# Adaptive Intelligence : Assignment 2

Maxime Fontana
*University of Sheffield*
Sheffield, United Kingdon
mfontana1@sheffield.ac.uk

***Abstract*—This report provides an implementation of the Reinforcement Learning algorithms described in the assignment as well as a discussion of the results. The main task of this study will be to teach a robot to move in a square-shaped area to find a battery charging location**

## I. Introduction

Reinforcement Learning is a way to learn what to do through interacting with an environment given a set of actions. Throughout exploring this environment, the agent will be either rewarded or penalised depending on the outcomes of the chosen actions carried out by the agent. That type of learning therefore stands out and is, in essence, different from both **supervised learning**, in the sense that the agent does not learn from pre-given labeled data, and **unsupervised learning** as it does not explicitly learn from intrinsic characteristics.
On the other hand, it is possible to embody such an algorithm into a Neural Network, a computational model that consists in successive layers made of neurons through which data is going to be processed.
Once the neural network's architecture has been defined and the initial weights initialised (see later), the training can start. In this study, we present the way this learning strategy can be associated with neural networks, therefore giving birth to what is known as *Deep Reinforcement Learning*.

In order for this strategy to fit into the Neural Network landscape, we need to derive the main property of an Artificial Neural Network (ANN), that being, the learning rule, the error function as well as a description of the algorithm we implement here.

### A. Immediate Rewards

First of all, Reinforcement Learning is driven mainly by the estimation of the rewards, these are called the **Q-values**. Indeed, the agent must explore the expected rewards after the completion of every action that this agent is capable of doing at a specific time. Therefore, Q-values are expressed in relation with their respective state **s** and action **a** : **Q(a,s)**. If we assume that in the initial conditions, we do not know about the probabilities and the rewards, we must therefore learn these Q-values. This process, that is going to be performed iteratively can be summarised as below :

$$Q_K(s,a) = \frac{1}{K}\sum_{t}^{K} R_t$$

With **K** being the number of times this learning is repeated and **R** being the estimated reward at each time step.
This equation is similar to the one below by introducing a learning rate and a much more practical computational learning rule. This equation will represent the **immediate reward**.

$$\Delta Q(s,a) = \eta(r - Q(s,a))$$

Where **Q(s,a)** is the expected reward, **r** is the received reward and $\boldsymbol{\eta}$, the learning rate.

### B. Future Rewards

In this study, we will implement the **SARSA** algorithm, this algorithm aims at taking into account future rewards and this brings along a slightly different equation for our learning rule. Indeed, the former will now become :

$$\Delta Q(s,a) = \eta[r - (Q(s,a) - \gamma Q(s',a'))]$$

That implies that weights must be updated after the future reward estimation has been carried out. Moreover, $\boldsymbol{\gamma}$ introduces a **discount factor** which represents the strength of importance given to the future reward.

### C. Error Function

An important aspect of incorporating Reinforcement Learning in an ANN is to set up the notion of the error function, in other terms, what we are looking to optimise in terms. Therefore, the error function could be described as the minimisation between the estimation of our reward and our received reward. Therefore, this could be translated into the equation below :

$$E(w) = \frac{1}{2}((r + \gamma Q(s',a',w) - Q(s,a,w)))^2$$

if it were to be calculated with respect to the *SARSA* algorithm.

### D. SARSA

The Reinforcement-Learning algorithm at the heart of the study is *SARSA*, this is an on-policy algorithm as it aims to improve the same policy they use to both select actions and the target policy they use for learning.
Below, we provide a brief description of this algorithm embodied in a ANN framework.

## II. Basic Algorithm

### A. The $\epsilon$-greedy Policy

In Reinforcement Learning, the *exploitation vs. exploration* trade-off is important, and is justified by the fact that there is a need to both exploit the good solutions and therefore follow the most optimal path at given time $t$, but also, there is a need to exploit less-optimal actions at this given time $t$ in order to maybe find better solutions later-on, this can only be achieved by exploring the exponential-increasing landscape. Therefore, this policy states that, if a uniformly distributed computed random number is larger than the hyper-parameter $\epsilon$, then it will opt in for the best action from Q given $t$, otherwise, it will choose at random.

### B. Description

The algorithm implemented here is the *SARSA* algorithm described earlier, it can summed up as below :

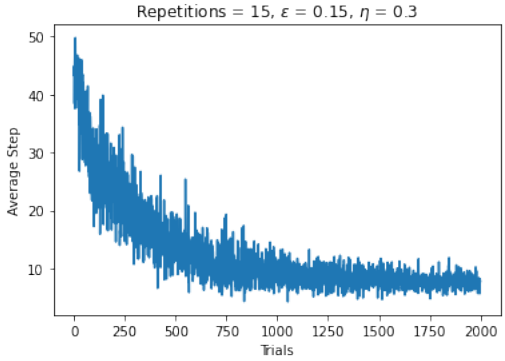
**Algorithm 1** SARSA

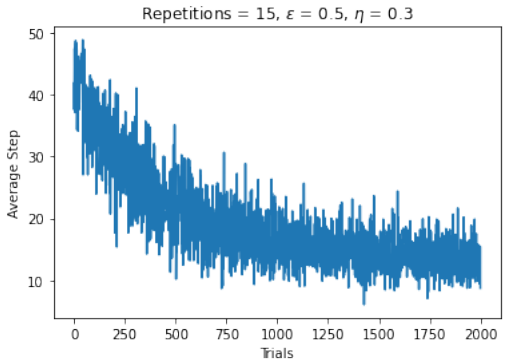
**for** $episode = 1, 2, \ldots$ **do**
  starting state s
  compute Q
  choose a from s based on policy from Q
  **while** $no - success$, $or$, $limit - not - reached$ **do**
    update Q
    take action a, observe r and s'
    choose a' from s' based on policy from Q
    Q(s,a) = Q(S,a) + $\eta$ [r+$\gamma$Q(s',a') - Q(s,a)]
    s = s'
    a = a'
    **if** $success$ **then** update Q(s,a) with r = 1
    **end if**
    **if** $limit - reached$ **then** update Q(s,a) with r = -1
    **end if**
  **end while**
**end for**

In this article, the algorithm is tuned by a few hyper-parameters :

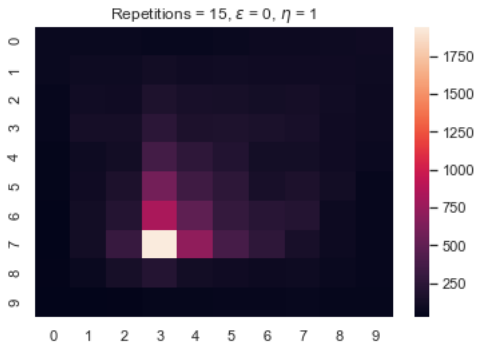
- $n_{episodes}$ = number of episodes to run the algorithm for
- $n_{steps}$ = number of trials to perform given a starting point
- $\eta$ = learning rate (strength of update rule)
- $\epsilon$ = epsilon for policy
- $\gamma$ = discount factor introduced by *SARSA*

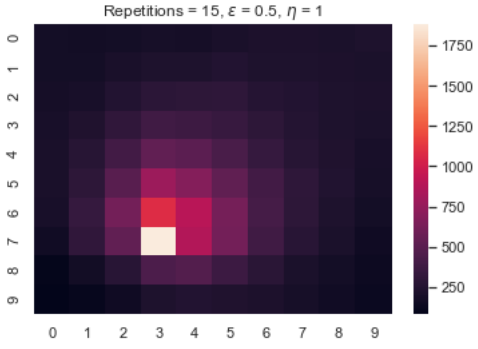
In order to show that our algorithm is exploring more states, we show the average number of steps taken (averaged over 15 runs of the algorithm) as a function of the number of trials with 3 different values of $\epsilon$.

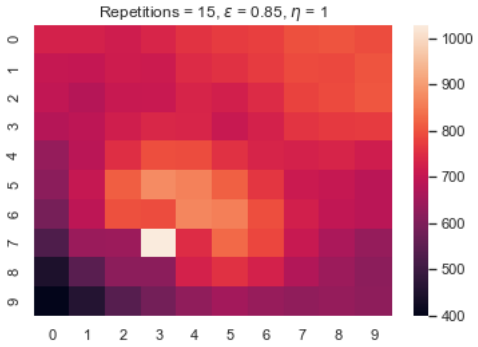



Taking values for $\epsilon$ to be 0.15 and 0.5 given the same learning rate $\eta$ = 0.3, we can see that as $\epsilon$ increases, in other terms, the chance to choose an action at random increases, we can see the number of steps become noisier and takes more time to converge, however, even though the number of steps taken converges fast we cannot be sure that the best solutions have been exploited.
Similarly, we can show that our model is exploring more states by showing the distribution of covered states throughout training, we show the results below averaged over 15 runs :





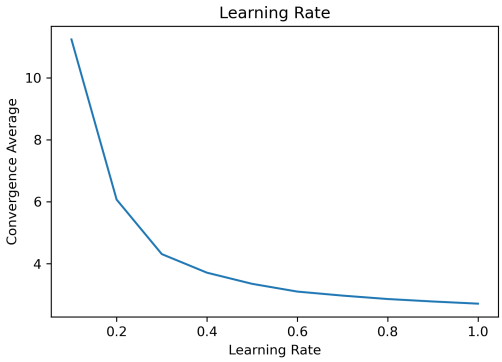

We can see clearly on those heat-maps that the greater the hyper-parameter $\epsilon$, the greater is the explored search space.

Moreover, the number of steps taken per trial is not necessarily representative of the quality of the learning, we are therefore to measure the efficiency of the learning by implementing another metric.

In order to do that, we will plot the **the number of extra steps over the optimal path per trial averaged over each run**. This minimum distance is also called the **Manhattan Distance** in this context, we will therefore count the number of steps over this *l1-distance*.
Starting with the following parameters over 2000 trials :

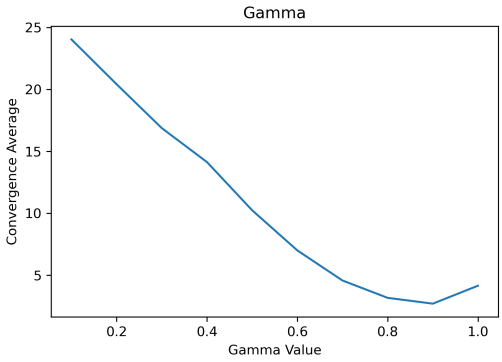
- $n_{episodes}$ = 15
- $n_{steps}$ = 50
- $\eta$ = 0.3
- $\epsilon$ = 0.25
- $\gamma$ = 0.9

Firstly, we start by tweaking the learning rate to find the most optimal value for this hyper-parameter, the following plot shows our obtained results.



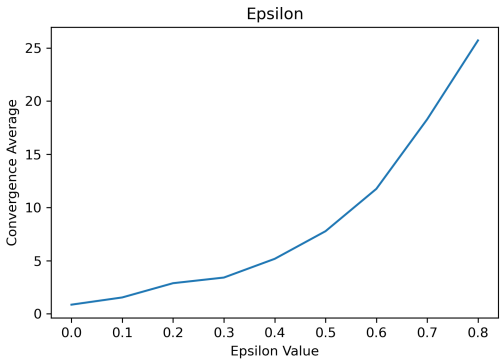

This plot represents the different tested values for the hyper-parameter, the *y-axis* represents the mean of the convergence, that is the 500 last values of the extra steps averaged over 15 repetitions.
We observe that the most optimal value is : $\boldsymbol{\eta = 1.0}$, this is rarely the case, but in the context of this assignment it can be explained by a large number of trials and also because the problem size is small. We will therefore keep this parameter for the remaining of the experiments.

As for the hyper-parameter $\gamma$, the following plot represents the values computed for it's optimal chosen value.

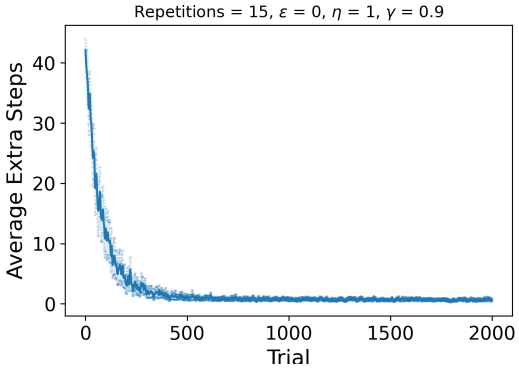



We can notice that the optimal value is : $\boldsymbol{\gamma = 0.9}$. We will therefore keep this parameter for the remaining of the experiments.

Now, repeating the same process for the $\epsilon$ hyper-parameter controlling the action-choosing policy.

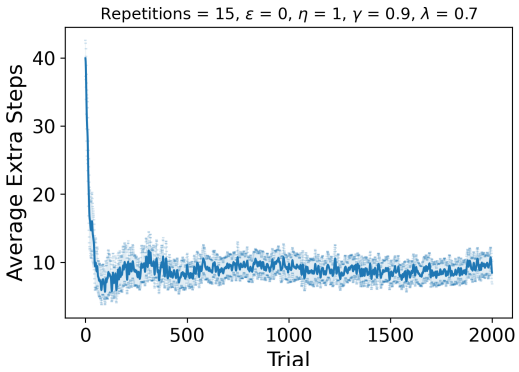


We thereby observe, the most optimal value is : $\epsilon = 0$, again, that is explainable by the relatively small problem size and therefore the need for exploration is diminished.
Thereby, we show the learning accuracy curve given the most optimal values gathered so far :

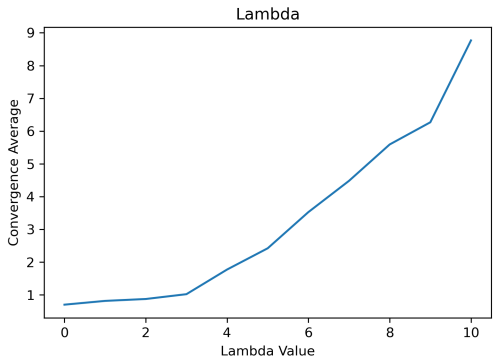



## III. Eligibility-Trace Algorithm

Now, we are looking at an extension of the already existing algorithm, we aim at implementing an *Eligibility-Trace*, put simply, this extension looks at the trajectory executed so far and attributes different levels of importance depending on the recentness of those values, there is a decreasing factor controlled by a new hyper-parameter : $\boldsymbol{\lambda}$. We therefore present how the model performs on the previously-computed best values for our hyper-parameters and a value of $\boldsymbol{\lambda = 0.7}$.




Now, we aim at tweaking this parameter in order to compute it's best value :

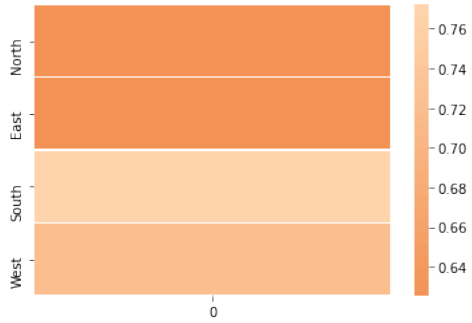



We therefore notice the best value is $\boldsymbol{\lambda = 0}$, again this result is explainable by the small research space and we will be using this value for the remaining of the experiments.

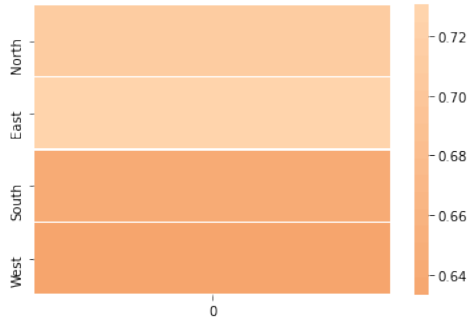
## IV. Q-Values

In the context of an ANN, we process our weights to give our *Q-values*, these are stored in the matrix we are choosing our actions from, therefore, we can process those in order to see how our Reinforcement Learning model behaves in terms of the average choice of actions, in order to do that, we can simply keep track of the *weights* in our Q-values matrix and average it out over multiple runs to observe the trends in the allocation of weights, where the weights our big in our Q-values matrix which is of shape (4,1) in our context as there are 4 actions to choose from, that would mean that the algorithm will most likely or surely (depending on $\epsilon$) choose these actions.

Then we show the results averaged over 15 runs with our current model and the previously computed set of hyper-parameters, in the form of a heat map :




The above heat-map demonstrates the actions taken in the case where **location = [7,3]**. We can clearly see that the most chosen actions are **South** and **West** as indeed, as we initially chose a position at random throughout our grid (problem space) it is most likely that the agent needs to choose these 2 actions.
Similarly, we can use the same method for our **location = [3,7]**




We can see that results are coherent.
***Note*** **:** Referring to the code, it might be possible that our agent lands on the correct ending state and therefore our code will not run the main while loop and leads to 'use before

assignment' error, however, this is really unlikely and should not happen too much.

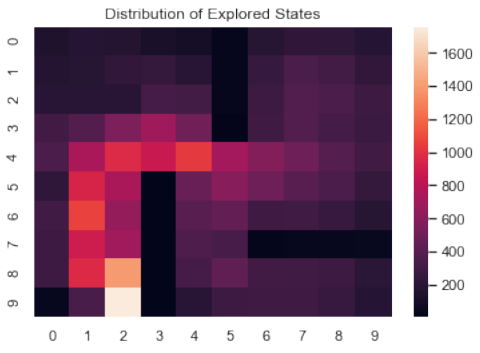
## V. Approximation in Reinforcement Learning

When the problem becomes too large, it is almost impossible for our algorithm to learn because the number of actions and therefore, the number of possible states is too large to be stored in memory.
A solution to this is to use **function approximators**, in other words, we aim at training another neural network so that it learns what do given certain circumstances.

This is done by having a model that can predict the *Q-learning* strategy update step. We add an approximation term to the learning rule. Also, this requires domain or historical knowledge which is something obtainable with our current model, so in this case, we would aim at approximating our Error function given parameters of our learning rules according to this knowledge. By having this model estimating of the outcome for our current state, this would save a tremendous amount of computational labour and drastically improve the running time and make Q-learning good in practice for large problems.

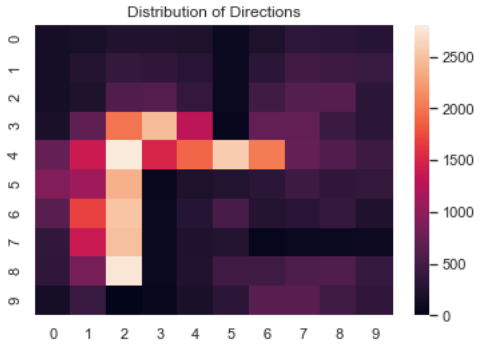
## VI. Adding Obstacles

We are now exploring an extension of our current space, it will consist in adding 3 **obstacles** throughout our 10 by 10 grid and see if our agent is capable of learning how to avoid these and retrieve the reward location at (3, 10).
We thereby model this by setting a negative reward of $r = -0.1$ whenever our agent bounces into a wall on the borders or an obstacle, and moving it back to it's previous location.
In order to show these added obstacles, we first show the distribution of explored states throughout training with the hyper-parameters tuned at their best values and averaged over 5 runs.




In the figure above, we can see the black trails, being the obstacles, are unexplored which proves that the agent has learnt to avoid them, we can also see that the reward location located at (3, 10) is extremely explored which means the agent is able to find it's battery location.

Then, we look at the **preferred direction for each state** throughout training.




By watching closely, we can see that the distribution of directions per square is coherent with the exploration of states, however a bit more details are in order here :

- Black : North
- Purple : East
- Orange : South
- Beige : West

We can neatly observe the tendency to move towards the reward location, also we can see how our agent is avoiding the obstacles, for example the wall on the left is being avoided by climbing north on it's right side, also, we can see the trail in between the left and upper wall that is going west.

```
def homing_nn(n_trials,n_steps,learning_rate,eps,gamma,lam,weights_display=False, tracking_matrix_display=False,
              direction_matrix_display=False):
    ## Definition of the environment
    N = 10                                  #height of the gridworld ---> number of rows
    M = 10                                  #length of the gridworld ---> number of columns
    N_states = N * M                        #total number of states
    tracking_matrix = np.zeros((N, M)) # A matrix to store the previously explored states
    direction_matrix = np.zeros((N, M)) # A matrix to store the preferred location per state
    states_matrix = np.eye(N_states)
    N_actions = 4                                           #number of possible actions in each state: 1->N 2->E 3->S 4
    action_row_change = np.array([-1,0,+1,0])               #number of cell shifted in vertical as a function of the ac
    action_col_change = np.array([0,+1,0,-1])               #number of cell shifted in horizontal as a function of the
    End = np.array([9, 2])                                  #terminal state--->reward
    s_end = np.ravel_multi_index(End,dims=(N,M),order='F')  #terminal state. Conversion in single index

    ## Rewards
    R = 1                                 #only when the robot reaches the charger, sited in End state

    ## Variables
    weights = np.random.rand(N_actions,N_states)
    learning_curve = np.zeros((1,n_trials))
    extra_steps = np.zeros((1, n_trials)) # Show the extra-steps vis-a-vis or shortest-path

    # Eligibility Trace Matrix
    e = np.zeros((N_actions, N_states))

    # Obstacles matrix
    obstacles = np.array(([0, 5], [1, 5], [2, 5], [3, 5],
                [5, 3], [6, 3], [7, 3], [8, 3], [9, 3],
                [7, 6], [7, 7], [7, 8], [7,9]))
```

Variables to set the scene for our environment

```
## TODO: check if state is terminal and update the weights consequently
if s_index == s_end:
    delta = R + (gamma * Q[action]) - Q_old
    e[action_old, s_index] += 1
    e = e * gamma * lam + (output_old.dot(input_old.T))
    dw = learning_rate * delta * e
    weights += dw
if step == n_steps:
    delta = -R + (gamma * Q[action]) - Q_old
    e[action_old, s_index] += 1
    e = e * gamma * lam + (output_old.dot(input_old.T))
    dw = learning_rate * delta * e
    weights += dw
```

The code implementation for our leaning rule in the ANN's framework

```
# move the robot to it's previous location if it bounces into a wall and give negative reward.
if any(np.equal(obstacles,state_new).all(1)):
    r_old = -0.1
    state_new = state
```

A way to check if a state belongs to on of our pre-set obstacles

```
#put the robot back in grid if it goes out and give negative reward
if state_new[0] < 0:
    r_old = -0.1
    state_new = state
if state_new[0] >= N:
    r_old = -0.1
    state_new = state
if state_new[1] < 0:
    r_old = -0.1
    state_new = state
if state_new[1] >= M:
    r_old = -0.1
    state_new = state
```

Manage the negative reward when bouncing in a wall

```
def get_min_steps(x, y):
    return (abs(x[0]-y[0]) + abs(x[1]-y[1]))
```

Computes the Manhattan-distance for our Shortest Path reference when calculating the number of extra steps