EX4 - Advanced Practical Course In Machine Learning
Reinforcement Learning

Daniel Afrimi
203865837

December 25, 2020

In order to learn the Q function from which it will be possible to obtain the optimal policy. I built a model that accepts batches of states and returns as a vector output the length of the number of actions (in our case 3 actions), so that in each coordinate i in the output appears the value we will get for the long run if we select action i. The model include 3 conv2D layers and 2 fully connected layers (with batchNorm for regularization) and Relu as an activation function. for the vanilla model there is softMax for getting a distribution vector

When I tried to train the agent at first, he did not learn anything too much, he would revolve around himself (the snake), or he would perform the same action for many steps. I finally played on Epsilon so that the snake would recognize situations he had not visited and see a "new world" and so I also increased the gamma (discount future rewards) and the model was able to practice and get a positive reward.

In general, most algorithms learn faster when they don't have to look too far into the future. So, it sometimes helps the performance to set gamma relatively low. Another thing is that for many problems a gamma of 0.9 or 0.95 is fine.

1 Q-learning

I trained the model for 10000 steps, with RMSprop Optimizer, MSE loss function and with various hyper parameters (the comparison can be seen below).

In **Figures 1** we can see that after training the model for 10000 steps, with lr 0.005 and gamma equal to 0.9 we obtained a positive reward.

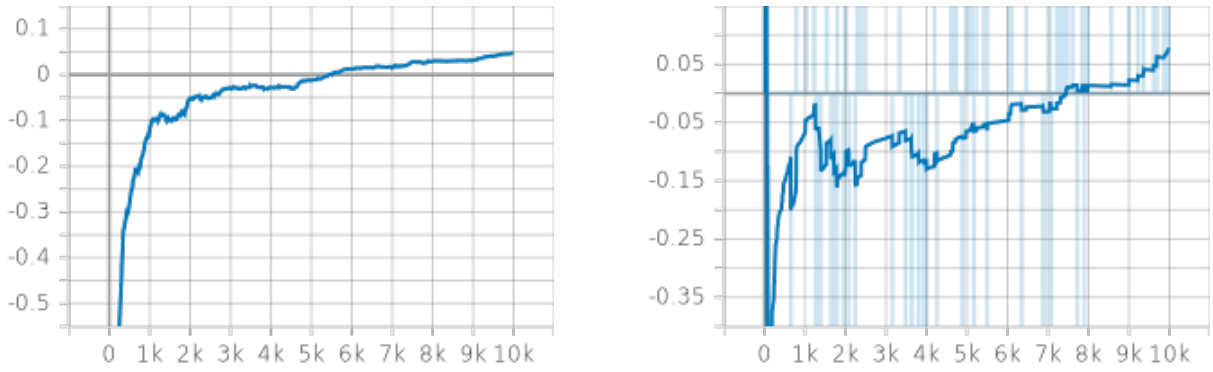


Figure 1: DQN. Average Reward (0.046, left), Reward per step (right)

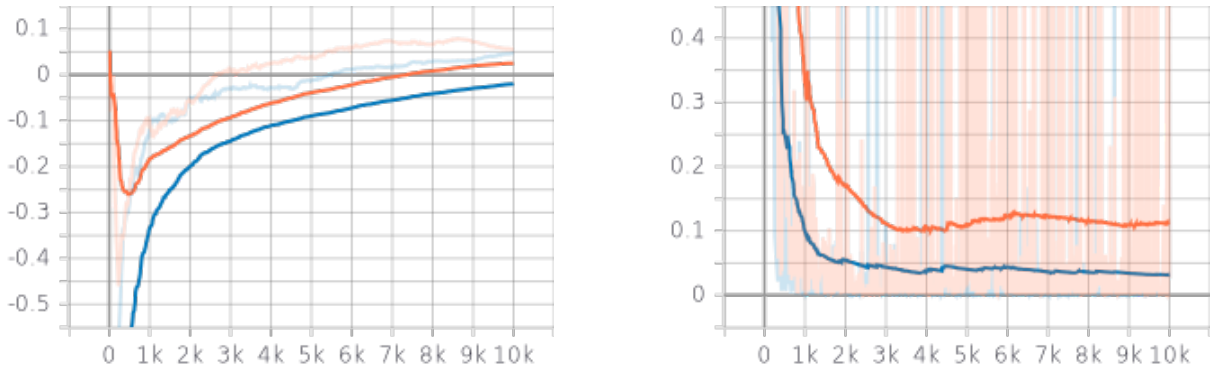


Figure 2: Different Window Size DQN. 9X9 window (orange), 3X3 centered (blue)

In addition I evaluated the model by a different window size (around the snake).In **Figures 2** we can see that the bigger window (9X9) gets better result on the reward.

1.1 Pseudo Code

Algorithm 1 Q-learning

initialize Q(s,a) randomly

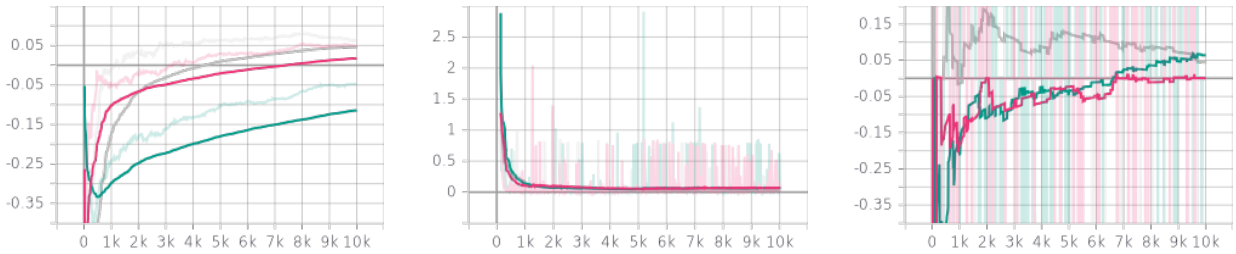
for $episode = 1, 2, \ldots$ **do**
    initialize S
    for $step = 1, 2, \ldots, episode$ **do**
        Choose action from S using policy derived from Q ($argMax_a = Q(s, a)$)
        take action a and observe the reward, S'
        Update (Optimization Step) $Q(s, a) = Q(s, a) + \alpha(reward + \gamma \cdot max'_a(S', a') - Q(s, a))$
        S $\leftarrow S' -$

1.2 Comparing Hyper-Parameters

In this section, there are 3 different metrics:

- Average reward - cumulative reward on all samples until that step.
- Training loss at each optimization step.
- Reward - The reward at each step.

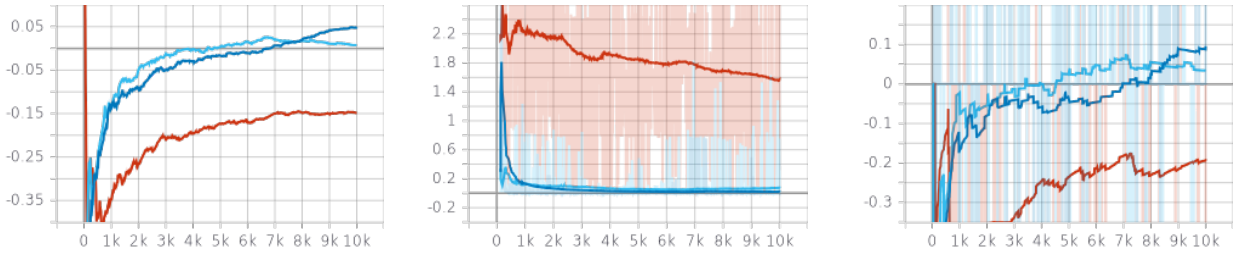
Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit. Improving the accuracy of the estimated action-values, enables an agent to make more informed decisions in the future. Exploitation on the other hand, chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates.
Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. In **Figures 3** it can be seen that for our agent the smaller the epsilon the more he relies on actions that have yielded higher reward in the past and he gets a higher average reward



(a) Average Reward (b) Training Loss (c) Reward

Figure 3: Different Epsilon factor, epsilon=0.3 (gray), epsilon=0.5 (pink), epsilon=0.7 (green)

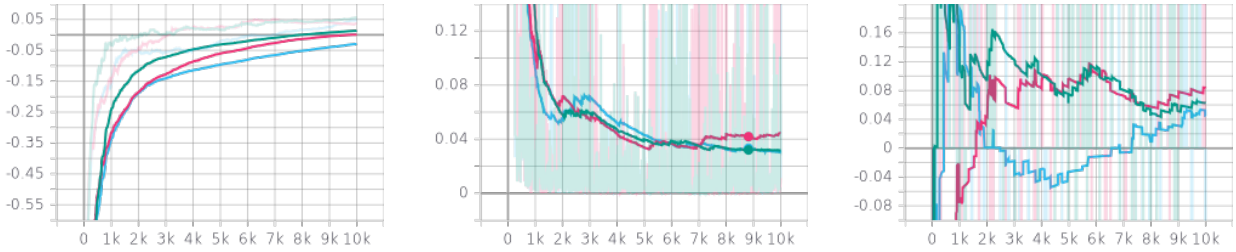
In **Figures 4** it can be seen that I tried to see how the effect of learning rates on the reward and the loss. The graphs suggest that the agent gets the best reward when the learning rate is 0.005.



(a) Average Reward (b) Training Loss (c) Reward

Figure 4: Different learninig rates, lr=0.1 (orange), lr=0.01 (blue), lr=0.0005 (red)

In **Figures 5** we can see that when gamma increased the reward of the agent was smaller.



(a) Average Reward (b) Training Loss (c) Reward

Figure 5: Different Gamma factor, gamma=0.6 (pink), gamma=0.9 (blue), gamma=0.3 (green)

2 Policy Gradient

The concept of Monte-Carlo method is where agent learn about the states and reward when it interacts with the environment. In this method agent generate experienced samples and then based on average return, value is calculated for a state or state-action.

Policy Gradients seeks to directly optimizes in the policy space. In Policy Gradients, we usually use a neural network to directly model the action probabilities. Each time the agent interacts with the environment. we tweak the parameters θ of the neural network so that "good" actions will be sampled more likely in the future. We repeat this process until the policy network converge to the optimal policy π^*. The objective of Policy Gradients is to maximize the total future expected rewards

I trained the model for 10000 steps, with RSMProp Optimizer, MSE loss function and with various hyper parameters (the comparison can be seen below).

In **Figures 6** we can the a positive reward for the model that trained according to gradient policy.

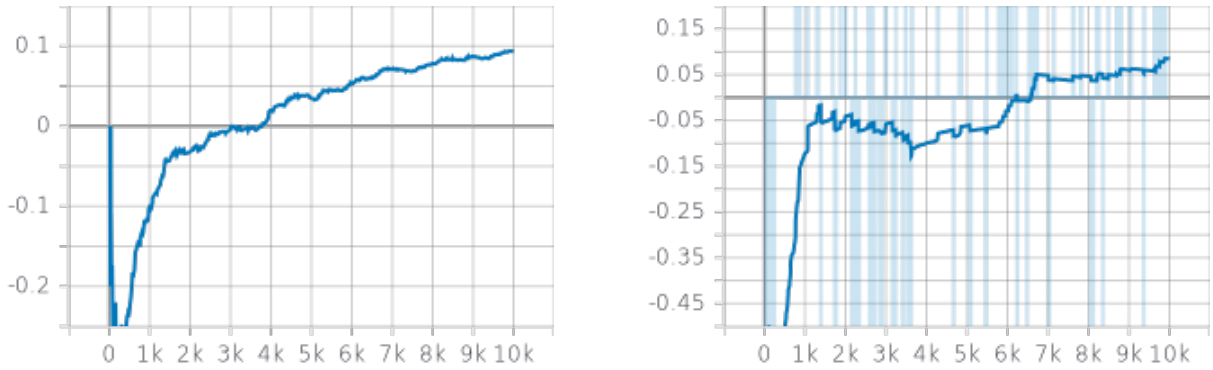


Figure 6: DQN With Policy Gradient. Average Reward (0.1, left), Reward per step (right)

2.1 Pseudo Code

In **Figures 7** we can see the Pseudo Code of the Monte-Carlo method (taken from the tirgul).

Algorithm 1: Reinforce / Monte Carlo Policy Gradient, single iteration

initialize buffer;
for *episode length* **do**
  $action \sim \pi(state)$ state, reward = env.step(action);
  buffer $\leftarrow$ (prev_state, action, reward, state);
end
for t **do**
  $V_t = \sum_{i=t}^{T} reward_i \cdot \gamma^{i-t}$
end
$objective_t = log(\pi(a_t|s_t)) \cdot V_t) + \alpha \cdot entropy(\pi(s_t))$
$objective = mean[objective_t]$
maximize *objective*

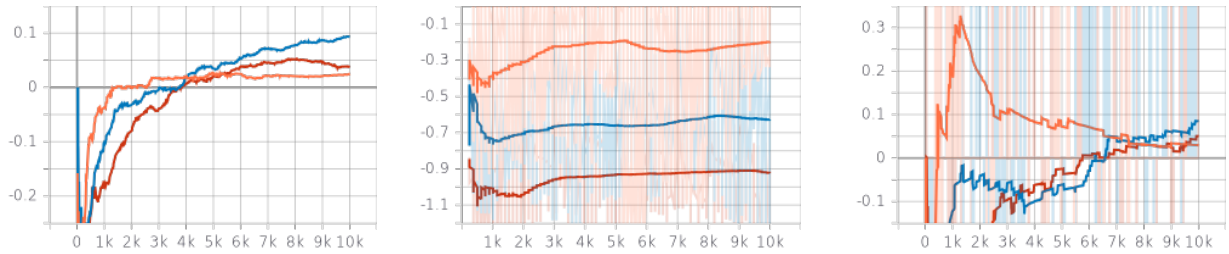
Figure 7: Pseudo Code

2.2 Comparing Hyper-parameters

In this section, there are 3 different metrics:

- Average reward - cumulative reward on all samples until that step.
- Training loss at each optimization step.
- Reward - The reward at each step.

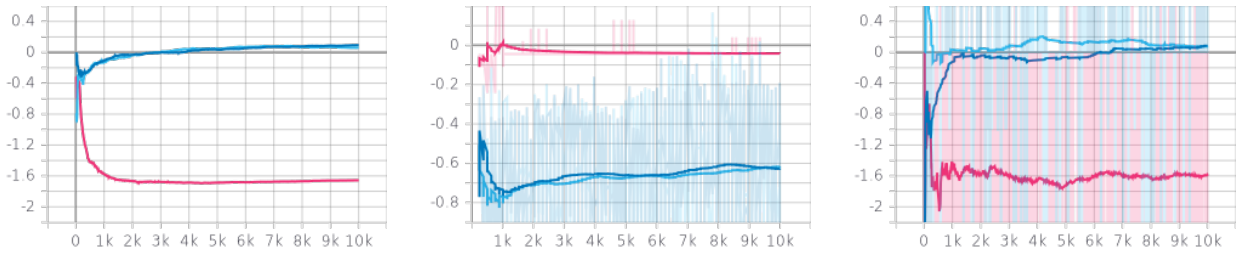
In **Figures 8** we can see comparison between different alpha that while using alpha equal to 0.4, we obtained the best result.



(a) Average Reward (b) Training Loss (c) Reward

Figure 8: Different alpha factor, alpha=0.4 (blue), alpha=0.3 (orange), alpha=0.7 (red)

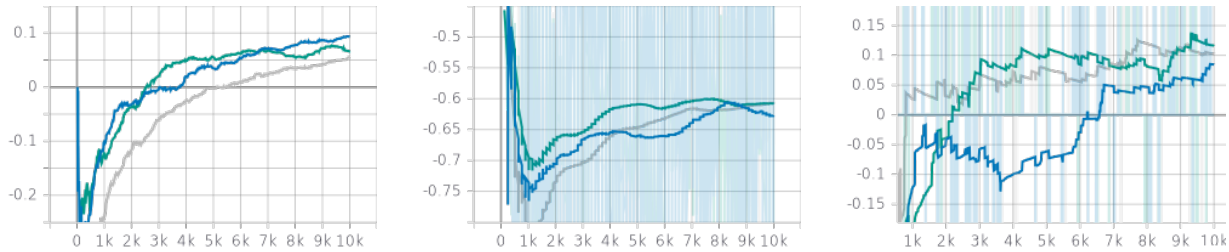
In **Figures 9** we can see that the best lr is 0.009.



(a) Average Reward (b) Training Loss (c) Reward

Figure 9: Different learning rates , lr=0.0009 (blue), lr=0.01 (pink), lr=0.001 (light blue)

In **Figures 10** we can see that while we defined the Episode size to be smaller the model obtained bigger reward.



(a) Average Reward (b) Training Loss (c) Reward

Figure 10: Different Episode Size , T=50 (blue), T=100 (green), T=150 (gray)

3 Comparison

Policy Gradients is generally believed to be able to apply to a wider range of problems. For instance, on occasions when the Q function (i.e. reward function) is too complex to be learned, DQN will fail. On the other hand, Policy Gradients is still capable of learning a good policy since it directly operates in the policy space. Furthermore, Policy Gradients usually show faster convergence rate than DQN, but has a tendency to converge to a local optimal. Since Policy Gradients model probabilities of actions, it is capable of learning stochastic policies, while DQN can't.

The biggest drawbacks of Policy Gradients is the high variance in estimating the gradient. each time we perform a gradient update, we are using an estimation of gradient generated by a series of data points accumulated through a single episode of game play (T). Hence the estimation can be very noisy, and bad

gradient estimate could adversely impact the stability of the learning algorithm.

From the tests I ran for both policies we received a higher reward for Policy Gradient

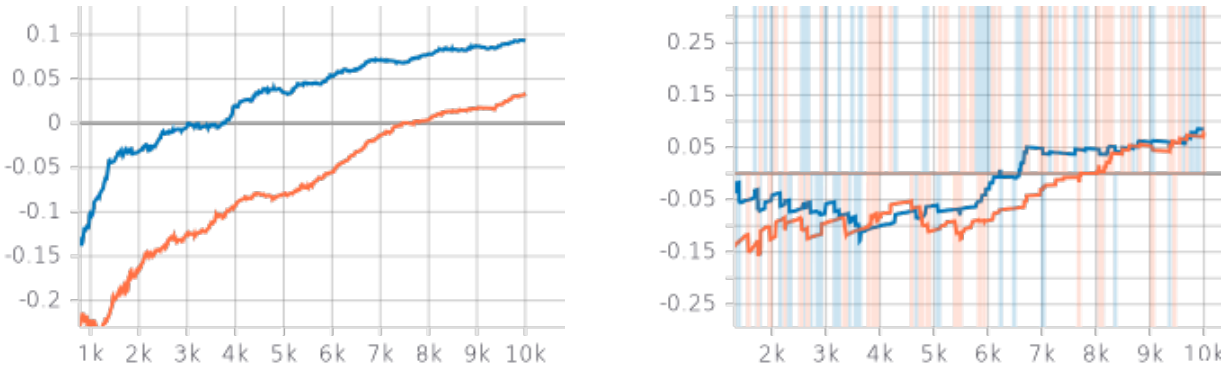


Figure 11: DQN With Policy Gradient. Average Reward (0.1, left), Reward per step (right)