Deep Q-Learning

Carlos Matheus Barros da Silva, *Computer Engineering Bachelor Student of ITA*
Prof. Marcos Ricardo Omena de Albuquerque Máximo

***Abstract*—This paper evaluates a Deep Q-Learning technique applied to solve the *Mountain Car* problem.**

The technique have been implemented and passed by the test executing the *Mountain Car* problem. Both techniques have been implemented and passed by some tests. On the end It was avaliated the learning with a car that needs to follow a track.

It was observed that the implementation worked as expected. The *Q-Learning* had a fast learning by the end being able to complete the challange in less than 50 traines.

***Index Terms*—Q-Learning, Reinforcement Learning, Mountain Car**

I. Introduction

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

It differs from supervised learning in that labelled input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "qualit" of an action taken in a given state.

II. Deep Q-Learning Implementation

The technique implementation can be seen on the file *dqn_agent* and on *utils.py*. The essence of the implementation can be seen from the Code 1 and to the Code 3.

```
    def make_model(self):
        """
        Makes the action-value neural network model using
    Keras.
        :return: action-value neural network.
        :rtype: Keras' model.
        """

        model = models.Sequential()

        # 1 layer
        num_neurons = 24
        model.add(layers.Dense(
                units=num_neurons,
                activation=activations.relu,
                input_dim=self.state_size,
            ),
        )

        # 2 layer
        num_neurons = 24
        model.add(layers.Dense(
                units=num_neurons,
                activation=activations.relu,
            ),
        )

        # 3 layer
        num_neurons = self.action_size
        model.add(layers.Dense(
                units=num_neurons,
                activation=activations.linear
            ),
        )

        model.build(input_shape=(1, 2))
        model.summary()

        model.compile(
            loss=losses.mse,
            optimizer=optimizers.Adam(lr=self.learning_rate
    )
        )

        return model

    def act(self, state):
```

Code 1. Code of the method *make_model*, that will crate the Neural Network

```
    def act(self, state):
        """
        Chooses an action using an epsilon-greedy policy.

        :param state: current state.
        :type state: NumPy array with dimension (1, 2).
        :return: chosen action.
        :rtype: int.
        """

        actions = self.model.predict(state)[0]

        if np.random.binomial(1, self.epsilon) == 1:
            action = np.random.choice([a for a in range(len
    (actions))])
        else:
```

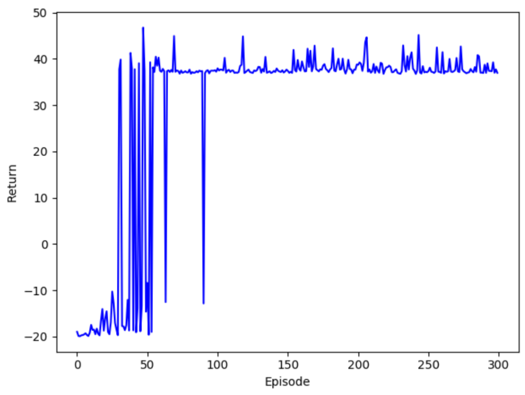

Fig. 1. Reward per training episode.

```
        action = np.random.choice([action_ for action_,
    value_ in enumerate(actions) if value_ == np.max(
    actions)])

        return action
```

Code 2. Code of the method *act* that will choose a ϵ-greedy action.

```
START_POSITION_CAR = −0.5


def reward_engineering_mountain_car(state, action, reward,
    next_state, done):
    """
    Makes reward engineering to allow faster training in
     the Mountain Car environment.

    :param state: state.
    :type state: NumPy array with dimension (1, 2).
    :param action: action.
    :type action: int.
    :param reward: original reward.
    :type reward: float.
    :param next_state: next state.
    :type next_state: NumPy array with dimension (1, 2).
    :param done: if the simulation is over after this
     experience.
    :type done: bool.
    :return: modified reward for faster training.
    :rtype: float.
    """

    r_original = reward

    position = state[0]
    velocity = state[1]

    next_position = next_state[0]
    start = START_POSITION_CAR

    r_modified = r_original + (position − start)**2 +
     velocity**2

    if next_position > 0.5:
        r_modified += 50

    return r_modified
```

Code 3. Code that implements the reward for a given state and next_state

III. Deep Q-Learning Analysis

A. Initial Training

In order to train the model, it was used the *dqn_training*.

With this, it was obteined the graph represented on the Figure 1 that is the reward per training episode.

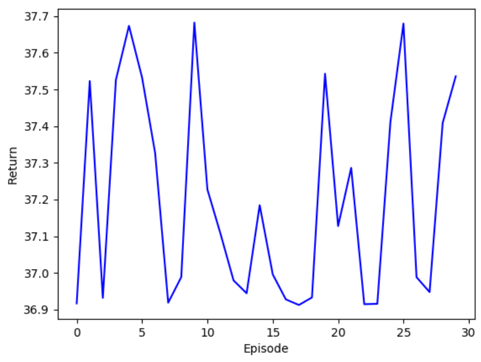

Fig. 2. Reward per evaluation episode.

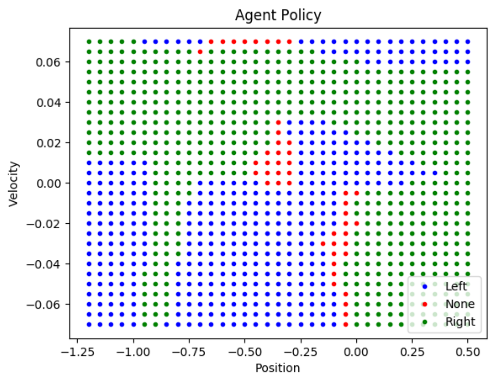

Fig. 3. Final Agent Policy.

As it is observed, the learning was fast, in way that after the 50th trainging the car was almost ways clibing the montain, and after the 100th it always climbed the mountain.

B. Evaluating the Trained Model

In order to evaluate the trained model it was used the script *evaluate_dqn.py*.

With it, was obtained the graphs represented on Figure 2 and on Figure 3.

It was observed that the convergency was good, the evaluation seen on Figure 2 show that the returned reward was stable during the 30 tests.

With the Figure 3 is clear the actions taken per region, it also has soft transitions with well determined actions regions, what makes sense to a well converged policy.

IV. Conclusion

Is was clear, therefore, that the implementation worked as expected. The *Deep Q-Learning* had a fast learning and good convergency, what garanteed a good solution to Montaion Car problem.

At the end of the learning the success rate was near 100% at the training, having 100% success rate at evaluation. Thus it is possible to conclude the algorithm solved the problem.