02/11/2022

Ben-Gurion University of the Negev

Faculty of Engineering Sciences

Department of Software and Information systems

Deep Reinforcement Learning

Assignment 1 – From Q-learning to Deep Q-learning (DQN)

General Instructions

1. The assignment should be submitted in pairs at the Moodle course site.
2. The submission will include a zip file containing:
   - A report in PDF format with answers to the questions appearing in the assignment, the requested outputs of the codes and a short instruction for running the scripts. The answers should be **short** (6-7 sentences max).
   - The scripts of your solutions.
3. The scripts should be written in Python. Use the TensorFlow library for Neural Networks. Use TensorBoard for the visualization and graphs.
4. The report can be written either in English or Hebrew.
5. Write your names and ID's in the report.
6. The final submission is due 23/11/2022

Introduction

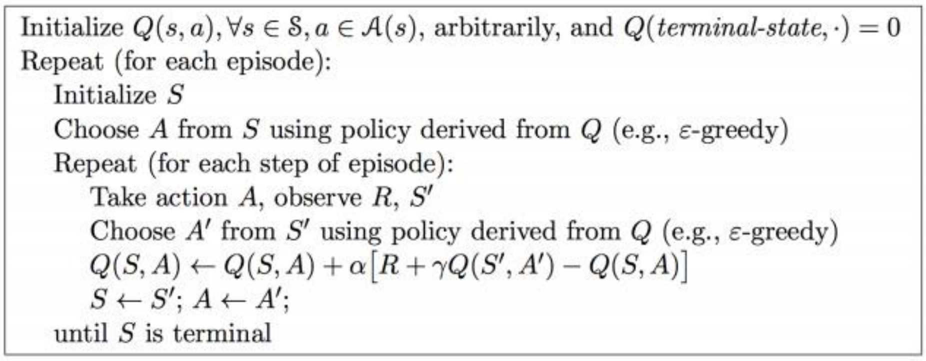
In this assignment you will get to know and experiment with **OpenAI Gym**, a testbed for reinforcement learning algorithms containing environments in different difficulty levels. You will first implement a tabular Q-learning model on a simple environment. Then, you will move on to a larger scale environment and use a NN function approximator of the Q-value, using the basic DQN algorithm. Finally, you will try to improve DQN with one of the state-of the-art algorithms from recent years or implement your own idea for improving DQN.

Section 1 – Tabular Q learning (25%)

Model-free methods in reinforcement learning are constructed for environments that their dynamics (the 'physics' of their world) are unknown or are too complicated to be handed to the agent.

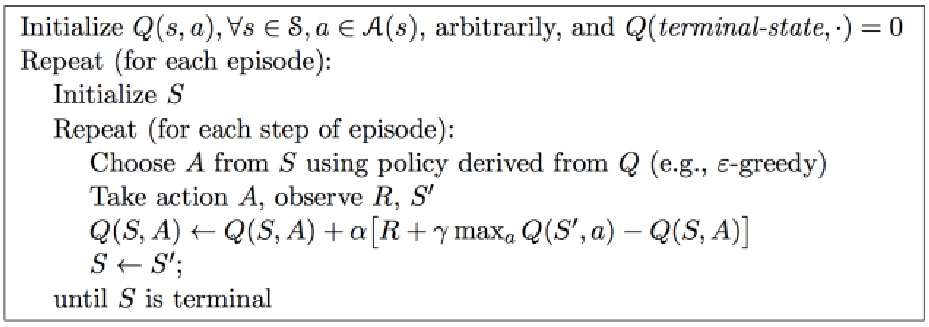
1. Why methods such as Value-Iteration **cannot** be implemented in such environments? Write down the main problem. (3%)

A few model-free methods have been suggested to overcome this issue, such as SARSA (figure 1) and Q-learning (figure 2).



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Figure 1 - SARSA algorithm



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

Figure 2 - Q-learning algorithm

Both algorithms' goal is to find the optimal Q-value for each state-action pair that the optimal policy will be derived from. As explained in class, Q-value is the expected discounted sum of rewards for an action taken in a specific state. Both algorithms are managing a lookup table for the current Q-value approximation of each state-action pair. In each step the approximation for the current state and action is being updated with respect to the **Target,** which is the immediate reward plus the maximum Q-value of the next state out of all possible actions, according to current approximation:

$$\boldsymbol{R + \gamma max_a Q(S', a)}$$

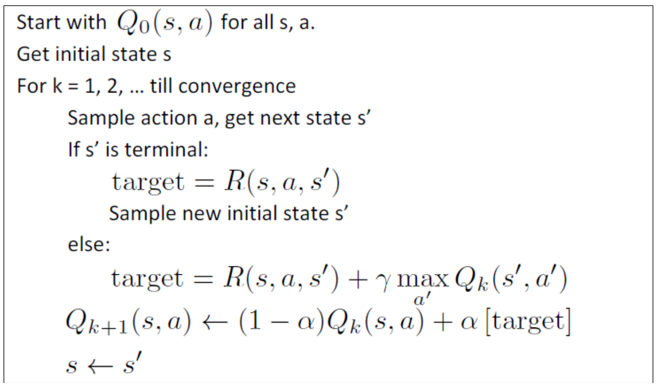
2. How do model-free methods resolve the problem you wrote in previous question? Explain shortly. (2%)
3. What is the main difference between SARSA and Q-learning algorithms? Explain shortly the meaning of this difference. (3%)

There are several ways for choosing (sampling) the next action that should be taken in each iteration of the algorithm, one of the most common is *decaying* $\varepsilon - greedy$ (i.e., the value of ε decreases over time.

4. Why is it better than acting greedily (choosing an action with $argmax_a Q(S', a)$)? (2%)

Write a short Python script (15%) implementing an agent using the basic **Q-learning** algorithm. The agent should run on the *FrozenLake-v0* environment, explained here.

- Create a lookup table containing the approximation of the Q-value for each state-action pair.
- Initialize the table with zeros.
- Choose initial values for the hyper-parameters: learning rate α, discount factor γ, decay rate for decaying epsilon-greedy probability.
- Follow the algorithm presented in figure 3. This is exactly the same algorithm from figure 2, written slightly differently.



```
Start with $Q_0(s,a)$ for all s, a.
Get initial state s
For k = 1, 2, … till convergence
    Sample action a, get next state s'
    If s' is terminal:
        $\text{target} = R(s,a,s')$
        Sample new initial state s'
    else:
        $\text{target} = R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$
    $Q_{k+1}(s,a) \leftarrow (1-\alpha) Q_k(s,a) + \alpha\,[\text{target}]$
    $s \leftarrow s'$
```

Figure 3 - Q-learning algorithm

- You should sample actions using *decaying* $\varepsilon - greedy$ or some other method of your choice.
- Optimize the hyper-parameters for optimal result
- **The Agent should train over 5000 episodes with a maximum of 100 steps in an episode.**

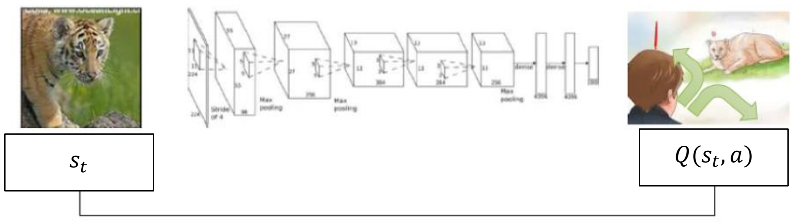
Add the following to your report:

- Hyper-parameters values that were used for the final solution.
- Q-value table after 500 steps, 2000 steps and the final table as a colormap (with the values).
- Plot of the reward per episode.
- Plot of the average number of steps to the goal over last 100 episodes (plot every 100 episodes). If agent didn't get to the goal, the number of steps of the episode will be set to 100.

Section 2 – Deep Q-learning (50%)

The main problem with Q-learning is its scalability. Most of the more interesting environments and problems have a very high number of states or have a continuous state distribution that is discretized (for example, the humanoid problem has a scale of 10^{100} states), and it is not feasible to represent them all in a lookup table.

The solution for this problem is a q-value function approximator. Instead of saving the current value of each state-action pair in a table, we will use a function to approximate the value. It will get state-action pair as an input and will output the approximate q-value (or get a state and output the q-value for each possible action). The function will have learnable parameters that will be optimized with each experience. That way, we can generalize across states.



In deep Q-learning we use a neural network as a function approximator. The loss function used for the network's gradient descent is MSE between the target and the q-value approximation:

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

In DQN, instead of updating the weights of the network in each step (as we did in the q-learning lookup table), we use two additional methods for stabilizing the network and making the model more like supervised learning:

1. **Experience replay**

   A large set of experiences that the agent went through in the past. An experience is a state-action-reward triplet. The set has a predetermined size. It is used in the algorithm to sample a minibatch of experiences in random order for the optimization process.

   1. Why do we sample in **random** order? (3%)

2. **Use an older set of weights to compute the targets**

   We separate the network into two different networks – one for computing the targets (using an older set of parameters θ^-) and one for predicting the q-value which is being updated every minibatch optimization (using parameters θ). Instead of updating the 'old' parameters of the target network each step, we do it only once every C steps by performing $\theta^- \leftarrow \theta$.

   2. How does this improve the model? (2%)

The algorithm is detailed in figure 4.

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode 1 to $M$ do
    for $t$ from 1 to $T$ do
        With probability $\epsilon$ select random action $a_t$ otherwise select
          $a_t = \max_a Q^*(\phi(s_t), a, ; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$, state $s_{t+1}$,
          and episode termination signal $d_t$
        Set $s_{t+1} = s_t$, $a_t$, $d_t$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1}, d_t)$ in $\mathcal{D}$
        Sample minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1}, d_j)$ from $\mathcal{D}$
        Set
          $y_j = \begin{cases} r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta') & \text{for non-terminal transition} \\ r_j & \text{for terminal transition} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
        Every C steps, update target network $\theta' \leftarrow \theta$
    end
end
```

Figure 4 - DQN algorithm with experience replay

Write a Python script (45%) implementing an agent using the basic **DQN** algorithm. The agent should run on the *CartPole-v1* environment, explained here. This environment can be solved with simpler methods than DQN but is used for this exercise due to runtime considerations.

In your code, include the following functions and data structures:

- **Neural Network**: the network should take a state as an input (or a minibatch of states) and output the predicted q-value of each action for that state. The network will be trained by minimizing the loss function with an optimization method of your choice (SGD, RMSProp, …). **Try 2 different structures for the network, the first with 3 hidden layers and the second with 5**.

  Notice: the network will be used both as the q-value network and as the target network, but with different weights.
- **experience_replay**: a deque in a fixed size to store past experiences.
- **sample_batch**: sample a minibatch randomly from the experience_replay.
- **sample_action**: choose an action with *decaying* $\varepsilon - greedy$ method or another method of your choice.
- **train_agent**: train the agent with the algorithm detailed in figure 4.
- **test_agent**: test the agent on a new episode with the trained model. You can render the environment if you want to watch your agent play.

Add the following to your report:

- Hyper-parameters values that were used in the final solution (batch size, learning rate, decay rate of epsilon, etc.). Write which HP affected the most on the performance and how. Detail the structure of your NN's.
- Write the number of episodes until the agent obtains an average reward of at least 475.0 over 100 consecutive episodes.
- Plot of the loss in each training step
- Plot of the total reward of each episode in training.

Section 3 – Improved DQN (25%)

Since DQN algorithm was first published, a few improvements were suggested in different papers such as prioritized experience replay and DDQN.

Write a Python script implementing an agent using the **DQN** algorithm with one of the state-of-the-art improvements or with an improvement of your own. The agent should run on the same environment of section 2.

Add the following to your report:

- Short explanation on the suggested improvement and its purpose.
- Hyper-parameters values that were used in the final solution. Detail the NN structure.
- Write the number of episodes until the agent obtains an average reward of at least 475.0 over 100 consecutive episodes.
- Plot of the loss in each training step.
- Plot of the total reward of each episode in training.
- Comparison with the results of section 2.