Machine Learning Project

Fabio Massimo Ercoli

July 2024

1 Introduction

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

- Tabular based. Implemented using a numpy dictonary.
- Neural network based. Also known as Deep Queue 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 code:

```
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 episod starts from the initial state and ends in case of a termination state is reached or is trunkated since the **max episode steps** value is reached.

This is the first value we can change to tune the lerning, in general this value should be great enough to allow the agent to learn and to rollout correctly the

[1] $https://gymnasium.farama.org/environments/toy_text/taxi$

strategy it learned. We decide to increase it from 200, that is the default, to 500 and this change seems to provided better performance to the learning.

How can we evaluate the goodneed of the learning? In this case we simpy compare the average score for an episode perfomed using a random strategy (before to learn) with the average score obtanied after the learn.

2.2 The rollout and the score

The score is a crucial concept, since the learning activity is entirly oriented to maximaize 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. The score is averaged and retured 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 (observation)[2] an action among all possible actions to apply. The Q function associate to a given state and a given action the expected total score of taking the action in the state and then continue to choose optimally the futher actions. So we can use the Q function to choose the policy as the action that maximaizes the expected total score.

We call it the *greedy policy*, since it maximaizes the expected score not conisidering 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 of course are 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**. In this project 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 interchangebly

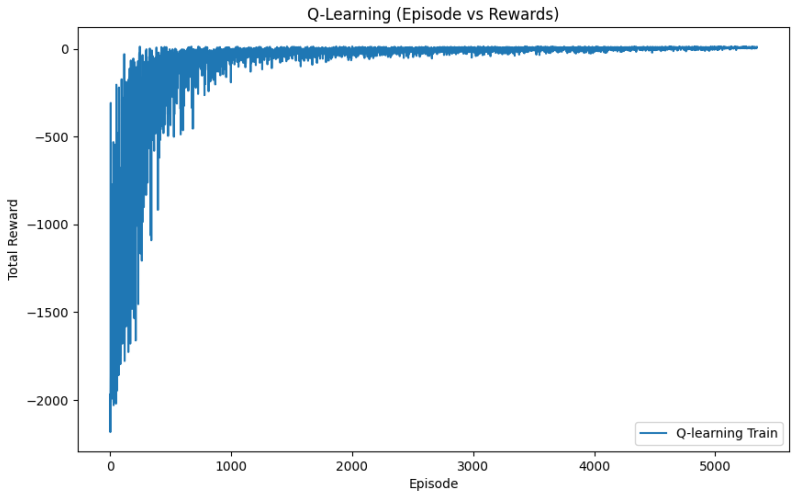

Figure 1: Q-Learning

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 action 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 we use in this project 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 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 code:

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

3.2 Network design

A completely different way to represent the Q function is to use a neural network to denote it. In this case the result of the learning will be 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.

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 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 benefiths of this approach are that:

- We save a lot computational resources, since we processes arrays of events, basically a matrix, at each $model.fit$ invocation.
- The fact that every time we select a random subset of events to train the network allows to implement the stocastic gradient discend, 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)
- A target network:
  - Synchronized each time only after we execute 10 episodes
  - Used to compute 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 countinous state values

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 opensource preexisting projects from the web. In particular from *minDQL*[1] on which I also to contribute with a *pull request*[2] to aling the source code with the last versions of TensorFlow and Gymnasium.

The stealed ideas was:

- 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 [3]

And I applied them to my project.

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

[2] My pull request: https://github.com/mswang12/minDQN/pull/9

[3] This is was also mentioned in the course, but the project helped me in having a reference implementation for the minibatch

3.7 DQL variables

learning rate	α_0	0.1
exploration / greedy	ϵ	linar 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