RL Report

Arina and Tarek

28 July 2023

1 Introduction

Reinforcement learning (RL) is a powerful paradigm that allows agents to learn and adapt to environments through interactions to achieve specified goals. Deep Q-Learning (DQN) is an example of RL that have gained much traction and attention in the recent years. The main strength of DQL is using and training a neural network to approximate the action-value function to make informed decisions in the environment. The combination of deep learning and reinforcement learning has shown remarkable success in various domains, from playing Atari games to controlling real-world robotic systems.

The main objective of this project is to 1) gain insights into the design of RL experiments, 2) investigate the impact of using different hyperparameters on the performance of the agent after training. In a DQN setup, there are quite a lot of hyperparameters and they play a critical role in the success of the learning process. Throughout this report there are some hyperparameters that we didn't tweak such as learning rate and batch size. And there were others that we were interested in and played with their values like discount factor (gamma), exploration rate (epsilon) and the loss function. Our goal was to examine how these parameters impact stability, convergence and overall performance of the DQN algorithm.

2 Methods

For our experiment, we used OpenAI Gym environment and PyTorch library. The OpenAI Gym provides a standardised interface for different set of environments. This allowed us to rapidly assess, as well as visualize, the DQN algorithm's performance across two different environments; 1D track and 2D track environments. In both of these environments, we had one agent with a set of actions, walls and reward blocks. The 2D track also featured poison blocks and a moving patrol agent that should be avoided. Pictures of both are appended at the end.

We implemented the Deep-Q-Learning algorithm within the PyTorch package. A basic online training loop was first developed. Each loop iteration composed of 1) Initializing the environment, 2) Running our Q network to decided on which action to take, 3) Executing this action within the environment, 4) Evaluating this action and updating the network using the MSBE loss function. An experiment is essentially running a number of episodes (specific number of loop iterations or until environment terminates), making sure our agent is learning.

The methods applied to the 1D track are exactly the same ones applied to the 2D track. We implemented different variations, mainly:

2.0.1 State Representation

How do we represent one state of our environment? There are different ways to interpret and represent a state. Let's take the 1D track as an example. If we were to implement a tabular Q-learning algorithm. We would think of our state as a an integer array of rewards and a pointer representing where the agent is standing. An action of the agent would be analogous to moving

the pointer, collecting an award and then updating the awards array by removing the collected award.

Representing the q-values would be challenging in this case though, because if we only have one Q-value per position/action pair, then it wouldn't be possible to distinguish between going right on a cell when we have all rewards collected or with zero reward. Because going right on that cell might mean different things depending on the state of our game. This means we need the history or an image of how the whole track looks when deciding whether to go right or left on every cell. This makes the tabular Q-value representation tricky in this situation. Of course if this is a small track it's manageable, but with bigger environments and more complicated actions it becomes more complex.

Therefore, the flattened representation we use in our neural network alleviate these issues. We pass a flatten array representing every position in the environment and which element occupies it. The output of the network are the q-values representing this state and the corresponding actions.

2.0.2 Evaluation method for the results

To evaluate the performance we plot the cumulative and discounted rewards after every N iteration steps, either N = 100 for Linear track or N = 1000 for 2D grid. We calculate the cumulative reward by running 3 episodes and taking the average of reward collected in these episodes. The length of these episodes are 20 for 1D track and 40 for 2D grid. With the help of discounted reward plot we can see how many steps it took the agent to achieve a reward in a given number of N overall steps.

2.0.3 Experience Replay

One very important aspect of our analysis was using Experience Replay (ER). This method ensures we use our previous experience in learning about the environment. The way we do this is by filling a First-in-first-out array with tuples of the current state, next state, action and reward. In every iteration of our loop we have two phases; an environment step and a learning phase. A learning phase is basically sampling a batch randomly from past experiences and using this batch to learn. As mentioned in DeepMind's famous 2012 Atari paper, ER is a way to remove correlations in the observation sequence and helps with convergance.

2.0.4 Exploration-Exploitation

Regarding the Exploration-Exploitation Tradeoff, The exploration-exploitation tradeoff is vital for RL agents to strike a balance between exploring new actions and exploiting known ones. We will be using an e-greedy policy with an epsilon being constant in the first 20% of the learning process, then an annealing epsilon that goes to zero in the rest of the learning process. Thus, ensuring the agent is more exploratory in the beginning and then ending with exploitative behaviour.

3 Results

We present our results best obtained results first. Then we go through different experiments. Within each comparison we talk about both 1D and 2D environments.

3.1 Best Results

Throughout the different trials we made, the best results were obtained by running the training using experience replay, a high γ value of 0.99 and running it for 500 thousand iterations. Figure 1 show the evolution of the accumulated reward as the discounted reward. The discounted reward line is going to make more sense when we start talking about the different γ values. Training on more update steps (500K) means better learning and ultimately finding better rewards.

An important question we had for the linear track was Why does the agent not collect coins after it goes and collects the diamond? It seems that the agent sees it unprofitable to do a lot of steps on the left and to get this diamond?

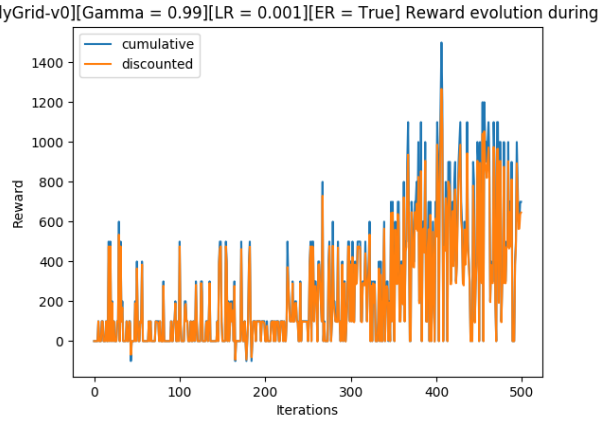


Figure 1: [2D Track] Best results obtained $\gamma = 0.99$ with experience replay in 500K steps

3.2 Different γ values

The Gamma γ parameter which controls the "memory" of the agent and how important future rewards are is a key variable to tweak. A max value of 1 means that the agent will evaluate every action based on the sum of all future rewards while a min value of 0 means the agent is myopic and has no "vision" for the future. When it is more profitable to get coins and not the diamond. Our steps are implicitly negatively discounted.

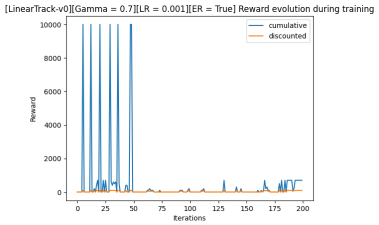
Initially we tried to experiment without exp replay, but in the best gamma case, 0.99, the agent still was going to collect all the coins and not the diamond given 20000 steps. So we decided to stick with experience replay regime.

Summary of experimentation with gamma and with experience replay and 20K loop steps. Figure 2 shows the effect of using different gammas on the discounted accumulated reward. It makes sense to see the discounted reward decreasing with gamma which

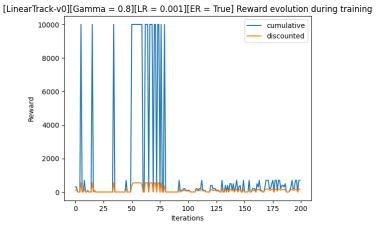
gamma	observation	max cumulative reward
0.99	the most far seeing, remembers that it's good to get a diamond, hence more hitting the diamond during training	1000
0.9	7	78
0.8	545	7787
0.7	just hangs around the starting point	1000

Similarly to Linear track, training on more update steps means finding better reward.

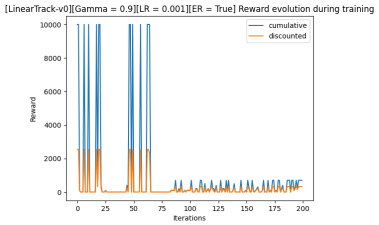
For the γ values experiments we decided to work with two "extreme" values, 0.99 and 0.7 due to the time constraints. Training was run on 100K steps with usual parameters and experience replay. In case of $\gamma = 0.7$ we could clearly see the difference between cumulative reward and discounted one, since the later one was considerably lower than the former one. This indicated the fact that the agent believe in the future rewards is much less that belief of the agent with larger γ. So in the case of small γ the agent is going to optimise with respect to the discounted



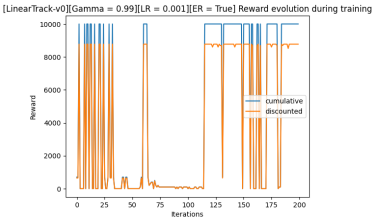
(a) $\gamma = 0.7$



(b) $\gamma = 0.8$

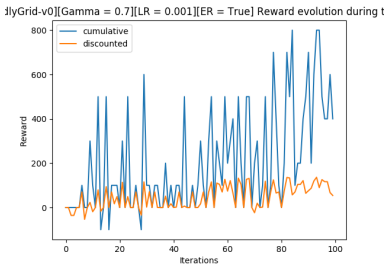


(c) $\gamma = 0.9$

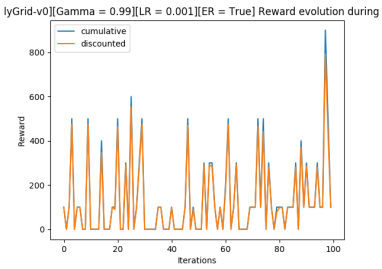


(d) $\gamma = 0.99$

Figure 2: [1D Track] Effect of γ on cumulative reward perception, 20K steps



(a) $\gamma = 0.7$



(b) $\gamma = 0.99$

Figure 3: [2D Track] Effect of γ on cumulative reward perception, 100K steps

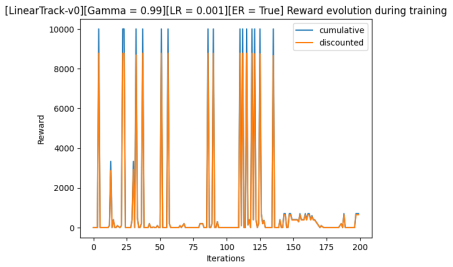
reward rather than cumulative one and will strive to get the best possible actions right now and right here.

To be fair, $\gamma = 0.7$ was event better on average than $\gamma = 0.99$ for relatively small number of training steps, 100K, with total reward 600 in comparison with 400. So in a way agent was quite cautious to take risks and go for nearby gold coins since it remembered the danger of gas cells and a patroller nearby.

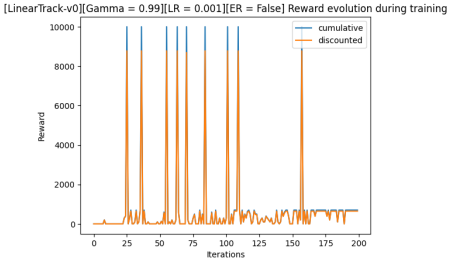
Very interesting thing happened when training for 1 million steps with experience replay and $\gamma = 0.99$: in some trials the agent was just standing frozen and in others it was not performing so well, only collecting 6 coins. For $\gamma = 0.7$ the agent collected 7 coins which was a better outcome.

3.3 With and without Experience Replay

Our results with experience replay was a bit confusing because it didn't necessarily affect the performance of the agent in all cases. With plotting the accumulated rewards for both environments we were able to see a pattern. The agent seems to learn faster and hit the higher rewards earlier and more frequent during the learning iterations. As we can see in 5, the average rewards of is a bit higher when using experience replay.

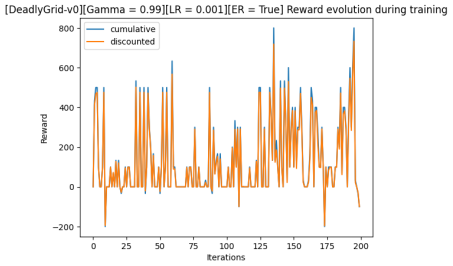


(a) With Experience Replay

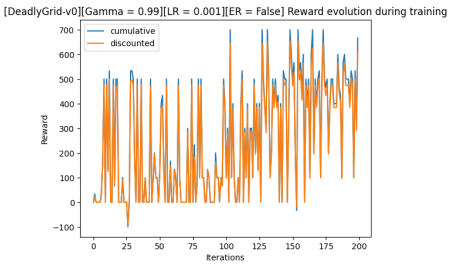


(b) Without Experience Replay

Figure 4: [1D Track] Effect of ER on learning progress



(a) With Experience Replay



(b) Without Experience Replay

Figure 5: [2D Track] Effect of ER on learning progress

3.4 Loss Functions

For most of our trials we were using the standard Mean Squared Error (MSE) where the error is calculated by subtracting the output of the network from the "desired" output and getting it's squared. We tried using the smooth L1 Huber's loss function, which would provide a ceil loss in case the loss value is higher than a specific constant (1 in this case). 6

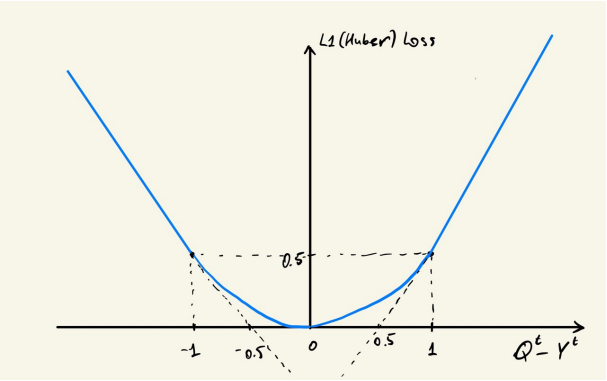
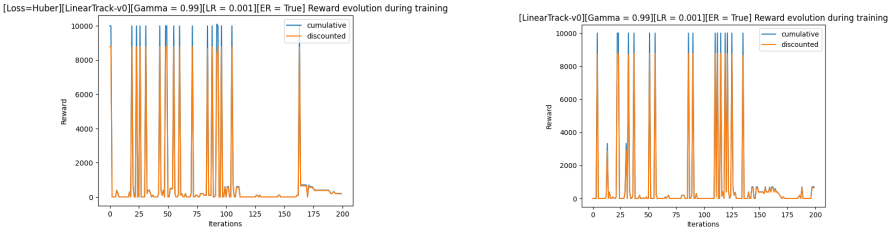


Figure 6: Huber loss function



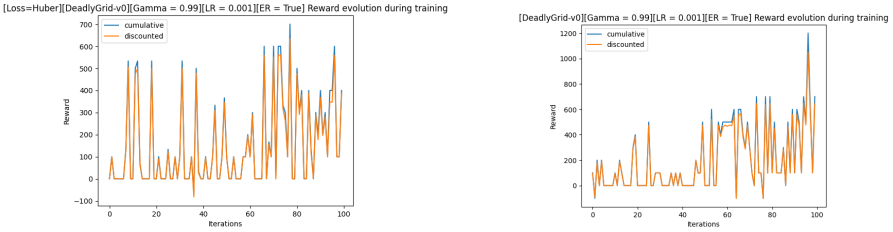
(a) Huber

(b) MSE

Figure 7: [1D Track] Different loss functions

3.5 General discussion

- In general, with introduction of experience replay we gained less noisy approximation of the gradient step, in comparison with online learning (batch size 1), which ideally helps the learning algorithm to arrive to the, possibly, global minimum faster.
- Interaction between learning rate and batch size: our default batch size was 128 elements, which is fairly big one for a small neural network. It gives us less noisy estimation of gradient which is good for a more accurate gradient descent. Since we are using "cleverer" optimisation algorithm rather than vanilla SGD, we can not worry too much about changing batch size if we change learning rate.
- If we don't detach some vectors we calculate, for instance, $\hat{Y}$, then the whole calculation flow will be screwed as change in $\hat{Y}$ is going to influence the update of network's parameters, which we do not want to, since the only meaningful indicator of "goodness" of our network is the maximisation of action-state function Q.
- Shallow network vs 1 hidden layer network:



(a) Huber (b) MSE

Figure 8: [2D Track] Different loss functions