Reinforcement Learning

Rainbow: Combining Improvements in Deep Reinforcement Learning

Authors:
Aurélien Renault
Ikhlass Yaya-Oyé
Victor Clermont

Professor :
Erwan Le Pennec

M2 Data Science

February 18, 2022

Contents

1 Introduction

Reinforcement learning is the sub-category of machine learning that has most intrigued AI researchers in recent years. Due to the great existing possibilities and the several areas of applications, reinforcement learning algorithms, methods and models have been elaborated and improved, especially with the emergence of deep learning models and neural networks which has resulted in what is called deep reinforcement learning.

The paper written by Hessel et al. [2018] we are going to talk about is named ”Rainbow: Combining Improvements in Deep Reinforcement Learning” and was released in 2017. The main objective of this paper is to briefly present a deep reinforcement learning algorithm, Deep Q network, describe some variations and extensions of this algorithm and finally, present a ”rainbow” algorithm which is the combination of all the presented extensions within one agent that outperforms all others.

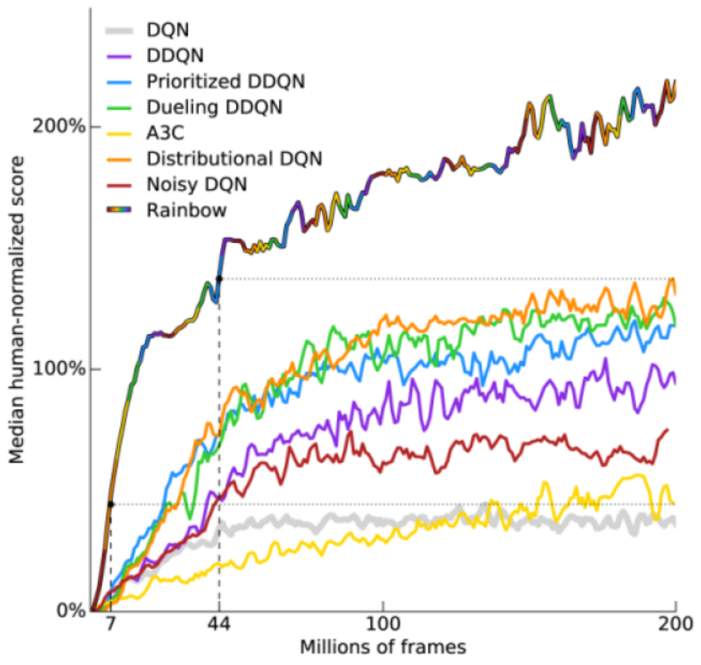


Image from Hessel et al. [2018]

We will introduce this report with a complete reinforcement learning state-of-the-art, then we will explain Q-Learning method and Deep Q network algorithm concept, talk about the limitations of these techniques and present the extensions that come to overcome these weaknesses. After that, we will describe the rainbow agent and mainly analyse and compare the results presented in the paper with our implementation results. Finally, we will end with an overview of DQN possibilities and applications.

2 Reinforcement and Q-Learning

2.1 Basis of Reinforcement Learning

Reinforcement Learning (RL) main idea is to learn which decision should be took at each specific time, according to the context at this time, in order to define the "best" sequence of choices for a dynamical context (Sutton et al. [1998]). It is mainly based on Markov Decision Process (MDP) which is a mathematical framework to model sequential decision making.
To deal with reinforcement learning algorithms and apply them we have to manage several components:

- The **agent** is the trained model inside the reinforcement learning algorithm. It takes decisions at each step of the process. Its objective is to maximize the reward.
- The **Reward** is a the decided metric / function to evaluate the performance of our agent. It can be positive or negative according to the agent decisions and the problem we are trying to solve.
- An **Action** is seen as a decision made by the agent. Each action made by an agent leads to a state.
- A **State** gives information on the situation at the step where the agent currently is and on all the possible actions the agent can make.
- Then the **environment** gathers all the components of the process. It is the real world containing all the possible actions and possible states where the agent evolves, and with which it interacts.
- Finally, we have the **Policy** which is what we want to determine and that can be defined as the strategy our agent is following to choose actions.

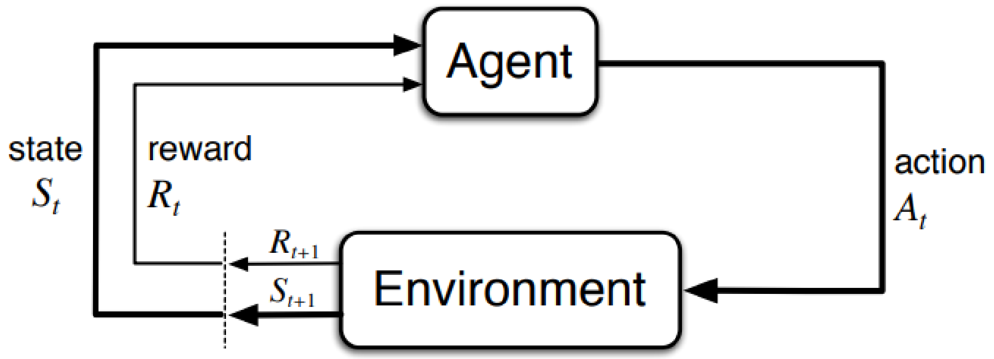


Figure 1: Reinforcement Learning process

RL problems, based on MDP, represent a succession of interactions between the agent and the environment : given a state the agent do an action, and given a state and the chosen action the environment reply by returning a new state (Figure 1). Each time the environment returns a new state, the current step is ended and a new one starts. The algorithm runs until the goal or the maximum number of iterations is reached.

To solve RL problems we can use different approaches that can be represented along two axes. First we have to define if our approach is based on the model or not.

Model-based approaches are applied when we know the environment's inner workings : for each current state we can define the reward and the next state following the chosen action. We often call them Planning solutions. The most often simple board games as tic-tac-toe or connect 4 can be solved with model-based solutions.
In contrast model-free solutions, considered as real RL solutions, are used for very complex environments where the inner dynamics are unknown. Then to solve the problem, the algorithm interacts with the environment and has to discover its behavior by trying : chose an action, see the results coming from this action and then repeats the process numerous times. As a human, the algorithm ends up deducing positive or negative conclusions from its experiences.

The second way to differentiate RL algorithms is according to what do we want to estimate at the output of our model. We have the Prediction problem where given a Policy as input, we want to estimate the Value function of this Policy. Then there is Control problem where we don't have any inputs and we just want to estimate the optimal Policy for a specific context.
In other words, in a Prediction problem we test a strategy and we want to estimate the function which will allow us to determine how good each possible state of our problem is. While in a Control Problem we want to find the best strategy; the strategy that will allow us to maximize our reward on mid, long-term according to our problem.

Most of real world problem that need RL solutions are Model-free Control problems, finding the best strategy without knowing the environment inner workings. The Q-Learning algorithm that we are going to study is able to solve these kind of problems.

2.2 Q-Learning algorithm

Q-learning is one of the most popular RL technique, and has been first introduce in Watkins and Dayan [1992]. It is part of the model-free and off-policy learning algorithm : it means that the evaluated and improved Policy is different than the Policy used by the agent to select an action, it allows to explore some others paths than the optimal one and learn from it. As a model-free algorithm, it does not use the reward system to learn, but rather, trial and error.
This learning technique is based on the Q function, also named action-value function which takes as parameters the state in which we are, s and the action a that we want to perform. This function returns the discounted sum of future rewards $G_t = \sum_{k=0}^{\infty} \gamma_t^k r_{t+k-1}$, ($\gamma$, being the discount factor, giving more importance to direct rewards rather than further ones) that the agent will collect by using our policy π. The Q function then is sort of modeling the "quality" of one action/state pair.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{1}$$

Temporal difference learning : One of the important elements to understand before jumping into Q-learning details is Temporal difference learning (TD). TD learning, on the contrary of Monte Carlo methods, performs updates differently, waiting only one time step to increment the value (or Q) function. In a TD(0) framework, for example, we are looking back at one step behind when updating the Q value. Furthermore, instead of one-step updates, one could use multi-steps updates, which we'll be interest in further on (Sutton et al. [1998]).

It's interesting to notice that TD(n), when n goes to infinity, is actually converging to the Monte-Carlo learning framework. Q-learning, then, is just applying TD learning to the Q function, using Bellman optimality, as follows.

$$Q^{new}(s_t, a_t) \longleftarrow Q^{old}(s_t, a_t) + \alpha(r_t + \gamma \max{}_{a'} Q(s_{t+1}, a') - Q^{old}(s_t, a_t)) \tag{2}$$

Note that SARSA (Rummery and Niranjan [1994]), named for State-Action-Reward-State-Action in Sutton et al. [1998] is the on-policy equivalent of Q-learning algorithm. The difference between off and on-policy using Q-learning and SARSA algorithm is the fact that in Q-learning (off-policy) we estimate a return for state-action pairs assuming we are following the best policy even if our algorithm is not following it, while in SARSA (on-policy) the return is estimated assuming the current policy is continuously followed.
Q-learning is usually faster to learn, you can introduce randomness with, for example, an ϵ-greedy algorithm to converge faster. While SARSA is most likely to learn slower but is, in a way, safer.

In practice, when we want to solve the policy optimization problem using the Q-learning algorithm, one create a so called Q table. The Q table is a matrix with number of states as rows and number of actions as columns and associate a quality value Q for each possible state-action pairs. Then, we're optimizing the policy by playing a bunch of games and going through the Q table, updating each Q value while the agent explore various solutions, taking multiple paths.
Even if Q-learning is a very good algorithm to learn a policy for several problem, there is a major inconvenient with it : the more possible states we have in our environment and the bigger will be our Q table, which will drastically increase time for learning. For example, a simple tic-tac-toe with a 3×3 grid has 19683 possible states and 5477 legal games states taking into account the rules of the games. Then use it on games like connect 4, chess or even shogi become more space-intensive and time-consuming. And for videos games like Pacman, Pong or simple Mario-like plateformers, Q-learning becomes totally inefficient as we can't define all possibles states in these kind of environment.

Hopefully RL has taken advantage of the breakthroughs and innovations brought by deep learning, resulting in the sub-branch of Deep Reinforcement Learning.

3 Deep Q-learning

3.1 Deep Q network

To solve the problem of large Q table we can replace it by a Q function doing the the same thing by taking a state-action pair and returning this quality value. Then the mission is to be able to estimate this Q function and here is the benefit of using deep learning. As Neural Networks are able to modelling complex mathematical functions, one can estimate our Q function with a neural network named Deep Q Network (Mnih et al. [2013]).
This Deep Q Network (DQN) trains itself and learns parameters using gradient descent and a loss function to minimize. If the training can determine efficient parameters, the model is able to return the best Q values for each possible actions in the specific state took as an input.

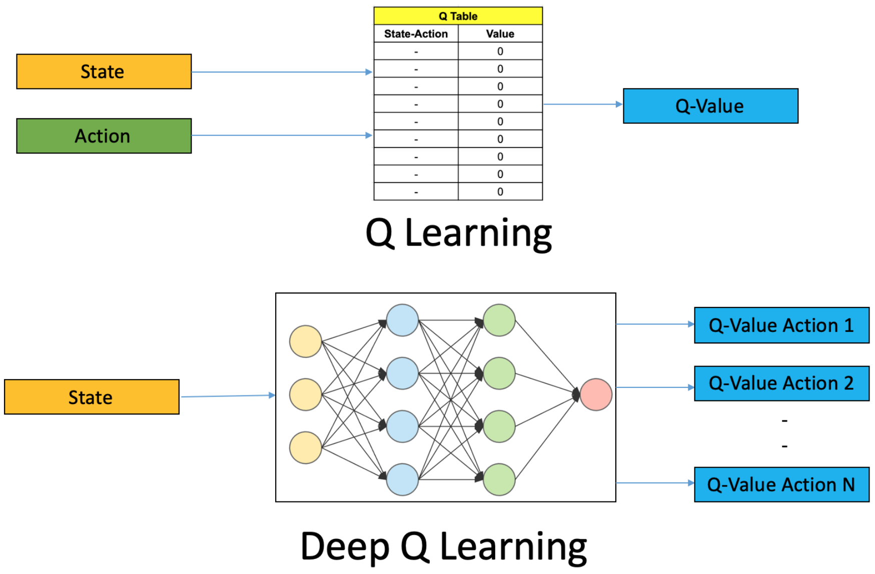


Figure 2: Q-Learning vs Deep Q-Learning schemes (*source*)

In DQN, the layers used in the neural networks depend on the shape of state data. If we can clearly define a step by numerical variables, we can build a multi-layers perceptron with fully connected layers. However in some cases, it is easier to represent state as an image and then use convolutional layers. If we take Pong game, our state can be an array with ball (x, y) coordinates and both player paddles y-coordinates, as well as a frame of the game.
While training a DQN architecture we have to deal with three components : Q network, Target network and Experience replay.

DQN Components

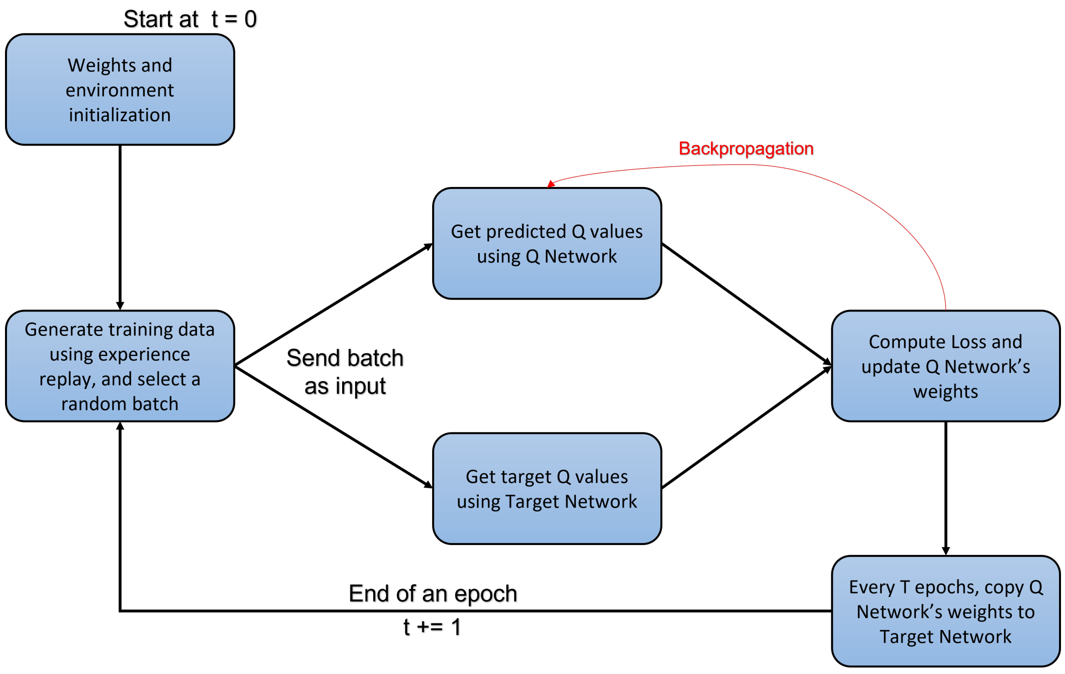
The Experience replay is a mechanism that interacts directly with the environment to generate a training sample which will be used by the Q network for training. It stores all the observations and actions made by the agent from the beginning (depending on the storage space allocated to the experience replay) and gives a shuffled random batch of samples to the Q network. If we give a list of successive actions to our Q network, we are not protected against the fact that the parameters' optimization is done only according to a specific case of our environment. So, experience replay is used to ensure the model's parameters allow it to generalize well and compute relevant Q values for all possible states.

Both networks, the Q and Target ones, have the same architecture but different objectives. The Q network takes and current state and a chosen action as input to predict the associate Q value for this pair, while the Target Network uses the next state as a result of the action performed in the current state and predicts the best Q value out of all possible action in the next state. In this DQN configuration, Q network is trained while Target Network is not. One could not use the Target Network and take the Q network to return predicted Q value and target Q value. However by doing this, when the network learns and its weights are updated to improve the predicted Q value, as the target and predicted Q value shared the same model, this will also change the Target Q value. All of this meaning try to reach a fluctuating value.

This is the reason why a not trainable Target Q network is used, ensuring stable target Q values. Instead, because the target Q values remains estimates and can be improved, it has been decided that after a specific number of steps during training, the weights of the Q network are copied on the Target Q network.

DQN algorithm

Using the Figure 3, we can describe how works the DQN workflow. First Q network's weights are randomly initialized and copied into Target Network.



Start at t = 0
Weights and environment initialization
Backpropagation
Get predicted Q values using Q Network
Generate training data using experience replay, and select a random batch
Send batch as input
Compute Loss and update Q Network's weights
Get target Q values using Target Network
End of an epoch
t += 1
Every T epochs, copy Q Network's weights to Target Network

Figure 3: Deep Q Network one epoch pipeline (*by authors*)
This process is done until the end of an episode. Then a new episode begins.

Then we create the training data using the experience replay. Taking the current state of the environment, we give it as input of Q network and obtain the predicted Q values for all actions. We select an action using ϵ-greedy method, experience replay executes this action and receives a reward and also the next state from the environment.
Each row of the training data contains a state s_t, the ϵ-greedy action a, the resulting reward r and the next state s_{t+1}, with t a time-step (Figure 4).

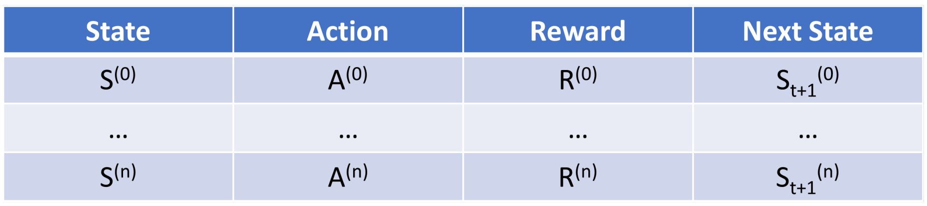
After that comes time to the computation of predicted and target Q value. A random batch of samples is drawn from training data and for each sample the following computations are done :

- The sample's state is given to the Q network which predicts Q values for all possible actions.
- The Q value of the sample's action is selected and defined as the Predicted Q value.

- The next state of the sample is given to the Target Network which sent us back Q values for all actions.

- The max output Q value is selected and added to the sample's reward to defined the Target Q value.

Note that in general, those steps are done on each row of training data in a parallel way using GPU.



State	Action	Reward	Next State
$S^{(0)}$	$A^{(0)}$	$R^{(0)}$	$S_{t+1}^{(0)}$
...	...	...	...
$S^{(n)}$	$A^{(n)}$	$R^{(n)}$	$S_{t+1}^{(n)}$

Figure 4: Visualisation of of a random batch of size n (*by authors*)

Loss

Now we have to compute the loss of our model. To do so, we use a function, here the mean squared error, and compute the loss between the Target Q value and the predicted one for each training data's sample :

$$\begin{aligned}\mathcal{L} &= MSE(Q^{(i)},\ \hat{Q}^{(i)}) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(q^{(i)} - \hat{q}^{(i)}\right)^2\end{aligned} \tag{3}$$

with q_i and $\hat{q}_i$ the target and predicted Q value of the *i-th* sample, and n the number of sample in the randomly sampled batch. Using all the terms' definition above, we can define the loss for a sample i as :

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(R^{(i)} + \gamma \max{}_{a'} Q(S_{t+1}^{(i)}, a') - Q(S_t^{(i)}, A^{(i)})\right)^2 \tag{4}$$

with $S_t^{(i)}$ the current state, $S_{t+1}^{(i)}$ the next state, $R^{(i)}$ the reward and $A^{(i)}$ the chosen action of the *i-th* sample.

Using the obtain loss value, we can back-propagate it and update the Q Networks's parameters θ. This conclude one epoch of training and it is repeated a several number of times. To finish, during the training every T epochs the Q Networks' weights are copied on Target Q Network to readjust all targets Q values.

3.2 Limitations of DQN

- **Overestimation bias** : One of the issues discovered using Q-learning algorithm is linked to the overestimation of some action values, which consequently led to poor performance. This maximization bias is introduced since Q-learning algorithm is using the maximum action value to approximate the maximum expected action value. It's unclear if overoptimism is a problem in itself, indeed, if all values were uniformly relatively higher, then we wouldn't expect the resulting policy to perform worse. Though, if the overestimations are not uniform, they actually might affect the policy badly (Thrun and Schwartz [1993]). Even though the introduction of deep neural networks, which should have, theoretically, reduce the issue, providing a potential low estimation error, it has been shown that even in some Deep Q network framework, the overestimation bias would still occur (Van Hasselt et al. [2016]). Double Q-learning targets more specifically this problem.

- **Experience replay memory is sampled uniformly** : In practice, when using replay memory, the experiences are sampled uniformly at random during updates. Thus, it gave the exact same importance to all transitions, not identifying the transitions from which there is much to learn (Mnih et al. [2015]). Hence, this motivate the use of a more sophisticated way to sample experiences, giving more attention to the important transitions.

- **Slowness of learning** : In practice, DQN architectures, working on a single-machine set up, are slow to train due to the number of transitions needed to learn a decent policy alongside the online interaction with the environment (robotics). For example, first vanilla DQN version took around 14 days to train on a single Atari game (Mnih et al. [2015]). However, deep learning offers us the possibility to parallelize the computations and many distributed RL techniques have been successfully proposed, creating multiple agents interacting with different instances of the same environment (Nair et al. [2015]).

4 DQN variants

4.1 DQN extensions used to build Rainbow

Double Q-learning

The basic idea behind DDQN, as stated in his first tabular setting (Hasselt [2010]) is to use two separated Q value estimators, each of which is used to update the other. Using two independent Q value estimators is supposed to unbiased the estimations of action values by calling the opposite estimator. The proposed DDQN algorithm which is used in the paper is then an DQN udapted version, which is generalizing the concept to arbitrary function approximation, including deep neural networks (Van Hasselt et al. [2016]). In this updated version of the DDQN, we are estimating two different value function with two set of weights θ and θ'. Then for each update, one set of weights is used to determine the greedy policy and the other to determine its value. Thus, the target of Q-learning can be rewritten as

$$Y_t^{DDQN} = R_{t+1} + \gamma_{t+1} Q_{\bar{\theta}}(S_{t+1}, \text{argmax}_a Q_\theta(S_{t+1}, a)) \tag{5}$$

with $\bar{\theta}$ being the weights of the target network and θ the ones from the primary network. Then, we are finally minimizing the following loss

$$\mathcal{L} = (Y_t^{DDQN} - Q_\theta(S_t, A_t))^2 \tag{6}$$

This change has prove to reduce the maximization bias effect present in DQN alongside improving global performances.
Since the publication of the paper, others DDQN alternatives has appeared, improving the state-of-the-art a little (Fujimoto et al. [2018]).

Prioritized replay

Prioritizing transitions samples from which we learn rather than using an uniform sampling has prove to improve experience replay, making it more efficient (Schaul et al. [2015]). Practically, the transitions are sampled using the absolute *TD error* of one transition, which is supposed to act as a decent proxy for the expected learning progress. Finally, transitions at time t, are sampled with probability p_t defined as

$$p_t = |R_t + \gamma_t \text{max}_a Q_{\bar{\theta}}(S_t, a) - Q_\theta(S_{t-1}, A_{t-1})|^\alpha \tag{7}$$

with α being one hyper-parameter determining the shape of the distribution. Some variants remains possible such as stochastic prioritization or importance sampling, which in some cases could be efficient as well (Schaul et al. [2015]).

Dueling networks

To fully understand the idea behind dueling networks, we first need to introduce the Advantage function, relating the value and Q functions as follows (Baird III [1993]):

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{8}$$

The Q function, on one hand, models the quality of one given action/state pair, the value function, on the other hand, quantify how good it is to be in state s. Thus, subtracting those two quantities returns our advantage function, which, in some way, is representing the relative importance of one action.
This is actually the main idea of dueling networks, using this decomposition, we split the original single DQN stream which returns the estimated Q values into two separate streams, one returning the estimated value function while the other is returning the advantage function estimation (see Figure 5).

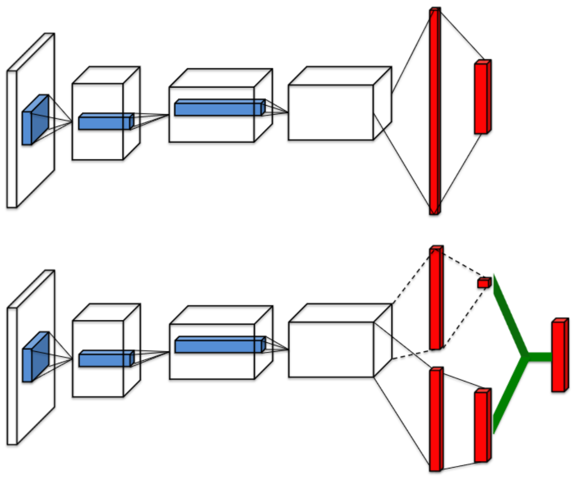


Figure 5: Classic DQN architecture (top) and dueling architecture (bottom) from (Wang et al. [2016])

In practice, this kind of dueling architecture is able to learn which states are valuable to be in, without necessarily computing each action for each state. More specifically, one algorithm using this architecture will not waste resources trying to learn from a state in which none of the possible actions can affect the environment in a relevant way. Thus, we are tempted to plug (8) directly to our model, replacing Q function by

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a, \theta, \beta) \tag{9}$$

α and β being respectively the parameters of the advantage stream and of the value stream, θ being the shared set of (usually convolutionnal) layers. However (9) suffers form an identifiability problem, which make it not convenient to use in practice. To target this issue, one can force the advantage function estimator to have zero advantage at the chosen action (Wang et al. [2016])

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + (A(s, a, \theta, \alpha) - \max_{a'} A(s, a', \theta, \alpha)) \tag{10}$$

One alternative way, which stabilize the optimization and which has chosen to be used in Hessel et al. [2018] consist in replacing the max by the average function

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + (A(s, a, \theta, \alpha) - \frac{1}{N_{actions}} \sum_{a'} A(s, a', \theta, \alpha)) \tag{11}$$

One can note that, the dueling architecture is sharing the same input/output structure as classic DQN, thus, we can use the same learning algorithms to train our dueling networks.

Multi-step learning

In previous sections, we presented the Q-learning as a one step Q algorithm. The TD method, however, can be generalized, bootstrapping over longer time intervals. It has been show that, in many cases, it is possible to reach better performances and accelerate the learning by wisely choosing your backup length parameter n (Sutton et al. [1998]). This alternative n-step DQN target is written

$$Y_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k-1} + \gamma_t^{(n)} \max_{a'} Q_{\bar{\theta}}(S_{t+n}, a') \tag{12}$$

Distributional RL (Categorical DQN)

In distributional RL, the basic idea is to learn the probability distribution Z of the returns rather than the expected returns. In some complicated environments, Q values can be stochastic and thus, simply learn the expectation will not capture enough information, such as variance for example, to choose best decisions.

Distributional Bellman operator. To evaluate the value distribution Z, we can apply recursively the Bellman operator $\mathcal{T}$. Let us define the transition operator P^π as well as the distributional Bellman operator $\mathcal{T}^\pi$

$$\begin{cases} P^\pi Z(s, a) \overset{D}{:=} Z(S', A'), \quad S' \sim P(.|s, a), \quad A' \sim \pi(.|S') \\ \mathcal{T}^\pi Z(s, a) \overset{D}{:=} \mathcal{R}(s, a) + \gamma P^\pi Z(s', a') \end{cases} \tag{13}$$

Parametric distribution. In categorical DQN, the distributions are supported over a finite support of *atoms* $z = \{z_1, z_2, ..., z_N\}$ taking values in $[V_{min}, V_{max}]$, N actually corresponds to the number of discrete bins, with $z_i = V_{min} + i\frac{(V_{min}-V_{max})}{N-1}$. Then, in a DQN architecture, one should have N output neurons for each action in order to encode the distributions of the returns. A *softmax* function can be applied over each action dimension to get the probabilities $\{p_i(s,a)\}_{1\leq i\leq N}$ of each bin z_i, the distribution Z is then the sum over the atoms of all probabilities (with δ_{z_i} the Dirac distribution)

$$Z_\theta(s,a) = \sum_{i=1}^{N} p_i(s,a,\theta)\delta_{z_i} \tag{14}$$

These probabilities being properly estimate, one can easily recover the Q value as the mean of the distribution and use it for action selection as in regular DQN

$$Q_\theta(s,a) = \mathbb{E}[Z_\theta(s,a)] \tag{15}$$

Projected Bellman update. The Bellman update for each atom i can be easily computed (see figure 6)

$$\mathcal{T}z_i = r + \gamma z_i \tag{16}$$

Finally we have to minimize the distance between the target distribution $\Phi\mathcal{T}Z_\theta(s,a)$ and $Z_\theta(s,a)$ (Φ is the the L2-projection of the Bellman update onto the support z), using the KL divergence.

$$\mathcal{L} = D_{KL}(\Phi\mathcal{T}Z_{\bar{\theta}}(s,a)||Z_\theta(s,a)) \tag{17}$$

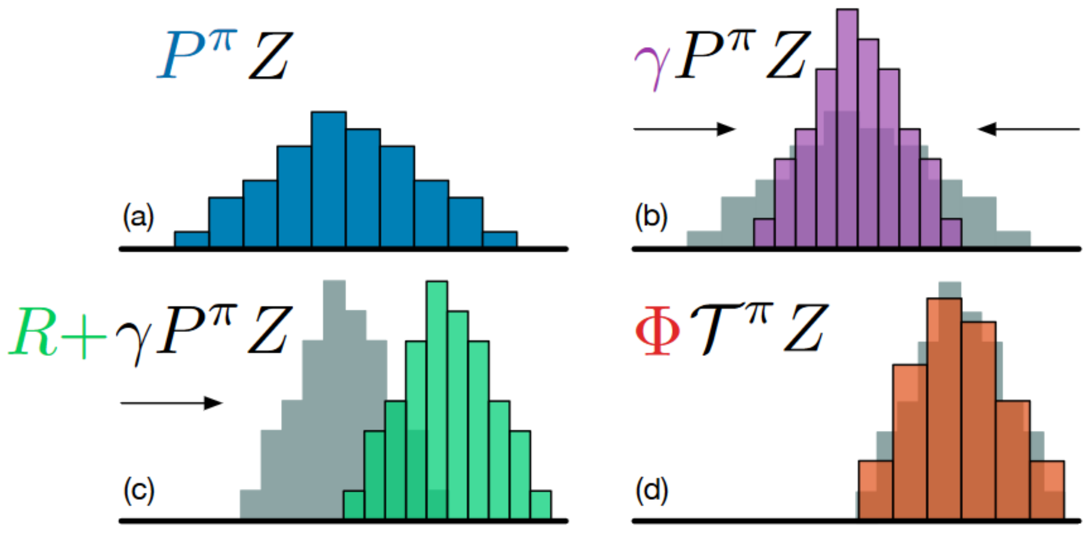


Figure 6: (a) Transition to next state distribution, (b) Discounting/Shrinking, (c) Reward shifting, (d) Fitting and projection from Bellemare et al. [2017]

In this part, we were only interested in the categorical DQN, which represents only one instance of distributional RL, as we have many other options in order to visualize probability distributions, Quantile Regression Network (QR-DQN) as well as Implicit Quantile Network (IQN) are other instances of distributional RL.

Noisy Nets

The ϵ-greedy policies may, in some cases, be inefficient, especially when a relatively high number of action is needed to obtain the first reward for example. Fortunato et al. [2017] proposed the so called NoisyNet, which basically introduce parametric noise to the deep neural network weights in order to promote a more effective exploration, often resulting in a better policy. In practice, we are replacing the standard linear transformation formula $y = Wx + b$ by

$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b \tag{18}$$

with $\zeta = (\mu, \Sigma)$ a set learnable parameters, while ϵ^w, ϵ^b are zero-mean noise random variables with fixed statistics and $\odot$ denoting the element-wise product. The variances Σ of perturbations can be seen as the energy given to the injected noise ; in other words, the model is automatically tuning the amount of noise during the optimisation. Hence, we do not require any hyper-parameter tuning, on the contrary of using a standard ϵ-greedy method. Many other exploration techniques were proposed such has Bootstrapped DQN or count-based exploration which could as well perform well (Osband et al. [2016], Ostrovski et al. [2017], Bellemare et al. [2016]).

4.2 Others extensions

Averaged-DQN

In order to improve stability and efficiency, the Averaged-DQN proposed to use the K previous Q values estimates and averaging them to get our Q value estimation.

$$Q_i(s, a) = \frac{1}{K} \sum_{k=1}^{K} Q(s, a, \theta_{i-k}) \tag{19}$$

This method addresses the previously discussed issue of overestimation bias, which happens to occur when the variance of the Target Approximation Error (TAE) is high. The Averaged-DQN has prove to significantly reduce this variance, leading to better policies, in the same way as a DDQN algorithm could do (Anschel et al. [2017]). This algorithm, however, is not an ideal combination with the dueling networks extension, inducing $2K$-fold more forward passes through the Q network, which, when it comes to computational effort led us to prefer a DDQN framework which are relatively performance equivalent.

Episodic control

In a near deterministic environment, Episodic control proposes to improve data efficiency by recording experiences in which the agent received high returns. Hence, if the agent find itself in the same state at some point during training, it will attempt to recreate the past successes based on the previous successful sequences and potentially improve the learned policy faster. Even though it's seems unlikely to re-encounter twice the exact same state in real-life problems, it's actually the case in the Atari environment, in which one agent revisits one state it has already been in, between 10-60% of the time (Blundell et al. [2016]). In practice, Episodic control is simply a growing table indexed by states and action, where each entry contain the highest return of action a in state s. We update the episode memory as follows

$$Q^{EC(s_t,a_t)} = \begin{cases} R_t & \text{if } (s_t, a_t) \notin Q^{EC}, \\ \max(Q^{EC}(s_t, a_t), R_t) & \text{otherwise} \end{cases} \tag{20}$$

To limit the memory used, the maximum table length is finite and the least recently updated entries are forgotten while the agent explores new states.
In large scale problem, new states are frequent, thus, Episodic controller estimate the Q^{EC} by averaging it's k nearest neighbors based on some distance metric. Thus, if s_t is one new state

$$\hat{Q}^{EC}(s_t, a_t) = \frac{1}{k} \sum_{i=1}^{k} Q(s_t^i, a_t) \tag{21}$$

where $\{s^i\}_{1 \leq i \leq k}$ are the k nearest states.
More recently, one can also mention the Episodic Memory Deep Q network (EMDQN), which is inspired by the human brain mechanism, which use both striatum and hippocampus respectively for reflex and memory to learn better, quicker (Lin et al. [2018]). This combinations of Episodic control and DQN is supposed to be more flexible : one can adjust a hyper-parameter whichever of memory module or general system decision are required.

Hierchachical RL

The main idea behind Hierarchical RL is to achieve the goal of the task not using only one policy, but using many sub-policies working together in a hierarchical way. In this way, one agent can learn simultaneously at various resolution both in spsace and time. This learning structure can improve data efficiency, with low-level policies hiding irrelevant information to higher-level policies. Hierarchical RL is also combining quite smoothly with transfer learning : trained low-level policies could be re-used to complete some other tasks.
In practice, there are many different approaches to implement Hierarchical RL, one successful example of combination of HRL and DQN is the FeUdal Network, inspired primarily by Feudan RL, in which high-level managers learn how to formulate tasks to their sub-managers who, in turn, learn how to complete them (Dayan and Hinton [1992]). In FeUdal Networks, the Manager Neural Network is taking one stat/reward pair as input and return an embedding of the sub-goal to be achieve by the Worker ; the Worker NN is then trying to complete the task assign to him by its manager (Vezhnevets et al. [2017]).

5 Rainbow Integrated agent

The *Rainbow* agent is eventually combining all the previously mentioned components, the n-step target distribution can be written : $d_t^{(n)} = \mathcal{R}_t^{(n)} + \gamma^{(n)} Z$, we have then the next loss we want to minimize

$$\mathcal{L} = D_{KL}(\Phi_z d_t^{(n)} || d_t) \tag{22}$$

Considering prioritize replay, *Rainbow* is no longer using the absolute *TD error* (even though it could have been computed, using the mean action values) but the KL loss to prioritize transitions. Thus, at time t, transitions are sampled with probability p_t

$$p_t = (D_{KL}(\Phi_z d_t^{(n)} || d_t))^{\alpha} \tag{23}$$

This choice of KL loss might actually reveal itself more robust, especially considering very noisy environments. In order to get the probability $p_i(s,a)$ for each *atom* z_i, dealing with a dueling architecture, is finally resulting in applying a $softmax$ function over the aggregated stream, containing both value and advantage streams, as in regular DQN.

$$p_\theta^i(s,a) = softmax\left(V^i(s,\theta,\beta) + \left[A^i(s,a,\theta,\alpha) - \frac{1}{N_{actions}}\sum_{a'} A^i(s,a',\theta,\alpha)\right]\right) \tag{24}$$

with θ the parameters of the shared representation and α, β respectively the parameters of the advantage and value streams.
Last thing to do is then to introduce the noisy layers, with a perturbation following a factorised Gaussian distribution, in order to limit the number of independent random variables. Considering a linear layer with p inputs and q outputs, factorised method is allowing to have $p+q$ units of Gaussian variables while simply using independent Gaussian noise, we would have $pq+q$ variables (Fortunato et al. [2017]).

6 Empirical results

6.1 Experimental set-up

6.1.1 Atari 2600 : a good option for reinforcement learning

When we want to test a RL agent, a widely used by the scientific community way to do is test the agent on Atar 2600 platform games.
Atari 2600 is a video game console, released in 1977, offering at its launch more than 500 games, and which continues to be developed today. This is the console that popularized the use of CPUs in game consoles. The game screen is 160 pixels by 210 pixels with 128 colors. The games are played with a joystick made up of 18 “actions” and a button. The game returns 60 frames per second, To define the action A_t, at time t, is to recover the last few frames (for example we can take 4, in the case of game ”pong”) in a way a be able to analyze the movement of the game and adapt its At action accordingly.
Thus, the Atari 2600 games are not simple basic games, but are endowed with a certain complexity thanks to the game environment and the multiple possible actions. However, this complexity remains affordable, and represents a very good balance to be able to perform modeling and learning, as we wish.

6.1.2 Evaluation metrics

Compare two agents between them is not an easy task. Different games have different point systems. Each time the agent aims to maximize a score, but this score can turn out to be very different between several games. There are a few metrics that allow to successfully compare algorithms.

- **Normalized Scores :** We consider the score the algorithm i at the game g, given by $s_{g,i}$. And so our goal is to compare S_1=[$s_{g_1,1}$,...,$s_{g_n,1}$] and S_2=[$s_{g_1,2}$,...,$s_{g_n,2}$]. To do that we can normalized $s_{g,i}$ by $z_{g,i}$, and then we could compare the normalized score across the different game. And ideally, we will have $z_{g,i}$=$z_{g',i}$ for different game g and g'.
  So to normalized the score, we use the notion of score range [$r_{g,min}$;$r_{g,max}$] on each game. And then we normalized $s_{g,i}$ with the following operation : $z_{g,i}$=($s_{g,i}$-$r_{g,min}$) / ($r_{g,max}$-$r_{g,min}$)

  - **Normalization to a reference score :** A simple way to normalized the score is to take a a reference score done by a random agent along the different games.
    But with this straightforward approach, some issues appear. Indeed, the scale of normalized scores could be very large and normalized scores are generally not translation invariant.
  - **Normalization to a baseline set :** Previously, we normalized the score by a single reference, here, instead we will normalize by a set of reference. Let $b_{g,1}$,...,$b_{g,k}$ be a set of reference score. A method's baseline score is computed using the score range [min_i $b_{g,i}$; max_i $b_{g,i}$]
    If the set of reference is enough rich, this normalization allows to reduce the score for the most game, and to gives comparable quantities of performance.
  - **Inter-Algorithm Normalization :** A third way to normalize the score is to use the score of the algorithms. For example, we take n algorithms, each one performing a score $s_{g,i}$ on the game g. The inter-algorithm score is defined by using the score range [min_i $s_{g,i}$; max_i $s_{g,i}$].
    As the inter-algorithm scores are bounded, this normalization is interesting to compare relatively different algorithms, and see which one is the best compare to the other. But the drawback with this approach, is that we can't know the absolute performance of an algorithm. We can know that algorithm A is better than B, but not how far it is better.

- **Aggregating Scores :** Once a normalized score for each game has been obtained, the next step is to generate a metric that reflects how each agent performed on that set of games. We now describe 3 ways to aggregate normalized scores.

  - **Average Score :** The easiest way to aggregate normalized scores is to calculate their average. However, without perfect score normalization, average scores tend to be heavily influenced by games with higher base scores (for example "Zaxxon"). The average of the inter-algorithm scores avoids this problem because all the scores are between 0 and 1.
  - **Median Score :** The median score is generally more robust to outliers than the mean score. The median is obtained by sorting all normalized scores and selecting the middle item (if the number of scores is even, the average of the two middle items is used).
  - **Score Distribution :** The aggregation of the score distribution is a natural generalization of the median score: it shows that the algorithm achieves a certain normalized score or a better game score. It is essentially a quantile plot or an inverse empirical

CDF. Unlike mean and median scores, score distributions accurately represent agent performance, regardless of how individual scores are distributed.

6.2 Our DQNs implementation

To build our different DQN extensions, we inspired from this github repository that presents all extensions presented in Rainbow paper. Our implementations have been done in Python, using the PyTorch library (Paszke et al. [2019]) to deal with the networks architecture and agent's training.

Configuration

As we do not have any computer with the needed resources to deal with RL problems solving, we manage to set up our work on Google Colab notebooks. By this way, we had access to free graphics and central processing unit and manage to train our agent in some environment, dealing with Google Colab resource limitations.

Classes

Each DQN implementation is based on three main class that are then modified according to the added extension : Replay Buffer, Network and Agent.
The first two classes are the Replay Buffer and the Network. Replay buffer contains a specific number of element sets from where we will draw training samples, while the Network class defines the neural network architecture used to approximate the Q function.
An instance of both classes are then used as an attribute of the last one which is the Agent itself.
Here is a brief explanation about these three classes and their main attributes and methods :

- **Replay Buffer :** This class has 5 numpy array as main attributes to store current observations, chosen actions, resulting rewards, next observations and the environment state. The numpy arrays storing actions, rewards and environment states are 1-dimensional with a size equals to the defined memory max size, and respectively store integers, floats and booleans. The arrays storing current and next observations are 2-dimensional in order to save an array of observation which is itself encoded as an array : in CartPole for example, an observation is an array with four float values that are cart position, cart velocity, pole angular and pole angular velocity.
  About the method, Replay Buffer class has a method to add a new set of element and a method to sample elements from the memory storage.
  Note that we renamed what we previously called 'state' as 'observation' to avoid misunderstanding with 'environment state', which tell us if the game is ended or not.

- **Network :** This class inherits from Torch Module class. It contains the layers of the network as attribute. The basis architecture is two dense layers with 128 neurons, each of them followed by a ReLU function, and the final dense layer with a number of neurons equals to

the number of possible actions.
Then we just had to overwrite the forward method. This function allows us to compute the Q value for each action according to the observation given as input.
In some DQN extensions, the network architecture may change as for the Noisy DQN where we use Noisy linear layer or in categorical DQN where the distance is computed in the forward method.

- **Agent :** Finally we have the main class, the agent. It contains the environment, the replay buffer, Q and target Q network as main attributes. Then, it has also several "informative" attributes as the discount future reward γ, the epsilon value with its bounds, or even the number of steps after which we update the target Q network weights.
As the agent has the train method, it is also implemented with several helpful methods as select an action, interact with environment by doing an action, compute Q network loss, update Q network using back-propagation and copy Q network's weight into tartet Q network. For the extensions were the loss is not altered, we defined it as the smooth L_1 loss. We also have one method to plot the scores and the losses through time-steps and a method to test the agent during one episode without training.

Smooth L1 Loss

This loss is defined as a squared term if the absolute element-wise error falls below a parameter λ, and a L_1 term otherwise. So with $X = (x_1, ..., x_n)$ a batch of size n containing element's features and $Y = (y_1, ..., y_n)$ the labels of the elements in this batch, we have $\ell(X, Y) = l_1, ..., l_n$ with :

$$l_i = \begin{cases} 0.5(x_i - y_i)^2 * \frac{1}{\lambda}, & \text{if } |x_i - y_i| < \lambda \\ |x_i - y_i| - 0.5 * \lambda, & \text{otherwise} \end{cases} \tag{25}$$

Finally, we compute the loss value as the mean of all element loss l :

$$\mathcal{L} = \text{mean}(\ell(X, Y)) \tag{26}$$

The benefit of this loss are the facts that it is less sensitive to outliers than a mean squared error loss, and it prevents from exploding gradients (Girshick [2015]).

Environment and hyperparameters' value

To the setup of the environment where our agent evolves we used the OpenAi Gym library giving access to several environments and Atari games. We first wanted to train each DQN extension on Pong games to compare their score and loss curve and then make some match between the extensions to see how they are doing at equal training time and hyperparameters value. However, we figured out that it takes about 1 million steps for a DQN to converge to the Pong solution. As the available resources does not allow us to train as much, we finally decide to test all the extensions on a simpler case which is the CartPole test.

As described by Gym developers, we have a pole attached by an un-actuated joint to a cart. This cart is moving along a friction-less track and the goal is to prevent the pole from falling. We took the 'CartPole-v1' version as environment for our implementation.

About the hyperparameters of each DQN agent, here are the values for the hyperparameters shared by all the extensions :

- **Number of episodes :** Each agent training has been done on **100** episodes.
- **Memory size :** The maximum size of the memory was **1000** set of elements.
- **Target update :** We copy the weights of Q Network into Target Q Network every **500** frames.
- **Batch size :** At each training step, we draw **256** samples from the memory.
- **Discount future reward :** The γ parameter value was set at **0.99**.
- **Learning rate :** We use the Adam optimizer and set the learning rate to **0.001**.
- **Epsilon values :** For agent working with ϵ, the max, min and decay value are respectively **1**, **0.1** and **0.001**.

For the prioritized experience replay extension, we set the parameter α which determines how much we use prioritization to **0.2** and β which determines how much importance sampling is used to **0.6**.

About the categorical DQN, we define the support V_{min} and V_{max} value as **0** and **200** and the number of $atoms = \mathbf{50}$.

Finally, in the multi-steps learning extension, we fixed the n-steps value calculating n-step temporal difference error to **3**.

6.3 Results analysis and comparison to Rainbow

As we said, we train 8 agents according to the basic DQN, the six extensions and the rainbow agent on 'CartPole-v1' during 100 episodes. At the end of each training episode we stored in a list the final score and the episode loss by storing loss at the end of each frame and averaging all the frame loss of one episode. Here we have on the Figure 7 the score curve through training for each agent.

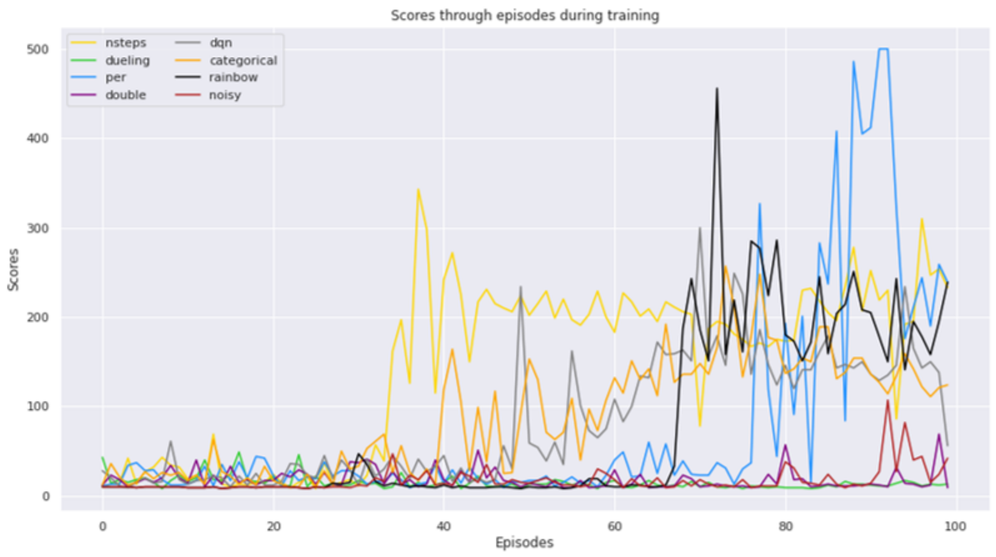


Figure 7: Curves of agent scores on each training episodes (*by authors*).

From this plot we can see a lot of variations that make it difficult to read. Then to alleviate this problem we compute for each line a n-smoothed curve. To do so we create a function to use a slicing window that takes n values of the curves and compute the mean of those n values before stepping forward by 1. We then obtained the Figure 8 which is far more readable than the previous plot by using a n-smooth function with $n = 4$.

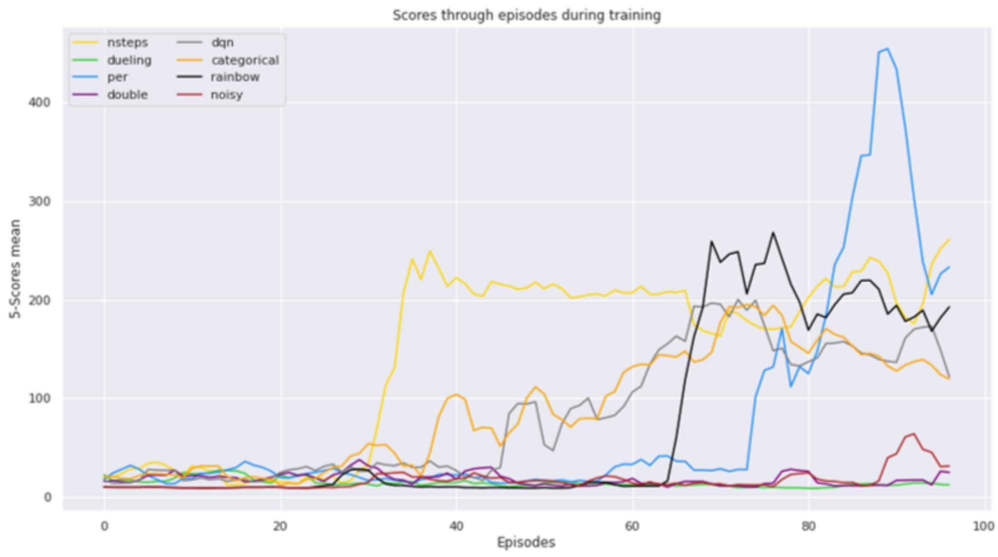


Figure 8: Smoothed curves of agent scores on each training episodes (*by authors*).

Looking at the plot we can see our results are very different from those presented in Rainbow paper (Hessel et al. [2018]). Our test show us that the algorithms that converge faster are Prioritized, Multi Step Learning and Rainbow DQN agents while the best algorithm in the paper is the Rainbow, far ahead, followed by Categorical, Dueling and Prioritzed DQN.
We confirm the plot analysis by using the CartPole test defined by OpenAi : we consider an agent

masters the CartPole test if it reach a minimal average score of 195.6 over 100 episodes. We tested each of our trained agents and mentioned the results in table 1. We can see from the table that only the prioritized and the Multi Steps DQN have passed the test, while double, dueling and noisy have very poor results.

Agent	Average Score	Video test Score
DQN	136.15	115
DDQN	10.1	12
Prioritized DQN	**200.16**	219
Dueling DDQN	12.55	12
Noisy DQN	9.56	9
Categorical DQN	121.33	124
Multi Steps	**196.11**	203
Rainbow	164.12	152

Table 1: Average score over 100 episodes and resulting score of the saved test episodes (link to videos)

From all of this, we can conclude by saying that our results are very different from the paper, however there are several reasons that can explain this conclusion. And all of theses reasons are related to a major difference between authors and us : computational resources.

As we said, our best machine to do costly operations was a Google Colab notebook, while nowadays RL algorithms are trained on servers using several processes. Then to deal with this issue we made several changes that may have an impact on our final results. Among those changes, there is the environment where agents evolve. The paper results have been reached by training on Atari games whereas our environment was CartPole.
Another change, which is a consequence of the first one, is that in CartPole an observation is a four values array where Atari games observations are mostly images (3D array, with height, width and channels). Then authors might have used convolutional operation in their network architecture while we only used Dense layer.
The last modification we did to deal with resources management is about a specific hyperparameters, the number of frames : in their work, authors trained algorithm on hundreds of millions of frame to see algorithm converges at most. To give an example, we trained each agent on 100 CartPole episodes which approximately correspond to 9k frames. Train agent on 50 millions of frame, which is the point where Rainbow outperforms others extensions maximum performance, represent around 625k episodes on CartPole. Unfortunately, this is not a thing we had the opportunity to test with the available resources.

6.3.1 Ablation studies

Even if we didn't obtain similar results with paper due to the mentioned reasons, we can nevertheless make an interesting observation on our results and the ablation agent trained by authors.

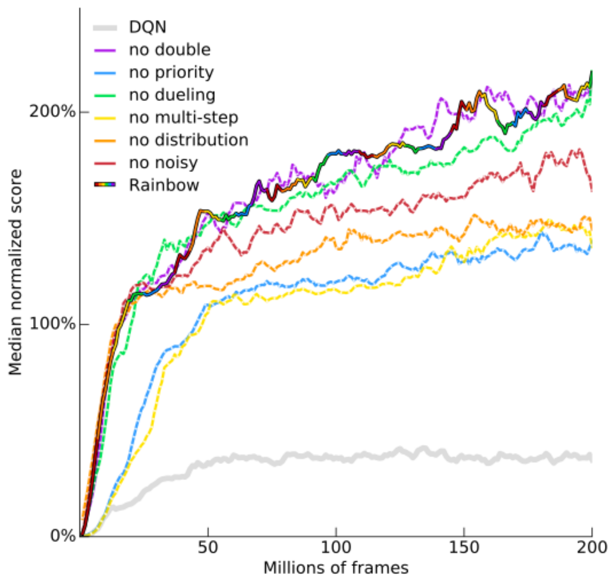


Figure 9: Median human-normalized performance across 57 Atari games, as a function of time. From Hessel et al. [2018]

Ablation agents are Rainbow agent for which authors removed only one extension. The goal behind this is to have a look on which extensions impact more than others. By analysing the Figure 9, we can see that the worst performing ablation agents are the Rainbow agent without priority and the one without multi-step. And, by making the connection with our implementation, those two extensions are the extensions that worked best and the only two having passed the CartPole test during test.
This relation between our implementation results and the ablation study led by authors can allow us to ask us some questions that can represent new leads to explore, like the following : Are the extensions that allow an agent to learn quickly at first the ones that contribute the most to its training ?

7 A concise overview of applied DQN

Paying board and Atari games is good, but what about RL when it comes to real-life problems? Is it good, effective? In this section, we will be interested in exploring a non exhaustive list of examples of DQN algorithms use cases to give a glimpse of what deep RL could actually be used for in real-life application. We'll review some instances from the financial industry as well as for robotics or energetic domain.

Robotics

Robotics is one natural application area for RL. Some papers already explore the possibility to apply DQN architectures to train 3D simulation of mechanical harm, so that it can operate objects manipulation for example. In Popov et al. [2017], an agent is trained in order to grab a Lego block and then stack it on another. Experiments show however, that there are many cases where the agent learn bad policies, such as grabbing the block in a way you can't stack it after for instance. This highlight the fact that, in such complex 3D environments, reward function is hard to design. In order to force the agent to learn good policies, one could make the reward sparse, but it make the optimisation more difficult at the same time.

Smart Grid / Energy management

Ecological matters, energetic transition are subjects that are getting more and more attention these days. RL applied to energy management is as well a great research topic currently. Mocanu et al. [2018] proposed to use DQN to create energy scheduling to maximize buildings power efficiency. In this context, one aims to minimize load peaks, and thus, minimize the cost of energy. The state space represents the building energy consumption as well as the price of electricity at time t ; the action space relies on electric devices. This method has show to perform very well at different scales (one building or an aggregation) and could actually propose real-time feedbacks to consumers to encourage a more efficient use of their electricity. Those type of problems are perfect examples in which RL/deep RL approaches work very well and for which we likely see future developments.

Trading decision using DQN

Financial markets provide us with great stochastic environment, as well, financial trading is far from following a deterministic policy. Those are the reasons why RL/Deep RL applications on this field are study and interest a lot. One relatively recent example of DQN application in financial industry consist in making trading decisions based on some DQN algorithm. In this context, the state space $\{S\}_{1\leq t\leq T}$ represents the returns of one given stock, while the action space simply contains three possible actions $A = \{BUY, HOLD, SELL\}$. Then, one can easily construct a DQN architecture in order to predict the Q value and the action to take on one hand, and, one the other, predict the number of shares one should include in the action (see 10). This DQN type of approach has prove to outperform fixed-number strategies as well as more standard RL approaches (Jeong and Kim [2019]).
Many other tasks within the financial context could be/have been explore from portfolio management to options pricing or even hedging strategy.

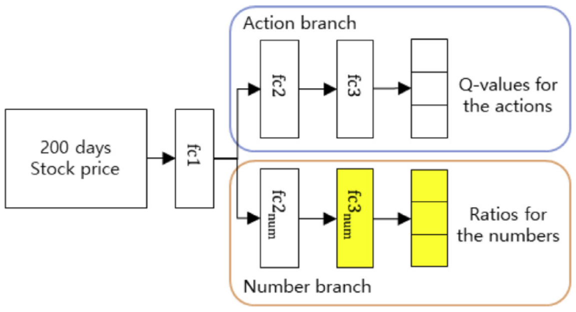


Figure 10: DQN architecture for trading decision (Jeong and Kim [2019])

References

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

Richard S Sutton, Andrew G Barto, et al. Introduction to reinforcement learning. 1998.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. Citeseer, 1994.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, volume 6, 1993.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

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. *nature*, 518(7540):529–533, 2015.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

Leemon C Baird III. Advantage updating. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems*, 29, 2016.

Georg Ostrovski, Marc G Bellemare, Aäron Oord, and Rémi Munos. Count-based exploration with neural density models. In *International conference on machine learning*, pages 2721–2730. PMLR, 2017.

Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *International conference on machine learning*, pages 176–185. PMLR, 2017.

Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control. *arXiv preprint arXiv:1606.04460*, 2016.

Zichuan Lin, Tianqi Zhao, Guangwen Yang, and Lintao Zhang. Episodic memory deep q-networks. *arXiv preprint arXiv:1805.07603*, 2018.

Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. *Advances in neural information processing systems*, 5, 1992.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549. PMLR, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

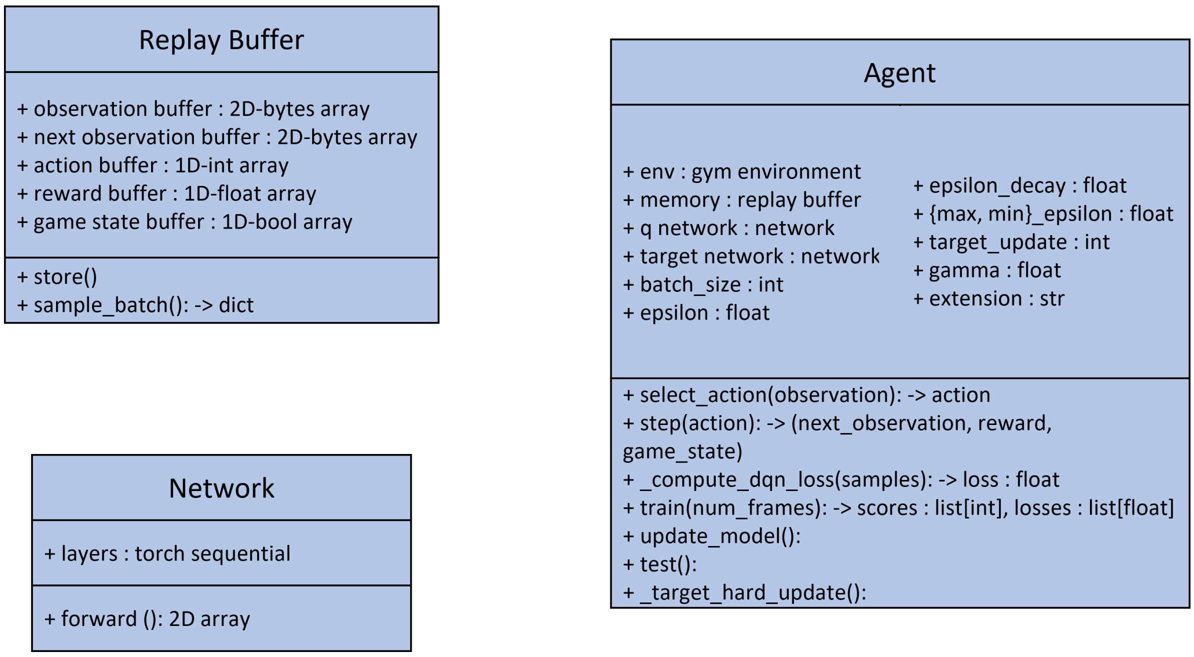
Ivaylo Popov, Nicolas Heess, Timothy Lillicrap, Roland Hafner, Gabriel Barth-Maron, Matej Vecerik, Thomas Lampe, Yuval Tassa, Tom Erez, and Martin Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint arXiv:1704.03073*, 2017.

Elena Mocanu, Decebal Constantin Mocanu, Phuong H Nguyen, Antonio Liotta, Michael E Webber, Madeleine Gibescu, and Johannes G Slootweg. On-line building energy optimization using deep reinforcement learning. *IEEE transactions on smart grid*, 10(4):3698–3708, 2018.

Gyeeun Jeong and Ha Young Kim. Improving financial trading decisions using deep q-learning: Predicting the number of shares, action strategies, and transfer learning. *Expert Systems with Applications*, 117:125–138, 2019. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2018.09.036. URL `https://www.sciencedirect.com/science/article/pii/S0957417418306134`.

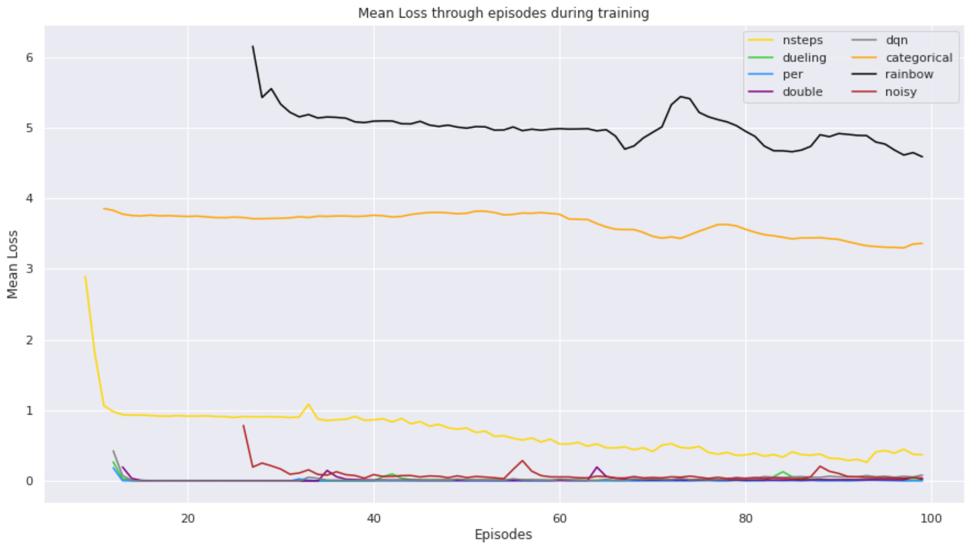
Appendix

Appendix 1 - DQN Classes schema


Replay Buffer
+ observation buffer : 2D-bytes array
+ next observation buffer : 2D-bytes array
+ action buffer : 1D-int array
+ reward buffer : 1D-float array
+ game state buffer : 1D-bool array
+ store()
+ sample_batch(): -> dict
Network
+ layers : torch sequential
+ forward (): 2D array
Agent
+ env : gym environment
+ memory : replay buffer
+ q network : network
+ target network : network
+ batch_size : int
+ epsilon : float
+ epsilon_decay : float
+ {max, min}_epsilon : float
+ target_update : int
+ gamma : float
+ extension : str
+ select_action(observation): -> action
+ step(action): -> (next_observation, reward, game_state)
+ _compute_dqn_loss(samples): -> loss : float
+ train(num_frames): -> scores : list[int], losses : list[float]
+ update_model():
+ test():
+ _target_hard_update():

Appendix 2 - Agents loss curve

The loss values of an algorithm does not mean that it is not training : rainbow has the higher loss curve values even if it is in 3^{rd} position when talking about score. It might be due to the fact there is several extensions, method and hyperparameters used by the agent. Then the most important is the curve shape and we clearly see that rainbow loss is decreasing.


Mean Loss through episodes during training
Mean Loss
Episodes
nsteps
dueling
per
double
dqn
categorical
rainbow
noisy