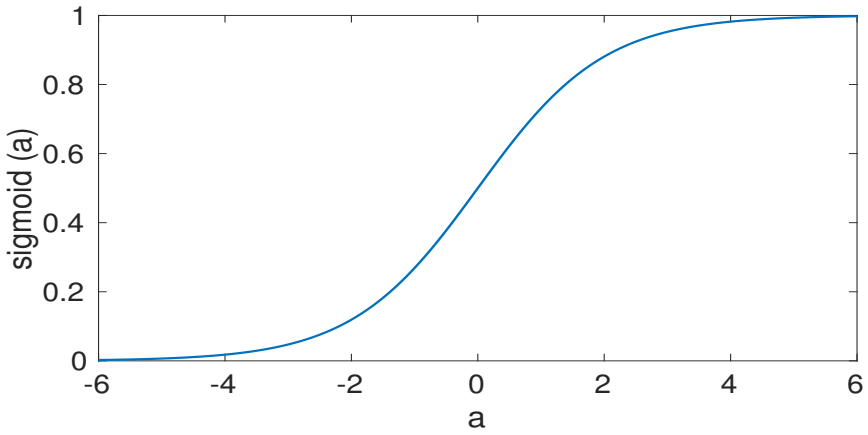
$$w_{ji}^{(t+1)} = w_{ji}^{(t)} - \alpha \frac{\partial E\left(\mathbf{w}^{(t)}\right)}{\partial w_{ji}^{(t)}} \tag{2.5}$$

where α is the *learning rate*.
Consider the same K-dimensional multi-classification problem used above in explaining the forward-pass, t_k as the ground truth value provided by the supervisor for the k-th class and the euclidean error is the loss functions suited for this problem and it is computed as follows:

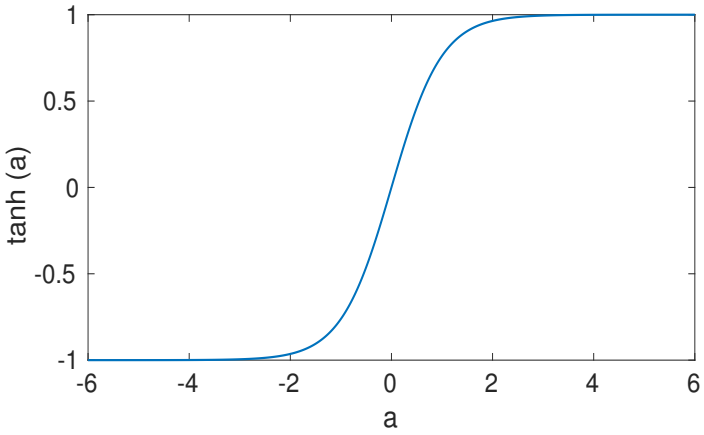
$$E = \frac{1}{2} \sum_{k=1}^{K} (y_k - t_k)^2 \tag{2.6}$$

where, for this passage, it is imposed $y_k = a_k$, the procedure starts by determining the derivative with respect to the weights:

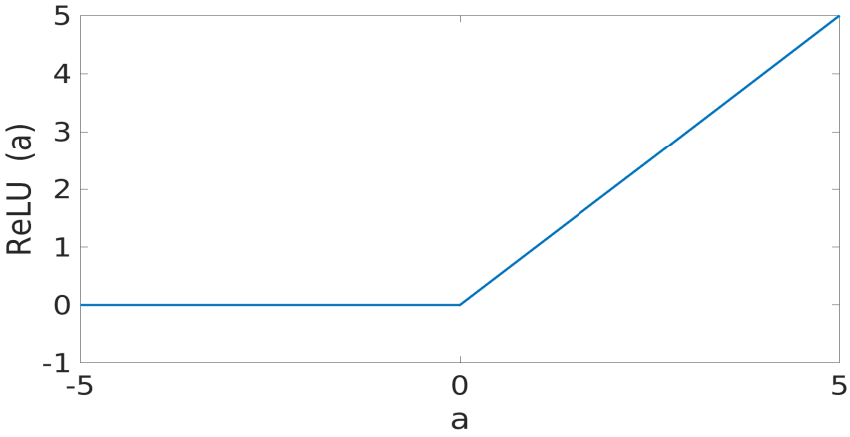
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} = (y_j - t_j)\ z_i \tag{2.7}$$



(a) Sigmoid $\sigma(a)$



(b) Hyperbolic tangent $tanh(a)$



(c) Rectifier Linear Unit $ReLU(a)$

Figure 2.2: The most famous and simple activation functions

now, using a more compact notation:

$$\delta_j = \frac{\partial E_n}{\partial a_j} \tag{2.8}$$

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j \, z_i \tag{2.9}$$

in output units, the δ_k calculation is trivial:

$$\delta_k = \frac{\partial E_n}{\partial a_k} = y_k - t_k \tag{2.10}$$

in the hidden units is a little more complex:

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_{k=1}^{K} \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{2.11}$$

the use of Equations 2.1 and 2.2 and the definition in Eq. 2.8 takes to the conclusion given by the *backpropagation* formula:

$$\delta_j = h'(a_j) \sum_{k=1}^{K} w_{kj} \delta_k \tag{2.12}$$

Up to now, a structure with just one hidden layer has been showed, these are usually referred with the term *shallow*[23]; instead structure with more and more hidden layers are called *deep feed-forward* neural networks or multi-layer perceptron (MLP), Figure 2.3 represents how it would look like.

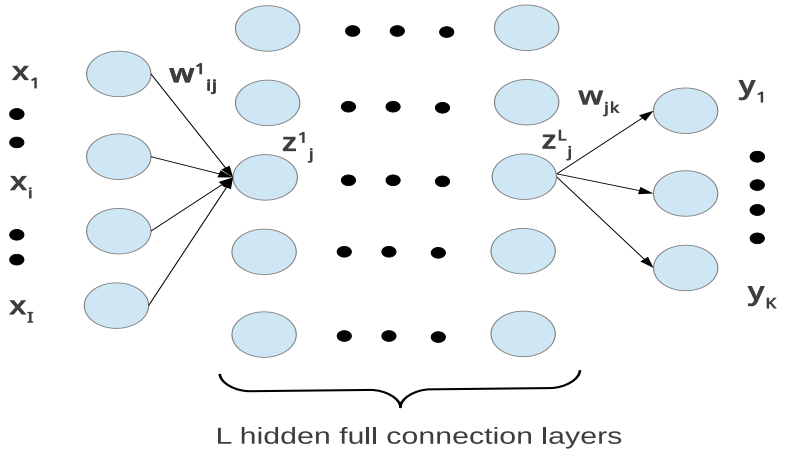


Figure 2.3: Schematic representation of multi-layer perceptron

The concept of depth in neural networks is, in general, addressed to a sequential application of the non linear functions in a single iteration in order to be able to get compositional representations of the input. In other words, adding hidden layers correspond to further utilization of the selected non linearity and it contributes to the depth of the structure in a linear way.
Neural networks with greater depth can execute more instructions in sequence. Sequential instructions offer great power because later instructions can refer back to the results of earlier instruction [24]. In comparing two structures, a common convention is to state that the first is heavier than the second if it has a higher number of weights, meanwhile the second one is lighter.
Suppose a starting structure is shallow with m units, it is generally proven that deeper structures performs better than wider ones even when the former has a lighter structure, in Chapter 3 a practical example of this statement will be showed in the procedure of shaping the final model proposed.

2.2 Recurrent Neural Networks

The peculiarity of recurrent neural network (RNN) is that it is present an internal mechanism of feedback information from the previous computational step. In this sense, the recurrent network structure is the same of the previous feed-forward but cyclical connections are added, meaning learnable parameters multiplying the unit value of older time steps are added to the linear combination computation of the current time step unit activation.
Hence, the first enhancement that can be introduced is modelling an input sequence to an output sequence, rather than a single input to a single output like the perceptron does.
The simplest form of recurrent unit include just a cyclical connection from the previous time step, it can be considered a general dynamic system described by the following equation:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, w) \tag{2.13}$$

where h represents the state of the unit, x is the input, w is the weight used to parametrize f and t is the time step considered, the function f is a transition function, usually a non-linear transformation like *sigmoid* or hyperbolic tangent function.
Consider a recurrent neural network with one hidden layer containing exactly H units, while the input unit $\mathbf{x}$ has dimension $\mathbf{I}$ and the output unit $\mathbf{y}$ has dimension K. The input layer connects to the hidden layer with a input-to-hidden weight matrix $\mathbf{U} \in I \times H$ and the hidden layer to the output layer with a hidden-to-output weight matrix $\mathbf{V} \in H \times K$, while the previous state of the unit is parametrized with the weight matrix $\mathbf{W} \in H \times H$. Figure 2.4 summarizes these connection relationships.
An important structure property is parameter sharing through the whole time sequence, so that the weights composing the matrix $\mathbf{W}$ are the same when they connect the hidden units at different timesteps.

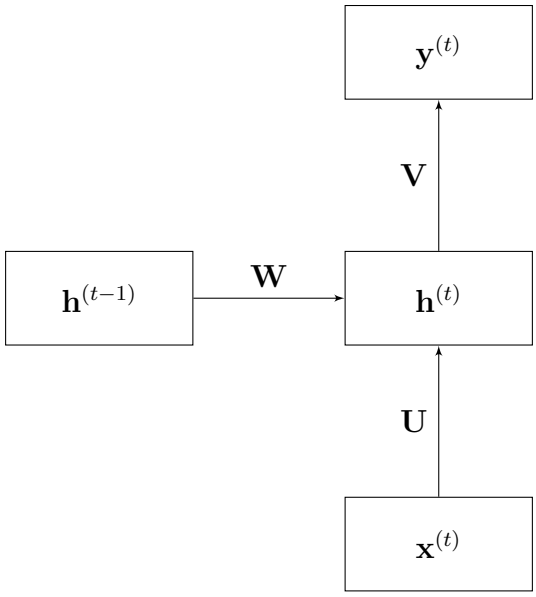


Figure 2.4: Graphical representation of a recurrent neural network

The forward pass of a recurrent is defined from Eq. 2.15 to Eq. 2.16 and can be easily represented by unfolding the computational graph of Figure 2.4 that leads to Figure 2.5 where a time portion of three time steps is considered:

$$s_j^{(t)} = b_i + \sum_{i=1}^{I} u_{ih}\, x_i^{(t)} + \sum_{h=1}^{H} w_{hj}\, h_h^{(t-1)} \tag{2.14}$$

$$h_j^{(t)} = \tanh\left(s_j^{(t)}\right) \tag{2.15}$$

$$y_k^{(t)} = c_k + \sum_{h=1}^{H} h_h^{(t)} \tag{2.16}$$

where b_i and c_k terms act as the biases.

The backward pass consists on repetitively applying the procedure used for the perceptron through the entire time sequence having a length of T time steps, this outlines what is called Backpropagation Throught Time (BPTT) algorithm [25].
Given a loss function E differentiable with respect to the network outputs, very similarly to Equations 2.8 and 2.10, the backward pass for a single timestep t can be defined as follows:

$$\delta_k^{(t)} = \frac{\partial E}{\partial y_k^{(t)}} \qquad \delta_j^{(t)} = \frac{\partial E}{\partial h_j^{(t)}} \tag{2.17}$$

$$\delta_j^{(t)} = \sigma'(s_j^{(t)}) \left(\sum_{k=1}^{K} \delta_k^{(t)}\, w_{jk} + \sum_{h=1}^{H} \delta_h^{(t+1)}\, w_{jh} \right) \tag{2.18}$$

where the complete sequence of δ terms is calculated starting from timestep $t = T$ to $t = 1$, backward in the time sequence.

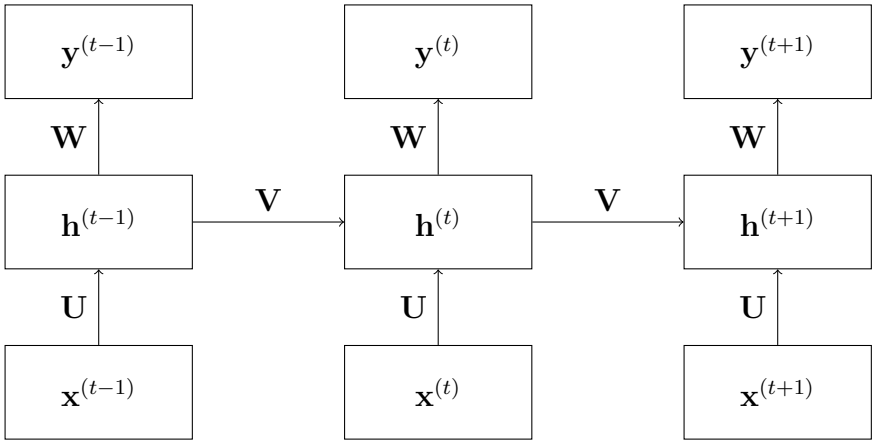


Figure 2.5: Unfolding the graph as forward pass representation

The last step is to calculate the derivative of the loss function with respect to the weights, for the ones included in input-hidden-connection it would be analogous to the feed-forward network case, while for the weights in hidden-to-hidden connections it would be a summation over the time sequence reminding that weights are shared [26]:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial E}{\partial s_j^{(t)}} \frac{\partial s_j^{(t)}}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^{(t)} \, h_i^{(t)} \tag{2.19}$$

Eventually, the weights can be update following the gradient descent method, same rule defined for the perceptron in the previous section in Eq. 2.5.
The most occurring problem that affects the recurrent neural network training is the gradient vanishing and gradient exploding, which leads to the lack of learning the long term dependencies [27] [28]. Focusing on the hidden-to-hidden weights matrix $\mathbf{W}$ in a single unit layer, after t steps, the value of the hidden unit would include the result of repeatedly multiplying the matrix $\mathbf{W}$, this is equivalent to have $\mathbf{W}^t$. Suppose an eigen decomposition such that $\mathbf{W} = \mathbf{V}\text{diag}\,(\lambda)\,\mathbf{V}^{-1}$, this would mean that:

$$\mathbf{w}^t = \left(\mathbf{V}\text{diag}\,(\lambda)\,\mathbf{V}^{-1}\right)^t = \mathbf{V}\text{diag}\left(\lambda^t\right)\mathbf{V}^{-1} \tag{2.20}$$

where the eigenvalues λ_i will cause gradient exploding if their absolute value is too larger than 1, on the opposite gradient vanishing if it is smaller than 1.
A graphical representation of the latter is illustrated in Figure 2.6, where different shades of gray mean proportional sensitivity with respect to the input of time step equal to one, as the sequence goes forward new inputs overwrite the activations of the hidden layer and so input of step one will weight less and less.

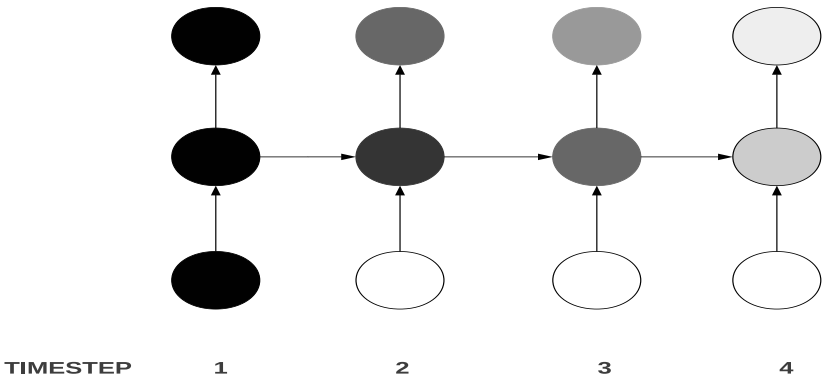


Figure 2.6: Sensitivity decaying over the time sequence with respect to the input at time step equal to one

2.2.1 The Long Short-Term Memory Neural Network

The Long-Short Term Memory (LSTM) neural network network belongs to the Gated Recurrent Networks (GRN) group, a variant of the basic recurrent neural network, as a common characteristic this group have in their single unit structure, called *memory cell*, additional elements working as gates, the LSTM case includes gates applied to the input, previous state and output.
From the first idea of LSTM memory cell by Sepp Hochreiter and Jürgen Schmidhuber in 1997 [29], many variants have been proposed, the most common is showed in Figure 2.7 and refers to the original structure.
The input and output signal, and the previous state value are multiplied by their own related gate function: input gate, output gate and forget gate, respectively.
Suppose a LSTM network has an input layer with I values and an hidden layer with H memory cells, the internal state $s_i^{(t)}$ of the i-th cell is updated by the following equations [30]:

$$s_j^{(t)} = f_j^{(t)}\, s_j^{(t-1)} + g_j^{(t)}\, \sigma\left(b_j + \sum_{i=1}^{I} u_{ij} x_h^{(t)} + \sum_{h=1}^{H} w_{hj} h_h^{(t-1)}\right) \tag{2.21}$$

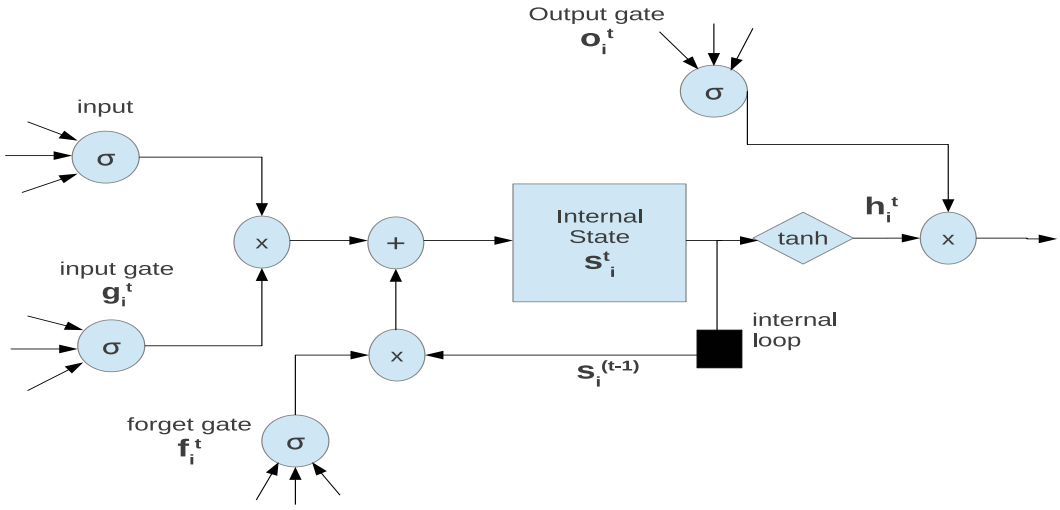


Figure 2.7: Diagram representation of the LSTM "memory" cell

where $f_j^{(t)}$ and $g_j^{(t)}$ are, respectively, the forget gate and the input gate:

$$f_j^{(t)} = \sigma\left(b_j^f + \sum_{i=1}^{I} u_{ij}^f x_i^{(t)} + \sum_{h=1}^{H} w_{hj}^f h_h^{(t-1)}\right) \tag{2.22}$$

$$g_j^{(t)} = \sigma\left(b_j^g + \sum_{i=1}^{I} u_{ij}^g x_i^{(t)} + \sum_{h=1}^{H} w_{hj}^g h_h^{(t-1)}\right) \tag{2.23}$$

where σ refers to the *sigmoid* non linear function. The forget gate can be seen as a sort of factor multiplying the previous state just like in a running average, this allows to have a variable weight associated with the past data, if $f_i^{(t)} \to 0$ the past state values information is quickly forgotten, while if $f_i^{(t)} \to 1$ past story has an important influence on the next state.
Now, the output of the cell can be calculated as follows:

$$h_j^{(t)} = \tanh\left(s_j^{(t)}\right) q_j \tag{2.24}$$

$$q_j^{(t)} = \sigma\left(b_j^o + \sum_{i=1}^{I} u_{ij}^o x_i^{(t)} + \sum_{h=1}^{H} w_{hj}^o h_h^{(t-1)}\right) \tag{2.25}$$

where $q_j^{(t)}$ represents the output gate. In order to have more global view of how the whole structure works, in Figure 2.8 a generic LSTM neural network with two cells is showed along with all connections, where arrows with initial point disconnected refer to biases terms, input and $h_i^{(t)}$ terms are displayed in vector notation and output layer configuration is related to a multiclass classification problems like the one employed in this thesis, this last concept will be described in the next section.

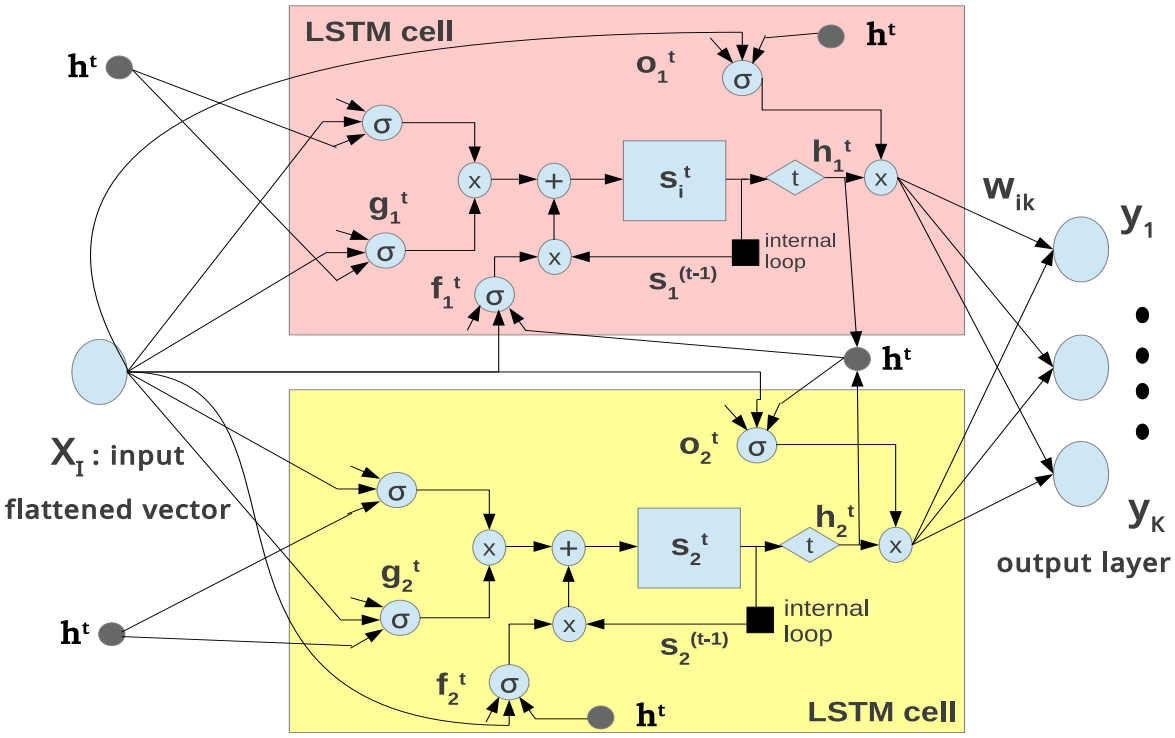


Figure 2.8: Representation of the connections in a two-cells LSTM neural network

The total number of weights is strongly increased with respect to the basic recurrent neural network, but the sensitivity problem introduced in Figure 2.6 is resolved thanks to the variable gates behaviour, which are responsible of the "Long Short Term" name of this kind of network, the input gates at time step higher than one can shut off the input in order to allow a continuous flow of the information relative to the input at time step one, the example is showed in Figure 2.9. More generically speaking, the LSTM network is capable of learning better the long-term dependencies.
Depth in neural networks is, in general, addressed to a sequential application of non linear functions to data generated in a single iteration in order to be able to get compositional representations of the input.
Given this definition, every kind of recurrent neural network has already this kind of structural property in the temporal dimension: unfolding through the whole time sequence [31]. However, it is possible to define specific structures of deep recurrent neural network depending in which position of the structure new layers, generally full-connected ones, the following structures represent the simplest deep LSTM neural network to build:

- deep input to hidden, abbreviated *deep-in*
- deep hidden to output, abbreviated *deep-out*
- stacking LSTM layers, abbreviated stack

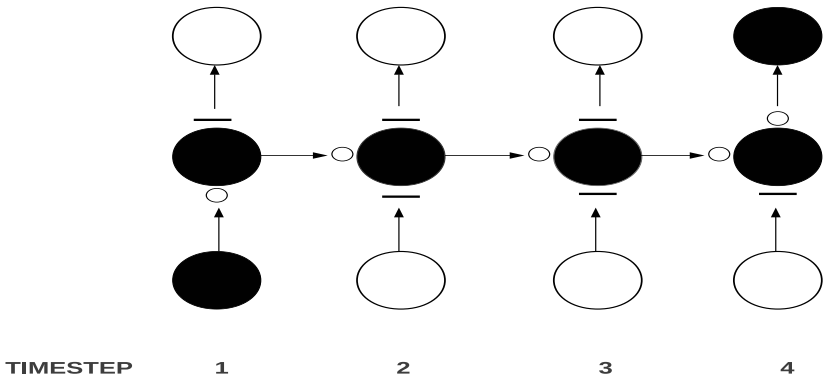


Figure 2.9: Representation of LSTM case behavior for long term dependencies learning, “o” means gate open and “-” means gate closed

In Figure 2.10 are showed the graphical representations of the three structures mentioned above. In order to not be repetitive, the same structures but just with recurrent network layer, discussed above, are addressed to be deep recurrent neural network.
More detailed and articulated discussion about deep recurrent - and LSTM - neural networks can be found here [32] [33].

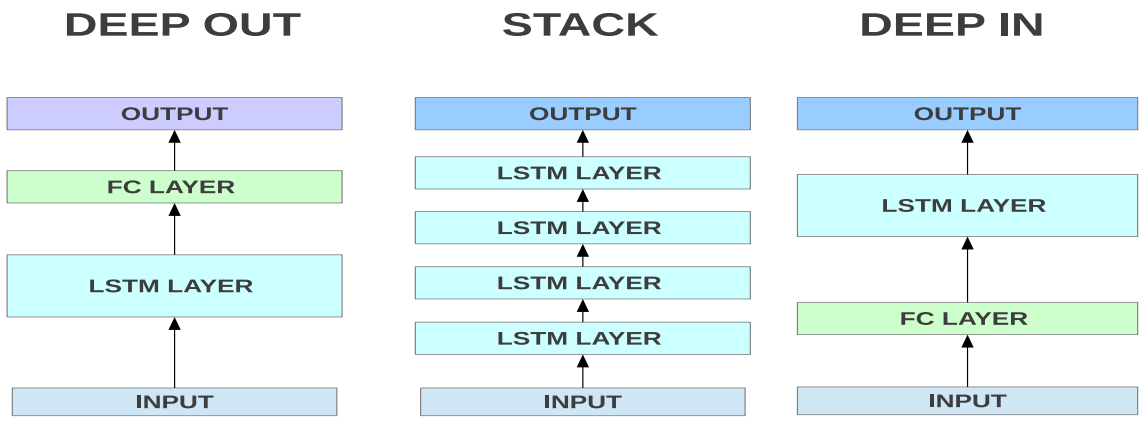


Figure 2.10: The deep LSTM neural network easiest to build

2.3 Neural networks as path planning agents

It is important to recall here that a strong hypothesis of the problem is that the proposed model operates in unknown environment, in Section 1.2.3 it has been addressed as a distinctive characteristic of online search.
So, in this thesis, neural networks will be evaluated to work as this kind of agent.
Previous sections were dedicated to generic models, from now on will be described neural networks to map input states into actions in order to assume the path planning agent role. Following Figure 2.12, with a very shallow but essential sequence of concepts I can motivate why they can be applied, to tackle the exploration problem as a method to make the agent learn by experience during the training process.
The multilayer perceptron has been successfully applied in similar application [34] [35] [36] [37], meanwhile recurrent neural networks take full advantage of one simple concept, showed in Figure 2.11, instead of learning state by state independently, I want to solve a sequence of states to a sequence of actions problem.
The main reason is because the sequence to sequence problem can be referred also to, at first glance, really different applications like speech recognition, handwriting recognition, machine translation, video captioning and so on; where the recurrent neural network family has obtained state of the art performance [38] [39] [31] [19]. Furthermore, the LSTM neural network has already been employed in robots path planning [40] [41] [42].

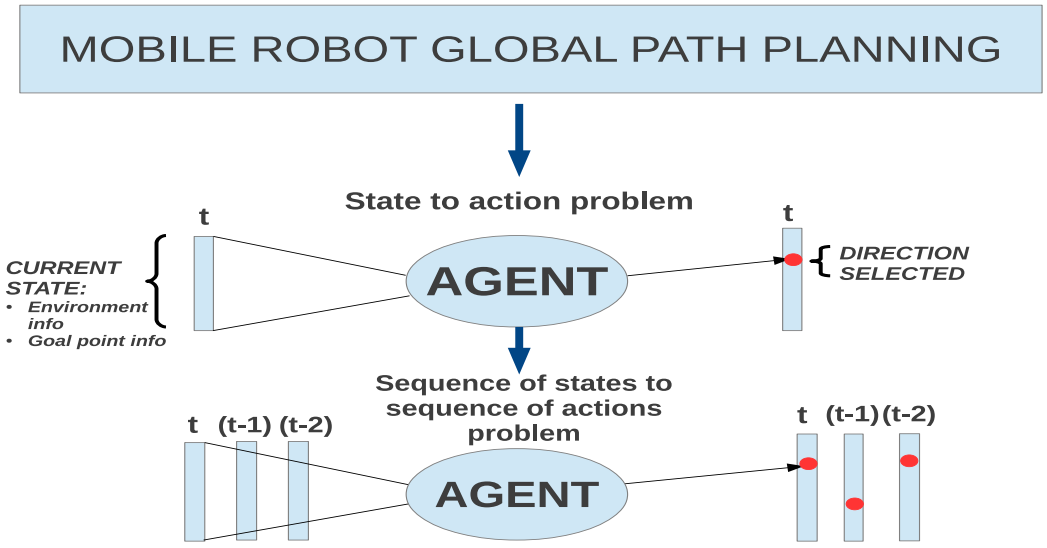


Figure 2.11: A sequence to sequence problem

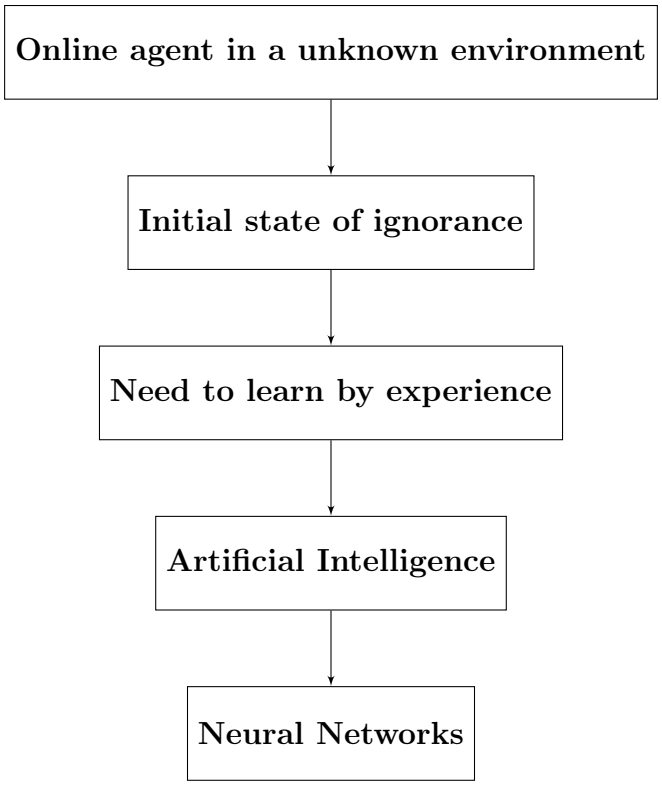


Figure 2.12: From the principle to neural networks

Both *supervised learning* and *reinforcement learning* have been applied to solve this kind of problem. They build completely different neural network configuration but they both are special applications of the decision theory.
In its simplest form, decision theory deals with choosing among actions based on the desirability of the immediate outcomes of a specific *utility function* associated to the agent. Hence, the overall neural network can be seen as a utility function $U(s)$ which generates the outcomes as preferences for each action allowed, these are values to express the desirability of the next state s to which the action will take the agent.
Subsequently, the *expected utility* $EU(a|e)$ is the average utility value of the outcomes in terms of probabilities given the input observation e. Eventually, the principle of *maximum expected utility* (MEU) says that an agent should choose the action that maximizes the agent's expected utility [43]:

$$a_{next} = \underset{a}{\operatorname{argmax}} \left(EU(a|e)\right) \tag{2.26}$$

2.3.1 Supervised learning approach

Among the many variants, I describe here in details the most common and simple configuration, implemented in this research as well.
Considering the reference frame centred in the current position of the robot, the input state is a flattened vector comprising values dedicated to surrounding environment and goal point informations. The former could be a series of proximity ranges to obstacles or a grid map assuming a binary variable for each pixel,for example "1" stands for obstacle and "0" for free space; the latter could be just an heuristic distance estimation from current to goal position like the variants described in Section 1.3.1. However, adding the relative angle calculated with *arctan2*() is useful to build a univocal association between current position of the robot and goal point, because a situation where two points have same distance to the goal and same surrounding environment but different relative angle is likely to occur, specially when complex maps and long paths are taken into account like in Chapter 3.
On the other side, the output is the choice of action calculated by the neural network. Regardless of the neural network structure type, the output layer is a full connection type - like the one showed in Figure 2.3 and Figure 2.8 - with K units as many as the total number of allowed actions as being at this point a K-dimensional classification problem and each unit value has the following expression:

$$y_k = \mathbf{w}_k^T \mathbf{z} + w_{k0} \quad \forall k = 1, ..., K \tag{2.27}$$

where $\mathbf{z} \in H$ is a the vector of the unit values of the last hidden layer composed by H units, $\mathbf{w}_k \in H$ is the vector of the weights related to the k-th output unit and w_{k0} is the bias term. The values y_k are quantities of preference that the network associates to each action a_k like the outcomes of the utility function, the posterior probabilities $p(a_k|y_k)$ are calculated thanks to the *softmax* transformation:

$$p(a_k|y_k) = \phi_k(\mathbf{y}) = \frac{\exp(y_k)}{\sum_{j=1}^{K} \exp(y_j)} \forall k = 1, ..., K \tag{2.28}$$

where $\mathbf{y}$ is the K-dimensional vector containing all the outcomes y_k.
Hence, considering the results from equations above, the action with highest probability represents what the neural network calculates as most right, therefore d correspond to the action taken:

$$d = a_k \text{ s.t. } k = \underset{k}{\text{argmax}} \left(\phi_k(\mathbf{y})\right) \text{ with } k \in [1, K] \tag{2.29}$$

In Figure 2.13 a summarizing representation of the configuration described is showed. It is important to have also clear how the error in a multiclass classification problem is computed. Consider a batch of samples with size N, for each sample, the ground truth is a K-dimensional vector of binary values $\mathbf{t}_n$ where just the value t_{nk} related to the right class assume value equals one, while the others are set to zero. The objective of the neural network is to maximize the likelihood function, notation in Eq. 2.27 is used:

$$p\left(\mathbf{T}|\mathbf{w}_1 ... \mathbf{w}_K\right) = \prod_{n=1}^{N} \prod_{k=1}^{K} p\left(a_k|\mathbf{z}_n\right) = \prod_{n=1}^{N} \prod_{k=1}^{K} \phi_{nk} \tag{2.30}$$

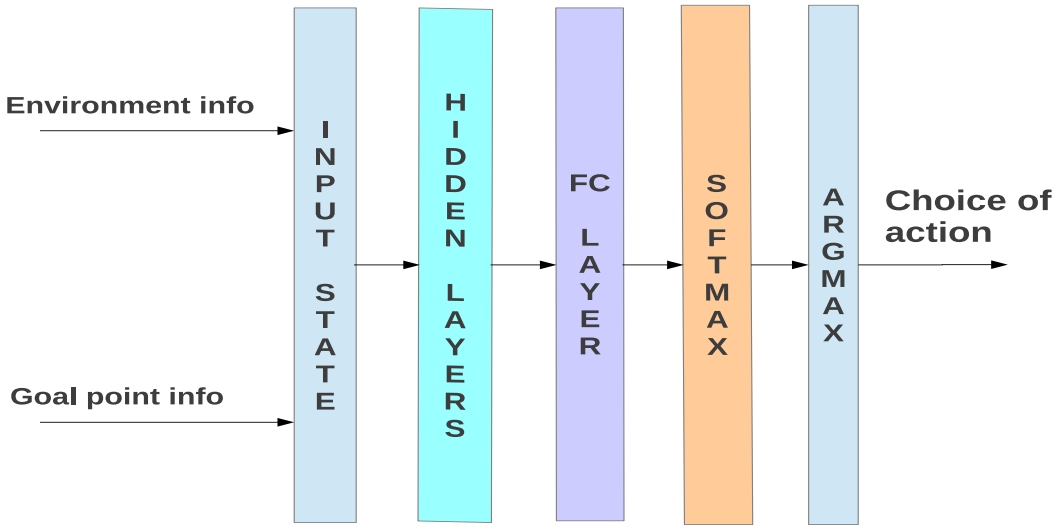


Figure 2.13: General configuration for supervised learning approach

where $\phi_{nk} = \phi_k(\mathbf{y}_n)$ and $\mathbf{T} \in N \times K$ is the matrix of target variables with elements t_{nk}. Maximizing the likelihood function correspond to minimize the negative logarithm:

$$E\left(\mathbf{w}_1, ..., \mathbf{w}_K\right) = -\ln\left(p\left(\mathbf{T}|\mathbf{w}_1 ... \mathbf{w}_K\right)\right) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln\left(\phi_{nk}\right) \tag{2.31}$$

this expression is known as the *cross-entropy* error function for the multiclass classification problem. Eventually, the rule to update the output layer weights needs the gradient of the error with respect to the vector $\mathbf{w}_j \; \forall \, j = 1, ..., K$:

$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, ..., \mathbf{w}_K) = \sum_{n=1}^{N} \left(\phi_{nj} - t_{nj}\right) \phi_n \tag{2.32}$$

where ϕ_n is the K-dimensional vector of posterior probabilities evaluated for the n-th sample. Eventually, the weights update at timestep t has the following expression:

$$\mathbf{w}_j^{(t+1)} = \mathbf{w}_j^{(t)} - \alpha \, \nabla_{\mathbf{w}_j} E(\mathbf{w}_1, ..., \mathbf{w}_K) \tag{2.33}$$

where α is the learning rate [44]. All the other internal weights are updated following the backpropagation formula introduced in Section 2.1.
This approach can implement an offline training process, faster and easier to manage, datasets are realized running the the supervisor algorithm as a path planner, then the neural is trained, eventually it is deployed on the robot for the online testing.
On the downside, it presents an important limit: experienced pairs state-action during training are limited to the ones decided by the supervisor, so it leads to a more probable wrong decision during effective deployment when the neural network agent gets itself in a state not experienced in the training process.

2.3.2 Reinforcement learning approach

This approach is an example of how an agent can learn what is the best decision in absence of labelled samples. During the training process, a supervised learning agent needs to be told the correct move for each step, so when its decisions are compared with the ground truth provided by a supervisor, a feedback can be built in terms of error to be minimized, eventually depending on the feedback value the internal weights are corrected.
Without a supervisor, the agent has no grounds for deciding which move is better to make, but after taking a move it can be evaluated if the outcome that is the next position is good or bad. This kind of feedback is called *reward*, or *reinforcement*.
The concept of rewards were first introduced in Markov decision processes (MDPs). The task of reinforcement learning is to use observed rewards to learn an optimal, or suboptimal, next policy for the current environment [45].
The most applied agent of this kind is the Q-learning agent that learns an action-utility function, called Q-function as well. Consider a total number of K actions allowed, the notation $Q(s, a_k)$ represents the value of doing an action a_k while standing in state s, Q-values are directly related to utility values $U(s)$ as follows:

$$U(s) = \max_{a_k} Q\left(s, a_k\right) \text{ with } k \in [1, K] \tag{2.34}$$

A highest Q-value indicates the action a_k is judged to yield the best result with the agent currently being in a state s.
The Q-values are updated via the *Bellman equation* [46]:

$$Q\left(s, a_k\right) = Q\left(s, a_k\right) + \alpha \left(R(s) + \gamma \max_{a_k} Q\left(s', a'_k\right) - Q\left(s, a_k\right) \right) \tag{2.35}$$

where α is the learning rate and γ is the discount rate, usually both tuned manually. The term $\max_{a'} Q\left(s', a'_k\right)$ outputs the maximum Q-value over all possible successive actions a'_k in the resulting state s' and $R(s)$ is the reward function that has a particular expression for every application defined by its designer. For example, in a path planning problem the expression below has been proposed [36]:

$$R(s) = -\sum_{r=1}^{R} \tau_i r_i \; + \tau_d d_{ag} + \tau_\theta \theta_{ag} \tag{2.36}$$

where r_i are the proximity ranges to obstacles, d_{ag} is the heuristic distance from current agent position to the goal point and θ_{ag} is the relative angle between agent and goal position. The respectively related weights τ are manually tuned and are sort of influence scales the designer wants to associates to each value inside the reward function.
As a conclusion, considering a particular input state s, the Q-learning agent selects the choice of action d as the one associated to the highest Q-value:

$$d = a_k \;\; \text{s.t.} \;\; k = \underset{k}{\operatorname{argmax}}\, Q\left(s^{(t)}, a_k\right) \text{ with } k \in [1, K] \tag{2.37}$$

A neural network can be used to estimate the Q-values, it would be a model parametrized by weights and biases, collectively called learnable parameters and denoted with $\mathbf{w}$. Q-values are estimated online by querying the output units of the network after performing

a forward pass given a state input s. Such Q-values are denoted $Q(s, a|\mathbf{w})$. Instead of updating individual Q-values, updates are now made to the parameters of the network to minimize a differentiable loss function $L(s, a_k|\mathbf{w})$ [47]:

$$L(s, a_k|\mathbf{w}^{(t)}) = \left(R + \gamma \max_{a_k} Q\left(s', a_k|\mathbf{w}^{(t)}\right) - Q\left(s, a_k|\mathbf{w}^{(t)}\right) \right)^2 \tag{2.38}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \nabla_{\mathbf{w}} L\left(\mathbf{w}^{(t)}\right) \tag{2.39}$$

The first advantage of this learning approach implements a model-free optimization, the agent does not train depending on an entity labelling which action is correct, which also means that it is not affected to restraints and disadvantages of a particular supervisor selected in the previous approach, this represents a major flexibility advantage.
Furthermore, recent works on reinforcement learning with deep neural networks, widely known as *deep reinforcement learning*, have shown outstanding performance in decision making problems so that they talk about reaching equal performance or even beating human control level in simulation environments concerning generic arcade games [48] [47]. Very recently, deep reinforcement learning has been applied to mobile robot navigation [49].
The training process can be difficult to perform for a mobile robot in unknown environments: it has to be online and Q-values update has to occur - normally - after every single iteration. Indeed, before reaching convergence, the output calculated by a neural network modelling a Q-learning agent is known to be oscillating or diverging in the worst case [50].
I would admit that the overall complexity of reinforcement learning in both training the agent and effective implementation restrained me in selecting this approach for the model proposed, however it is also my opinion that, given recent researches results, reinforcement learning can be a real valid player for path planning more than supervised-based methods and maybe better than traditional but still competitive heuristic algorithms, two of which described in Section 1.3.

Chapter 3

Simulation

3.1 The ASCII maps environment

The simulation environment takes full utilization of an already built and available collection of maps realized with ASCII characters, along with each map there is a predefined list of problems specifics in terms starting point, goal point and optimal length [51].
This collection is really helpful for such kind of researches, it is mainly used to test every kind of global path planning algorithm, offline search in general, furthermore official competitions are held in this environment to evaluate the best one.
A fundamental rule set for these maps is that the dot character "." means free space, while all the other ones are labelled as obstacles, it is also important that the simulation environment developed deals only with static maps.
The categories taken into account in this section are the following:

1. discretized from videogames
2. mazes
3. random filled maps

The three paragraphs below aim to explain to the reader some details about the map categories listed above with examples to clear up any doubt about.

Maps discretized from videogames

The maps belonging to this category and used here are discretized versions of environments included in the Dragon Age videogame.
I would refer to these maps as the ones with largest open space regions, in its general meaning, but they also include regions with narrow corridors. Another feature is that they don't have fixed size, but it ranges from tens to thousands of steps, thus very little maps to very huge ones.
Figures 3.1 is showed to help the reader catch the description given above.

Figure 3.1: Example of a portion of a map taken from Dragon Age

Mazes

The mazes are the easiest to imagine, the general meaning is followed. An internal classification differentiates maps with respect to their corridor width measured in terms of pixels. Here are included maps having corridor width of 16, 8, 4 and 2 steps width, they all have size 512×512 steps. Figure 3.2 shows an example of a maze map.

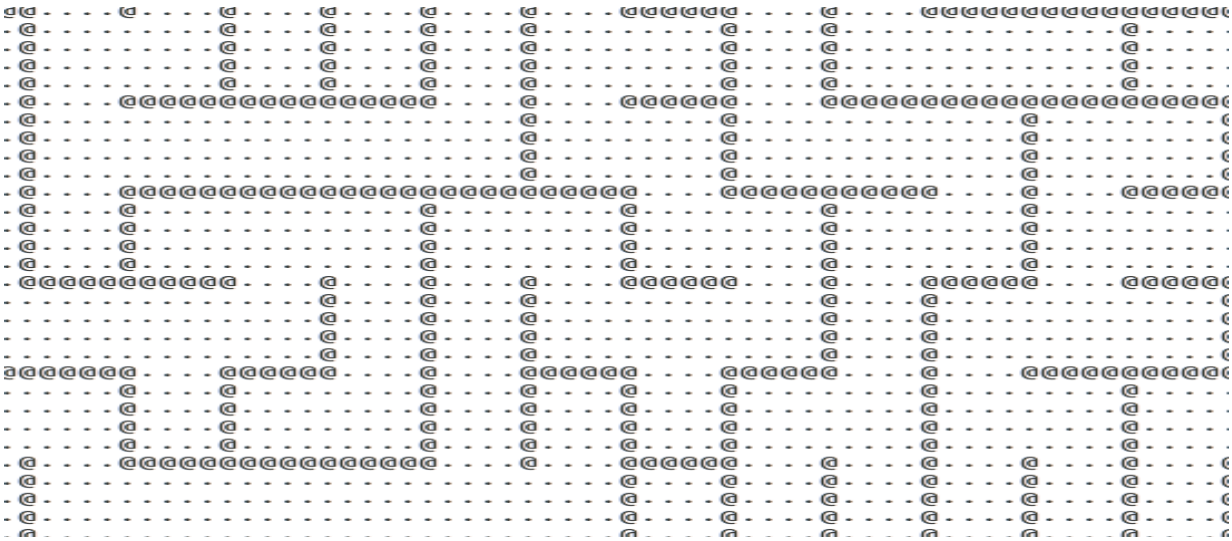


Figure 3.2: Portion of a maze map width corridor width 4 steps long

Random filled maps

The last category includes map where the characters standing as obstacles occupy a specified percentage of the overall map. In particular, I use maps having this specific parameter equal to 25 % and 40 %, like the mazes, all of them have dimensions 512×512 steps. Figure 3.3 illustrates an example.

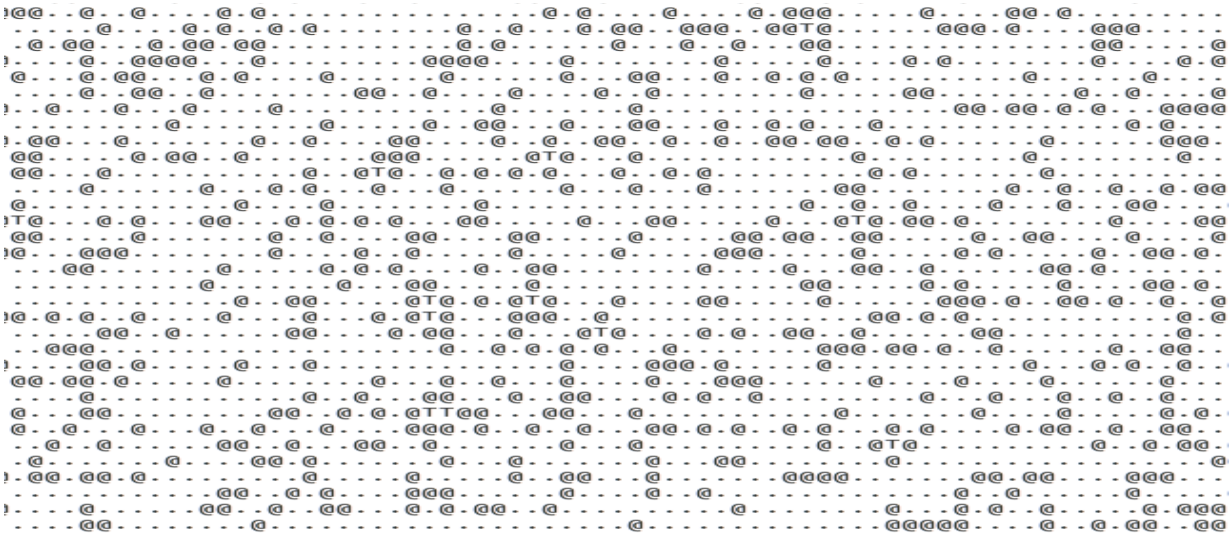


Figure 3.3: Portion of a random filled map for 25 % of its size

3.2 Training the neural network

At this point of the document, the reader should understand the difference between an online search agent and an offline search agent, Sections 1.2.3 and 1.2.2.
Every agent running in this environment is allowed to take diagonal and cartesian moves, in calculating the path cost, the scale parameter associated to the former is equal to $\sqrt{2}$, while the latter movements scale cost parameter is 1.
A supervised learning approach is selected and the ground truth data is provided by the conventional A* algorithm, Section 1.3.1, an offline agent. The only important detail to be mentioned about A* is that euclidean distance with a scale parameter equal to 25 is found to work better than any other configurations and so it is selected as heuristic distance function $h(\bullet)$.
So, the policy here assumed is: training the neural network with offline search related ground truth and then testing as online agent.
This is a really important design choice that influence the overall results described in the next section.

Here are listed the main reasons of this choice:

- offline search is easier, it gives a wide freedom in building any kind of training dataset
- online agents do not, generally, give optimal solutions and are harder to correctly implement
- teaching the neural network with optimal, or suboptimal, solutions

In line with the hypothesis of unknown environment, the input state of the neural network has to be defined. The grid map is a usual solution for obstacle awareness, it consists on occupancy squares labelling a certain region of the space as occupied by obstacles or free space, here in this simulation environment the latter is a pixel occupied by the dot character ".", while all the other characters refers to the former. However, in the real environment the grid map is built from sensor readings like proximity ranges from a local range finder (LRF) by appropriate algorithms included in the SLAM activities.
It was found that for obstacle awareness using directly ranges to obstacle improves performance with respect to a grid map [36], in addition they mean a less computational step in the real environment applications and such shallow environment perceiving increases the low resource usage by the proposed method. In ASCII maps, ranges considered are just in cartesian and diagonal directions and are calculated as number of pixels $n_{p_{A\to O}}$ from current agent position A to first obstacle encountered O in a particular direction with a scale parameter:

$$range_{\text{cartesian}} = 1 \cdot n_{p_{A\to O}} \qquad range_{\text{diagonal}} = \sqrt{2} \cdot n_{p_{A\to O}} \tag{3.1}$$

as a design choice, the maximum number of pixels considered in a single calculation is 50. Relative angle and euclidean distance between agent and target position are added to build the state. Furthermore, I noted that adding the previous directions taken into the current state helps the network to improve predictions.
Consider current agent position at coordinates (x_A, y_A) and goal position at coordinates (x_G, y_G), the standard input state input state components are defined below and represented in Figure 3.4:

- $d_i \; \forall \; i = 1, ..., 8 :=$ proximity ranges
- $\theta_{A\to G} = arctan\left(\frac{y_G - y_A}{x_G - x_A}\right) :=$ relative angle
- $e_{A\to G} = \sqrt{(y_G - y_A)^2 + (x_G - x_A)^2} :=$ euclidean distance

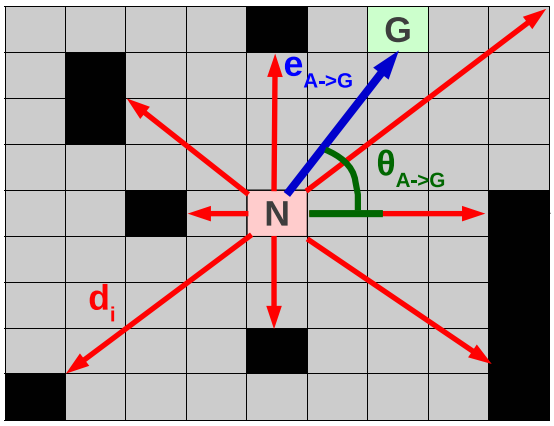


Figure 3.4: Building the input state

Meanwhile the available actions set is a 8-dimensional vector: four cartesian directions plus four diagonal directions. Recalling Section 2.3.1, the model calculates eight posterior probabilities a_k and selects the next direction d as the one having the highest value:

$$y_k \ \forall k = 1, ..., 8 \ := \ \text{eight values from the output layer} \tag{3.2}$$

$$p(a_k|y_k) = \phi_k(\mathbf{y}) = \frac{\exp(y_k)}{\sum_{j=1}^{K} \exp(y_j)} \forall k = 1, ..., 8 \tag{3.3}$$

$$d = a_k \ \text{ s.t. } \ k = \ (\phi_k(\mathbf{y})) \text{ with } k \in [1, 8] \tag{3.4}$$

Figure 3.5 is a graphical representation of the sequence of the operations to calculate the decision.

The cross-entropy error introduced in the previous chapter is essential to update the learnable parameters, however it is not suitable to evaluate the learning performance, specially because it is not "human readable" in the sense that it is not a direct relationship between direction chosen by the neural network and the correct direction provided by the supervisor.

Instead, accuracy is widely used for a classification problem, consider a batch of samples with size N, for each sample the ground truth, also called label data, is denoted with l_n and it refers to one of the eight allowed directions ($l_n = a_k$ with $k \in [1, 8]$). The n-th decision of the neural network is calculated as in Eq. 3.4 and here denoted with d_n, the accuracy related to the batch of samples supposed is computed as follows:

$$A_N = \frac{1}{N} \sum_{n=1}^{N} \delta\{d_n = l_n\} \tag{3.5}$$

$$\text{where } \delta\{\text{condition}\} = \begin{cases} 1 \text{ if condition} \\ 0 \text{ if } \neg \text{ condition} \end{cases} \tag{3.6}$$

The word *batch* is always used to indicate for how many samples the error is accumulated before taking the backward pass described in Sections 2.1 and 2.2.

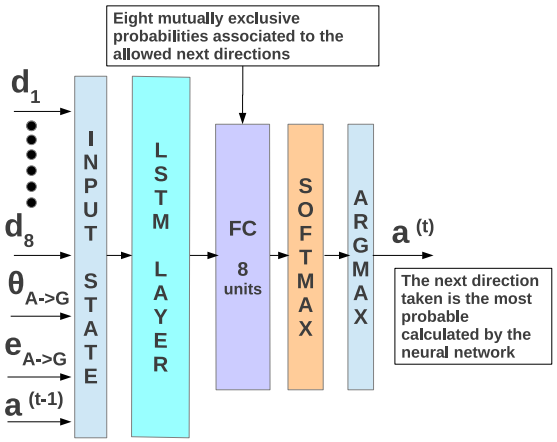


Figure 3.5: Operations from input to output

The details of how the training is performed are important to understand why there is a validation set and a section dedicated to online testing.
The general learning process of a neural network consists on working with two sets: the *training* set and the *validation* set. The former is used to effectively train the neural network, the cross-entropy error is calculated for every batch and the backward pass is computed in order to correct the weights, in other words the neural network builds its experience in the training set. Meanwhile, the latter is used as a test set, just the forward pass is computed for the batches analyzed in this set, the purpose is to verify if the neural network that is repeatedly updating the learnable parameters, has a successful performance also in samples not included in the training set.
However large it may be, the training set is not supposed to include every single sample the neural network may experience when it will be deployed, the hope and the basic principle is that learning allows to take the right decision even in situations not directly experienced.
If the performance is high in the training set and the low in the validation set, the neural network is *overfitting* or *memorizing* the former, it is learning to be perfect among the training samples but the validation set is considered as extraneous.
There could be many reasons causing overfitting, but in general the most important and most widely occurring are low training set size and neural network structure too articulated in terms of number of layers and units per layer. The best behaviour every designer aim for is achieving the highest possible accuracy value in both sets, it is called *generalization* to denote that the neural network has found the right weight values to take the correct decision for generic cases, not only the ones included in the training set and actively experienced.
A usual size ratio between the two sets is 10 to 1, which is used also in this section to build the proposed model.
Both training and validation sets are made of step samples coming from solution paths found by the A* algorithm, in addition to the entire map, the supervisor receives only the informations about starting point coordinates and goal point coordinates, picked randomly from the predefined list attached to each map.

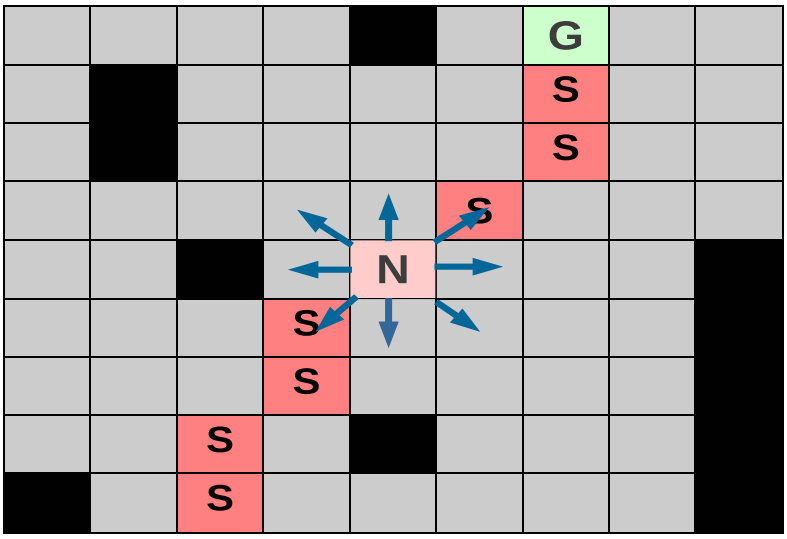


Figure 3.6: Neural network agent behaviour during the training process

During the training process, suppose the neural network is in the starting point of a new path, it senses the local environment tu build to input state, calculates a decision and eventually the cross-entropy error is computed. The next point is not the one related to the direction decided by the neural network, but it is the first point of the solution path found by the A* algorithm, an analogous procedure occurs for every sample. Figure 3.6 shows that the neural network agent, denoted with "N", has eight available direction and it travels following the coordinates included in the path found by the supervisor, denoted with "S".
In other simpler words, it is like the A* algorithm takes by the hand the neural network agent for the whole path, the latter can calculate its own decisions but the direction effectively taken is always chosen by the former. As the training process is implemented, this happens also in testing the agent in the validation set, the only difference is that there is not any calculation regarding cross-entropy error and weights update.
This last concept explained is the reason why in the validation set the neural network agent is not evaluated as a standalone agent, therefore, the next section is dedicated to the effective deployment of the neural network as an online search agent.

In Table 3.1, specifics of the training set and validation set assumed here as standard for the learning process are showed, maps in the training set are different with respect to the validation set but they are picked from the same categories: mazes with corridor having 4 steps width, mazes with corridor having 2 steps width, random filled for 25 % map size, random filled for 40 % of the map size. The paths having length higher than the maximum path length parameter are not included in the set, it can be considered as a sort of maximum complexity limit.

Specific	TRAINING SET	VALIDATION SET
Size (steps)	102400	10240
Number of maps	5	3
Maximum path length	300	300

Table 3.1: Training set and validation set specifics

Among the recurrent network family, the Long Short-Term Memory neural network is chosen over the basic RNN because of its advantage to learn features with longer time lags. The final model consist on a LSTM neural network, the procedure of shaping the structure will be explained in each step. One of the thesis purpose is to show that it can be applied as a valid agent, so the first thing to prove will be comparing learning performances with respect to the best multilayer perceptron (MLP) structure because it is one of the most applied neural network for path planning.
Regarding the training of the LSTM network, an important assumption is that weights are updated after sweeping a single time sequence.
Every test that will be presented from now on is conducted using the CAFFE framework (Convolutional Architecture for Fast Feature Embedding) [52] and the GTX 1080Ti is the graphic board unit used for calculations [53].
The hyperparameters final configuration to train the LSTM neural network is showed in Table 3.2. Except for the time sequence and the learning rate, all the other parameters were not previously introduced, since they are beyond the purpose to explain a thesis regarding neural networks to a reader having a beginner level knowledge on the argument. However, I will briefly introduce these parameters here with related reference, in case a deepening is desired.

Hyperparameter	Value
learning rate (fixed)	0.1
weight decay rate	10^{-5}
regularization type	L_1
momentum rate	0.95
epochs	500

Table 3.2: Hyperparameters configuration for tuning the LSTM network structure

First of all, an important trade-off about the learning rate has to be mentioned. It can be seen as a step size of the error minimization process along the gradient direction toward the minimum. If the step size is too large, it can "jump over" the minima we are trying to reach, this can lead to osculations around the minimum or in some cases to outright divergence. On the opposite, an higher step size can speed up the training attaining the

same accuracy value but in less time, this can be very important when deep neural networks are involved in the particular project because they may need several hours before reaching convergence. It is very rare see applications with learning rates equal or higher than one, in training the LSTM neural network for this application, a value equal to 0.1 showed to lead to the highest accuracy and to best reduce the training time.
Often the number of weights is used as a measure of the neural network *complexity*, when it is too high and the training set is not sufficient large and various, overfitting is likely to happen. Instead of reducing the complexity, an option is to limit the growth of the weights through regularization techniques like *weight decay* [54]. It should prevent the weights from growing too much unless it is necessary, it is realized by adding a term to the error function $E(\mathbf{w})$ that penalizes the weights:

$$E(\mathbf{w}) = E_0(\mathbf{w}) + \beta\, L(\mathbf{w}) \tag{3.7}$$

where $E_0(\mathbf{w})$ is the cross-entropy error function, $\mathbf{w}$ denote a vector of all the weights of the neural network, $L(\mathbf{w})$ is the weights penalization function and β is the weight decay rate. In the CAFFE framework, the $L(\mathbf{w})$ is either the L_1 or the L_2 normalization functions:

$$L_1(\mathbf{w}) = \sum_{i=1}^{W} |w_i| \tag{3.8}$$

$$L_2(\mathbf{w}) = \sqrt{\sum_{i=1}^{W} w_i^2} \tag{3.9}$$

where W is the total number of weights. In literature it is easy to find different expression for L, one above all is the one below:

$$L_{other}(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{W} w_i^2 \tag{3.10}$$

With the new term added in Eq. 3.7, the weight update rule for a generic weight w has a new expression:

$$w^{(t+1)} = w^{(t)} - \alpha \left(\frac{\partial E\left(\mathbf{w}^{(t)}\right)}{\partial w^{(t)}} - \beta \right) \qquad L_1 \text{ case} \tag{3.11}$$

$$w^{(t+1)} = w^{(t)} - \alpha \left(\frac{\partial E\left(\mathbf{w}^{(t)}\right)}{\partial w^{(t)}} - \beta\, w \cdot \frac{1}{\sqrt{L_2(\mathbf{w})}} \right) \qquad L_2 \text{ case} \tag{3.12}$$

The momentum method is a technique for accelerating gradient descent that accumulates a velocity vector $\mathbf{v}$, same dimension as $\mathbf{w}$, in direction of persistent reduction of the error function [55]. Given the error function in 3.7 to be minimized, the classical momentum method modifies the general weight update rule [56]:

$$\mathbf{v}^{(t+1)} = \mu \mathbf{v}^{(t)} - \alpha\, \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(t)}\right) \tag{3.13}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t+1)} + \mathbf{v}^{(t+1)} \tag{3.14}$$

where α is the learning rate and $\mu \in [0, 1]$ is the momentum rate. The momentum is also used to increase the size of the steps to avoid or however get out from local minima. Usually, the values most widely adopted are in the range $[0.9, 1)$.
Eventually, sweeping an epoch means every sample comprised in the training and validation sets is analyzed one time.
A fundamental rule drives the LSTM network structure shaping procedure: if it shows prediction improvements it is better to have structure deeper through the temporal dimension than through adding layers or cells. In different but maybe simpler words, this is related to the fact that recurrent network family shares the parameters through time, so, for example, suppose two networks, one having one more layer and a with respect to the other and they have the same prediction performance, in this case the one with less layers has less learnable parameters and so a lighter structure which means a faster computation of the forward pass. This atomic operation is extremely important because it is what the network will execute during online testing.
From now on the only parameter taken into account to evaluate the learning performance will be the accuracy achieved in the validation set, since it is the essential parameter to assess the generalization level of a generic neural network and in every test reached comparable value with respect to the accuracy evaluated in the training set, where for comparable I mean:

$$Accuracy_{train} - 2\% < Accuracy_{validation} < Accuracy_{train} \tag{3.15}$$

thus, no overfitting was experienced.
The first step of the procedure consists in taking one layer with a minimal number of cells equal to four and compare the learning performance with different time sequence lengths, in particular the following values are considered:

$$\mathbf{T}_{seq} = [128, 256, 512, 1024] \tag{3.16}$$

Figure 3.7 gives a graphical interpretation of the results.
So, the first conclusion is to proceed with the two highest values in $\mathbf{T}_{seq}$. Procedure starts now to get two directions, adding two more cells in the existing and only one hidden layer or stacking a new LSTM layer with two cells above the existing one. These tests form the second stage, results are summarized in Table 3.3.

	Time sequence = 512 *steps*	Time sequence = 1024 *steps*
1 LSTM layer - 4 cells	84.02 %	82.11 %
2 LSTM layers - 2 cells	76.71 %	75.52 %

Table 3.3: Validation accuracies of the second stage of the procedure

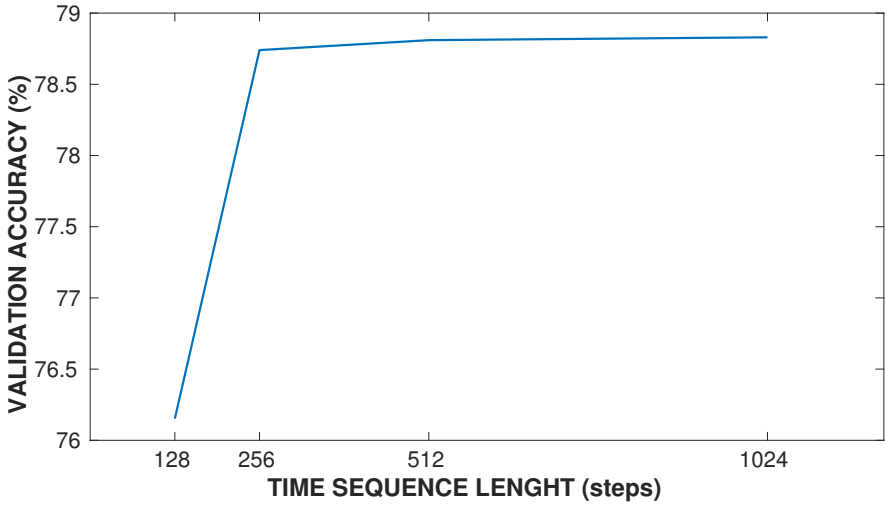


Figure 3.7: Increasing the time sequence with one layer and 4 "memory" cells

The next third stage is very similar to the previous one, but now the starting point is the structure referred to the first row of Table 3.4: meaning that it will consist on comparing one layer with 8 cells and 2 layers with 4 cells each one.

	Time sequence = 512 *steps*	Time sequence = 1024 *steps*
1 LSTM layer - 8 cells	87.50 %	86.94 %
2 LSTM layers - 4 cells	84.56%	85.90 %

Table 3.4: Validation accuracies of the third stage of the procedure

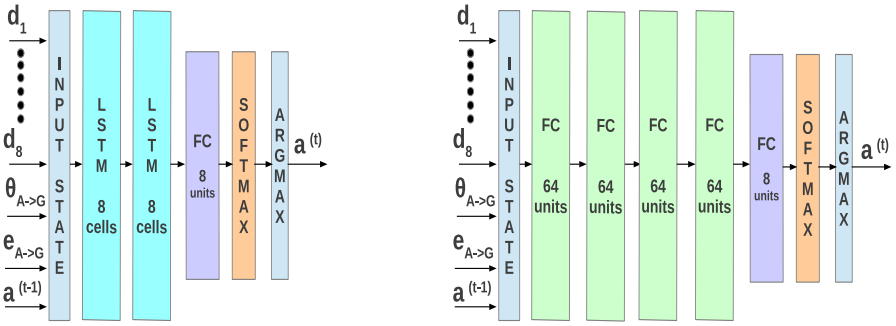
Keeping on following the same guideline also for a fourth stage, but in Table 3.5 it can be clearly seen that improvements are not so really higher than the third stage.

	Time sequence = 512 *steps*	Time sequence = 1024 *steps*
1 LSTM layer - 16 cells	88.37 %	88.47 %
2 LSTM layers - 8 cells	88.60 %	88.45 %

Table 3.5: Validation accuracies of the fourth stage of the procedure

Going over again brought just some minor changes in the first digit after the decimal point. So, I concluded that a convergence to these performance numbers has been set up and the best model based on LSTM memory cells is the one represented in Figure 3.8a because it has less weights compared to the other structure analyzed in the last fourth stage.
A very similar procedure has been applied to tune the best MLP structure, represented in Figure 3.8b: a very close result with respect to the best LSTM is found, precisely a

validation accuracy equal to 88.43 %. The important difference is that there is no time sequence parameter in training the a multi-layer perceptron, but the weights are update after sweeping a batch of samples. In this case, the best batch size value found was 512 samples.
The conclusion of this section is that a the final LSTM structure has achieved comparable prediction results with respect to the MLP with a lighter structure, this is considered enough to state that the former is more performs better than the latter, the proof is provided by calculating ratio regarding the total number of weights between the final MLP model and the proposed model based on LSTM.



(a) The final LSTM structure (b) The final MLP structure

Figure 3.8: The best models found the two neural network families treated in this thesis

Referring to Section 2.1, the number of weights n^l_{wFC} in a single full connection layer l is calculated below:

$$n^l_{wFC} = n^l_{units} \cdot \left(n^{(l-1)}_{units} + 1\right) \tag{3.17}$$

where n^l_{units} is the number of units in the l-th layer and $n^{(l-1)}_{units}$ in the $(l-1)$-th layer. Recalling that input layer consists on eleven values, the total number of learnable parameters comprised in the structure of Figure 3.8b is denoted with n^{MLP}_{tot} and equals to the following summation:

$$\begin{aligned} n^{MLP}_{tot} =& 64 \cdot (11+1) \; + && \text{weights 1st hidden FC layer} \\ & 64 \cdot (64+1) \; + && \text{weights 2nd hidden FC layer} \\ & 64 \cdot (64+1) \; + && \text{weights 3rd hidden FC layer} \\ & 64 \cdot (64+1) \; + && \text{weights 4th hidden FC layer} \\ & 8 \cdot (64+1) && \text{weights output FC layer} \\ =& 13768 \end{aligned}$$

Referring to Section 2.2.1, the calculation of total number of weights n^l_{wLSTM} for a LSTM

layer is a little more articulated:

$$n^l_{wLSTM} = n^l_{cells}\left[3 \cdot (1 + n^l_{cells} + n^{(l-1)}_{cells}) + (1 + n^l_{cells} + n^{(l-1)}_{cells})\right] \tag{3.18}$$

where the 1st term is dedicated to the three variable gates and the 2nd to the input value of the cell multiplying the input gate. The total number of weights of the proposed model in Figure 3.8a can now calculated:

$$\begin{aligned} n^{MLP}_{tot} =& 8 \cdot [3 \cdot (1 + 8 + 11) + (1 + 8 + 11)] + \quad \text{weights 1st hidden LSTM layer} && (3.19)\\ & 8 \cdot [3 \cdot (1 + 8 + 8) + (1 + 8 + 8)] + \quad \text{weights 2nd hidden LSTM layer} && (3.20)\\ & 8 \cdot (8 + 1) \quad \text{weights output FC layer} && (3.21)\\ =& 1256 && (3.22) \end{aligned}$$

Eventually the ratio gives:

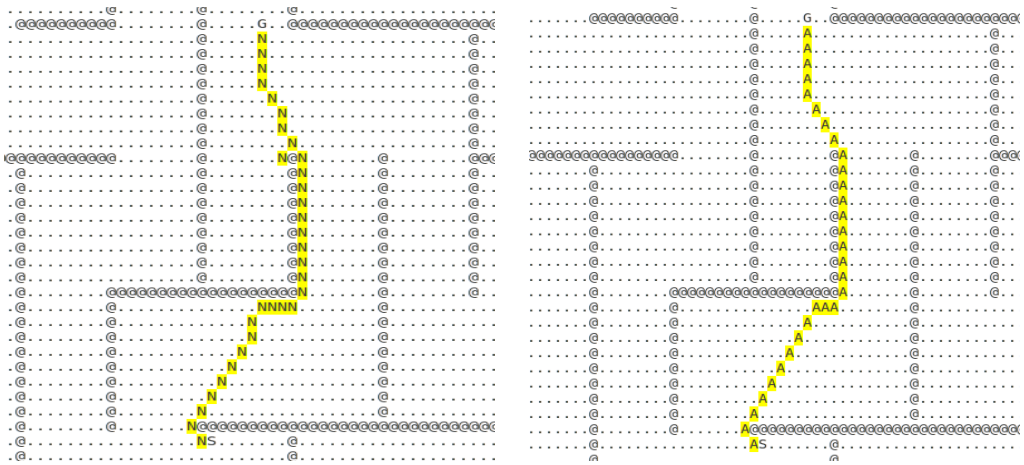
$$\frac{\text{tot. number of weights final MLP model}}{\text{tot. number of weights final LSTM model}} = \frac{13768}{1256} = 10.96 \tag{3.23}$$

the best MLP structure turns out to be more than ten times heavier than the LSTM structure proposed. This means that a sort of initial competition among the neural network based agents is won by the LSTM model that in next section will try to challenge the A* algorithm.

3.3 Online testing

Online testing is taken in map categories listed in 3.1. Before dealing with the performance analysis and comparison with the A* algorithm, I want to show an example of solving a path included or not in the training set by the proposed agent deployed in online search. The purpose is to demonstrate the different behaviour of the agent between cases actively experienced and never encountered during training. In order to show the whole path found from start to goal point, just simple examples will be showed.
The first one, Figure 3.9 shows the most complex path of this series, it consist on an example included in the training set, so the agent had the chance to repeatedly practice on this path, make mistakes, accumulate errors and correct the wrong moves occurring less and less until training convergence was attained. Eventually, it is evident that the solution found is quite similar to the one considered correct because it was found by the supervisor algorithm.
Figures 3.10 and 3.11 show examples of attempts to solve a path which was not included in the training set. In the first example, the LSTM agent finds a solution because the goal point is reached but the path is far from being similar to the one realized by A*, a sort of diverging towards the north direction is realized before tracing back and finding an acceptable pathway. The reason is because there may have been similar paths included in the training set, so the experience on these allowed the agent to try different movements to escape from the wrong positions reached at some point.

Meanwhile, the second figure shows a failure case caused by obstacle hit, in particular this is selected because it represents objectively an easy problem, but still the agent did not recognise the obstacle. The result of this attempt means that similar cases were not experienced during training, this conclusion arise a quite bad problem because such easy cases cannot be unsolved while lot more complex paths are found. Therefore, this last example leads to include a better composition of the training set in the future works list.

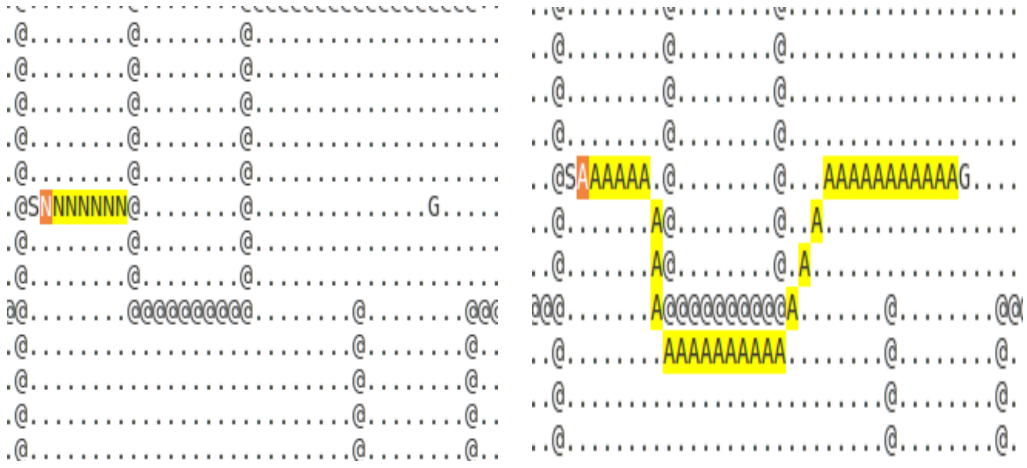


(a) Path found by the LSTM agent (b) Path found by A*

Figure 3.9: Comparing the solution of a solved case included in the training set

(a) Path found by the LSTM agent (b) Path found by A*

Figure 3.10: Comparing the solution of a solved case not included in the training set



(a) Path found by the LSTM agent (b) Path found by A*

Figure 3.11: Comparing the solution of a failed case by the LSTM agent not included in the training set

In order to better analyze the comparison between the proposed LSTM model and the A* algorithm, it is useful to recall the differences between each other.
The former is an online search agent, it can explore only nodes next to the current position, so more suitable for real world path planning because distance travelled is equal to the one explored, furthermore it can solve a path planning by storing just the current state into its memory, in other words the current input of the LSTM neural network.
Meanwhile the latter is an offline search agent, it examines the entire map stored in its memory and containing, at least, the supposed path from start to goal position. As a consequence it can expand nodes being in different part of the space explored, more than two actions away from each other. The essential distinction regards memory usage: in terms of states, the A* algorithm needs all the states corresponding to all the coordinates of the entire known map, meanwhile the proposed online agent stores in the memory just the current state. This is the main advantage of using online search agents. My conviction is that training a neural network with "memory", like the LSTM one, will help even in tests having this strong constraint of using just one sample at a time.
In Table 3.6, a summarizing parallel description of the two methods under performance comparison is presented. For the sake of precision, I report what here it is meant for "expanding a node" with respect to the two agents under comparison, the A* algorithm expands a node every time the following function is calculated for its adjacent nodes, in ASCII maps environment a total of eight adjacent nodes are taken into account:

$$f(n) = g(n) + h(n) \tag{3.24}$$

where $g(n)$ is the backward cost through the backpointers from the starting position to node n and $h(n)$ is the heuristic future cost that, in this study, is the euclidean distance from node n to the goal node, meanwhile the neural network expands a node one step at a time when a new state is built to be the the new input.

Property	A*	LSTM
Agent type	offline	online
Output nature	deterministic	stochastic
Environment	known	unknown
Solution calculation	The whole path	One step at a time
Node expansion allowance	more steps farther away	only the adjacents
Node expansion count increase	a node with minimal $f(n) = g(n) + h(n)$ is found	a new input state is built

Table 3.6: A properties and hypothesis comparison between the two path-planning agents

Each online test comprise a total amount of 50 paths picked from ten different maps, both selected randomly for each specific category. It is essential to state that none of the paths selected for online testing was included in the training set introduced in the previous section of this chapter, meaning that the neural network has not actively experienced these paths during the training process.
Tables 3.7, 3.8 and 3.9 show the results related to the categories below listed in the same order:

1. Dragon Age videogame
2. Mazes with corridor 8 steps long
3. Random with 25 % filler parameter

initial set up is to maintain the path length limit used in the training process, equal to 300 steps, the policy here is to lower this value in order to try to get better results if they are considered not acceptable.

Agent	Success Rate	Average time (s)	Average length	Average nodes expanded
A*	100 %	$3.37 \cdot 10^{-4}$	45.82	47.029
LSTM	68 %	$1.69 \cdot 10^{-3}$	53.39	44.41

Table 3.7: Online testing in Dragon Age maps

Agent	Success Rate	Average time (s)	Average length	Average nodes expanded
A*	100 %	$2.96 \cdot 10^{-4}$	50.90	53.69
LSTM	26 %	$1.98 \cdot 10^{-3}$	61.71	52.92

Table 3.8: Online testing Mazes corridor 8 steps long

Agent	Success Rate	Average time (s)	Average length	Average nodes expanded
A*	100 %	$2.84 \cdot 10^{-4}$	61.60	49.85
LSTM	82 %	$2.04 \cdot 10^{-3}$	71.57	56.02

Table 3.9: Random filled for 25 %

There is a clear and also expected low success rate in the mazes, because, as the reader may imagine, the chance to meet a blind pathway is higher than in all the other categories. A practical reason is that if the agent gets in a blind spot or simply a corner, it is a lot harder to find an escape way in maps like the mazes because it could be too much distant from the current position, indeed most failure attempts are deadlock situations like the example showed in Figure 3.12.

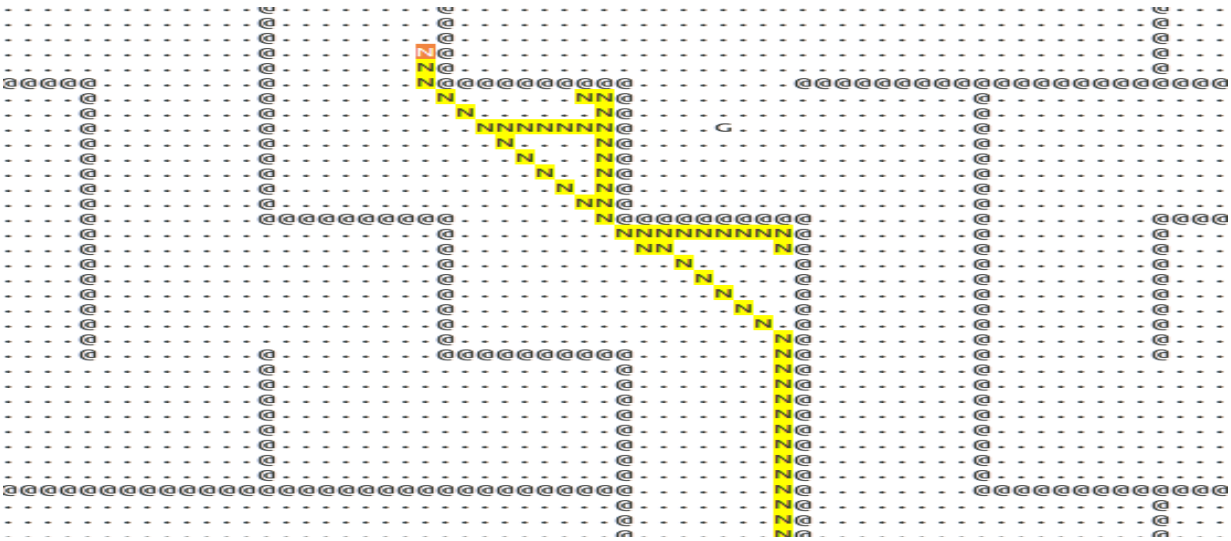


Figure 3.12: Deadlock situation in a maze

I write too distant because it is evident that such large workarounds, like they would be needed in these cases, are not experienced during the training process or very rarely. Since the supervisor algorithm consist in an offline and optimal search, the solutions found are almost always a straight way to the goal point, for sure they do not include escaping pathways from critical positions like the one showed in Figure 3.12.

However, remaining in mazes maps, several solved cases are completed because the agent was able to find an escape way relatively close to its position, an example of this kind is revealed in Figure 3.13: initially coming from the lower part of the figure, the agent initially seems to go directly into a corner like wall, but then it explores the free space on its right and find a way go round the wall and continue the path solving toward the goal point.
Meanwhile, the higher success rate in random filled maps can be explained with exactly the opposite reason the agent tends to fail in the mazes. Focusing on Figure 3.14, it is possible to see how the agent can find a way to go around every group of obstacles. In other words, I would conclude that the structure of random filled maps matches better the skills of the the developed method.

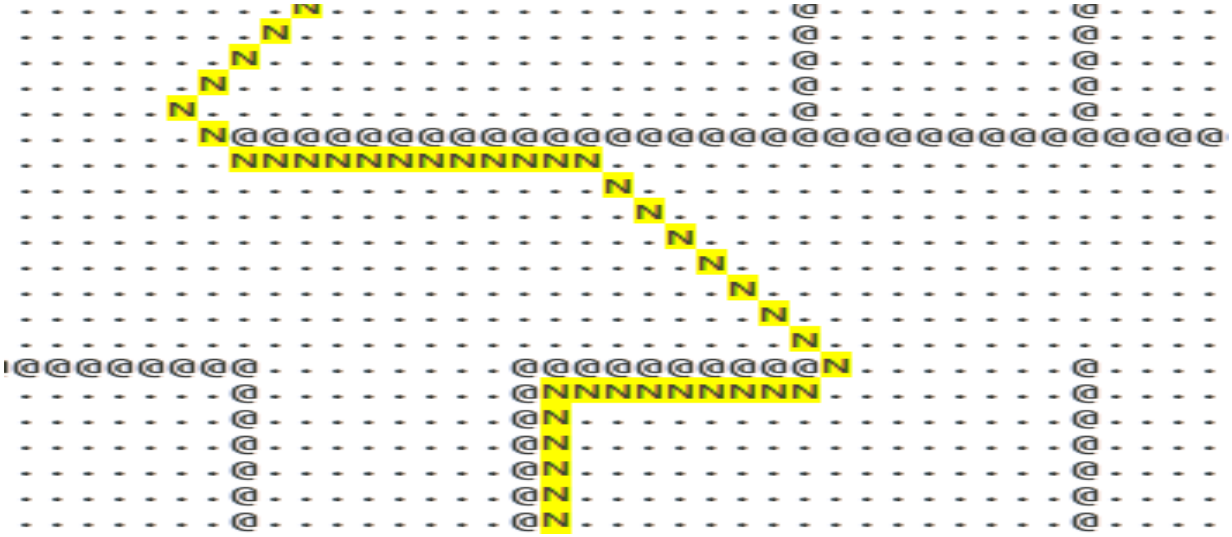


Figure 3.13: A way out from stuck position in a corner was found

Figure 3.14: Local escape ways are easier to find in random filled maps

The success rate by the LSTM-based agent is clearly the most important weakness, while the performance regarding the other parameters can be considered good, in particular the average computation time calculation is in favor of the A* because in a real robot it would have to include the time spent to build or update the map, on the contrary, here it is immediately available. A focus on the success rate is necessary, it was found, easily expected, that it increases if the maximum path length limit is reduced, the trend is showed in Figure 3.16 and 3.17 referring to mazes and random filled maps, respectively. While for Dragon Age maps, it is quite similar to the one for random filled maps (Figure 3.15). Furthermore, this analysis includes the trends referred to the online agent using the MLP model of Figure 3.8b. This last comparison states that LSTM sets an absolutely higher success rate with respect to the MLP.

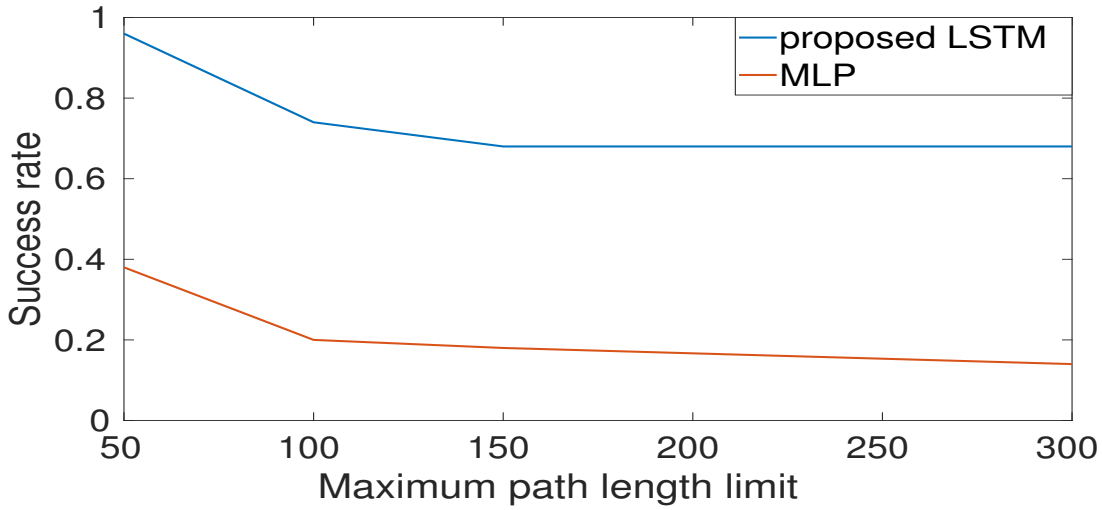


Figure 3.15: Accuracy trend in Dragon Age maps

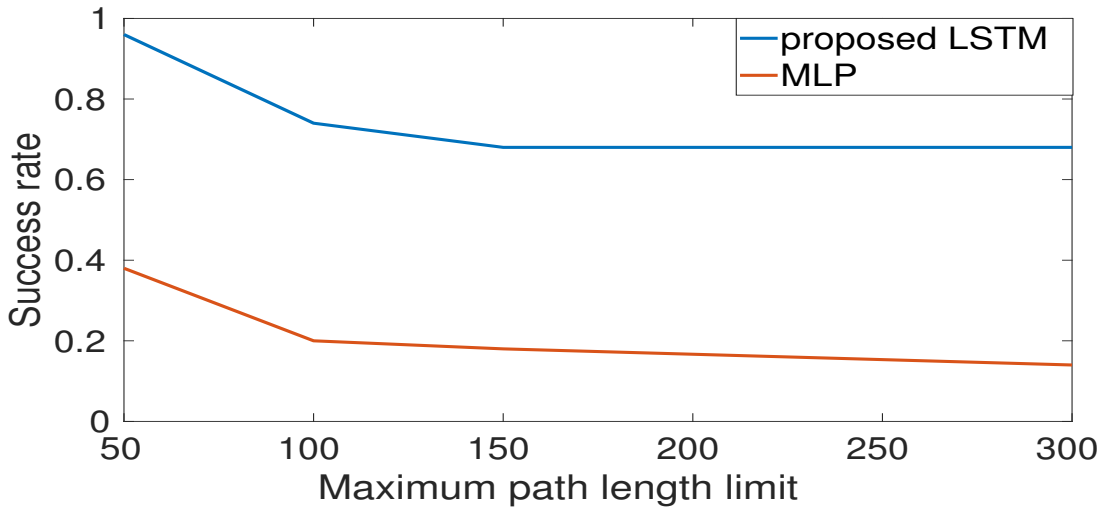


Figure 3.16: Accuracy trend in Mazes maps with corridor 8 steps long

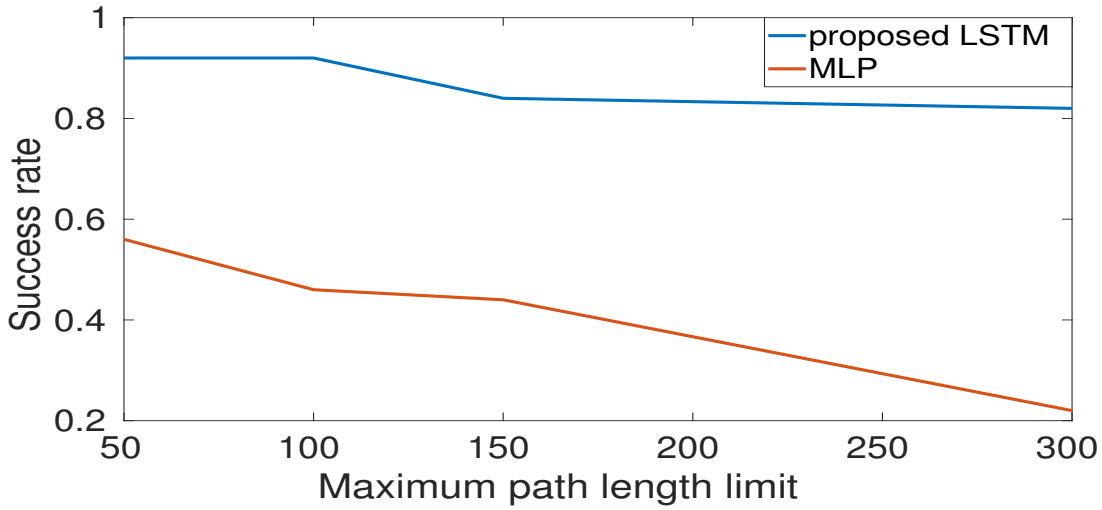


Figure 3.17: Accuracy trend in Random 25 % maps

Chapter 4

Experiment

This last chapter deals about the implementation of the proposed method on a real mobile robot platform. The mobile robot platform is the IRobot Create 2 and proximity ranges are generated by the Hokuyo UTM-30-LX-EW local range finder (LRF) sensor. The whole system for the real robot takes is implemented in the ROS framework, in particular for the characteristics listed below:

- data communication between processes
- an internal system for two-dimensional mobile robot navigation, called navigation stack
- available and tested driver for both robot and LRF sensor

4.1 Real robot set up

The tests are conducted in our laboratory as a good example of indoor map. Since the beginning, I tried keep the design flexibility as high as possible, in order to have a platform easy to understand and use without radical modifications.
This concept translated into the real world as performing test through a wireless connection with the robot, so that just a device to handle and send the data from the robot to the deep learning monitor station and viceversa would be connected to the robot with a cable. The desktop computer is connected via ethernet, meanwhile, on the robot hand, a wireless connection to a different segment of the same network is available. This means that a direct connection between the two by just setting the server ip and the client ip in the ROS configuration of the environment variables is not possible, however not without modifications of the network.
The simplest workaround consisted to configure a Virtual Private Network (VPN), where the desktop pc acts as the server and a laptop placed onto the Create 2 robot as the only client. Another small but nice advantage of the VPN is that ip assigning by DHCP is supported, in a sense that I don't have to change the ip address in a configuration file every

time it changes, however for external reasons the desktop pc has a fixed ip connection, but this detail is not so important for the purpose.
A third device is added to the complete the set up, it is a device with an SSH (Secure Shell) client installed - in this particular case an android tablet - that connects to the laptop through wireless in order to execute any terminal command is needed. This device helps in the overall project flexibility, two simple but practical cases represent a support to this choice:

1. if unfortunately the vpn connection drops or the robot driver node accidentally terminates or similar cases, the connection or the particular command can be restarted without interrupting the test

2. clock synchronization between laptop and desktop computer must be always maintained within a certain offset, this is achieved by running a command on a laptop terminal

in particular, case number two consists on a manual synchronization thanks to the Network Time Protocol (NTP), it is really essential because otherwise I would not be able to run the ROS navigation stack.
Figure 4.1 represents a schematic description of set up presented in this section, while Figure 4.2 shows the real devices to have a complete idea.

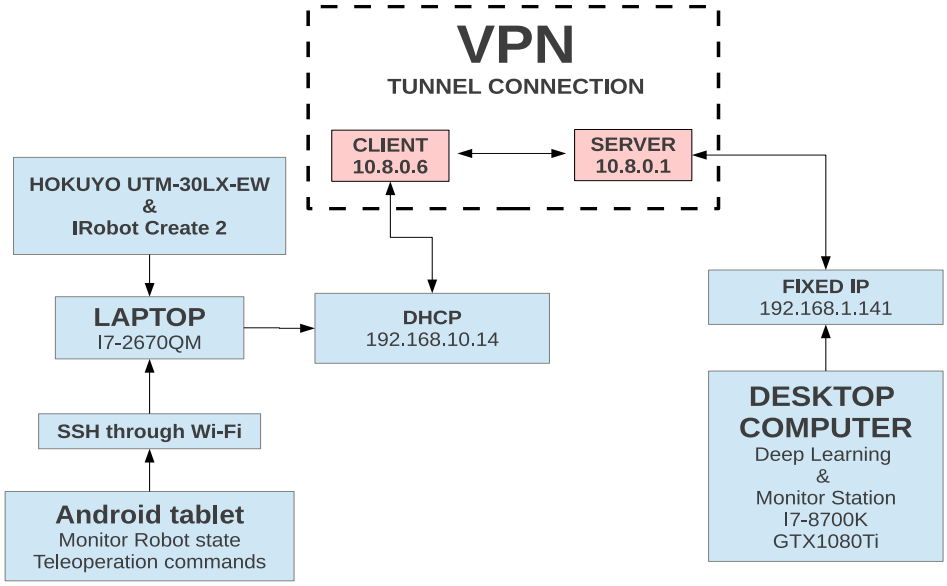


Figure 4.1: Schematic representation of the real world set up

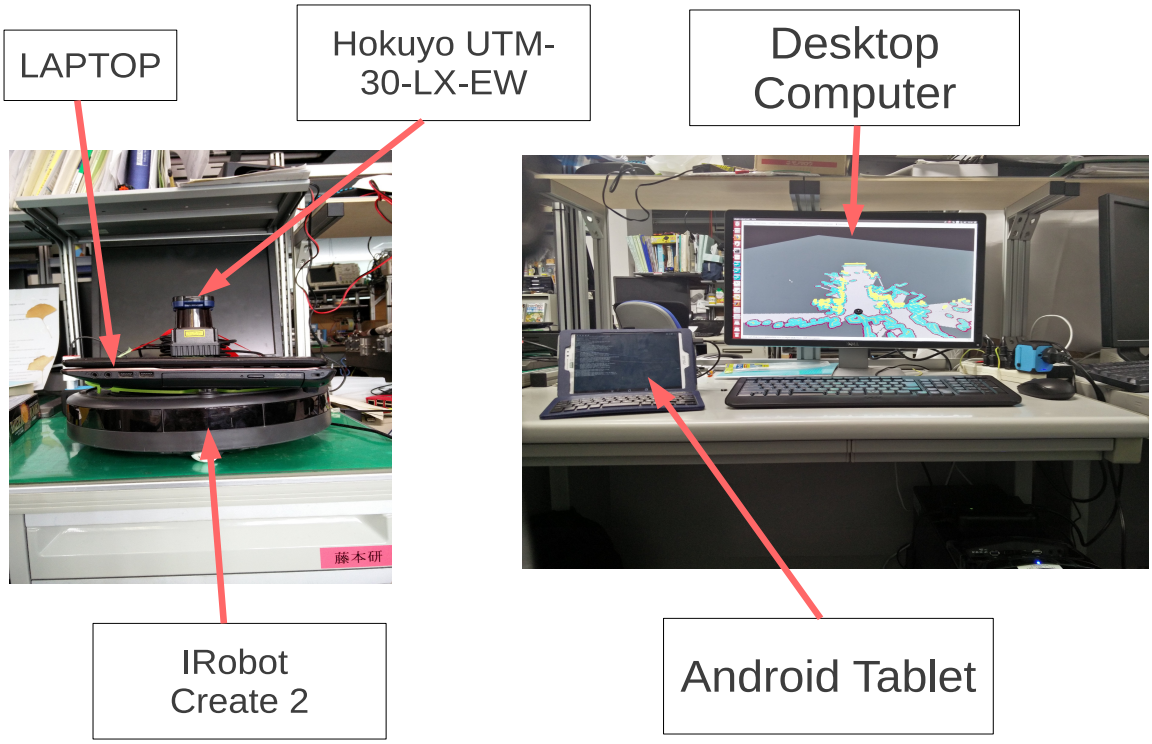


Figure 4.2: Devices utilized

4.2 Training and Testing

First of all, similarities and diversities with respect to the simulation environment have to be described. The nature of output of the neural network is essentially the same, here is more properly called "steering angle" as the two command signals that are given as input to the robot are the translational and rotational velocity commands.
The output directions set allowed are, here, reduced to just five steering angles comprised in $[-90°, +90°]$ and divided $45°$ from each other. Figure 4.3 shows an example of a sequence of steps, where θ is the relative angle between two sampled positions of the robot, one subsequent to the other.
The input state of the neural network include exactly the same elements defined in 3.4, a

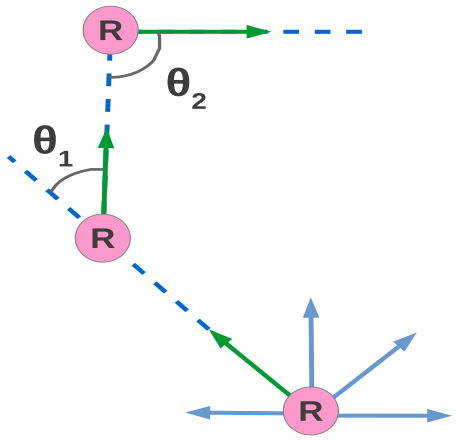


Figure 4.3: A sequence of steps in the real robot implementation

further element added is the current absolute orientation to give a consistent and unambiguous definition of the state.
The LRF is configured to limit the readings in the same angle span related to the output, the ranges taken into consideration are not single measurements but an average value calculated between readings of the same angular section, a total of 720 ranges are generated within the angle span considered. Figure 4.4 helps to catch the practical meaning. Concluding, a total number of eight sections is selected.
The most right way to use the ROS framework for two-dimensional robot navigation would be using the navigation stack as a whole (Figure 4.5). Extremely summarizing, it consists of a global planner responsible to analyze the entire map explored and find the best sequence of coordinates from the robot current position to the goal point and a local planner that generates the translational and angular velocity commands to make robot follow the global coordinates easily associated to a reference signal, eventually the costmap create a virtual inflation of the obstacles to aid related planners to calculate a path most possibly far away from them: the one related to the global planner takes care of the entire map, while the other updates with higher frequency a local squared map centred in the current robot position.

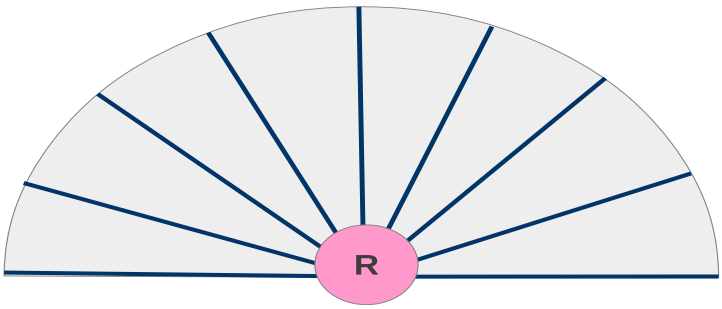


Figure 4.4: The average value of each section is an input of the neural network

Unfortunately, I couldn't find a proper configuration to make the trajectory controller work in our indoor environment, so I decided to keep using the navigation stack but the velocity commands are generated by teleoperating the robot remotely.
In order to highlight the low need of resources by the proposed method, in Figure 4.6 it is compared how the neural network sees the real world versus the human eye. The purple marks are local proximity ranges given as input to the neural network as described above referring to Figure 4.4. Meanwhile, the best way to evaluate the amount of resources needed by the conventional global planner in ROS is a view from above of the entire map of our laboratory explored by the robot (Figure 4.7a), in addition, the global costmap is showed in 4.7b as an enhanced version of the explored map used for the global search.

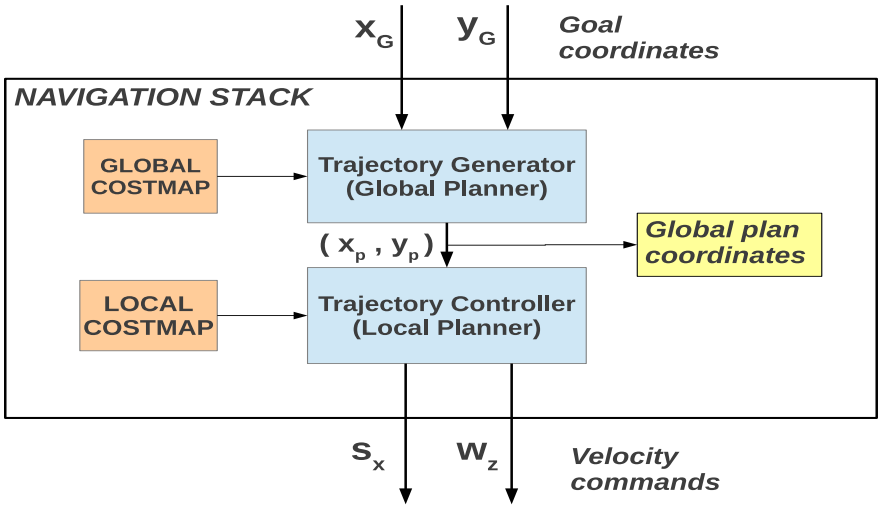
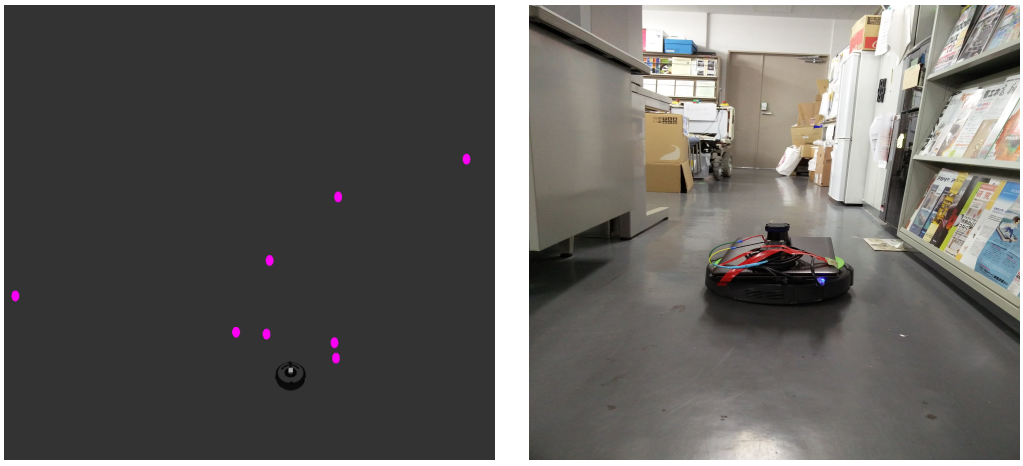


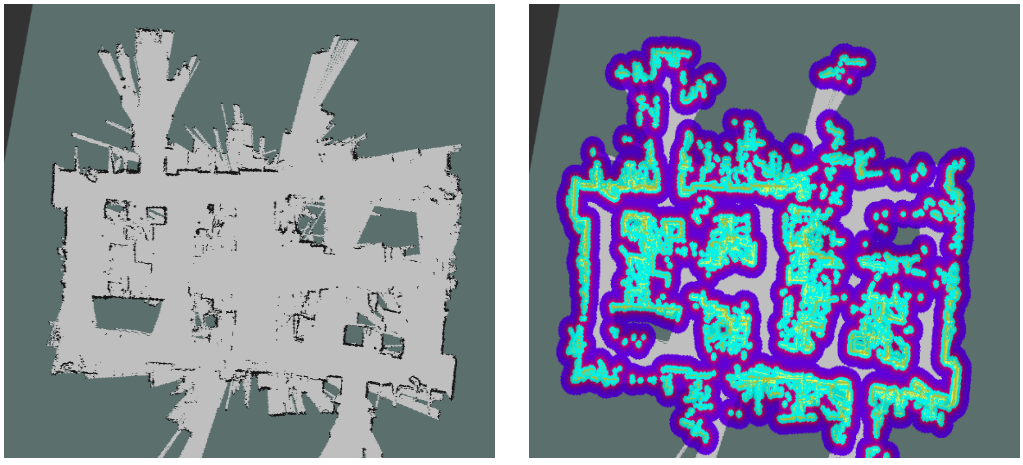
Figure 4.5: Navigation stack diagram block

The costmap is the result of obstacle inflation, the principle is to assign a maximum cost color to the points corresponding to the proximity ranges generated by the LRF sensor, then a descending cost function assigns different colors starting from the maximum threshold to the value considered as free space. Specifically, in the configuration used for Figure 4.7b, yellow stands as the proximity ranges, light blue as cost threshold still considered as obstacle, light purple is in the midway between obstacle and allowed region to navigate represented by the dark purple color.



(a) Neural network view (b) Human eye view

Figure 4.6: Comparing the view of the environment between proposed method and human eye



(a) Map without costmap (b) Map with costmap

Figure 4.7: Conventional global planner view of the entire map

Before effectively training a neural network, the training and validation datasets have to be built, I would like to express here the opinion that collecting good sample data and training the neural network for the real robot implementation was the most difficult thing of the overall research. Losing the partial observability of the environment is just the beginning, the following proportionalities, defined in Figure 4.8, summarize what the real challenge was about, they may ended up in simple decisions but the final configuration proposed was the result of many and many training and testing.

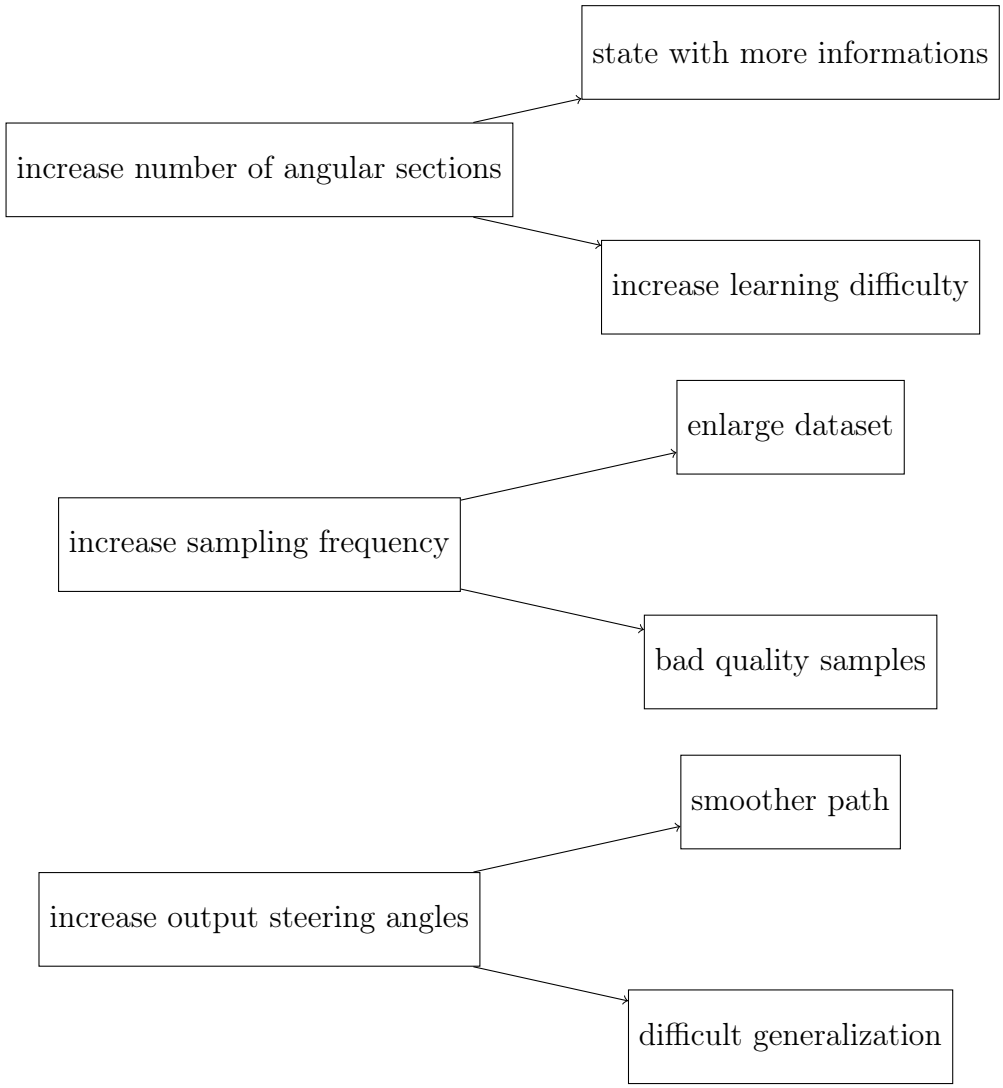


Figure 4.8: Dataset and training decisions consequences

Hence, the policy decided was to sample the data every 2 Hz and, if the robot has moved from the last step, data are collected and the step stored in the dataset. Following Figure 4.3, it means that every 0.5 s the input state elements are collected and stored in a specific sample.

Specific	TRAINING SET	VALIDATION SET
Size (steps)	3000	1000
Time required	$\approx$ 1.5 h	$\approx$ 40 min

Table 4.1: Training set and validation set specifics

Meanwhile, the ground truth is calculated when the robot reaches the next step, suppose an input state is built at timestep t in a position $p^{(t)}$, the ground truth data $l^{(t)}$ can be calculated after 0.5 s with the robot in a position $p^{(t+1)}$:

$$\gamma = relativeangle\left(p^{(t)}, p^{(t+1)}\right) \tag{4.1}$$

$$l^{(t)} = \underset{d_k}{\text{argmin}}\left(|\gamma - d_k|\right) \qquad \text{with } d_k \in D \tag{4.2}$$

where γ is the measure of the real relative angle between $p^{(t)}$ and $p^{(t+1)}$, and D is the set of the five $d_k\, k = 1, ..., 5$ directions allowed.
Table 4.1 shows the few important specifics for the training set and validation set. The traces where the robot moved were limited to the ones represented in Figure 4.9. The red points mean starting or goal points depending on the specific path decided to take and include in the set. The start and goal position new path are picked casually among the red points showed, then the robot travels and a certain amount of samples are stored.

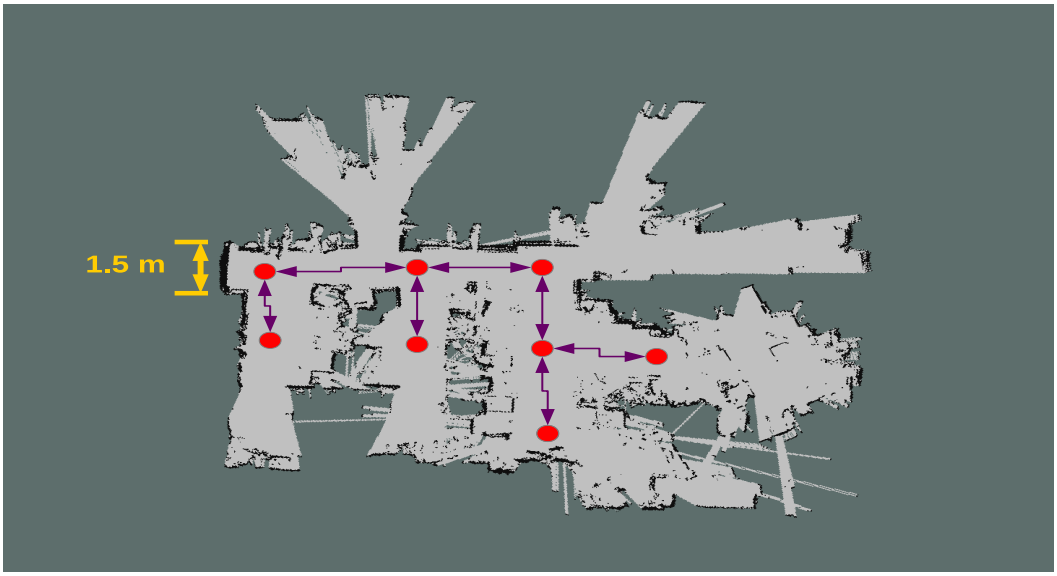


Figure 4.9: Indoor map with the traces followed

Then, the same structure showed in Figure 3.8a, is trained with the hyperparameters values of Table 4.2, the only difference from simulation environment parameters is the time sequence length, here reduced to just 30 steps.
The trend of training and validation accuracy is showed in Figure 4.10, the purpose of showing this graph is to make the reader see the various presence of instabilities during the training process, especially around the epoch[1] four hundred. To understand the complications generated by an oscillating behaviour of accuracy trend, it has to be explained what is the definition of training stability, how the trained weights are saved and how the accuracy itself is computed.
In Section 3.2, dedicated to training the neural network in the simulation environment, accuracy convergence was met in relatively few epochs and for the remaining training time just the first digit after the decimal point was affected by variations, this means a stable training.
Meanwhile, here, in the best-case scenario, there are fluctuations of $3 \div 4$ % and they occur after every single epoch, I consider it enough to influence the selection of the trained layers in a bad way. Concluding, I would define as training instabilities the fluctuations going beyond a threshold predefined by the neural network designer.
The weights of the trained neural network that will be deployed in online testing comes from saved snapshots[2] executed after a weights update during the training process, what can be changed is the rate of taking a snapshot. In practice, the minimal rate of taking the snapshots is usually once per epoch in order to limit the snapshots files generated and have a univocal pair epoch accuracy - epoch weights snapshot. However, when the training convergence gets more and more flat, synonym of stability, good snapshot saving rates can be also once every ten or fifty epochs.
The accuracy expression is recalled below:

$$A_N = \frac{1}{N} \sum_{n=1}^{N} \delta\{d_n = l_n\} \tag{4.3}$$

$$\text{where } \delta\{\text{condition}\} = \begin{cases} 1 \text{ if condition} \\ 0 \text{ if } \neg\text{condition} \end{cases} \tag{4.4}$$

where N is the batch sample size, l_n is the ground truth data and d_n is the direction decided by the neural network. The accuracy measure is an average, so the snapshot paired to an accuracy value is in turn associated to a specific local set of samples. More precisely, it is a multiple of the batch size, the total number of samples which the error is accumulated before applying the weights update and its accuracy value represents the atomic part of the average value calculated, for example, for an epoch:

$$A_{epoch} = \frac{1}{M} \sum_{m=1}^{M} A_N^m \tag{4.5}$$

where M is the number of batches in the whole training set, and A_N^m is the accuracy value of the m-th batch. In this specific case here implemented, the batch equal to the time sequence considered, in simulation environment it was set to 512, while here is reduced to 30 timesteps.

So, for the case depicted in Figure 4.10 you would set the minimal saving rate of one snapshot per epoch in order the catch the weights related to the highest accuracy value but the snapshot taken depends on the last time sequence of that particular epoch. With such high variance, inside the summation of the accuracy you can never be sure the saved weights selected turns out to correspond to the epoch accuracy.
Online testing, the final stage, gave acceptable results just for simple targets, like the cases showed in Figure 4.11 and 4.12, while Figure 4.13 represents a good attempt in the first steps but then decisions taken by the neural network brought the robot too far from the goal point to be considered a solved task.
In these last figures, the green arrow is the goal point while the red ones represents the trace marked by the robot and the purple points denote the averaged ranges of each sections in Figure 4.4 given as input to the neural network proposed.

Hyperparameter	Value
learning rate (fixed)	0.1
weight decay	10^{-5}
regularization type	L_1
momentum	0.95
epochs	500

Table 4.2: Hyperparameters configuration for training the LSTM neural network

[1]Sweeping an epoch means every sample comprised in the training and validation sets is analyzed one time

[2]The snapshot is a file containing the layers of the neural network which weights value correspond to the instant the snapshot was created

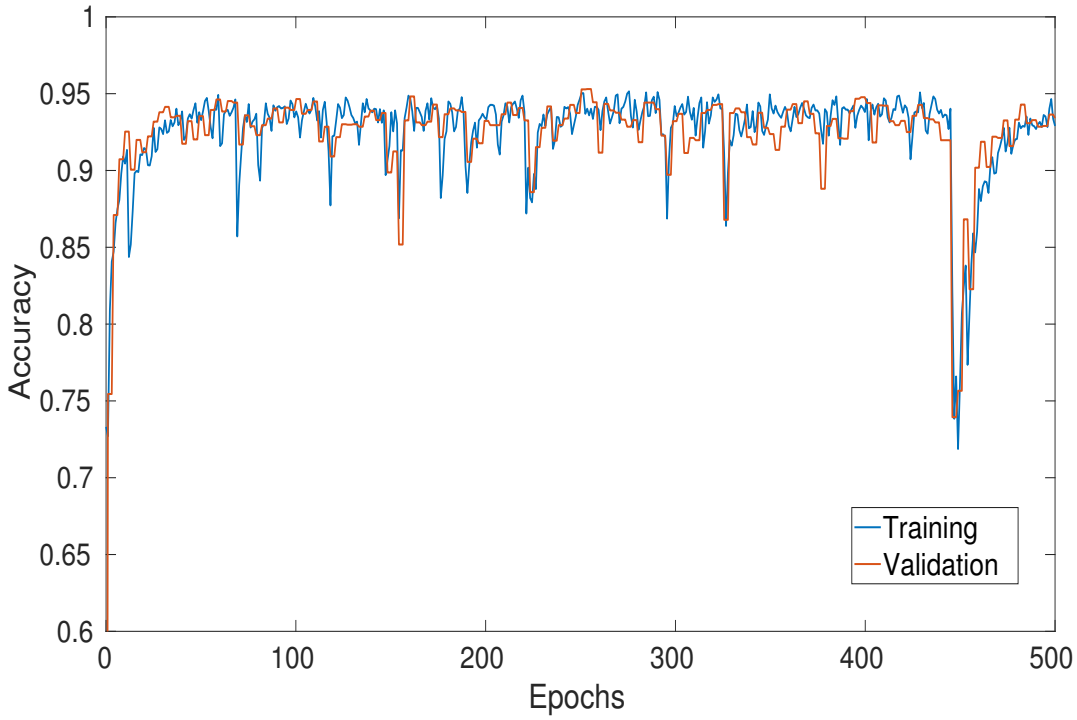


Figure 4.10: Training and validation accuracy trend for the real robot network training

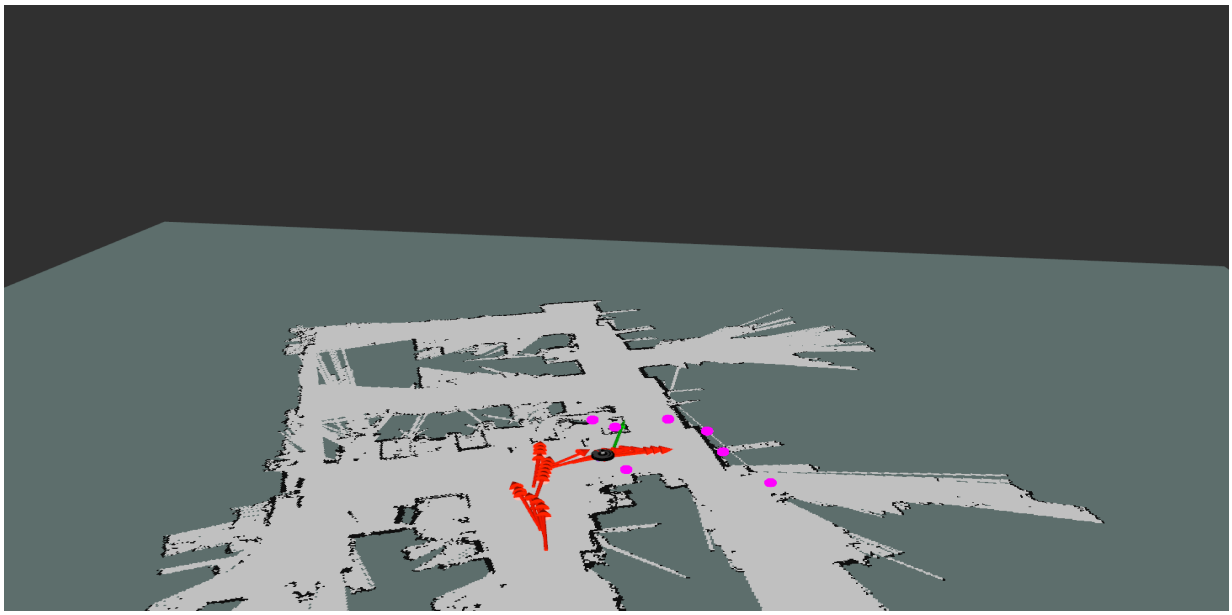


Figure 4.11: Example 1 online testing with the real robot

Figure 4.12: Example 2 online testing with the real robot

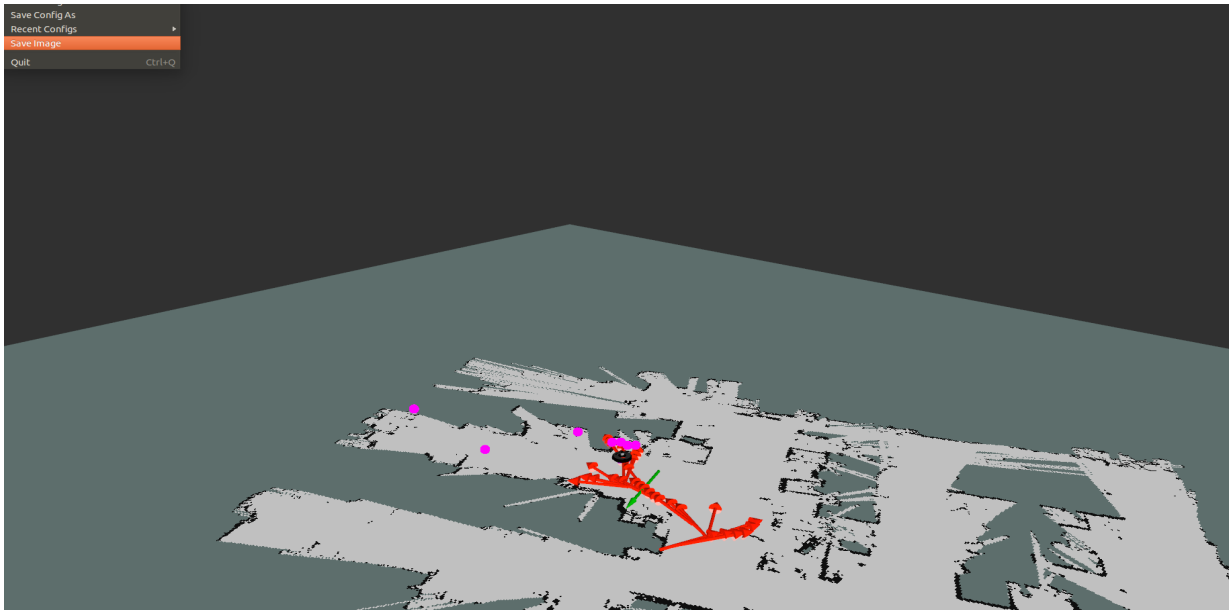


Figure 4.13: Example 3 online testing with the real robot

4.3 Comments

In this section I would like to express my a posteriori critical opinion regarding the application and implementation of the proposed method on the real robot. However, it has to be stated that the implementation here analyzed is the first attempt to realize the proposed method in the very complex real world, so it can be considered an entry level version which is absolutely normal it may have errors and bad performances.
First, the simulation environment and related conclusions were realized before starting to work with the real robot, so the principle followed was to translate the ideas developed in simulation into the real world. After the results attained with the Roomba and Hokuyo LRF were analyzed, the principle mentioned can be considered wrong or partially wrong or something did not work in realizing it or it maybe too ambitious.
The most important differences between the simulation world and the real world have to be searched in the nature of the input state elements, the output nature is left out since the final configurations are essentially the same: a certain number of allowed directions, each one meant to take the robot in a different next position with respect to any other possible direction.
Next paragraphs are dedicated to point out what changes when building each element of the input state. All the complications that will be mentioned from now on contributes to the fact that the associations state-direction in the real world are potentially infinite even in a limited are like the indoor map showed in Figure 4.9. Meanwhile, this aspect is much more limited in ASCII maps, because they are built to be discrete worlds where each pixel assumes a unique and never changing value of obstacle or free space and the robot performs its steps in pixel to pixel.

4.3.1 Proximity ranges

In ASCII maps, these elements are calculated in number of pixels, cartesian measures increase by 1, while diagonal measures increase by $\sqrt{2}$. In the real world, the Hokuyo sensor generates the ranges readings, then the average value on each angular section is an element of the input state.
Since the LRF is not ideal, an error term represented by a random variable has to considered in addition to the ideal reading. Supposing the robot starts in a certain position where a scan is saved and stored, then travel for some time and gets back to the starting position, a scan saved in this moment is different from the one stored before: the neural network will receive different input states related to the same physical position.
However, this effect is absolutely normal, it represents an attempt to read a continuous signal in a real world context with a non ideal sensor. As a conclusion, I would say that working with continuous signals as inputs to a neural network is a hard challenge. More specifically, proximity ranges from LRF sensor is a low level sensory input, while neural networks could be more suited for high level sensory inputs like some sort of events extraction, for example the idea proposed in [41]. An event could be a binary variable switching to the value of 1 if the proximity ranges of a particular angular sections are below a predefined threshold value.

4.3.2 Distance and relative angle

The problem relative to these elements can be more complex, but not so hard to understand. First of all, it is better to explain how they are calculated.
When the navigation system of the robot is activated at the beginning of a generic test session, the center of a new global frame is assigned to the starting real position of the robot. Thanks to the odometry feature of the robot and relative driver, the current position of the robot during travelling is always available. Meanwhile, the goal point is set from the remote monitor station by picking a point on the screen (Figure 4.2), even this point is relative to the global frame mentioned above.
However, my opinion is that in setting up the whole navigation system for building the datasets, the resolution of the odometry was too high. This is important, because remember that global planning computes the solution on a large scale and low resolution environment, indeed in ASCII maps the agent occupies an entire square grid.
In the real world, the map resolution was a lot higher than the robot size, the main effect is that for two apparently equal physical positions of the robot slightly different values of distance and relative angle to the same goal point are calculated, eventually the stored samples would result different. I think this bad set up effect increase a lot the potentially infinite real world cases the neural network is supposed to learn.
Concluding, the first attempt to fix this problem is to lower the resolution of odometry in calculating these particular values, however I think robot size could be a maximum resolution value.

Chapter 5

Conclusion & Future works

In this thesis a model based on the Long Short-Term Memory neural network is proposed as an online agent for the path planning task. A complete model tuning and structure shaping is developed in the dedicated simulation environment and then a performance comparison is analyzed.
A real robot implementation is realized but it is still strongly limited, so it can be analyzed just as a demonstration, no performance comparisons are carried on.
The first and most important problem is the success rate parameter, future enhancements should be in the direction of increasing it. In ASCII maps, the main problem is that in the training process the agent never experienced escaping scenarios from a blind spot or deadlock in a corner, the ground truth is given by an algorithm built to generate an optimal solution.
So, generating a dataset where many starting points are on purpose placed in blind spots could be a solution but maybe too hard to realize if these points are not automatically found in a map.
Meanwhile, a reinforcement learning stage could be the best solution. We would think about an initial stage of supervised training in order, then a second one based on reinforcement learning online training in order to have the chance to adjust its learnable parameters every time it will get in ambiguous positions like the situations mentioned above.
On the real robot side, the implementation has to be enhanced considering the proposed method just as a starting point, in Section 4.3 I point out the main problems and possible solutions. Furthermore, in order to build a completely autonomous system, a new and more suitable trajectory controller has to be designed well suited for this kind of online search method.
The source code of the research project developed can be found on my github page: *https://github.com/nikfio*.

References

[1] NASA/JPL-Caltech, ed. *Mars Science Laboratory Mission's Curiosity Rover*. 2011. URL: https://www.nasa.gov/mission_pages/msl/multimedia/gallery/curiositySAF_R.html.

[2] KUKA Robotics, ed. *Mobile robots from Kuka*. 2018. URL: https://www.kuka.com/en-hu/products/mobility/mobile-robot-systems.

[3] IBM, ed. *Multi-Purpose Eldercare Robot Assistant (IBM MERA)*. 2018. URL: https://www.ibm.com/blogs/age-and-ability/2016/12/08/tour-the-ibm-aging-in-place-environment/.

[4] W. Khaksar et al. "A review on mobile robots motion path planning in unknown environments". In: *2015 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*. Oct. 2015, pp. 295–300.

[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. http://aima.cs.berkeley.edu. Prentice Hall, 2010. Chap. Solving Problems by searching.

[6] Akshay Kumar Guruji, Himansh Agarwal, and D.K. Parsediya. "Time-efficient A* Algorithm for Robot Path Planning". In: *Procedia Technology* 23 (2016). 3rd International Conference on Innovations in Automation and Mechatronics Engineering 2016, ICIAME 2016 05-06 February, 2016, pp. 144–149. URL: http://www.sciencedirect.com/science/article/pii/S2212017316300111.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. http://aima.cs.berkeley.edu. Prentice Hall, 2010. Chap. Beyond Classical Search, pp. 429–436.

[8] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. URL: http://dx.doi.org/10.1007/BF01386390.

[9] Amit Patel. *A*'s Use of the Heuristic*. Ed. by Red Blob Games. Amit's A* Pages. Stanford University. 2018. URL: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107.

[11] Anthony Stentz and Is Carnegle Mellon. "Optimal and Efficient Path Planning for Unknown and Dynamic Environments". In: *International Journal of Robotics and Automation* 10 (1993), pp. 89–100.

[12] Warren S. McCulloch and Walter Pitts. "Neurocomputing: Foundations of Research". In: ed. by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chap. A Logical Calculus of the Ideas Immanent in Nervous Activity, pp. 15–27.

[13] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pp. 65–386.

[14] Bernard Widrow. "Adaptive Filters". In: *Aspects of Network and System Theory* (1960).

[15] Bernhard Scholkopf, Christopher Burges, and Alexander Smola. "Advances in Kernel Methods - Support Vector Learning". In: *MIT Press*. Dec. 1998.

[16] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[18] Dan C Cireşan et al. "Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks". In: *Medical image computing and computer-assisted intervention : MICCAI ... International Conference on Medical Image Computing and Computer-Assisted Intervention*. Vol. 16. Sept. 2013, pp. 411–8.

[19] A. Graves, A. r. Mohamed, and G. Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. May 2013, pp. 6645–6649.

[20] Alex Graves and Navdeep Jaitly. "Towards End-To-End Speech Recognition with Recurrent Neural Networks". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1764–1772. URL: http://proceedings.mlr.press/v32/graves14.html.

[21] Bastien Moysset et al. "The A2iA Multi-lingual Text Recognition System at the Second Maurdor Evaluation". In: *Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR*. Vol. 2014. Sept. 2014.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1". In: ed. by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press, 1986. Chap. Learning Internal Representations by Error Propagation, pp. 318–362.

[23] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (2014), pp. 85–117.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. introduction, pp. 5–8.

[25] Ronald J. Williams and David Zipser. "Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity". In: *Back-propagation: Theory, Architectures and Ap- plications* (1995).

[26] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Vol. 385. Jan. 2012. Chap. Recurrent Neural Networks, pp. 18–20.

[27] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning Long-Term Dependencies with Gradient Descent is difficult". In: *IEEE Transactions on Neural Networks* (1994).

[28] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the Difficulty of Training Recurrent Neural Networks". In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. 2013, pages. URL: http://dl.acm.org/citation.cfm?id=3042817.3043083.

[29] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computations* 9.8 (Nov. 1997), pp. 1735–1780.

[30] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Vol. 385. Jan. 2012. Chap. The Long Short-Term Memory, pp. 31–38.

[31] J. Donahue et al. "Long-Term Recurrent Convolutional Networks for Visual Recognition and Description". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.4 (Apr. 2017), pp. 677–691.

[32] Razvan Pascanu et al. "How to Construct Deep Recurrent Neural Networks". In: *International Conference on Learning Representations 2014 (Conference Track)*. Apr. 2014. URL: http://arxiv.org/abs/1312.6026.

[33] X. Li and X. Wu. "Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Apr. 2015, pp. 4520–4524.

[34] Danica Janglova. "Neural Networks in Mobile Robot Motion". In: *Neural Networks in Mobile Robot Motion* (2014).

[35] S Hamid Dezfoulian, Dan Wu, and Imran Shafiq Ahmad. "A Generalized Neural Network Approach to Mobile Robot Navigation and Obstacle Avoidance". In: *Advances in Intelligent Systems and Computing*. Vol. 193. Jan. 2013, pp. 25–42.

[36] H. Xiao, Li Liao, and F. Zhou. "Mobile Robot Path Planning Based on Q-ANN". In: *2007 IEEE International Conference on Automation and Logistics*. Aug. 2007, pp. 2650–2654.

[37] Mihai Duguleana and Gheorghe Mogan. "Neural networks based reinforcement learning for mobile robots obstacle avoidance". In: *Expert Systems with Applications* 62 (2016), pp. 104–115.

[38] Y Fan et al. "TTS synthesis with bidirectional LSTM based Recurrent Neural Networks". In: *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*. Jan. 2014, pp. 1964–1968.

[39] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition". In: *CoRR* abs/1402.1128 (2014). URL: http://arxiv.org/abs/1402.1128.

[40] Masaya INOUE, Takahiro YAMASHITA, and Takeshi NISHIDA. "Robot Path Planning by LSTM Network Under Changing Environment". In: (2017). URL: http://lab.cntl.kyutech.ac.jp/~nishida/paper/2017/CSADC2017_38.pdf.

[41] B. Bakker, F. Linaker, and J. Schmidhuber. "Reinforcement learning in partially observable mobile robot domains using unsupervised event extraction". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 1. 2002, 938–943 vol.1.

[42] Bram Bakker. "Reinforcement Learning with Long Short-Term Memory". In: *Advances in Neural Information Processing Systems 14*. Ed. by T. G. Dietterich, S. Becker, and Z. Ghahramani. MIT Press, 2002, pp. 1475–1482. URL: http://papers.nips.cc/paper/1953-reinforcement-learning-with-long-short-term-memory.pdf.

[43] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. http://aima.cs.berkeley.edu. Prentice Hall, 2010. Chap. Making Simple Decisions.

[44] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. Chap. Linear Models for Classification, pp. 208–210. ISBN: 0387310738.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. http://aima.cs.berkeley.edu. Prentice Hall, 2010. Chap. Reinforcement Learning.

[46] Richard Bellman. "The Theory of Dynamic Programming". In: (1954).

[47] M. Hausknecht and P. Stone. "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *ArXiv e-prints* (July 2015). arXiv: 1507.06527.

[48] Mnih Volodymir, Kavukcuoglu Koray, and Silver David. "Human-level control through deep reinforcement learning". In: *Nature* (2015).

[49] Piotr Mirowski et al. "Learning to Navigate in Complex Environments". In: *CoRR* abs/1611.03673 (2016). arXiv: 1611.03673. URL: http://arxiv.org/abs/1611.03673.

[50] John N. Tsitsiklis and Benjamin Van Roy. "An Analysis of Temporal-Difference Learning with Function Approximation". In: *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*. Vol. 42. 5. May 1997, pp. 674–681.

[51] N. Sturtevant. "Benchmarks for Grid-Based Pathfinding". In: *Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 144–148. URL: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

[52] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (2014).

[53] NVIDIA Corporation, ed. *GEFORCE GTX 1080 Ti*. 2017. URL: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/.

[54] Anders Krogh and John A. Hertz. “A Simple Weight Decay Can Improve Generalization”. In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*. Morgan Kaufmann, 1992, pp. 950–957.

[55] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1139–1147.

[56] Boris Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr Computational Mathematics and Mathematical Physics*. Vol. 4. Dec. 1964, pp. 1–17.