Probabilistic reasoning and learning project report: play Atari games using artificial neural networks

a.a. 2017-2018

Jary Pomponi

1 Introduction

Reinforcement learning is the area of machine learning that focuses on how agents take action in an environment in order to maximize a value called reward. Those agents should learn which action is better in each situation. In this project I have implemented different methods to teach an agent how to play Atari games just by observing the video and knowing the number of possible actions, without further information about them. In particular the game used is Pong. This approach follows the one proposed in [1] and studied deeper in [2].

2 Q learning

The agents act in an environment, which is formulated as a Markov decision process and defined as a tuple (S, A, P, R, γ) in which:

- S is the set of possible states.
- A is the set of possible actions.
- $P(s, a, s')$ is the probability of transition from state s to state s' taking the action a.
- $R(s, a, s')$ is the immediate reward obtained after transition from s to s' taking the action a.
- γ is the discount factor (how much the future rewards are important).

At time step 0 the agent is in the initial state s_0 with a given probability. Then the agent pick an action a_t, get the reward r_t and transit to the next state s_{t+1}. This is done until the terminal state is reached. But how the action a_t should be chosen? To choose the best action in each state a function π, called policy function is used. This is a function that specify which action to take in a given state. The optimal policy π^* is the one that maximizes the cumulative discounted reward over time: $\sum_{t\geq 0} \gamma^t r_t$. Formally:

$$\pi^* = \arg\max_{\pi} \mathbb{E}(\sum_{t\geq 0} \gamma^t r_t | \pi)$$

Following a policy produces a path in the state space. To know how good is a state a function called value function is used:

$$V^{\pi}(s) = \mathbb{E}(\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi)$$

It is also possible to know how good a pair state-action is. This function is called Q function and is the one that I will mainly use:

$$Q^{\pi}(s, a) = \mathbb{E}(\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi)$$

If the optimal state-action value for the next time step is known then the optimal strategy is taking the action that maximizes the expected value of $r + \gamma Q^*(s', a')$. So:

$$Q^{\pi}(s, a) = \begin{cases} r + \max_{a'} \gamma Q^*(s', a') | s, a & if\ not\ terminal \\ r & otherwise \end{cases} \tag{1}$$

So the optimal policy can be found by iterating over all the possible pairs and computing the best action in each state, but this is not feasible if the states space is not known or if it is huge, like in the Atari case. To overcome this problem a function approximator can be used to estimate $Q(s, a)$ and $V(s)$. I will do this by using artificial neural networks.

3 Deep Q-learning (DQN)

The basic architecture that I have used to approximate the Q function can be seen in Figure 1a. Given a state the networks calculate the expected cumulative reward for each possible action.

The problem is handled as a supervised learning one and the used loss is the so called squared error:

$$L(\Theta_i) = ((r + \gamma Q(s', a'; \Theta_{i-i}) - Q^*(s, a; \Theta_i))^2$$

Where Θ are the weights of the network. This is good because when a larger error shows up the network tries to minimize it, but doing this the weights change radically and with these also the target value. To overcome this problem and improve stability and convergence properties it is useful to clip rewards in -1 and 1 and to use another loss function called Huber. It takes into account the difference between large and small errors by decoupling the calculation:

$$L(\Theta_i) = \begin{cases} \frac{1}{2}e^2 & if\ |e| < 1 \\ |e| - \frac{1}{2} & otherwise \end{cases}$$

where $e = r + \gamma Q(s', a'; \Theta_{i-i}) - Q^*(s, a; \Theta_i)$. This loss gives the same weight to low and high errors, improving stability and convergence.

In the following sub-sections I will explain the used technique and architecture used in the train process.

3.1 Memory of the past experiences

The approximation of Q function with an artificial neural network has a problem: while the agent learns how to act in the current state it also forgets the previous explored states. This does not allow the agent to learn how to act properly. To overcome this problem the passed states are saved in a ring buffer memory with a fixed length. These experiences are randomly sampled each time the network is trained. A passed experience at time t is saved as a tuple $(s_t, a_t, r_t, s_{t+1}, ending)$ where $ending$ indicates if the state is an ending one or not and s_t are the states from $t-3$ to t stacked. A subset of the memory is randomly extracted each time a network is trained.

Fixing the length of the memory is crucial since a small one can not solve the agent's memory problem but a huge one will slow the learning process, and make this impossible in the worst case, since it will contain too old states not useful in the last phases of the training process.

This method can be further improved by sampling from the memory using a probability distribution instead of doing it randomly. To do that I have implemented the proportional prioritized experience replay memory presented in [3]. The main idea is that a transition is more desirable in the training process if the network can learn more from it. In order to know how much the network can learn from an experience it is saved in the memory with an error:

$$error = |Q(s, a) - T(s)|$$

Where $T(s)$ is the estimated reward obtainable from s. If the learning process is not started $T(s)$ is simply the immediate reward r. Then this error is converted to a probability $p_i = (error_i + 0.001)^\beta$ where β weights how much the higher errors are more important than the others. The sample process is the following. Each time a sample is needed a random number s, $0 \leq s \leq \sum_i p_i$ is picked and the memory would be walked summing the probability of visited experiences until the sum exceeds s, then the current transition is picked. To do that the memory should be sorted based on the p_i since transitions with high error should be favored. This is very expensive and another approach involves the using of a binary unsorted tree structure where each node contains the sum of its children. When a node is updated the change propagates through all the tree. With this structure the operations are faster and the maintenance easier.

3.2 Double DQN (target network)

To choose the action a that should be picked in a state s and to estimate the future reward obtainable in the new state, the same network is used. This network is like a cat chasing its own tail since the network sets itself its targets and then follow them. This could lead to instability, oscillations or divergences. This problem can be solved by using another network to estimates the future reward.

This new network is simply a copy of the first one frozen in time, so it won't be update using a train procedure, but after several steps the weights from the first network are copied in the second one. With this improvement the Q function becomes:

$$Q(s, a) = r + \gamma \max_{a'} Q'(s', a')$$

Where Q' is estimated with the second network, called target network. A drawback is that it substantially slows down the learning process because change in the Q function is propagated only after the target network update. The intervals between updates are usually in order of thousands of steps, so this can really slow things down. But this improves the stability so it is worth to use it.

3.3 Duel DQN (DDQN)

Dueling DQN uses a specialized Head in order to separate Q into A (advantage) stream and a V stream. Adding this type of structure to the network Head allows the network to better differentiate actions from one another, and significantly improves the learning. With this network the Q function becomes:

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{N} \sum_{a'}^{N} A(s, a'))$$

The main improvement of this approach is the faster learning. In DQN the network updates the Q values only for the specific actions taken in those states. This results in a slower learning as we do not learn the Q values for actions that were not taken yet. Dueling architecture starts learning the state-value even if only a single action has been taken in current state by decoupling the Q function. This architecture are shown in Figure 1b.

4 Training process

As environment I have used the ai gym library with some preprocessing. The first one consists in transforming the images into gray scale and resizing them to 84×84. It should be then taken into account that the Atari games are designed for TVs, so the update frequency are high and this could lead the experience memory to store too many similar images. To overcome this problem only the 4th image is taken, skipping the three before it. In addition to this, it is defined a state that is the stack of the last previously three not skipped images and the current one, giving the agent an overview of what happened before and helping it in taking the right decision. This helps the agent to know, for example, where the ball is moving.

The agent cannot starts to learn how to move in a state space that is unknown. So it is important to explore the state space, even if it is huge, before starting to learn how to act in it, otherwise the agent will move randomly since it does not know how the game is structured. To do that I have implemented the following approaches, all used at the same time:

1. Populating replay memory: the train process starts only when the memory contains a certain amount of experiences. This helps the agent in knowing how the state space is structured before it starts acting.

2. No actions taken: at the beginning of the game the agent stays stationary for a random amount of frames from 1 to 30. These frames are not added to the memory. This helps exploration and train speed, since the starting states are all very similar.

3. Random action picking: the probability of picking a random action in a state starts from a ϵ value and decrease linearly until a minimum is reached. This is the crucial exploration strategy since allows the agent to explore a large amount of states and associated actions while collecting the rewards. This is important since the agent needs to know which are the useful action in the states.

If the memory is a prior one then after the update of the network the tree of probability should be updated. More than that if a target network has being used, then after the update of network weights these should be copied in the target network, if it is time to do it.

The final algorithm is:

Algorithm 1: Q-DQN generic algorithm

```
-Initialize memory M to capacity N
-Initialize action-value function with random weights
for episode=1 to Eps do
  get preprocessed state s
  while game is not done do
    -with probability ϵ get a random action
    otherwise select a_t = max_a' Q(s, a')
    -perform action, collect reward r and
    next state r_n
    -store experience (s, a, r, s_n) in
    memory M
    if enough experience in M then
      -sample random batch from M and
      update batch targets
      using formula (1)
      -update network weights according
      to loss
    end if
  end while
end for
```

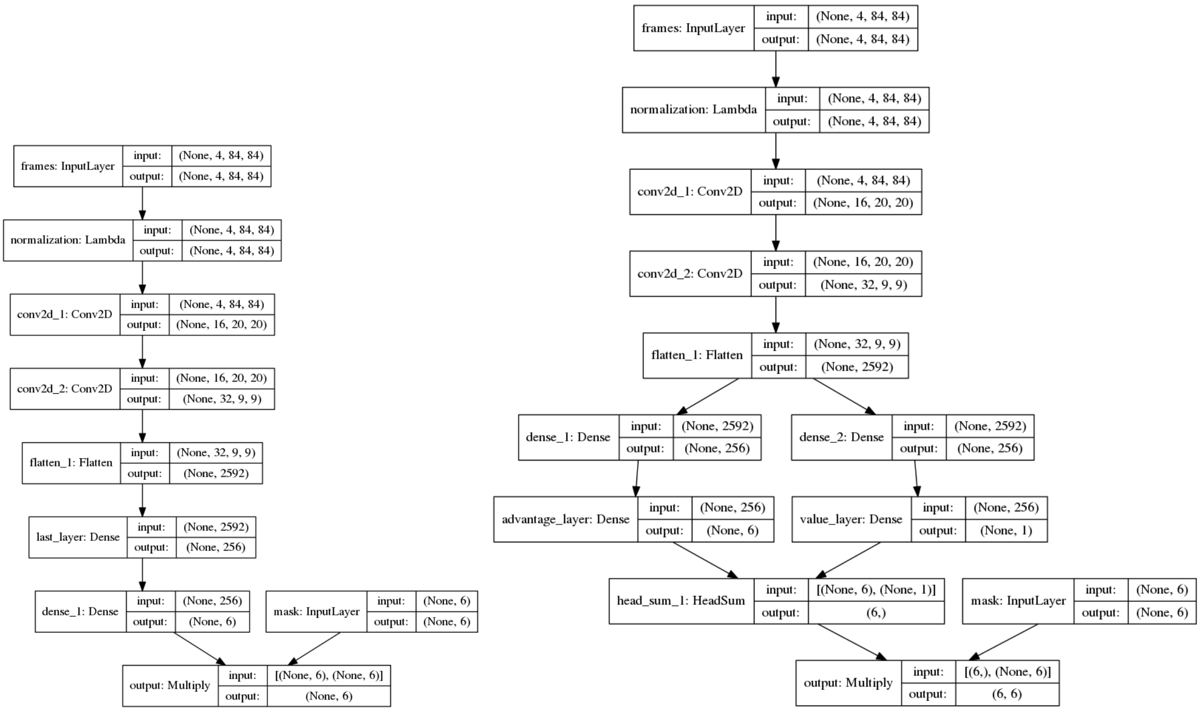
5 Empirical evaluations

To study which network is better I have fixed the parameters after some preliminary empirical evaluations. The length of the memory is set to 1 million and the reasons are two: the memory available on my machine is limited and the game is not hard and states tend to be all very similar. The ϵ value starts from 1 and decreases in 1 million step and reaches 0.02 as minimum. The batch sampled from the memory contains 96 transitions and the train process starts only after 10000 observations. Then the learning rate is set to $1e - 4$ and Adam optimizer is used. These values are the same for all the networks.

The experiments are divided in two set. In both of them the agent uses the two proposed networks with a target one, but in the first set of experiment a simple memory is used, while in the second one the memory is a prioritized one. The target network is used because it is know in the literature that using it leads to better results. The goal is to study which combination is better for the given game. The evaluation criteria are the following: if the network reaches stability, if the agent have learned how to beat the ai and how many episodes it needed to learn it. More that that each agent plays the same game and comparisons of the rewards, the q values and the stability reached are made. The videos of these games are in the video folder delivered with this report. Each experiment consists in the agent playing 300 games. The complete results can be viewed in figure 2 and 3.

About the first experiment, it is possible to notice how the agent using a duel architecture can achieve a positive mean in less time, compared to the agent using simply DQN. This is due to the fact that it learns Q value of actions not explored yet, as told before. But at the end of the training process the mean reaches the same value of the first network, this due the fact that the duel network is very unstable in the end of the experiment. Moreover it achieves a worst result in the final game. This could be due to the fact that the parameters chosen are not good for this type of architecture, for example the decay of the learning rate is not enough or the memory size is too small and a bigger one is needed in order to help the network to remember the correct Q value estimation of each action, since it looks like the networks overestimate these values. So even if the first network train is slower it should be preferable since it leads to more stable results. In fact this is the architecture used in [1] and [2]. The last mean value reached are 15.5 for the first network and 15 for the duel one.

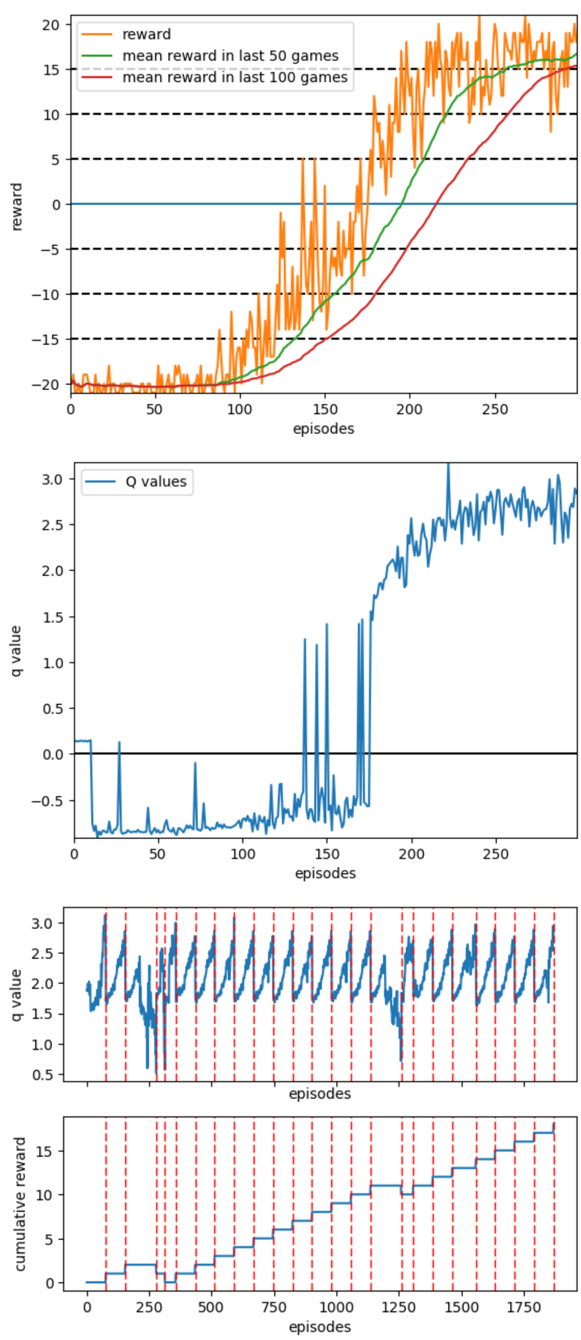
From the second set of experiments we expect that the agents will learn faster and better due to the prioritized replay buffer. In fact the prioritized DDQN reaches a positive mean after 100 games, compared to 190 needed by DDQN with simply memory. The final achieved mean is 17,5. In addiction to this, the network looks very stable in the long period. This network reaches better results even in the final game since the final score is 19, compared to 16 achieved by the DDQN. Comparing the duel architectures seem that the one that uses the prioritized memory tends to stabilize the mean in the late training process, but without exceed 16,5, even if the score achieved in the final game is the highest one. So in the and the preferred architecture should be the DDQN since the train process takes less time and it is more stable. In addition to this a prioritized replay memory can increase the score and decrease the training process time, since it helps the network to reaches good and stable results in less time.



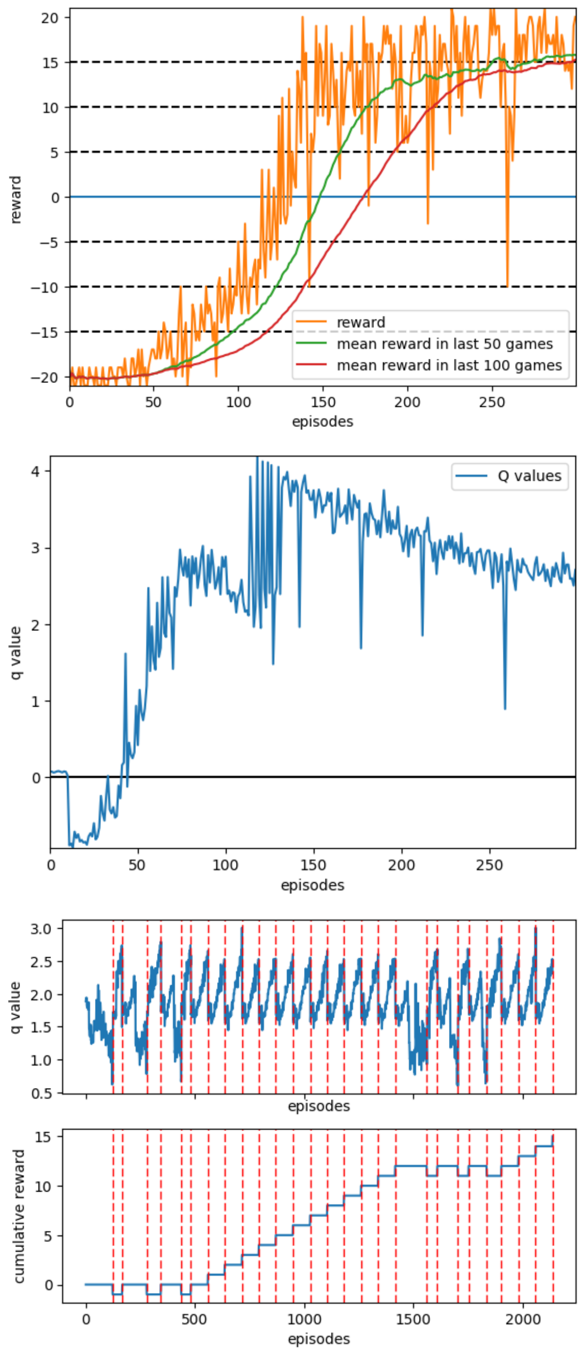
(a) DQN architecture

(b) Duel DQN architecture

Figure 1: Architecture implemented in this project

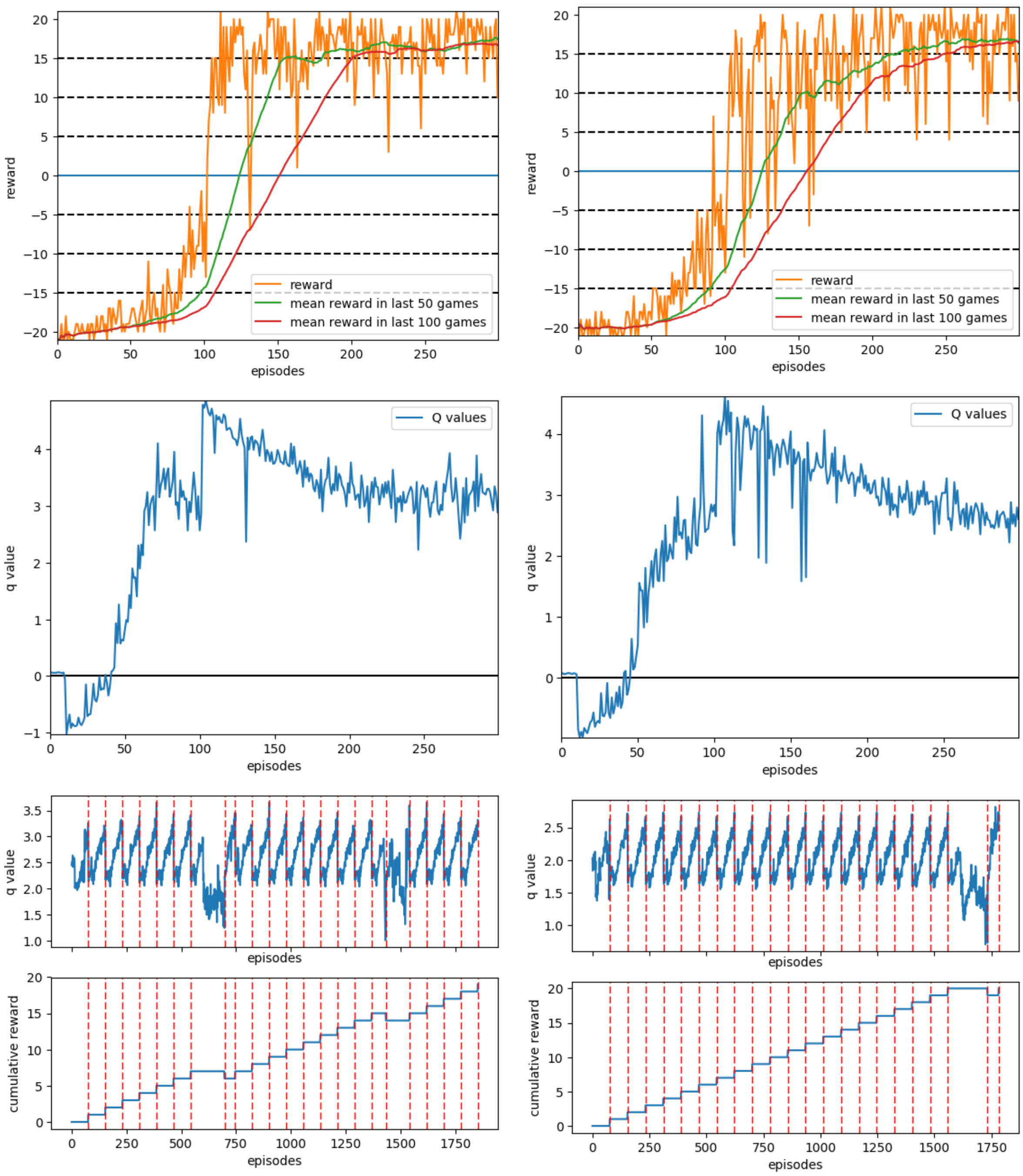


(a) DQN with target network results



(b) duel DQN with target network results

Figure 2: Experiments using simple memory buffer



(a) DQN with target network results

(b) prior duel DQN with target network results

Figure 3: Experiments using a prioritized memory buffer

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing atari with deep reinforcement learning*

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. *Human-level control through deep reinforcement learning.*

[3] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver *Prioritized Experience Replay*