Lab 5 report

Lucie Sénéclauze Bertolo

April 30, 2025

1 Introduction

In this report I describe the use of Deep Q-networks for the cartpole and atari pong environments. In the first part of the report, I go over the implementation of vanilla DQN with the gymnasium librairy. Then I explain how I went from vanilla DQN to enhanced DQN by adding multi step reward, prioritised experience replay and double DQN. In the second part of the report, I comment my results and model's performances and analyse the efficiency of the enhancements by performing an ablation study.

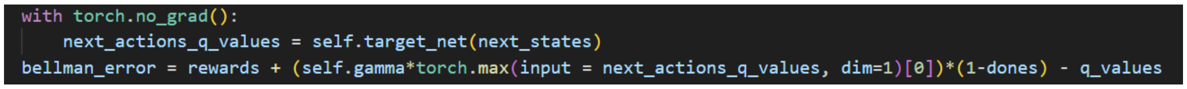
2 Implementation

2.1 Bellman error for DQN

In DQN the formula for the Bellman error is

$$\text{Bellman error}(\theta) = r + \gamma \max_{a' \in A} Q_{target}(s', a'; \bar{\theta}) - Q_{main}(s, a; \theta)$$

where D denotes the sampled batch of transitions from the replay memory, and Q_{main} and Q_{target} denote the main Q network with parameters θ and the target Q network with parameters $\bar{\theta}$, respectively. From this equation, the implementation of the Bellman error is rather straightforward given that we know the batch's reward and q values and we can compute the q values for the nest step with the networks, the current states and actions and the next states. In Python, this becomes :



```
with torch.no_grad():
    next_actions_q_values = self.target_net(next_states)
bellman_error = rewards + (self.gamma*torch.max(input = next_actions_q_values, dim=1)[0])*(1-dones) - q_values
```

Figure 1: Bellman error

2.2 From DQN to DDQN

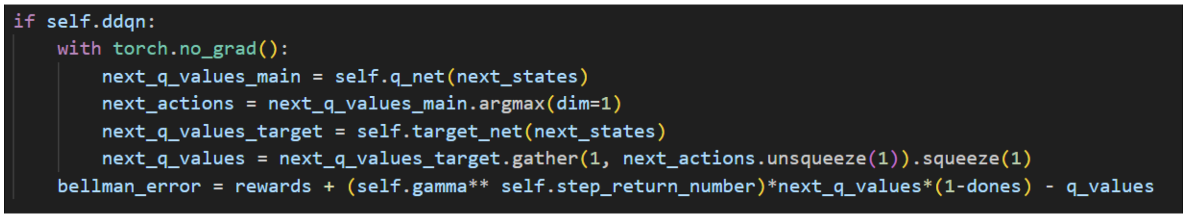
Since we already have a target network that we used to compute the Bellman error in simple DQN, we don't need to implement it from scratch for double DQN. Hence the only code difference between simple DQN and double DQN will be during the computation of the loss.
Indeed, for simple DQN, the loss is simply the sum of the squares of the Bellman errors :

$$\text{LDQN}(\theta) := \frac{1}{2} \sum_{(s,a,r,s') \in D} \left(r + \gamma \max_{a' \in A} Q_{target}(s', a'; \bar{\theta}) - Q_{main}(s, a; \theta) \right)^2$$

For DDQN, the loss becomes

$$\text{L}_{\text{DDQN}}(\theta) := \frac{1}{2} \sum_{(s,a,r,s') \in D} \left(r + \gamma Q_{target} \left(s', \arg\max_{a' \in A} Q_{main}(s', a'; \theta); \bar{\theta} \right) - Q_{main}(s, a; \theta) \right)^2$$

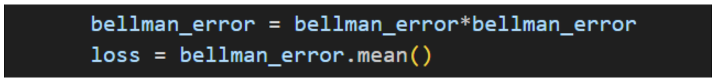
As we can see, we can simply change the computation of the Bellman error. In Python this becomes :



```
if self.ddqn:
    with torch.no_grad():
        next_q_values_main = self.q_net(next_states)
        next_actions = next_q_values_main.argmax(dim=1)
        next_q_values_target = self.target_net(next_states)
        next_q_values = next_q_values_target.gather(1, next_actions.unsqueeze(1)).squeeze(1)
    bellman_error = rewards + (self.gamma** self.step_return_number)*next_q_values*(1-dones) - q_values
```

Figure 2: Bellman error for double DQN

In my implementation of the loss, I changed the 1/2 by a mean over all the Bellman errors, as shown in figure 3 because it helped the loss to become smaller and make the computations faster and more efficient.



```
bellman_error = bellman_error*bellman_error
loss = bellman_error.mean()
```

Figure 3: Loss computation

2.3 Memory buffer for PER

The idea of using Prioritised experience replay is not sample transitions from the replay buffer, not randomly but proportionnaly to their Bellman error. This way, we investigate more the transitions that are the most "surprising".
In Vanilla DQN, I implemented the memory buffer as a simple list where I randomly draw batch_size transitions. For the PER DQN, I used the PrioritizedReplayBuffer class and filled its 3 functions :

- In the function *add*, I compute the priority of the new transition which is equal to $|Bellman\_error| + \epsilon$ with ϵ set to 10^{-6}. Then I replace the oldest transition in the buffer with the new transition and I put its error in the priority list.
- In the function *sample*, I select batch_size elements from the batch and each element has probability $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ to be selected where α is an hyperparameter. This function also return the IS_weights who are computed as $w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta$ with β an hyperparameter and who serve to cancel-out the induced bias in the loss caused by the PER method.
- In the function *update_priorities*, I recompute the priorities for each element in the replay buffer.
- I also added a *len* function to get the number of element in the buffer to simplify the computations later in the code.

2.4 From 1 step return to multi step return

To implement the multi step return, I created a small buffer (via a deque) of size step_return_number which is an hyperparameter. Then in the *run* step, where I used to add the transition with its associated error to the replay buffer (or the memory list before PER), I implemented the computation of the n_step error : $R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q(s_{t+n}, a')$ using the values stored in the buffer I created.
For cartpole I used 3-steps return and for pong I used 4-step return, thought I didn't get the time to explore the impact that this parameters could have on the efficiency of the multi-step return.

2.5 Weights and bias

Througouth the training, I used weight and bias to monitor the evolution of significant parameters such as the loss, the update count, the total reward, the step count, the evaluation reward, epsilon, the episode and the environment step count. This allowed me to see the evolution of these metrics in real time to debug and evaluate the performances.

3 Analysis and discussion

3.1 Training curves

3.1.1 Task 1

In figure 4, we can see the running average evaluation score versus environment step for task 1 - vanilla DQN for cartpole.

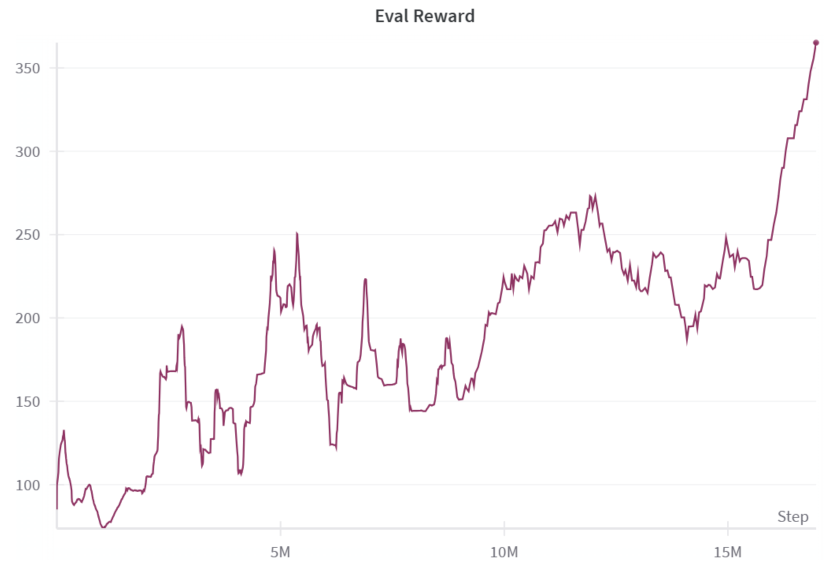


Figure 4: Training curve for task 1

3.1.2 Task 2

In figure, 5, we can see the training curve for task 2 - a vanilla DQN adapted for a Pong problem. In terms of implementation, the difference between DQN for cartpole and DQN for pong is that pong has an image pre-processor to pre-process the RGB colors (cartpole doesn't need one as all the information about the system's state can be easily computed in a vector) and that the Q-function approximator is a CNN for Pong but a NN for cartpole.
On this figure we can also see the impact of ϵ in the training. Indeed in this run, ϵ reached 0.05, its minimum value, after 3M steps and we can see that after 3M steps the evaluation reward ceases to increase and stagnates at a less than optimal evaluation reward. This is because when ϵ is really small our ϵ greedy policy is almost equivalent to a greedy policy thus the model does not explore and the evaluation reward does not increases. However I found that finding a good ϵ was tricky because if it decreases too slowly the model takes a lot of time to train so it is a tradeoff between training time and performances. If I had had more time, I would have run the program with a really small ϵ decay to see the outcome in hope of increasing my final performance result.

3.1.3 Task 3

In figure 6, I plotted the training curve for the enhanced DQN (using double DQN, prioritised experience replay and multi step return at the same time) for the Pong environment. To put it in perspective,

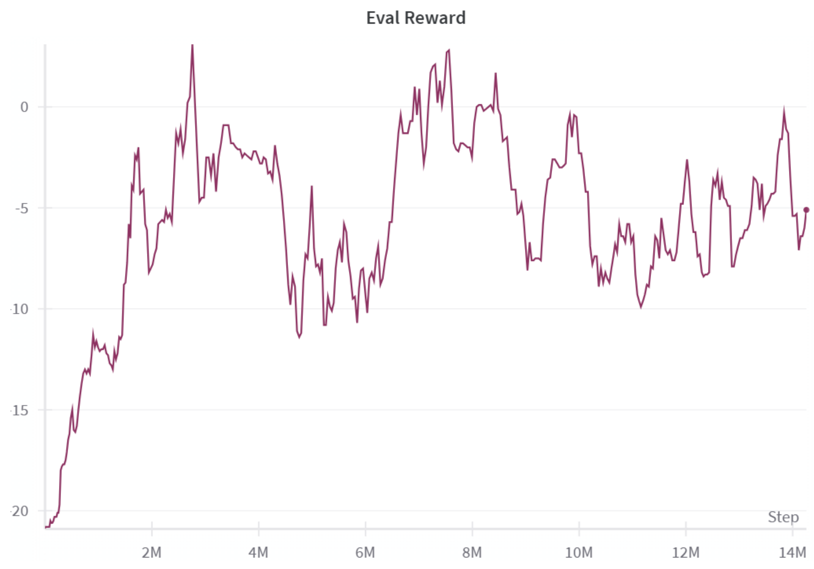


Figure 5: Training curve for task 2

in figure 7 I plotted the comparison between the vanilla and the enhanced DQN.

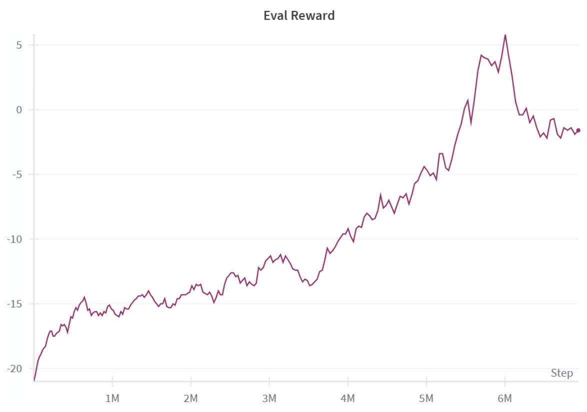


Figure 6: Training curve for enhanced DQN

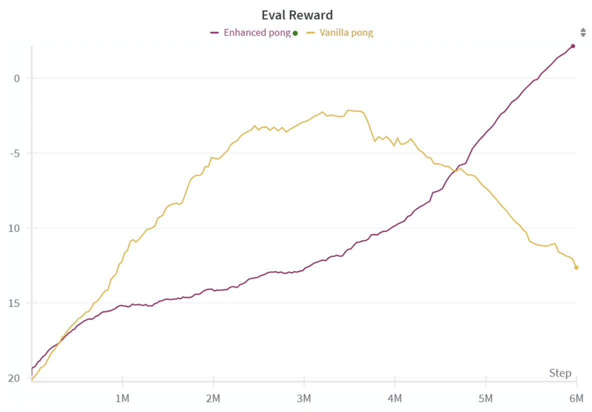


Figure 7: Training curve for both enhanced and vanilla DQN

As we can see, the enhanced DQN gives better results on the long term while vanilla DQN does no have a steady increase and decreases at the end.

3.2 Efficiency of DQN enhancement

As we can see in figure 7, the enhancement yields convincing results. It is also visible for the cartpole environment in figure 8 with the enhanced reaching the maximum score of 500 really quickly and the vanilla one not reaching above 300.

In order to visualise the imapct of each enhancement, I run several model each while enabling only one of the enhancement. In figure 9, we can see the training curve for each of them.

As we can see, the most useful enhancement is the multi step replay as its performances are just slightly below the one with all the enhancements. The second more useful enhancement is the prioritised experience replay and the last is double DQN with performances more or less equivalent to vanilla DQN. However this does not mean double DQN and PER are useless as their unconvincing results could be due to a poor hyperparameter choice. Moreover, we should note that this is the results for only one run of each and that instability and luck might impact the model's performance.

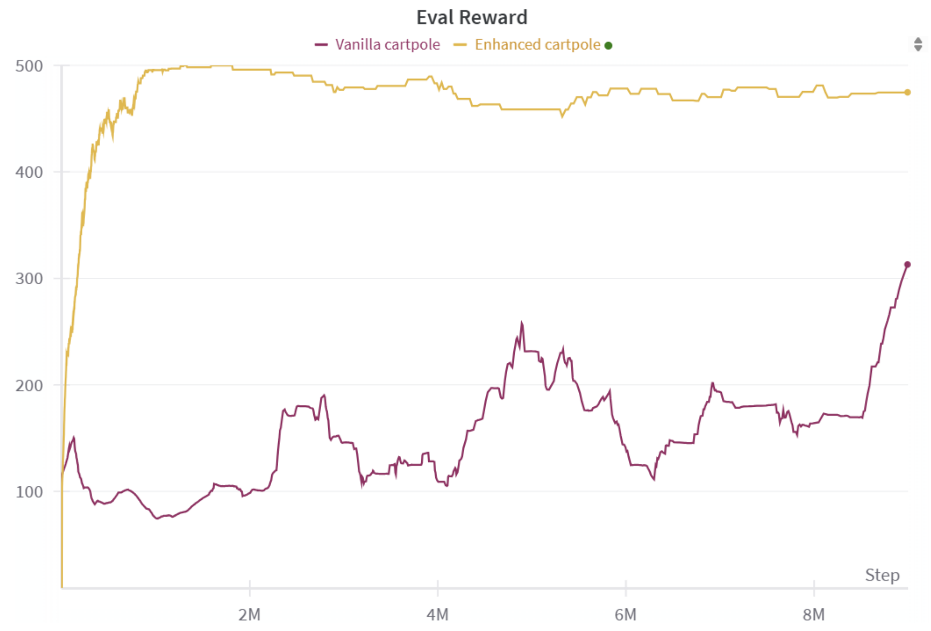


Figure 8: training curve for both enhanced and vanilla cartpole DQN

To illustrate this, figure 10 shows the training curve of two runs of vanilla DQN using the same parameters. As we can see despite the models being identical the outcomes are significantly different.

4 Conclusion

In this report, I showed that adding enhancements like multi step return, PER and double DQN improves the model's performance. The ablation study confirmed that multi step return was the most impactful. However, results can vary a lot depending on the run and the hyperparameters, such as ϵ, so further testing would be necessary.

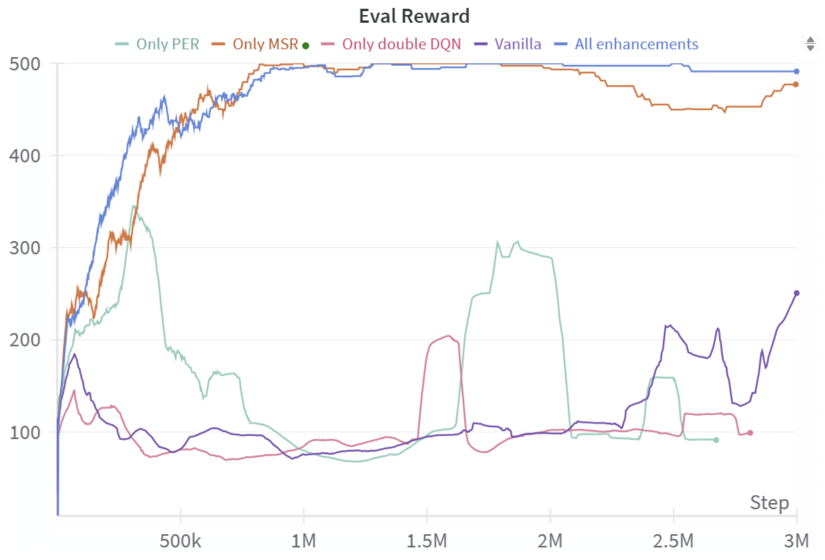


Figure 9: training curve for different enhancements

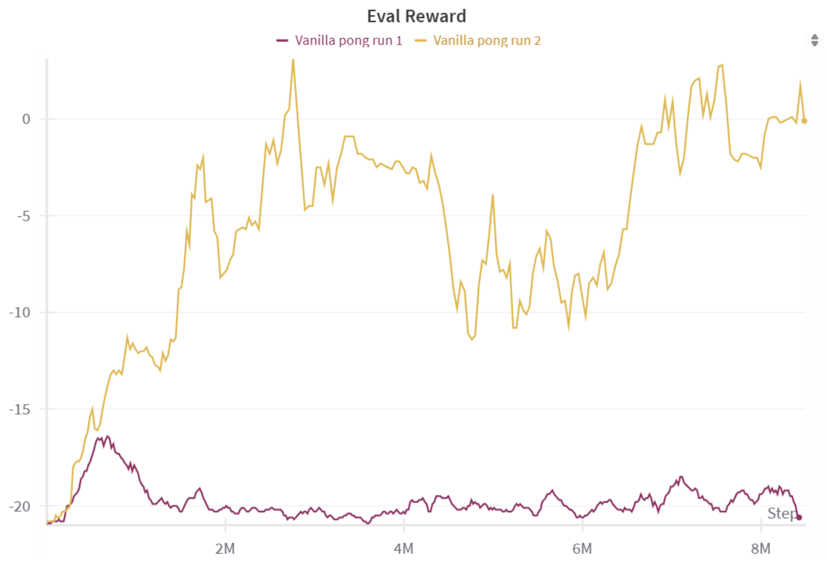


Figure 10: Training curves for two identical models