Use of Reinforcement Learning to improve GNSS satellites signal acquisition search strategy

Giovani Gogliettino
ADG – AAD GNSS R&D
STMicroelectronics
Arzano (NA), Italy
giovanni.gogliettino@st.com

Fabio Pisoni
ADG – AAD GNSS R&D
STMicroelectronics
Cornaredo (MI), Italy
fabio.pisoni@st.com

Domenico Di Grazia
ADG – AAD GNSS R&D
STMicroelectronics
Arzano (NA), Italy
domenico.di-grazia@st.com

2024 IEEE International Workshop on Metrology for Automotive (MetroAutomotive) | 979-8-3503-8498-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/METROAUTOMOTIVE61329.2024.10615859

Abstract— **Satellite acquisition is a crucial procedure in GNSS receiver operation, which becomes particularly challenging when no assistance information is available - the so-called cold start conditions. In this paper, a machine learning approach will be introduced to improve the acquisition strategy. In detail, a reinforcement learning algorithm will be implemented to learn an optimal acquisition strategy.**

Keywords — GNSS, GPS, GALILEO, GLONASS, BEIDOU, Positioning, GNSS Acquisition, Machine Learning, Reinforcement Learning.

I. Introduction

A crucial procedure in GNSS receiver operation is the satellites acquisition, where the visible satellites' signals are searched through the possible frequency ranges. The coarse doppler and code delay of the signal candidates are estimated to be passed to the subsequent tracking stages. The acquisition search strategy is a critical step in the GNSS receiver because it affects the receiver's performance in terms of the time it takes to achieve the *first fix*.

The *first fix* is the computation of the receiver's position and velocity for the first time after it has been turned on or reset. To achieve this, it needs to track a sufficient number of satellites, and to download the navigation data which are transmitted by each of them. Navigation data contains both the ephemeris, which provides accurate satellite coordinates, and the almanacs, which provide approximate information about the position of all satellites of the same GNSS system.

The acquisition step is particularly challenging in case of *cold start*. A *cold start* condition happens when the receiver is turned on for the first time or after a long time. In those cases, it has no historical records both about the satellites (i.e. ephemeris and almanacs) and about its own previous position. This latter information can be used, along the almanacs, to guess which satellites are visible from that position, and therefore searching only for them. Without such information, most receivers use a simple selection algorithm (for instance, a fixed or random selection of the next satellite to try to acquire). But trying to acquire a satellite which is not currently visible in the sky will take time and therefore will delay the time for the first fix.

In this paper, a machine learning approach will be introduced to improve the acquisition strategy. In detail, a reinforcement learning algorithm will be implemented to learn an optimal acquisition strategy. This strategy will be compared to a random acquisition strategy to verify the improvement gained by the ML approach.

A simulation of the acquisition environment has been implemented using the *Gymnasium* library [9], which is a maintained fork of the OpenAI's *Gym* library. Such an environment has been implemented to emulate both the sky, in terms of visible and not visible satellites, and the GNSS receiver acquisition block in terms of internal status and different types of signal searches. Cold start acquisition can be treated as a stochastic process.

A random acquisition strategy algorithm was implemented as reference. Such an algorithm generates a sequence of acquisition searches, where each search type in the sequence has been chosen randomly. It ends when a given number of satellites have been acquired, or after a given number of steps, if not enough satellites have been successfully acquired until then.

A reinforcement learning, known as *Deep Q-learning* [1] and [7], was used to find an optimal acquisition strategy. It is part of a class of machine learning algorithms which try to learn the optimal strategy to solve a problem by trial and error, based on feedback (reward) received for any given action performed in a given state of the problem environment.

Deep Q-learning can learn the value of any action for a given environment, assuming that a sufficient number of actions is attempted (environment exploration). It belongs to the model-free algorithm class (i.e.: it does not need a prior model of the environment to work) and, if the environment can be modelled as a finite Markov decision process, then the Deep Q-learning algorithm is able to find the optimal policy and maximize the expected value of the total reward, from the current state for all successive steps.

The Deep Q-learning algorithm proposed in this paper is based on a *Deep Q network*: it is an artificial deep neural network which learns the values of any action for any given state. During the training phase, the algorithm is left to explore the environment, trying random actions for the observation it gets and checking the rewards it gets for such action to a given observation. Then it trains the network by applying the *Bellman equation* [5] to the rewards it collects during such exploration. Once the training ends, the network has learned the optimal values for each action in each state, and can be used to deploy the learned policy, i.e.: it can be used to choose the best action given the current state during the actual acquisition process.

This paper describes the details of the environment and Deep Q-learning algorithm implementation, and shows that, in the simulated environment, the number of steps and the total time to acquire a given number of satellites is shorter when the Deep Q-learning algorithm is employed, compared to the random strategy algorithm.

II. Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning algorithm that enables an *agent* to learn how to make decisions in an environment by interacting with it. The goal of RL is to learn an optimal policy that maximizes the cumulative reward over time.

In RL, the agent learns by receiving feedback in the form of rewards or penalties for its actions. The agent takes actions in the environment, and the environment provides feedback in the form of a reward or penalty. The agent's goal is to learn a policy that maximizes the expected cumulative reward over time.

At each discrete time step t, the agent receives a representation of the environment's state, S_t, and on that basis selects an action, A_t. At the next time step, based on the chosen action, the environment finds itself in a new state S_{t+1} and produces a numerical reward R_t, which is delivered back to the agent.

The interface between agent and environment allows for a continuous exchange of states, actions, and rewards, as from Figure 1, This formalism is known as Markov Decision Process (MDP) [6].

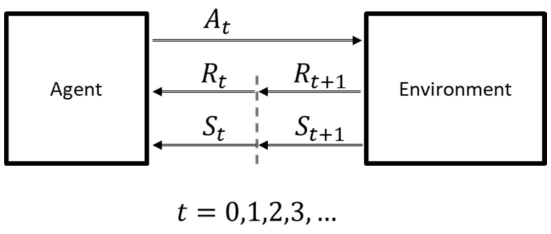


Figure 1 - MDP Agent-Environment interaction.

The state S_t is observed by the agent as a tuple O_t of state-variables [6]. The RL algorithm learns by iteratively updating the policy based on the feedback received from the environment. The policy can be represented as a mapping from states to actions, or as a function that takes in the state and outputs the action to take.

There are different types of RL algorithms, including model-based class and model-free class. Model-based ones learn a model of the environment and use it to plan actions. Model-free ones, on the other hand, learn the policy directly from experience without explicitly modeling the environment.

RL has been successfully applied to a variety of problems, such as game playing, robotics, and autonomous driving. It is a powerful and flexible algorithm that can learn from raw sensory inputs and can handle high-dimensional state and action spaces.

III. Acquisition Environment Simulation

To train the RL algorithm an environment is needed. Such an environment can be simulated to make it faster and more practical to train and test RL algorithms. In reinforcement learning, an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties.

The environment provides the agent with the observation of the current state, and the agent takes *an action* based on that observation. Then the environment provides *a feedback* in response to such action, in the form of a reward or penalty, which the agent uses to update its decision-making process.

Reinforcement learning environments can be simple or complex, depending on the problem being solved. For example, a simple environment might be a game like Tic-Tac-Toe, while a more complex environment might be a robotics task like navigating a maze.

OpenAI Gym and Gymnasium are examples of libraries that provide a collection of reinforcement learning environments that can be used to train and test reinforcement learning algorithms, as well as an API to develop custom environments.

In detail, Gymnasium is a Python library that is open source and used for developing and comparing reinforcement learning algorithms. It provides a standard API that allows communication between learning algorithms and environments, as well as a standard set of environments that are compliant with that API. Gymnasium is a fork of OpenAI's Gym library, which was handed over to an outside team for maintenance a few years ago, and in which any future maintenance of the library will be done.

The GNSS acquisition environment proposed in this paper is a simplified one, because a real GNSS acquisition is very complex, and this study aims to understand if the use of RL in GNSS acquisition can bring any improvement, and not to provide a complete solution out of the box. Nevertheless, the main aspects of GNSS acquisition have been considered, to provide useful information anyway.

The GNSS acquisition problem will be modeled as an MDP, and reinforcement learning will be applied to this formalism to learn an optimal policy. The GNSS receiver represents the environment. An acquisition agent will be developed to learn this optimal policy through interactions with the environment.

IV. MDP Definition

The state observations O_t is defined in Table 1.

Table 1 - MDP Definition of State Observation.

Observation State-Variable	**Note**
O_DONE	0 = current episode is still ongoing, 1 = current episode ended (episodic case)
O_FIX	Type of GNSS fix: 0 = No-fix, 2 = 2D-fix, 3 = 3D-fix
O_<sat_type>_RTA	Number of remaining satellites to be acquired for the <sat_type> GNSS constellation (with respect to the selected target)
O_<sat_type>_VNT	Number of satellites which are visible (i.e.: could still actually be acquired) but have not been acquired yet, for the <sat_type> GNSS constellation
O_<sat_type>_NFAIL	Number of consecutive failed acquisition attempts for the <sat_type> GNSS constellation

Where: *<sat_type>* can be one of: GPS, QZSS, GLO, GAL, BDS, IRNSS.

If any of these constellations are not configured to be used, then the corresponding input will assume special values to not interfere with the learning process.

The actions A_t are chosen from the a finite set:

TABLE 2 – SET OF ACTIONS.

Actions		
HOT search	**COLD search**	**FULL search**
A_GPS_HOT	A_GPS_COLD	A_GPS_FULL
A_GLO_HOT	A_GLO_COLD	A_GLO_FULL
A_GAL_HOT	A_GAL_COLD	A_GAL_FULL
A_BDS_HOT	A_BDS_COLD	A_BDS_FULL
A_QZSS_HOT	A_QZSS_COLD	A_QZSS_FULL

HOT, *COLD*, and *FULL* searches differ for the frequency bandwidth that shall be searched at the next acquisition. *COLD* will search through the maximum bandwidth (doppler + clock drift), *FULL* will search through an extended bandwidth, assuming higher clock drift, and *HOT* will search through a narrow bandwidth around the predicted satellite frequency.

The number of steps of non-coherent accumulations (i.e., the number of times that the magnitudes of the signal samples are summed up to increase the signal-to-noise ratio) shall be associated to each search type.

The reward *R* shall be generated based on heuristics and on the result of the last acquisition:

For every successful acquisition *R* is set to a positive value, as described below.

For every failed acquisition *R* is set to a negative value, as described below.

More in details, the simulation environment has been defined in the following fashion:

1. The satellite search list, which contains the list of satellites to search for in a predetermined order, and the visible satellite list, which contains the list of satellites which are currently visible for the current sky view, are initialized.
2. The satellite to be acquired is selected by satellite acquisition selection procedure described below.
3. If the selected satellite is not visible according to the current sky view the acquisition fails and the reward is set to -1.
4. If the selected satellite is not visible (according to the current sky view) then the max reward factor f_{max} is set to *2*, but the actual reward depend on the search action (f_{max} for *HOT search*, *1* for *COLD search* and *0.75* for *FULL search*), the acquisition is tried according to the description which is provided below.

   If the acquisition succeeded then the *Remaining To Acquire (RTA)* counter is decremented, the reward factor f is set to f_{max}, and the reward R is computed according to the following formula:
   $$R = \frac{1}{t_{acq}} + \frac{f}{f_{max}} \tag{1}$$

   where the acquisition time t_{acq} is set by drawing a sample from a Rayleigh distribution in which the σ parameter has been set to f (A large number of acquisition attempts have been observed in real GNSS receivers and their durations have been recorded, and analyzing those records it has been seen that Rayleigh distribution is a good approximation of their distribution).

   If the acquisition fails then the reward *R* is set to *-1* for the *FULL* search, *-0.75* for the *COLD* search actions, or to $-f_{max}$ for the *HOT* search action.
5. If all satellites have been acquired (*RTA* = 0) then the current episode ends. The episode will eventually end if a maximum number of steps has been reached (e.g. 100) regardless of the number of satellites acquired so far. In this latter case the reward *R* is set to *-10000.*

The procedure for the selection of the next satellite to be acquired is carried out according to the following steps:

1. The number of acquisition attempts is set. If the position fix has not been achieved yet then it is set to *3*, otherwise it is set to the length of the satellite search list.
2. The next satellite in the satellite search list is selected. In the case the position fix has already been achieved, then the next satellite in the satellite search which is also in the visible satellite list is selected.
3. If the selected satellite has already been acquired, then the next satellite is selected according to what is specified in the previous step. This step is repeated until a satellite which has not been acquired yet is selected or the number of times reaches the number of acquisition attempts set at the first point.

The satellites acquisition attempt is done in the following way:

1. In the case the selected satellite is not visible in the current sky view, then the acquisition fails.
2. If the selected satellite is visible, the factor f is set according to the following formula:
   $$f = \frac{C/N_0}{C/N_{0\,max}} \tag{2}$$

   where C/N_0 is the carrier to noise density of the current satellite and $C/N_{0\,max}$ is the maximum value that carrier to noise density can assume.
3. In case of the current action is an *HOT search*, then the value *+1* is added to the factor f in the case the position fix has already been achieved, otherwise *-1* is added to such factor.
4. The POD (probability of detection) is set to the non-normalized value (i.e.: it is not in the [0,1] range):
   $$POD = r + f \tag{3}$$

   where r is a sample drawn from a normal distribution with σ = 1.
5. If the POD computed in the previous step is greater than 0 then the acquisition succeeded, otherwise it fails.

V. Deep Q-learning

Q-learning is a reinforcement learning algorithm that learns to make decisions by estimating the expected reward for taking a particular action in a particular state.

For this study the *Deep Q-learning* algorithm [1] has been used. It is a type of reinforcement learning algorithm that uses a *deep neural network* to approximate the *Q-values* of state-action pairs in a given environment. The *Q-value* represents the expected reward that an agent will receive by taking a specific action in a specific state.

The Deep Q-learning algorithm works by iteratively updating the Q-values on the experience gained from interactions with the environment using the *Bellman equation* [5].

The Bellman equation describes the relationship between the value of a state and the values of its neighboring states. It is named after Richard Bellman, who introduced the equation in the 1950s.

It is a recursive equation that expresses the value of a state as the sum of the immediate reward obtained in that state and the discounted value of the next state. The discount factor is used to give less weight to future rewards, since they are uncertain and may not be realized.

The equation can be written as follows:

$$V(s) = max_a(R(s,a) + \gamma \cdot \sum_{s'}(P(s'|s,a) \cdot V(s'))) \quad (4)$$

where:

- *V(s)* is the value of the state *s*.
- *R(s, a)* is the immediate reward obtained by taking the action *a* in the state *s*.
- γ is the discount factor, which determines how much weight is given to future rewards.
- *P(s' | s, a)* is the probability of transitioning from the state *s* to the state *s'* when taking the action *a*.

The *max* operator selects the action *a* that maximizes the value of the next state, and the sum over *s'* represents the expected value of the next state, weighted by the probability of reaching that state.

It is worth noting that the *value* of a state is the expected future reward of following the optimal policy from that state, while the *Q-value* of a state-action pair is the expected future reward of taking a particular action in a particular state and then following the optimal policy.

The Bellman equation is used to iteratively update the values of the states until they converge to their optimal values, and it provides a way to estimate the value of a state based on its neighboring states.

Besides, the Deep Q-learning algorithm selects actions based on an *epsilon-greedy* policy [7], which means that it selects a random action with a certain probability *p* and selects the action with the highest Q-value with a probability *1-p*. It stores the experiences in a *replay buffer* and uses a batch of these experiences to update the neural network parameters using gradient descent. The goal of the algorithm is to learn the optimal policy for the environment, which is the policy that maximizes the expected cumulative reward over time.

Deep Q-learning algorithm is a powerful and flexible algorithm that can handle high-dimensional state and action spaces, in contrast to previous Q-learning algorithms, which use a table to store the Q-values and therefore require a very large amount of memory for high-dimensional state and action spaces.

VI. Algorithm and Agent Definition

In details, the algorithms used in this study works as described in the following steps:

1. First, the algorithm initializes a neural network with random weights. This neural network is called the *Q-network*, and it takes in the current state of the environment as input and outputs a *Q-value* for each possible action.
2. The algorithm then starts interacting with the environment and collecting experience tuples (state, action, reward, next state). These tuples represent the agent's experience in the environment and are stored in the *replay buffer.*
3. At each time step, the algorithm selects an action to take based on an *epsilon-greedy policy*. This means that the agent selects the action with the highest Q-value with probability *1-* ε and selects a random action with probability ε.
4. After taking an action, the algorithm observes the reward and the next state of the environment and stores the experience tuple in the replay buffer.
5. The algorithm then samples a batch of experience tuples from the replay buffer and uses them to update the Q-network. The update is done using the following formula, obtained from the Bellman equation (4):

   $$Q(s,a) = R + \gamma * max(Q(s',a')) \quad (5)$$

   where *Q(s,a)* is the Q-value for the current state and action, *R* is the reward for taking that action, *s'* is the next state, and γ is a discount factor that determines the importance of future rewards. The *max(Q(s',a'))* term represents the maximum Q-value for the next state.
6. The algorithm trains the Q-network using the sampled experience tuples and the updated Q-values. The training is done using a gradient descent algorithm to minimize the difference between the predicted Q-values and the target Q-values (i.e., the right-hand side of the Q-learning update equation).
7. The algorithm repeats steps 3-6 for a fixed number of iterations or until convergence.

Overall, the Deep Q-learning algorithm uses a neural network to approximate the Q-values for each state-action pair and updates the network using experience replay and the Q-learning update rule. This allows the agent to learn an optimal policy for the environment over time.

In summary, the RL Agent is defined as follows:

- *Episodic Case*: an *episode* is a sequence of steps starting from the beginning of the acquisition attempt and ending when the number of acquired satellites is equal to the respective RTA for each available constellation. The episode will also end if a maximum number of steps is reached (100 in the case described in this paper). In this latter case, the reward of the last step will be a large negative number (-10000).

- *Stochastic environment*: the GNSS acquisition, for its own nature, is a stochastic process.
- *Discrete Action Space*: all the possible actions are specified in Table 2.
- *Discrete State*: the state is described in Table 1, and each field of the table can assume a discrete value.
- *Algorithm*: Deep Q-learning.
- *Function approximation*: Deep Neural Network.

VII. Neural Network Implementation

In Deep Q-learning the agent is implemented with a *Deep Neural Network* (*DNN*); such a type of artificial agent is also called *Deep Q-Network* (*DQN*) [1]. A DNN is a type of artificial neural network which is organized in multiple layers of units, where each layer feeds the next one.

In this study the DNN has been implemented using *TensorFlow* [8], which is an open-source, end-to-end platform for machine learning.

The structure of the DNN used is shown in Figure 2 and its layout and number of parameters are shown in Figure 3:

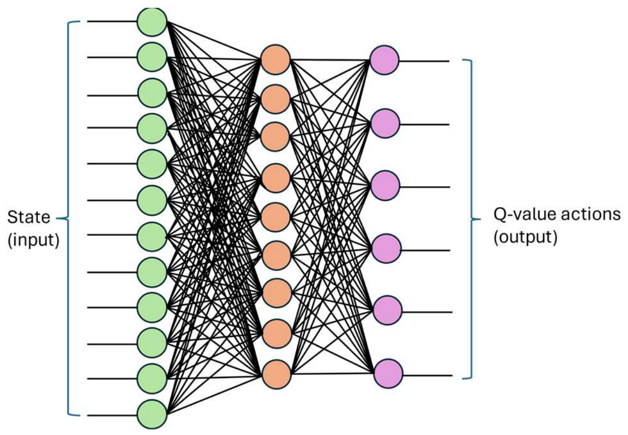
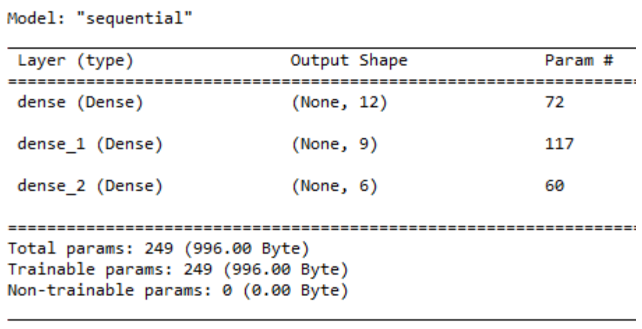


Figure 2 – Structure of the DNN used.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 12)                72

 dense_1 (Dense)             (None, 9)                 117

 dense_2 (Dense)             (None, 6)                 60

=================================================================
Total params: 249 (996.00 Byte)
Trainable params: 249 (996.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 3 – DNN layout and number of parameters.

The activation functions used are the *Rectified Linear Unit* (*ReLU)* for the first and second layer, and the *linear* activation function for the output layer. Note that, because of the linear activation function in the output layer, the sum of the outputs will not be normalized to 1.

The DNN's inputs are a representation of the state of the environment, while the outputs represent the Q-values of each action for that state. Therefore, the action to undertake for a given inputs is the one corresponding to the largest Q-value. For example, if the output of each output unit (from #1 to #6) are [0.4, 0.6, 0.2, 0.01, 0.03, 0.004] the action #2 should be chosen because it has the highest Q-value (0.6).

For the network weights initialization, the He Uniform distribution [2] has been used, while the *Huber* loss [3] has been used as loss function, and the *ADAM* algorithm, which is a stochastic gradient descent method based on adaptive estimation of first and second-order moments, has been used for optimization [4]. Such configuration allows to get better performance from the DQN algorithm implementation.

In Deep Q-learning two neural networks with the same architecture are used for the learning process: the *main* network and the *target* network. Those networks have different weights, and every *N* steps, the weights from the *main* network are copied to the *target* network. That is because it has been observed that the use of two networks brings more stability to the learning process and allows it to improve learning effectiveness. In the implementation presented here, the weights of the main network are replaced with the ones from the target network every 50 steps.

VIII. Training

To perform the DNN's learning process a training dataset (which has been generated using the simulated environment described in the paragraphs III and IV) has been used, which contains the actions done along with their respective rewards. As wrote in paragraph VI, a replay buffer has been used to build up the training dataset by collecting a set of examples along with their own rewards.

More in details, after that an action has been selected using the epsilon greedy policy, the agent to perform such action, stores it in the replay buffer along with the previous state, the reward it gets from such action and the new state after that action, Then, periodically (in our case each four steps, or when an episode ends), the main networks is trained by sampling a mini batch from the training dataset, which is built by getting the states from the replay buffer as examples, and getting the labels (i.e.: the respective rewards) from the same buffer and using the (5) to compute them. Besides, every 50 steps, the weights of the main network are copied to the target network.

Furthermore, the ε value used in the epsilon greedy policy is not constant, but it is decreased episode by episode, from an initial value set to 1.0 to a minimum of 0.01 with a decay rate of 0.02. That has been done because at beginning of the training a large ε is preferred to allow the agent to explore the environment while, as the training proceeds, the knowledge learned so far from the network can be exploited. Figure 4 shows the value of ε in function of the training episode number.

The training ends after a predetermined number of episodes, which has been set to 100 because it has been seen that it provides good results. Figure 5 shows the training loss for the main network, where each epoch corresponds at each time when main network is trained.

Since the maximum number of steps in an episode is set to 100 as well, then the number of total steps for the training phase is equal to or less than 10,000.

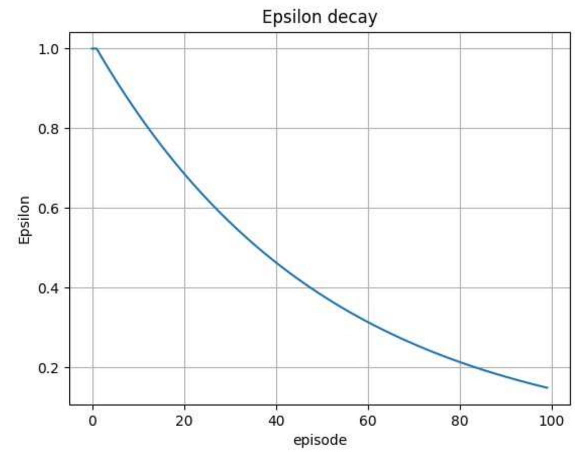


Figure 4 - Epsilon decay during training.

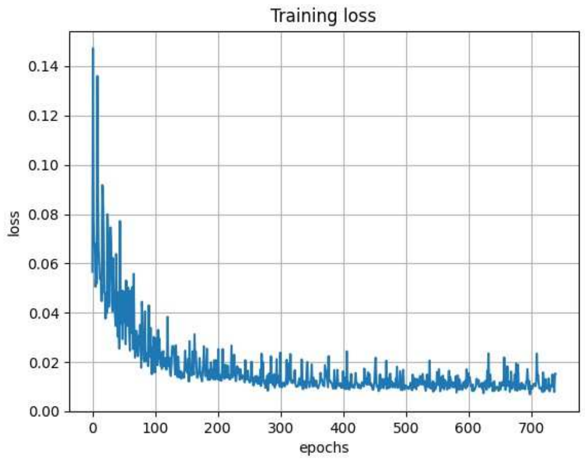


Figure 5 - Training loss.

IX. Results

For the testing of the network performance, GPS and Galileo constellations has been used because they allow to simulate a multi-GNSS satellites' constellation environment and, at same time, offered a faster training and testing time compared to the use of all satellite's systems.

The test has been carried out by playing 100 episodes of maximum 100 steps each (i.e., with a maximum of 10,000 steps in total), and by calculating statistics over them. The parameters considered were:

- The *mean* and the *standard deviation* of the number of steps to complete the episode (i.e.: after the number of satellites acquisitions equal to the RTA value set at beginning for each available satellite constellation).
- The *mean* of the total reward collected for each episode.

As benchmark, two other acquisition strategies have been performed on the same number of episodes:

- A *random acquisition strategy*, which chooses the next action by sampling an integer number from zero to number of all possible actions minus one. This may be considered as the baseline.
- A *manually engineered strategy*, which performs FULL and COLD searches action on all available constellation until the position fix is achieved, and the perform HOT searches actions. Due to some simplification that has been done in the modelling of the environment, this strategy is particularly successful for such an environment. However, this strategy does not account for the stochasticity of the environment.

In Table 3 are reported the results of such test:

Table 3 - Results

Strategy	Mean of the # of steps	STD of the # of steps	Mean of the Total reward
DQN	22.29	6.169757	9.363001
Random	27.20	9.389356	-3.3351036
Man. Eng.	23.23	6.540420	2.0169490

X. Conclusion

The results reported in IX showed that the DQN can perform not just better than a random GNSS satellites signals' acquisition strategy, but also of a strategy which has been manually engineered using the knowledge of the simulated environment, because DQN can better model the stochasticity of the environment itself.

However, as already mentioned, the simulation environment is a simplification of the actual environment, including some assumptions that have been made in terms of probability distributions of the events.

Nevertheless, this study has shown that a Deep Q-Learning solution, in addition to take into account the deterministic part of the environment, can better model the stochastic part, which is more difficult to figure out in the real world.

So, it is worth a further and more profound investigation into such kinds of solutions, using a more detailed and comprehensive simulated environment, or a real environment as well, to explore and assess them more in depth.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Delving Deep into Rectifiers, "Surpassing Human-Level Performance on ImageNet Classification" (2015). https://doi.org/10.48550/arXiv.1502.01852

[3] Peter J. Huber, "Robust Estimation of a Location Parameter" (1964). Ann. Math. Statist. 35(1): 73-101 (March, 1964). DOI: 10.1214/aoms/1177703732

[4] Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization (2014). https://doi.org/10.48550/arXiv.1412.6980

[5] Bellman, Richard (1954) "The theory of dynamic programming", Bulletin of the American Mathematical Society, 60 (6): 503–516, doi:10.1090/S0002-9904-1954-09848-8

[6] Richard S. Sutton, Andrew G. Barto, "Reinforcement Learning: An Introduction" (Adaptive Computation and Machine Learning series) 2nd Edition, Bradford Books, 2018, ISBN 978-0262039246

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, "Playing Atari with Deep Reinforcement Learning" (2013), arXiv:1312.5602v1. https://doi.org/10.48550/arXiv.1312.5602

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

[9] https://gymnasium.farama.org/