Machine Learning Project

Fabio Massimo Ercoli

July 2024

1 Introduction

We're presenting two possible Q-learning implementation approaches:

- Tabular based. Implemented using a numpy dictionary.
- Neural network based. Also known as Deep Q-Network (DQN).

2 Tabular Q-Learning

2.1 Running the project

To trigger the learning process run from the *machine-learning-project* directory the following python command:

```
python qlearn-app.py
```

You should see a message similar to the following:

```
observation space Discrete(500)
action space Discrete(6)
avg return before learning -92.98608555399927
avg return after learning 2.5552244875845997
```

The observation and the action space description is related to the *Gymnasium* environment we used to train and test our agent, that is the *Taxi-v3 environment*[1] that has 500 discrete possible states and 6 possible actions.

The *avg return* is the average of the score gained rolling out a series of episodes. In the case of this project 5 episodes are executed before the training and 20 after the training. The expectation is that the agent can get a better score after we properly trained it.

An episode starts from the initial state and ends in case of a termination state has been reached or can be truncated if the **max episode steps** value has been reached.

[1] $https://gymnasium.farama.org/environments/toy\_text/taxi$

This is the first value we can change to tune the learning, in general this value should be great enough to allow the agent to learn and to rollout correctly the strategy it learned. We decide to increase it from 200, that is the default for this environment, to 500.

How can we evaluate the goodness of learning? In this case we simply compare the average score for an episode performed using a random strategy (before learning) with the average score obtained after the learning.

2.2 The rollout and the score

The score is a crucial concept, since the learning activity is entirely oriented to maximize this value.

The score is the summation of all the reward we get from the environment every time we execute an action (notice that can be negative or positive), multiplied by the **discount factor** γ.

The meaning of this value is to promote not only the rewards collected, but also the speed with which we get them. This is another value that we need to balance, since if it is too low, we can penalize the learning of long term goals.

The score depends on the actions the agent chooses, those are randomly selected before the training and after the training they are chosen according to the learned policy.

The rollout procedure will collect the score for each epoch, applying the current discount factor to the reward. Finally, the value is averaged and returned to the caller.

2.3 The learning and the Q function

The output of the learning is a policy to choose for any given state (or observation)[2] an action among all the possible actions that are possible to apply. The Q function associate to a given state and a given action the expected total score of taking the action in the given state and then continue to choose optimally the further actions. So we can use the Q function to choose the policy as the action that maximizes the expected total score.

We call it a *greedy policy*, since it maximizes the expected score not considering the fact that unexplored paths may possibly lead to even greater scores, improving the Q function. At the begin of the learning we want always to apply a random strategy to learn as much as possible ($\epsilon = 0$). At the end of the learning on the other hand we want to exploit the knowledge we've acquired to perfectionate the policies on areas that we already know to be good ($\epsilon \approx 1$). In this project we use a linear decay from 0 to 1 of the **greedy factor** ϵ for the learning. Other decay functions are of course possible.

How should the learning last for? In this project the learning finishes as soon as we run a number of actions equals to the **learning steps**, and we set this value to 200,000.

[2]In the context of this project that state is always fully observable, so we will use the term state and observation interchangeably

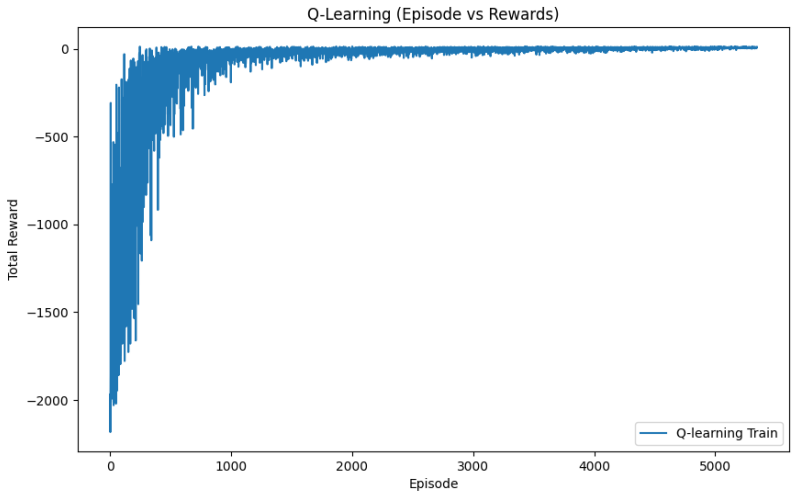

Figure 1: Q-Learning

The last crucial setting we present in this chapter is the **learning rate** α_0. The Q function is updated for supporting the indeterministic environments according to the formula:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{\text{a'}}(Q(s',a'))] \tag{1}$$

Where s' is the state we get from s applying the action a, and the $\max(Q(s',a'))$ is calculated among all the possible actions a' executable from s'.

2.4 The tabular approach

In this implementation the values for the Q function are stored as table items, in particular in order to represent only the subset of state we're interested in. In this project we use a dictonary (instead of an array).

This means that every time we update an entry on the Q function we operate on a discrete value of the observation and on a discrete value of the action.

Thus in order to support with this approach continuous environment, we need first to apply a discretize of the state (and/or the action).

Figure 1 presents the graph of the sum of the total rewards (not discounted) per episode get during the learning process.You can see the improvement given by the fact that the:

- Moving forward the actions are increasingly greedy and less random.
- The agent is actually learning how to act better.

3 Deep Q-Learning

3.1 Running the project

To trigger the learning process run from the *machine-learning-project* directory the following python command:

```
python dqn-app.py
```

You should see a message similar to the following:

```
observation space Box([. . . .], [. . . .], (4,), float32)
action space Discrete(2)
avg return before learning 13.160926128670493

... a long series of logs from TensorFlow ...

avg return after learning 98.27961935316044
```

The observation and the action space description is related to the *Gymnasium* environment we used to train and test our agent, that is the *CartPole-v1*[1] that has 4 continuous variables to denote the state and 2 discrete possible actions.

The *avg return* is the average of the score gained rolling out a series of episodes. In the case of this project 5 episodes are executed before the training and 20 after the training. The expectation is that the agent can get a better score after we properly trained it.

An episode starts from the initial state and ends in case of a termination state has reached or is truncated if the **max episode steps** value has been reached.

3.2 Network design

A completely different way to represent the Q function is to use a neural network. In this case the learning process consists in the training of the network.

In the implementation provided by this project the neural network has been designed to have:

- One input node for each state value, in the *input layer*.
- One output node for each action that can be performed, in the *output layer*.
- Two middle layers of size 24 each.

[1] $https://gymnasium.farama.org/environments/classic\_control/cart\_pole$

In the output layer we use a linear function (returning any real value) for each action, since the result of $Q(s, a)$ can assume any real value.

In the middle layers we use the *relu* activation function, that provides the fundamental non linear property to the network.

The network is dense, so the nodes between two adjacent layers are fully connected.

3.3 Mini batch approach

Each time we execute a step among the environment we collect the tuple:

```
< state, action, reward, next_state, done >
```

in the *replay buffer*. We want to keep only the last *50,000* events, so we implemented it as a bounded deque.

We train the network using a minibatch strategy that consists in randomly selecting a fixed size batch of events, in our case *128*, from the replay buffer and use it to train the network, as a unit training step.

The benefits of this approach are that:

- We invoke the *predict* function on an array of observations, instead of invoking a *predict* on each observation.
- We apply the changes to the network, using *fit* on an array of inputs and on an array of outputs, instead of on a single single input and a single output.
- The fact that every time we select a random subset of events to train the network promote the stochastic gradient descent, that allows to be not trapped in a local minimum when we compute the loss function.

3.4 Policy and target networks

In this project we used two networks, having the same architecture but different weight:

- A policy network, also called the main network:
  - Updated every very few steps (4 in our project)
  - Used to get the $Q(s, a)$ current value
- A target network:
  - Synchronized each time only after we execute 10 episodes
  - Used to predict the $\max_{a'}(Q(s', a'))$, when we calculate a new Q value

This strategy leads to more stability in the learning process.

3.5 Difference with the tabular approach

There are two main difference, compared with the tabular Q-learning:

- The Q function values are approximated, not exact
- With the DQN we can train agent in environments having a continuous state values without the need of discretize the inputs

On the other hand other aspects are in common between the two approaches, such as the formula to compute the new $Q(s, a)$ value, or the application of the epsilon strategy to choose between the exploration action or a greedy (policy) action during the learning process.

3.6 DQN project models

I tried to steal some ideas from some open source pre existing projects from the web. In particular from *minDQL*[2] on which I also to contribute with two *pull requests*[3] to align the source code with the last versions of TensorFlow and Gymnasium and to plot the graph of the total rewards per episode during the training.

The stealed ideas were:

- To use two neural networks instead of one to stabilize the learning (see Policy and target networks)
- To run the replay procedure from the replay buffer not at after each step to make the learning faster
- To train the network on an array of Q values instead of on each at the time [4]

And I applied them to my project.

3.7 DQL variables and results

The following are the parameters we use to run the learning:

[2] minDQL https://github.com/mswang12/minDQN

[3] My pull requests: https://github.com/mswang12/minDQN/pull/9, https://github.com/mswang12/minDQN/pull/10

[4] This was also mentioned in the course, but the project helped me in having a reference implementation about how to pass the matrix parameters to the networks

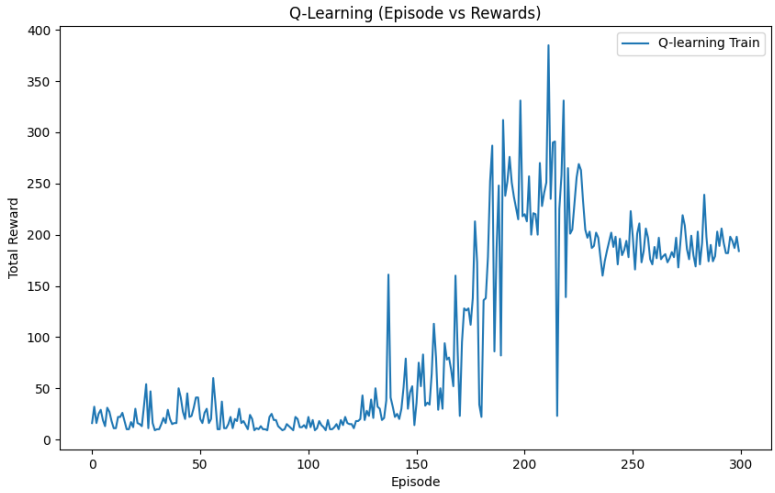

Figure 2: DQN learning - execution 1

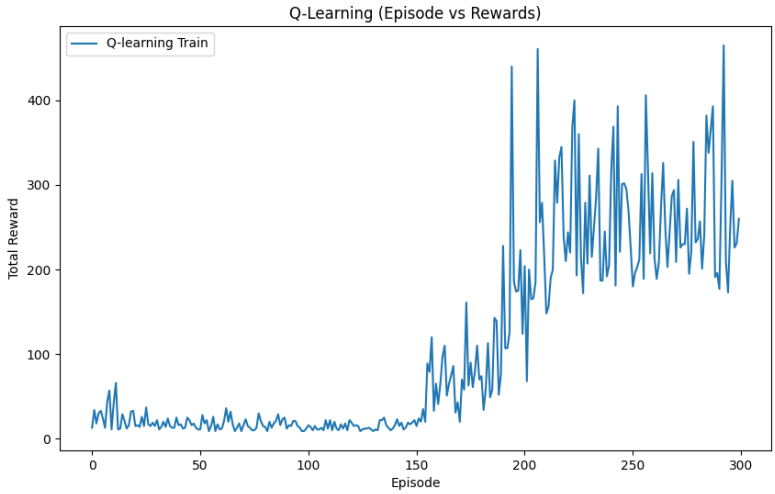

Figure 3: DQN learning - execution 2

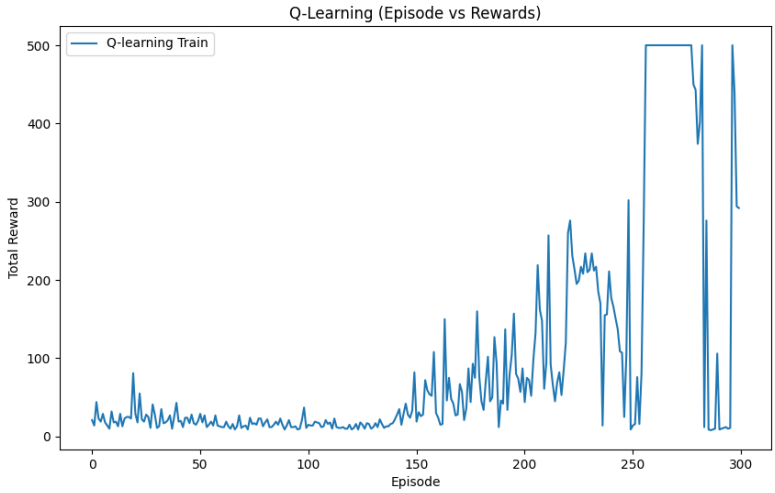

Figure 4: single network + small replay buffer

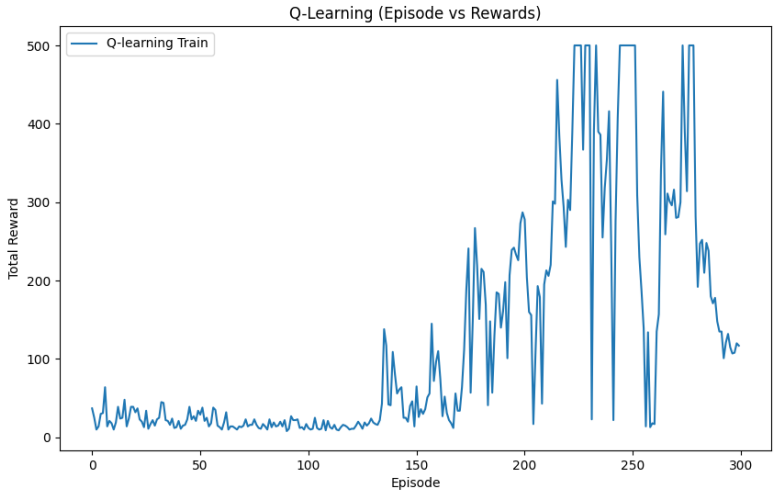

Figure 5: small replay buffer

learning rate	α_0	0.1
exploration / greedy	ϵ	linear decay from 0 to 1
discounting factor	γ	0.95
replay buffer size	-	50,000
min replay buffer size	-	1,000
mini batch size	-	128
train episodes	-	300
steps to update policy model	-	4
steps to update target model	-	100
max episode steps	-	500
network learning rate	-	0.001

The total rewards (not discounted) per episode get during the learning process using these parameters are presented in figures 2 and 3.

While figures 4 and 5 are the result of two learnings runned setting by mistake the replay buffer size to 1,000 instead of 50,000.

We can see that the total rewards can even reach the value of 500, but the learning is in general less stable. This is an example about the fact that the changing the values of the parameters can seriously affect the learning process.

4 Play with the parameters

In this final section we're going to explore the possibility of changing some of the parameters and try to evaluate the effects on the learning process, looking at the learning curve.

Raise the learning rate The first attempt we made is to raise the learning rate from 0.1 to 0.7.

The result was the graph depicted in figure 6. Even if the change looks quite significant (from 0.1 to 0.7) we can see that the graph is pretty similar for instance to 5. Anyway the graph looks pretty nice, so we can keep this change.

Lower the disconting factor The second attempt we made was to lower the disconting factor from 0.95 to 0.618. But the result, depicted in figure 7 was not so good.

We introduced some instability in the learning, so we decided to rollback this change.

Higher the disconting factor Since raising the disconting factor seems to be not a good idea. We tried to raise it to 0.99.

The result is depicted in figure 8 and it looks nice to me, since the score is always higher than 200 in the last 50 episodes and reaches also very high rate. So we decided to keep this change.

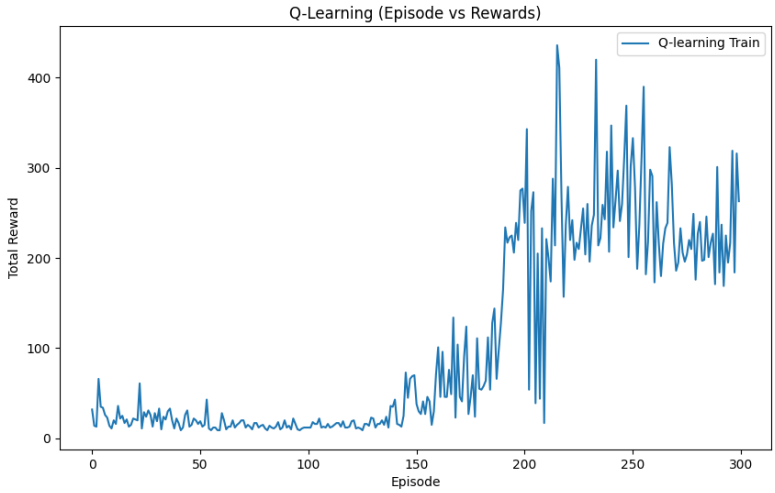

Figure 6: learning rate from 0.1 to 0.7

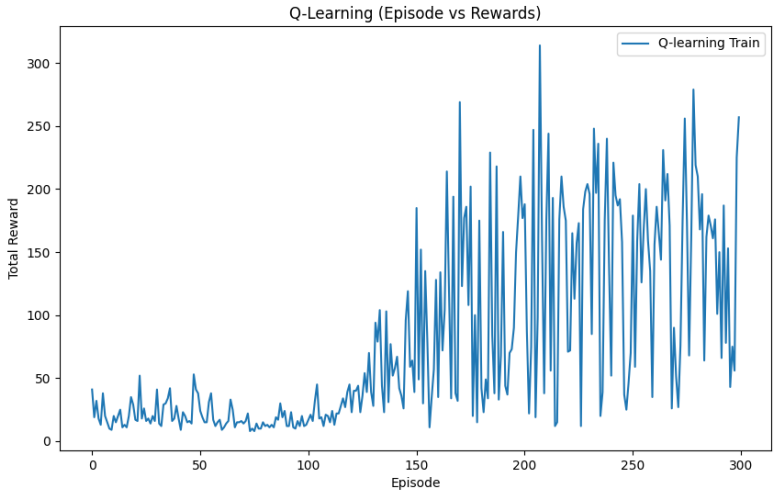

Figure 7: disconting factor from 0.95 to 0.618

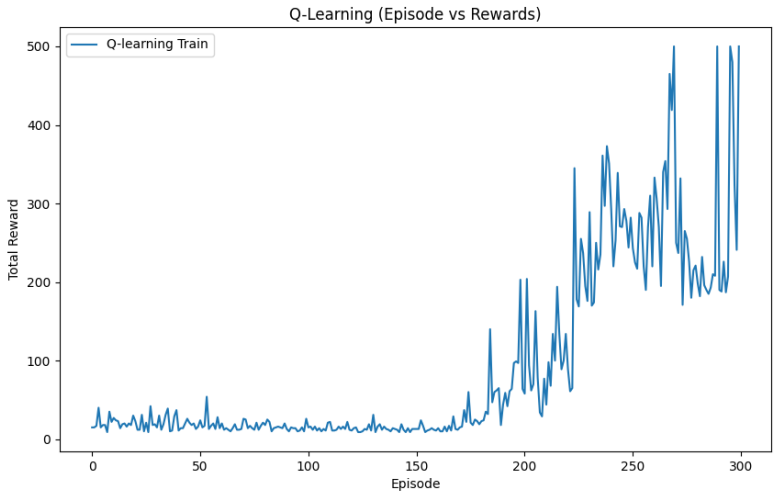

Figure 8: disconting factor from 0.95 to 0.99

Lower the replay buffer size The experience we had with the mistake of lowering too much the size of the reply buffer made me thinking about the fact that we try something in the between, so we tried to lower the reply buffer size from 50,000 to 35,000.

The result is depicted in figure 9. We decided to rollback this change since even if the graph showed a nice stability from episode 200 to the end, the final total rewards look less high.

Lower the network learning step Sometimes makes sense to lower the learning step of the network to reach an ideal local minimum, we called this value the network learning rate. Tring to lower it, we did get good results.

The corresponding graph is depicted in figure 10. The results are great only in the very last episodes. So I decided to rollback this change.

The final change This is last attempt we made playing with the parameters to improve the learning. We said that one idea we stealed from the *minDQL* project was the the fact of not updating for every step the neural network, but instead doing that every 4 steps to make the learning faster.

This time we try to lower this value to 2 and see the effects. But not only. We also change the learning rate from 0.7 to 0.3 and the replay buffer from 50,000 to 35,000.

The results are depicted in figure 11 and I think that they are really nice.

We can see that the total rewards are very high starting from episode 170, providing both stability and the high rates at the same time. So the following

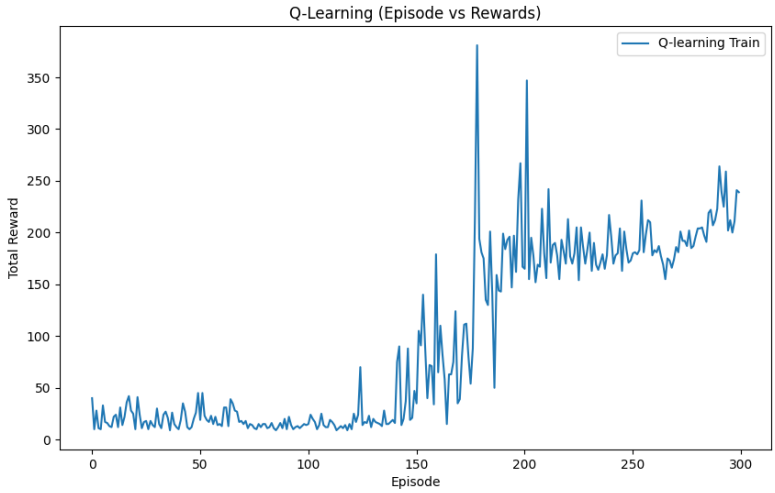

Figure 9: replay buffer from 50k to 35k

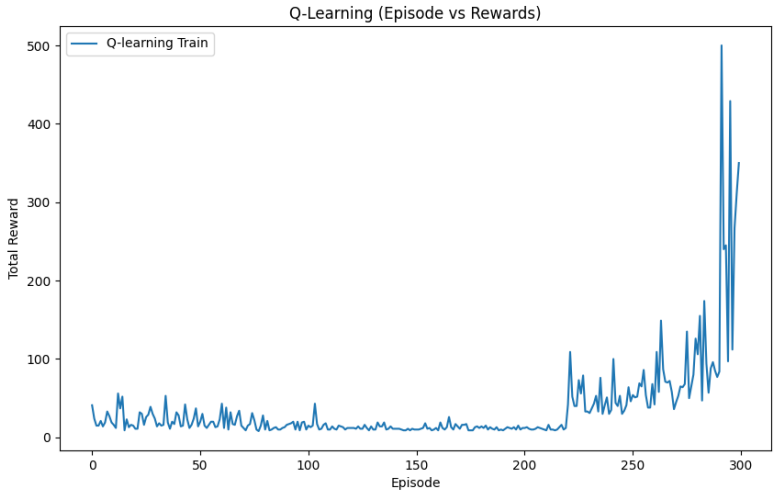

Figure 10: network learning step to 0.0005

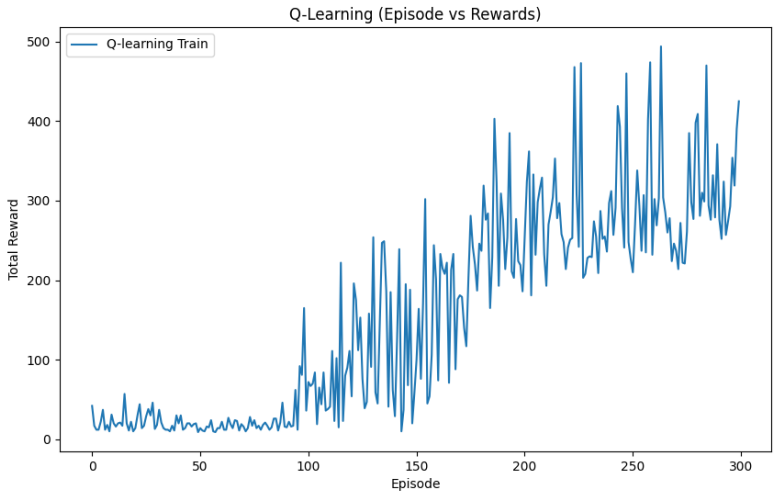

Figure 11: final parameters

are the parameters we decided to keep for this project:

learning rate	α_0	0.3
exploration / greedy	ϵ	linear decay from 0 to 1
discounting factor	γ	0.99
replay buffer size	-	35,000
min replay buffer size	-	1,000
mini batch size	-	128
train episodes	-	300
steps to update policy model	-	2
steps to update target model	-	100
max episode steps	-	500
network learning rate	-	0.001