Navigation Problem - Banana Collector

1 Reinforcement Learning Framework

The Banana Collector agent interacts with the environment according the current state, then it receives a new state and a reward along with it, this behaviour continues in iterative way forming a a sequence like the one below:

$$S_0, A_0, R_1, S_1, A_1, R_2, ..., S_T, R_T$$

For the rest of this report:

S or s: stands for state
A or a: stands for action
R or r: stands for action
Q or q: stands for action-value function or it refers to neural network approximator of this function
G : stands for value of future rewards

This sequence is called an episode and the goal of the agent is to maximize the overall reward at the end of the episode. The future rewards at certain time t can be compute with the equation :

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, where \gamma \in [0, 1].$$

What this equation does is to take the present value of all future rewards using a discount factor of γ.

Now let's assume the agent has a criteria to determine his actions, this criteria is formally called a policy π. How could we determine what actions are the best to take at each certain state? The answer is taking the expected future rewards given that the agent takes certain action and follows the policy afterwards. The next equation summarize this idea and is referred as **action- value function** or simply as **Q function**.

$$Q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$$

An optimal action value function is such that $Q^* = max_\pi Q_\pi(s, a)$ for all $s \in S$ and $a \in A(s)$.Likewise an optimal policy is such that $\pi_* \geq \pi$ for all π and it is guaranteed to exist but no to be unique, the expected return of the future rewards it is the same for any optimal policy.

To obtain the optimal policy we part from an optimal action-value function or an estimate of it which is obtained by the interaction of the agent with the environment.

$$Interaction \rightarrow q_* \rightarrow \pi_*$$

. To go from the q_* to π_* the next criterion is used:

$$\pi_*(s) = arg\ max_{a \in A(s)}\ q_*(s, a)$$

1.1 But how can Q^* be found?

Before go any further is paramount to define what an **epsilon-greedy policy** is. So, an $\epsilon - greedy$ policy π selects randomly among all the actions in $A(S)$ with probability $1 - \epsilon$, otherwise selects the action that yields the maximum q-value. So:

$$\pi(s \mid a) = \begin{cases} 1 - \epsilon + \epsilon / |A(s)| & if\ a\ maximizes\ Q(s,a) \\ \epsilon / |A(s)| & otherwise \end{cases}$$

for each $s \in S$ and $a \in A(S)$.

Broadly speaking, given a certain policy π a $Qfunction$ is computed, then π is improved by changing it to be $\epsilon - greedy$ with respect to the Q,then this new policy π' is used to interact with the environment and the experience procured is used to compute a new action-value function Q', following these steps iteratively we'll get the optimal policy $\pi*$ eventually.

1.2 Q-Learning

The Q-Learning algorithm also known as SARSA-MAX is a Time-Difference method used to update and improve the action-value function Q, and is the core of DQN algorithm used to solve the Banana collector environment.These kind of methods take into account not only the current state-action pair to update q-values but also the next pair, that's why the name SARSA $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$. Q-learning only needs to know the next state S_{t+1} and not the action A_{t+1} because uses the $a \in A(S_{t+1})$ that yields the maximum for $Q(S_{t+1}, A_{t+1})$, because of this SARSA MAX is considered an off-policy algorithm.

An important concept is needed to jump into the SARSA algorithm is the definition GLIE, which stands for **Greedy on the Limit with Infinite Expectation**. SO a policy is considered to be GLIE if and only if:

- All state-action pairs are visited infinitely
- As $t \to \infty$, the learning policy converges to a greedy policy with respect to the action-value function.

Now let's introduce SARSA MAX algorithm:

Algorithm 1 SARSA MAX[1]

Input: policy π, positive integer *n-episodes*, small positive fraction α, GLIE ϵ_i.
Output: action-value function Q, if *n-episodes* large enough $Q \approx Q*$
Initialize $Q(s,a) = 0$ for all $s \in S$ and $a \in A(s)$
for $i \leftarrow 1$ **to** *n-episodes* **do**
    $\epsilon \leftarrow \epsilon_i$
    Observe S_0
    repeat
        Choose action A_t using policy derived from Q
        Take action A_t and observe R_{t+1}, S_{t+1}
        $Q(S_t, S_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t))$
        $t \leftarrow t + 1$
    until S_t is terminal
return Q

1.3 DQN algorithm

The Deep Q Network algorithm or DQN training algorithm is built upon the SARSA algorithm but uses a neural network as non linear approximator of the action-value function Q.To avoid the divergence of this neural network it uses a couple of techniques, **experience replay** and **fixed Q-targets**.

1.3.1 Experience Replay

The DQN algorithm uses tuple experiences as its input and these are defined as $e_t = (s_t, a_r, r_{t+1}, s_{t+1})$, the naive Q-learning algorithm also learns from each of these experience tuples but in sequential order runs the risk of getting swayed by the effects of this correlation.Instead DQN keeps track of a past experiences using **replay buffer** and **experience replay** to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

The **replay buffer**, D contains a collection of experience tuples (S, A, R, S'). The tuples are gradually added to the buffer as we are interacting with the environment.Usually the size if the buffer is fixed keeping the last N experiences.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, **experience replay** allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

1.3.2 Fixed Q-Targets

The other modification to the naive Q-learning is to model the TD targets, $R_{t+1}+\gamma \max_{a\in A} Q(S_{t+1}, a; \theta^-)$ with a different set of weights $\theta^- \neq \theta$.The idea behind this is that naive Q-learning in, we update a guess with a guess, and this can potentially lead to harmful correlations. So each certain C number of steps the network representing Q is cloned into a network Q', and this last network is used to compute the TD targets.

1.3.3 Deep Q-Learning with experience replay

Algorithm 2 deep Q-learning with experience replay.[2]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
for $episode = 1\, M$ **do**
  for $t = 1\, T$ **do**
    With probability ϵ select a random action a_t
    Execute action a_t in emulator and observe reward r_t and s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions (s_j, a_j, r_t, s_{j+1}) from D
    $$Set\ y_j = \begin{cases} r_j & if\ episode\ terminates\ at\ state\ j+1 \\ r_j + \gamma max_{a'} Q'(s_{j+1}, a'; \theta^-) & otherwise \end{cases}$$
    Perform a gradient descent step on $(y_i - Q'(s_j, a; \theta))$ with respect to the network parameters θ
    Every C steps reset $Q' = Q$

2 Solution to the Banana Collector Environment

As the README of this repository describes, the environment's state is represented by 37 dimensional vector, the learning algorithm is directly derived from Deep Q-learning algorithm described in 1.3.3 which is the same as presented in the original paper[2] but without the preprocessing steps for the states because we are not using image as input but the raw observation vector provided by the environment.

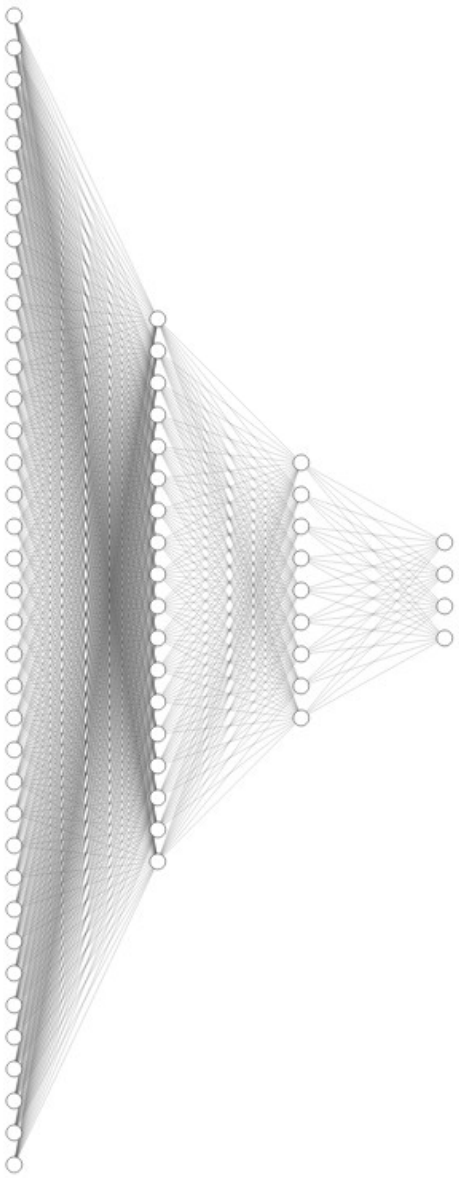
2.1 DQN Implementation

To train the agent, it must perform an action following $\epsilon - greed$ policy, at the start this action is completely picked at random, then the action is sent to the environment and returns a next state and reward to the agent along the information to know if the task has been done already.After a step is taken the experience $(state, action, reward, next_state, done)$ obtained is stored in the replay buffer, then every fixed number of steps (in our case 4) the learning phase is executed and a new batch of experiences from the buffer are selected. This batch of experiences are used to feed the Q-network of expected q-values and the Q'-network of targets, then the Mean Square Error between the outputs of both networks is computed. Then back propagation is used on the Means Square Error of the last step and the weights of the Q-network are updated using an improved version of the Stochastic Gradient Descent called ADAM that leverages a technique known as momentum to avoid local minima.Finally the Q' targets network is update as convex linear combination of its own weights and the Q-network's weights. This behaviour is repeated until an average scored is accomplished over 100 consecutive episodes.

2.1.1 Hyper-parameters

Hyper-parameters		
Parameter	Value	Description
BUFFER_SIZE	100,000	replay buffer size
BATCH_SIZE	64	minibatch size
GAMMA	0.99	discount factor used in TD targets
TAU	.001	for soft update of target parameters
LR	.0005	learning rate used by the optimizer algorithm of the neural network
UPDATE_EVERY	4	how often to update the network
eps_start	1.0	epsilon applied in first episode
eps_end	0.01	inferior limit for epsilon values
eps_decay	0.995	decay rate applied at the of each episodes to value of current $\epsilon - greedy$ policy
p_drop	0	probability for nodes of hidden layers to be zero out in a training step
HIDDEN_LAYER_1	18	number of nodes for the first fully connected layer
HIDDEN_LAYER_2	9	number of nodes for the second fully connected layer

2.1.2 Network's architecture



Layers	Nodes	Activation function
Input layer	37	Rectifier Linear Unit
First hidden layer	18	Rectifier Linear Unit
Second hidden layer	9	NA
Output layer	4	NA

Additionally to the described architecture above a drop out methodology was implemented for each layer with the exception of the output layer, but at the the moment the networks were trained we used a drop out probability of zero and given that the result were good enough to resolve the environment, there was no need to tune this hyper-parameter.

2.2 Double DQN

Another approach for the learning algorithm to solve the banana's environment is presented, this solution the same hyper-paramaters and network's architecture proposed for the DQN solution. The popular Q-learning algorithm is known to overestimate action values under certain conditions[3]. Double DQN proposes a reformulation of TD target basically using another set of parameters that in the long run prevents the algorithm to propagate high incidental rewards that may been obtained by chance. So basically it's like having 2 different q-value function approximators that try to agree on the best action.

$$reward + \gamma * Q'(s', argmax_a Q(s, a; \theta), \theta^-)$$

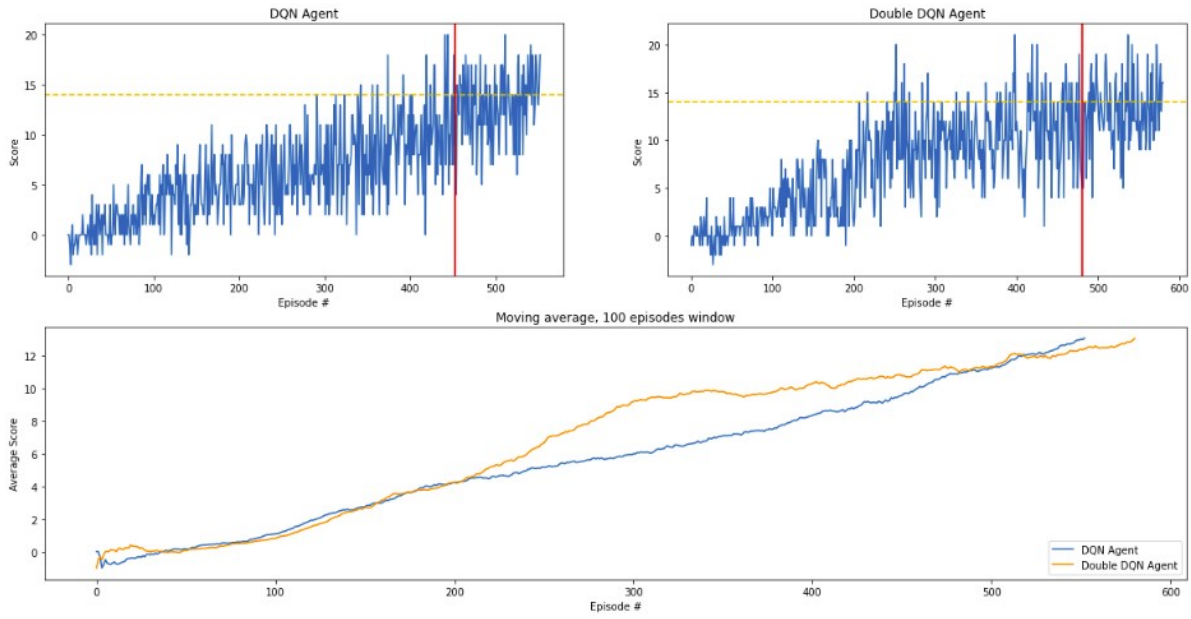
We use the local network to get the best actions and the we evaluate on the target network, getting in this way the TD targets.

3 Results

Following can be observed how the agents perform through the training phase, their average performance over a window of 100 hundreds episode and the moment they solved the environment.

DQN Agent		Double DQN Agent	
Episode	Average Score	Episode	Average Score
100	1.06	100	0.80
200	4.21	200	4.20
300	5.92	300	9.14
400	8.28	400	10.21
500	11.23	500	11.30
553	13.05	581	13.03

Environment solved in 453 episodes! by the **DQN Agent with** Average Score: 13.05
Environment solved in 481 episodes! by the **Double DQN Agent** with Average Score: 13.03



4 Future Work

The following steps for this exercises would be:

Implement the variations: Dueling DQN and Prioritized Experience Replay. Run Google's Rainbow algorithm with this environment. Solved the the environment using raw image inputs, implement a convolutional network to approximate the action-value function.

References

[1] Richard S. Sutton and Andrew G. Barto(2018). Reinforcement learning. MIT Press, page 131.

[2] Mnih, V., Kavukcuoglu, K., Silver, et. al. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529–533.

[3] Hado van Hasselt, Arthur Guez and David Silver (2015).Deep Reinforcement Learning with Double Q-learning. Journal CoRR, Volume abs/1509.06461.