Faculty of Applied Sciences

Optimal Decision Making for complex problems

INFO8003-1

Assignment 3 : Deep Reinforcement Learning with Images for the Car on the Hill Problem

---

Authors :
Delcour Florian - s181063
Makedonsky Aliocha - s171197

Instructors :
Ernst Damien
Teaching assistants :
Aittahar Samy
Miftari Bardhyl

April 2022

1 Domain

The new challenge we have to face for this *car on the hill problem* is that our agent has no longer access to the state of the game, only to the image representing the state. In order to take the right decision regarding to the actual state of the game, the agent won't be able anymore to train with the real state of the game (i.e. its position `p` and its speed `s`) but only with the image representing the state.
On the Figure 1 is one of the multiple possible images :

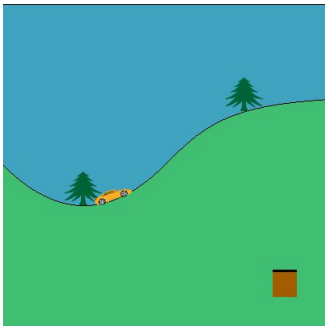

Figure 1: Image representing the state of the game

There are two markers of the state of the game : the position of the car, which corresponds to the position of the agent, and the gauge on the bottom right of the image, indicating the speed of the agent.
So we will generate some trajectory according to the true state of the game (the p and the s) and we will transform it into a 100x100 RGB image thanks to the function `save_caronthehill_image` we were given in the `car_on_the_hill_images.py` file, and we will put the image in the form of a tensor, containing the RGB value of each pixel of the image, and that will be used in the following for training the agent.

We still take $\gamma = 0.95$ as the discount factor.

2 Deep Q-Learning

For the agent to learn to take the right action whatever its situation, we need to create a neural network that will be trained on the images representing trajectories. This neural network is a convolutional neural network (CNN) which follows a general scheme that we saw in the course INFO8010-1 Deep Learning. This scheme consist in `input -> ((convolutional layer with RELU activation function)*N + optional pooling layer)*M + (fully connected layer with RELU activation function)*K + fully connected layer -> output`. The part in red represents a multi-layers perceptron (MLP).
This CNN will take one or multiple images as input, those images all transformed as tensors, and it will output 2 values for each input image, those 2 values being the Q-function for each possible action (either u = -4 or u = 4), and the biggest value is the action that

the agent should take in its current state.

To train our model we will use an algorithm similar as the one in Figure 2 (described in this paper) :

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a 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 and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: Algorithm used

With 100 episodes of 25 epochs each.
We need a replay memory that will store up to N transitions, transitions composed of `(state, action, reward, next_state)` with the state represented as an image. We can take random samples of a certain batch size (we put 64 here) from this replay memory, and if we reach the capacity of the memory and we want to add new transitions, we will replace the oldest.
At the begining of each epoch, the sequence is initialized as in the assignment 2, so with a `p` uniformly drawn from the interval [-0.1 ; 0.1] and with a `s` equal to zero.
We then generate a trajectory with an ϵ-greedy policy, with an ϵ that decreases from 0.9 to 0.05 linearly with the episode we are currently in (this way, at the beginning, when the parameters of our CNN are bad, the agent will prioritize exploration, and when more training is done, it will prioritize exploitation). The generation stops when we reach a terminal state. At this moment we add this final transition to the replay memory and we skip to the next epoch.
Each transition is added to the replay memory, and when the length of the replay memory is bigger than or equal to the batch size (so basically after the 64^{th} transition (in total, not of each episode/epoch)), we can start training our model with samples taken from the replay memory.
Our CNN will give us the value of $Q(state', u')$ for both possible actions, with `state'` the next state, and `u'` the next action. We then take $\max_{u'} Q(state', u')$ (so the maximum of the outputs) and from it we can compute y_j as described in the algorithm, which will be equal to r_j if we are in a terminal state (which is the case when $r_j \neq 0$), and to $r_j + \gamma \max_{u'} Q(state', u') = \gamma \max_{u'} Q(state', u')$ when we are in a non-terminal state

$(r_j = 0)$.
Then our CNN will predict the Q-function and will be trained by performing gradient descent with the MSE as loss function between y_j and $Q(state_j, u)$, both computed by the CNN.

When all the epochs and episodes are done, we generate all the values of `p` between -1 and 1 with a step of 0.01, and all the values of `s` between -3 and 3 with a step of 0.01 as well. Then with our trained CNN, we can predict the Q-function $\hat{Q}$ for all the possible combinations of those values, for each of the two possible actions. Here is the heatmap of $\hat{Q}$ for each of the possible actions :

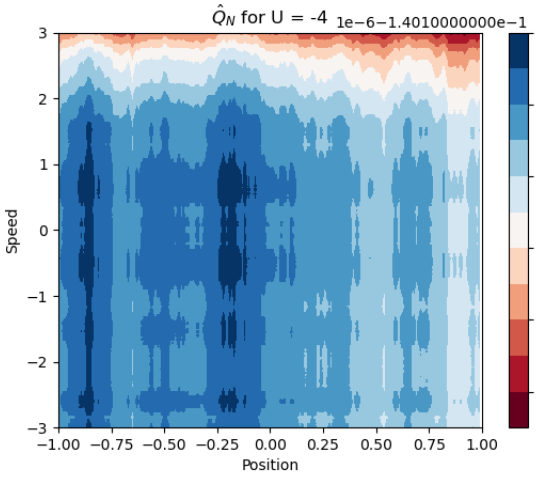

Figure 3: $\hat{Q}$ for $u = -4$

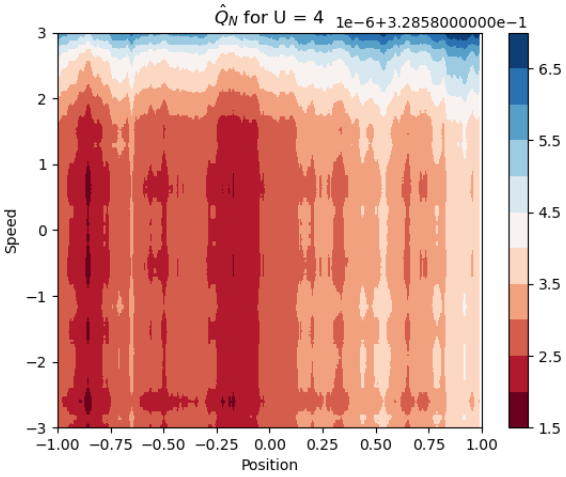

Figure 4: $\hat{Q}$ for $u = 4$

Don't be fooled by the colors, take a look at the scale of those colors, we see that $\hat{Q}$ is close to 0 everywhere.

From those Q-functions we can induce the optimal policy μ :
Seeing the Q-functions the estimated policy couldn't be good either, and indeed it indicates that the agent should always accelerate, which is not the optimal solution to reach the top of the hill (as we could see in the assignment 2).
The expected return $J_{\hat{\mu}}$ is 0 for any pair position-speed, with 150 time steps, not surprising either with this bad policy.

3 Deep Q-Learning vs FQI-Trees

3.1 FQI with Extremely Randomized trees with image

We take the code of the assignment 2 section 4 and we update it to take the constraint that the agent can't access the real state of the game, only the image representing it. The code is in the `section2_FQI.py` file.

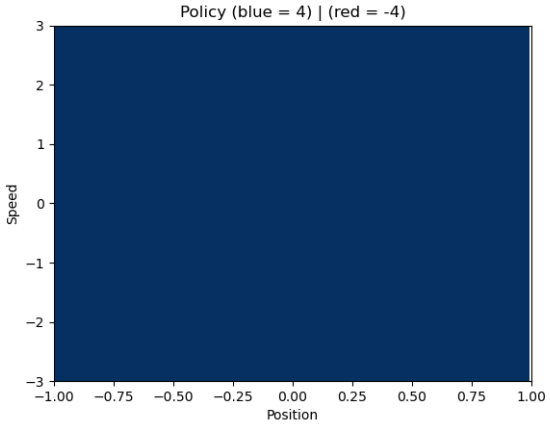

Figure 5: Estimated policy $\hat{\mu}$

Just as in the section 2, the input of our FQI with Extremely Randomized Trees as model is an image, that was transformed into an array containing the RGB values of each pixel.
Since we its an offline algorithm, we directly generate all our trajectories, before feeding those entirely to the model for it to train. We generate 100 trajectories, each starting with a `p` drawn uniformly between -1 and 1, and with a `s` equal to 0. Each trajectory contains up to 150 transitions (it may be less because we stop when the agent reaches a terminal state). A transition is composed of the state of the game, that is transformed into a 100x100 RGB image `x`, the action taken `u`, the reward obtained `r`, and the next state reached, also transformed into a 100x100 RGB image `x'`. It gives us transitions in the form of `(x, u, r, x')`

Once we have generated all those trajectories we need to train our model with those. To do so, we build an array to store all the inputs, that are an array containing the image `x` (the RGB value of all its pixel) and the action taken `u`, and we feed this whole array of inputs to the model once.
We then build an array that will contain all the outputs. The output is $r+\gamma \max_{u'} Q(x', u')$ where $Q(x', u')$ is the output of the model when we feed `x'` and `u'` to it.
Once we have done that for all the transitions in our trajectories, the model trains with our array of inputs and our array of outputs.
We loop like N times, where N is given by $\dfrac{2\gamma * N * Br}{(1-\gamma)^2} = 60$ where $Br = 1$ is the bound on the reward.

With the final model we obtain, we can display the Q function for each possible action like in the previous section, and here are the results we get :

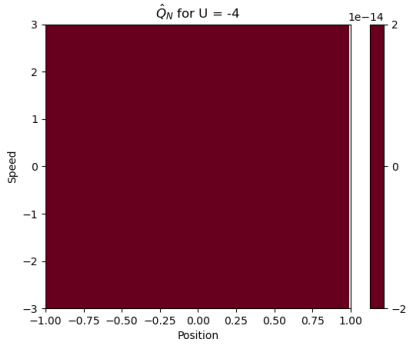

Figure 6: $\hat{Q}$ for $u = -4$

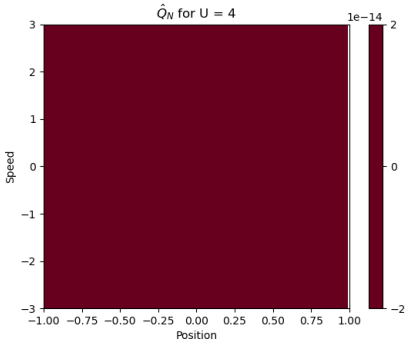

Figure 7: $\hat{Q}$ for $u = 4$

And for the estimated optimal policy :

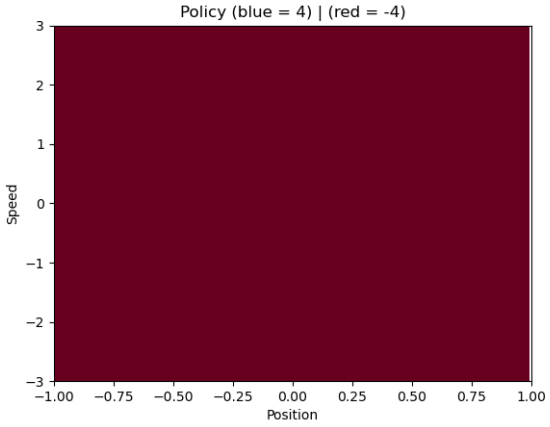

Figure 8: Estimated policy $\hat{\mu}$

With an expected return of 0.

As you can see we don't have much to compare since the results we get with the current algorithms are just really bad, and thus are worst than the results we got before.

3.2 Extremely Randomized Trees and Q-Learning with real state

This time the agent has access to the real state of the game, not only to the images representing the state. It corresponds to the sections 4 and 5 of the assignment 2.
As a reminder we got the following results for the FQI with Extremely Randomized Trees :

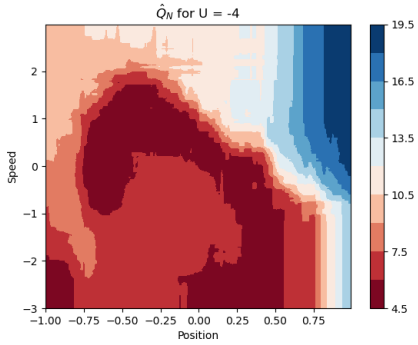

Figure 9: $\hat{Q}$ for $u = -4$ for FQI with Extremely Randomized Trees

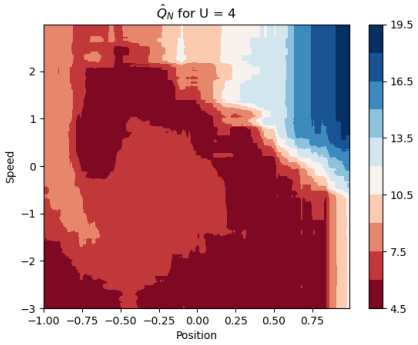

Figure 10: $\hat{Q}$ for $u = 4$ for FQI with Extremely Randomized Trees

And for the estimated optimal policy :

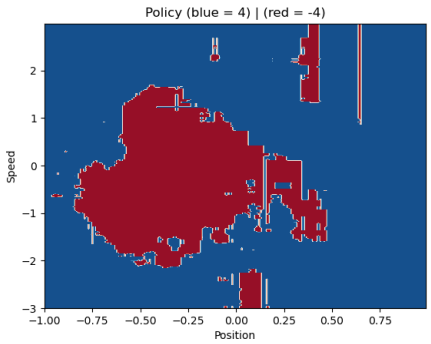

Figure 11: Estimated policy $\hat{\mu}$ for FQI with Extremely Randomized Trees

With 0.395 as expected return.

Whereas for the Q-Learning with neural network :

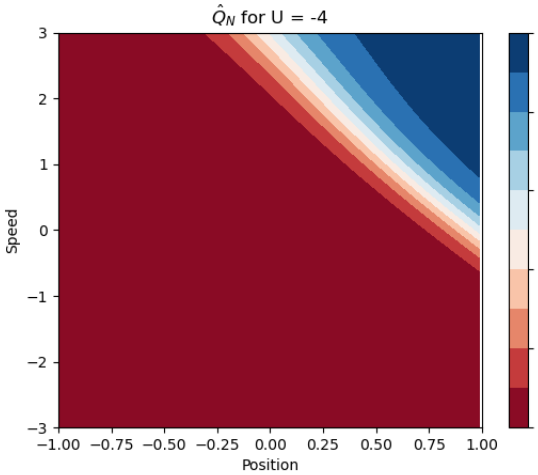

Figure 12: $\hat{Q}$ for $u = -4$ for Q-Learning neural network

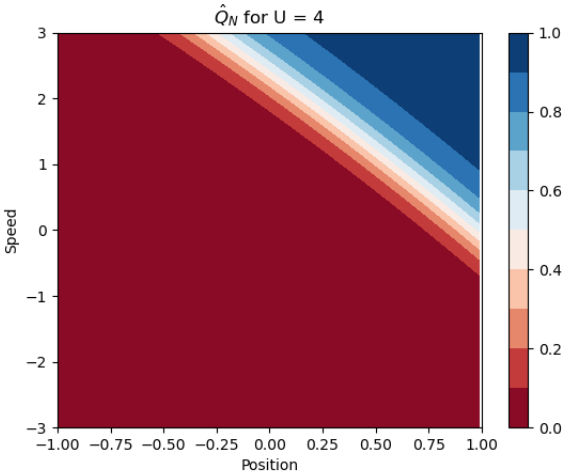

Figure 13: $\hat{Q}$ for $u = 4$ for Q-Learning neural network

And for the estimated optimal policy :

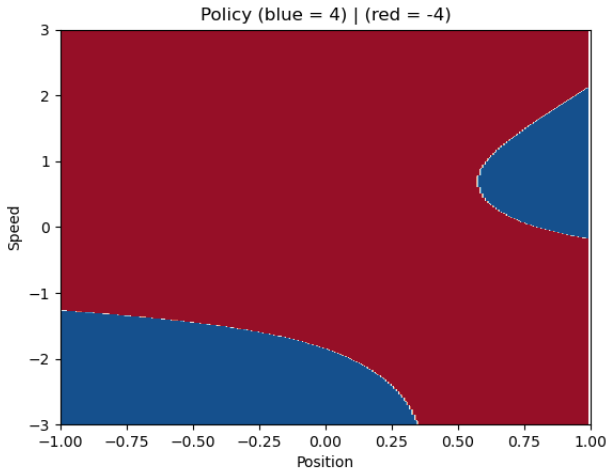

Figure 14: Estimated policy $\hat{\mu}$ for Q-Learning neural network

With 0.2 as expected return.

As you can see we don't have much to compare neither since the results we get with the current algorithms are just really bad, and thus are worst than the results we got before. It seems that our algorithms are really bad with images.

4 Deep Q-Learning VS Deep Q-Network

The implementation of the DQN is in the `DQN.py` file.
A common implementation of DQN is double Q-learning (as explained here), or in the lectures, or even here), which separates the CNN computing the predicted value and the one computing target value (used in the loss function), to avoid instabilies. We thus create a target network, which has exactly the same architecture than our CNN, but that isn't updated simultaneously with our CNN, but updated only periodically after some steps. We chose to update it every 20 steps.

Here are the results we got :

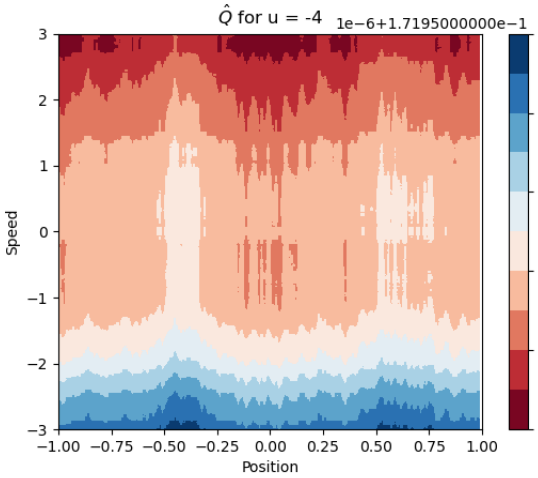

Figure 15: $\hat{Q}$ for $u = -4$ for DQN

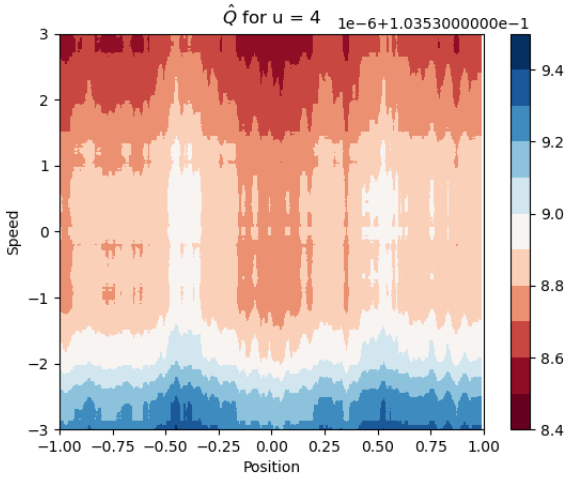

Figure 16: $\hat{Q}$ for $u = 4$ for DQN

And for the estimated optimal policy :

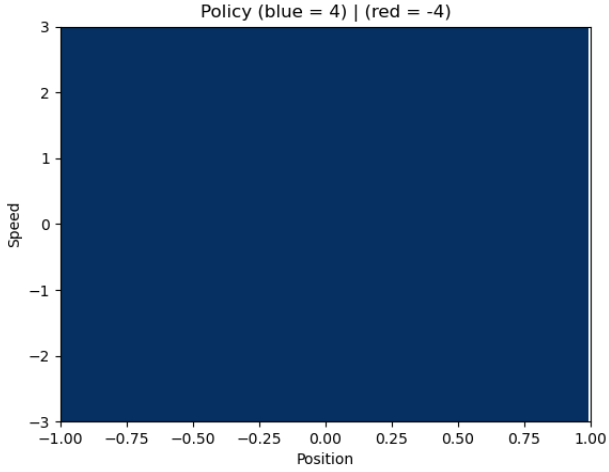

Figure 17: Estimated policy $\hat{\mu}$ for DQN

With 0.0 as expected return.

We obtain the same results than before, that are really bad too.

5 Conclusion

Our algorithms seem to work really bad when the input is an image representing the state of the game, instead of the real state of the game.
There may be multiple reasons for that :

- It's an image, it can't be as accurate as the real state of the game
- The image are harder to compute, they take more memory so there is a limit in the generation of our trajectories (for example, I tried to generate 50 000 transitions but my program just crashed), where we would need longer trajectories than with the real state of the game.
- Our CNN architecture may not be optimal. We could have wider hidden layers and more of them, but once again we are limited with the computational capacity of our computers.