Polo del Conocimiento

Pol. Con. (Edición núm. 70) Vol. 7, No 7
Julio 2022, pp. 2732-2770
ISSN: 2550 - 682X
DOI: 10.23857/pc.v7i6

# *El Agente Rainbow en pocas palabras: de RL a la combinación de mejora de DQN*

## *The Rainbow Agent in nutshell: From RL to DQN's enhancement combination*

## *Agente Rainbow em poucas palavras: Combinação de aprimoramento de RL para DQN*

Alex Eduardo Pozo-Valdiviezo [I]
eduardo.pozo@espoch.edu.ec
https://orcid.org/0000-0003-0480-5669

**Correspondencia:** eduardo.pozo@espoch.edu.ec

Ciencias Técnicas y Aplicadas
Artículo de Investigación

*** Recibido:** 23 de mayo de 2022 ***Aceptado:** 12 de junio de 2022 *** Publicado:** 31 de julio de 2022

I. Escuela Superior Politécnica de Chimborazo (ESPOCH), Ecuador.

## Resumen

En este documento, nuestro objetivo es centrarnos en alguna noción básica sobre el Aprendizaje por Refuerzo particularmente el algoritmo Q-learning. Luego cruzaremos el marco de RL dentro de la tecnología Neural Network, ya que un entorno de estado fuerte como los videojuegos no se puede administrar dentro de una simple optimización de la tabla Q. Este proceso de acoplamiento les da nacimiento a la Deep-Q-Network (DQN) que son la primera etapa de la llamada "Inteligencia Artificial". Vamos a aprovechar el algoritmo Vanilla DQN como punto de referencia y presentaremos las mejoras DQN más famosas y las clasificaremos dentro de la mejora de pérdida de error TD o la mejora de arquitectura: Double Deep Q-Network, Dueling Network, Priority Experience Replay, RL distributional, Dueling Network, aprendizaje Multi-Step Q. Cómo podemos combinarlos genuinamente para superar todos los algoritmos de márgenes y nuestro benchmark Vanilla DQN, llamaremos a esta combinación inteligente de todas estas mejoras como el agente Rainbow. Analizaremos su rendimiento utilizando líneas de base y comprenderemos el peso efectivo de cada componente mediante el método de ablación. También se examinaría el establecimiento de la metodología de los hiperparámetros. Se daría un código y una aplicación para la mayoría de los métodos. Nuestro punto de partida es el artículo de Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., & Silver, D. (2018, April), del cual tomamos la iniciativa de enfatizar un poco este efecto de investigación ilustrando el uso extensivo del agente de Rainbow en la mayoría de los concursos de Game como Sonic podría testificar.

**Palabras clave:** Agente Rainbow; Framework RL; Algoritmo Vanilla DQN.

## Abstract

In this document, we aim to focus on some basic notion about the Q Reinforcement Learning Algorithm. Then we will cross the RL framework within the Neural Network technology since a strong state environment like video games cannot be managed within a simple optimization of the Q table. This coupling process gives birth to the Deep-Q-Network (DQN) which are the first stage of the so-called "Artificial Intelligence". We will exploit the Vanilla DQN algorithm as a benchmark and present the most famous DQN improvements and classify them within the TD error Improved loss or improved architecture: Double Deep Q-Network, Dueling Network, Priority Experience Replay, Distributional RL, Dueling Network, Multi-Step Q learning. How we can genuinely combine them to overcome all the margin algorithms and our benchmark Vanilla DQN, we will call this intelligent combination of all these enhancements the Rainbow agent. We will analyze their performance using baselines and understand the effective weight of each component using the ablation method. The establishment of the hyperparameter methodology would also be considered. A code and application would be provided for most methods. Our starting point is the article by Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., & Silver, D. (2018, April), of which we took the initiative to emphasize a little this research effect by illustrating the extensive use of the Rainbow agent in most Game contests as Sonic might testify.

**Keywords:** Rainbow agent; RL framework; Vanilla DQN Algorithm.

## Resumo

Neste documento, nosso objetivo é focar em algumas noções básicas sobre Aprendizado por Reforço, particularmente o algoritmo Q-learning. Em seguida, cruzaremos o framework RL para a tecnologia Neural Network, pois um ambiente de estado forte como videogames não pode ser gerenciado dentro de uma simples otimização Q-table. primeira etapa da chamada "Inteligência Artificial". Vamos tomar o algoritmo Vanilla DQN como referência e introduzir as melhorias DQN mais famosas e classificá-las em melhoria de perda de erro TD ou melhoria de arquitetura: Double Deep Q-Network, Dueling Network, Priority Experience Replay, RL Distribution, Dueling Network, Aprendizado em várias etapas Q. Como podemos combiná-los genuinamente para superar todos os algoritmos de margem e nosso benchmark Vanilla DQN, chamaremos essa combinação inteligente de todas essas melhorias como o agente Rainbow. Analisaremos seu desempenho usando linhas de base e entenderemos o peso efetivo de cada

componente usando o método de ablação. O estabelecimento da metodologia de hiperparâmetros também seria considerado. Código e implementação seriam fornecidos para a maioria dos métodos. Nosso ponto de partida é o artigo de Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., & Silver, D. (2018, abril), de que tomamos a iniciativa de enfatizar um pouco esse efeito de pesquisa, ilustrando o uso extensivo do agente da Rainbow na maioria dos concursos de jogos, como o Sonic pôde testemunhar.

**Palavras-chave:** Rainbow Agent; RL Framework; Algoritmo Vanilla DQN.

## Introducción

**Q-learning Algorithm and SARSA.** Q-Learning is a Value-based Reinforcement Learning algorithm with off policy selection. SARSA is a Value-based Reinforcement Learning algorithm with on policy selection. Legitimate question can appear, what is the difference between both algorithms. Let first introduce the Q function, Q reminding the "Quality of the action", at this stage one can imagine that Q-learning Algorithm would be an enriched Value functional within the knowledge of the action. The Action Value Function (or "Q-function") takes two inputs: "state" and "action". It returns the expected future reward of that action at that state.

$$Q^{\pi}(s_t, a_t) = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k \ R_{t+k+1} | \, s_t, a_t] \qquad (1)$$

We remind that both **SARSA** and **Q-learning** are Value-Based Algorithm, the main difference is that they differ in term of policy. On-policy and off-policy learning are only related to the only one task: evaluating $Q^{\pi}(s_t, a_t)$, then starting from this common point, the procedure diverge in term of methodology.

1. In **Q-learning** that is a learning, the $Q(s_t, a_t)$ function is learned from different actions (for example, random actions). We even don't need a policy at all.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r + \gamma \max_{a} Q(s'_t, a'_t) - Q(s_t, a_t)\right] \qquad (2)$$

2. In **SARSA** on policy learning the $Q(s_t, a_t)$ function is learned from actions, we acted using our current policy π, we learn using the TD framework:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma Q(s_t', a_t') - Q(s_t, a_t)] \qquad (3)$$

where $a_t'$ is the action that was taken to policy choice $\pi$.

We remind that the **Update policy** is how your agent learns the optimal policy, and **behavior policy** is how your agent behaves.

3. In **Q-Learning**, the agent learns optimal policy using absolute greedy policy and behaves using other policies such as $\epsilon$ - greedy policy. As the update policy is different to the behavior policy, so Q-Learning is off-policy.

4. In **SARSA**, the agent learns optimal policy and behaves using the same policy such as $\epsilon$ - greedy policy. As the update policy is the same to the behavior policy, so **SARSA** is on-policy.

At this stage, we will now focus on the Q-learning Algorithm and we will try to understand how we can improve most of its weakness when environment is quite heavy in term of size, this will lead us to consider Deep Q network.

First let have some familiarity within the Q-table output by the Q-learning Algorithm.

**Q-table and Q-learning algorithm procedure:** The task is to build a table where we are mandated to compute the maximum expected future reward, for each action at each state.

Q-table is a matrix within row are flagged as the states and the column are flagged as actions. Q-table is just a matrix of storage for the $Q(s_t, a_t)$ at each time step. The value of each element of the matrix would then be the maximum expected future reward for that given state and action. Each Q-table score will be the maximum expected future reward that the agent would get if he took an action at a particular state with the best policy already fixed. We remind that Q-learning is an off policy so in practice, we don't implement a policy, instead of that, we just improve our Q-table to always choose the best action.

Now we proceed to indicate how we calculate the table at each time step:

- Step 1: **Initialization**

We build a Q-table, with $n$ rows ($n$ = # of states), $p$ cols ($p$ = # of actions). We initialize by setting all the values at 0.

$$\boldsymbol{Q(s_0, a_0) \in M_{n\times p}(0)}$$

$$Q(s_0, a_0) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad (4)$$

where the first two columns are the states and the other two columns the actions.

- Step 2: **Choose an action**, to repeat till the learning procedure is stopped.

Then we show how we can select an action from the uniform null states. Let us first use the $\epsilon$-greedy policy.

- We specify an exploration rate $\epsilon$, which we set to 1 in the beginning. This is the rate of steps that we'll do randomly. In the beginning, this rate must be at its highest value, because we don't know anything about the values in Q-table and we want to force the exploration phase. Picking high value of $\epsilon$ means we need to do a lot of exploration, by randomly choosing our actions.
- We generate a random number. If this number > $\epsilon$, then we will do "exploitation" phase (we use what we already know to select the best action at each step). Else, we will switch to exploration.

1. The strategy is a phase selection is set such that we must have a big epsilon at the beginning of the training of the Q-function, then, reduce it progressively as the agent becomes more confident at estimating Q-values.

Short Term we make the choice to be exploratory and then gaining confidence to be more exploitative on longer term horizon.

- Step 3 & 4: **Take an action** and **Update the Q-table**.

We take the action $a$ that we chose in the previous step 2, and then performing this action returns us a new state $s'$ and a reward $r$, then we update the function $Q(s, a)$ using Selection is done in respect of that rule: For all possible actions from the state $s'$ select the one with the highest Q-Value.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r + \gamma \max_a Q(s'_t, a'_t) - Q(s_t, a_t) \right]$$

Here is the pseudo code procedure for the Q-learning Algorithm:

1. Initialization of the Q-values for all states and actions couples.

While learning is not stopped do:

2. Choose an action $a$ in the current observed state environment based on the current Q-value estimates $Q(s,\cdot)$ within $\epsilon$ - greedy policy modularity.

3. Take the selected action $a$ coming from step 2 and observe the nature of the new outcome state $s'$ and reward $r$.

4. Update the Q-function using the Bellman equation:

$$New\ Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r + \gamma \max_a Q(s'_t, a'_t) - Q(s_t, a_t) \right] \quad (5)$$

$a$

To resume a bit: the Q-learning algorithm is based on a production and updating a Q-table, we fill this table in aim to find a maximum expected future reward.

As we are more familiar within the Q-learning Algorithm, natural question can then income, what would happen for a huge Q-table matrix. We can imagine that we could be interacting in a complex video game like in a shoot them all or a platform one like Sonic, Mario, or Doom, where the environment set is highly dimensioned (around 10e6 of different states). Creating or updating a Q-table for that kind of game would clearly not be efficient at all. We must revisit the problem using Deep-Neural Network technology in aim to retrieve the Q-value of the actions.

Create Neural Network that will approximate for a given state the different Q-values for each action. We then combine the Q-learning to Deep neural network giving then the Deep Q-Network Algorithm (DQN).

## Computation of Q-learning Algorithm application to openai Gym / Retro in Python

We will in this section give some material that would allow the user to train an agent in an old school video game environment. Our study has been done in Windows 10 environments using

Anaconda 3 package with Python version 3.7. Also, you must install or to be more specific. You can install it by running *pip install gym[atari].*

We will then describe here some of functionalities of the OpenAI gym package. We illustrate it shortly as https://www.gymlibrary.ml/content/api/ does it better and is complete in term of user guidelines.

Here's a bare minimum example of getting something running. This will run an instance of the MsPacman-v0 environment for 1000-time steps, rendering the environment at each step. You should see a window pop up rendering the classic PacMan adventure.

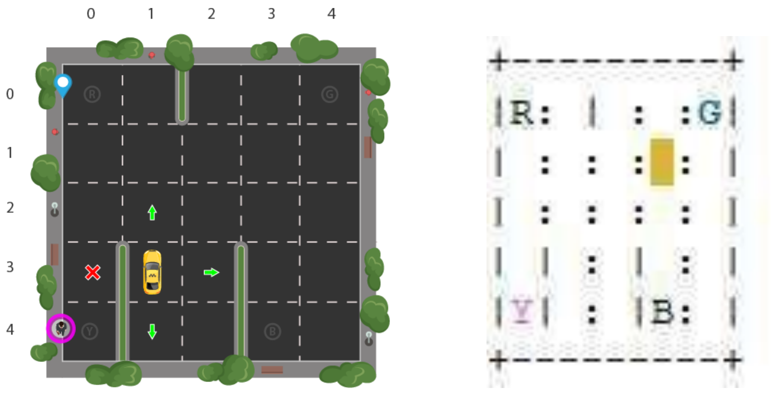
Every Gym environment will return these same four variables after an action is taken, as they are the core variables of a reinforcement learning problem.

**Source:** Own elaboration

**Figure 1:** Result of python render exit for Pacman

For Pacman, we can clearly not use the Q-learning Algorithm as the environment state is quite large: there is a total of 33,600 pixels with three RGB values that can have a range from 0 to 255. But we will run it within Taxi-v3 game, below we report the Q-learning Computation for 2K episodes starting from a reward strongly negative and progressively increasingly during experience.

**Source.** Own elaboration

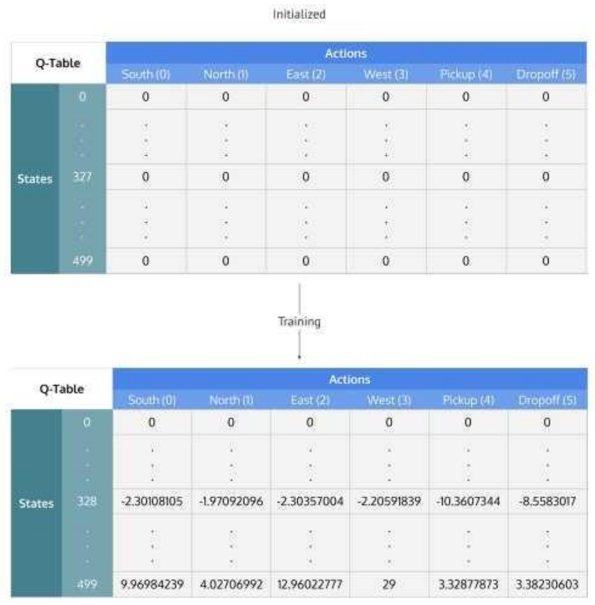


```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

**Figure 2:** Taxi-v3 environment and the result of python render exit for Taxi-V3

Within the Q-table filling:

**Figure 3:** Filling of the Q-table


Initialized
Training

| Q-Table | | Actions | | | | | |
|---|---|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) | Pickup (4) | Dropoff (5) |
| States | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | : | : | : | : | : | : | : |
| | 327 | 0 | 0 | 0 | 0 | 0 | 0 |
| | : | : | : | : | : | : | : |
| | 499 | 0 | 0 | 0 | 0 | 0 | 0 |

| Q-Table | | Actions | | | | | |
|---|---|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) | Pickup (4) | Dropoff (5) |
| States | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | : | : | : | : | : | : | : |
| | 328 | -2.30108105 | -1.97092096 | -2.30357004 | -2.20591839 | -10.3607344 | -8.5583017 |
| | : | : | : | : | : | : | : |
| | 499 | 9.96984239 | 4.02706992 | 12.96022777 | 29 | 3.32877873 | 3.38230603 |

The modularity of the Q-learning Algorithm is driven by the variability of the Hyper parameters of the methods.

- Learning Rate α, empirical study during the training phase would suggest a decreasing time depends on structure.

The larger the learning rate, the stronger is the influence of new data for updating.

- Discount Factor $\gamma$, empirical study during the training phase would also suggest a decreasing time depends on structure.

It allows to allocate a smaller weight on future reward.

- Exploration or Exploitation trade off $\varepsilon$ in the $\varepsilon$- greedy off policy for the Q-learning, empirical study during the training phase would also suggest a decreasing time depends on structure.

The Epsilon parameter could be seen as a probability to exercise an exploratory phase during the learning procedure.

## Hyper parameters tuning for Q-learning Algorithm and application to Taxi-V3 from openai gym

**Learning Rate** α **effect:** We will consider the case where gamma and epsilon would be fixed first, in aim to avoid being so much exploratory, we will keep epsilon close to 0.1, we will explain this choice a bit later, we remind that as we restricted the exploratory probability the computation is also less greedy. We will also publish another score measure that will the average score pro-rated by time; we will call it score over the time.

In the next graphic sequence, we will restrict our visualization on 2k episodes but computation has been done with 10k episodes step, we will first marginally have a idea of the learning rate parameter sensitivity.

**Source.** Own elaboration

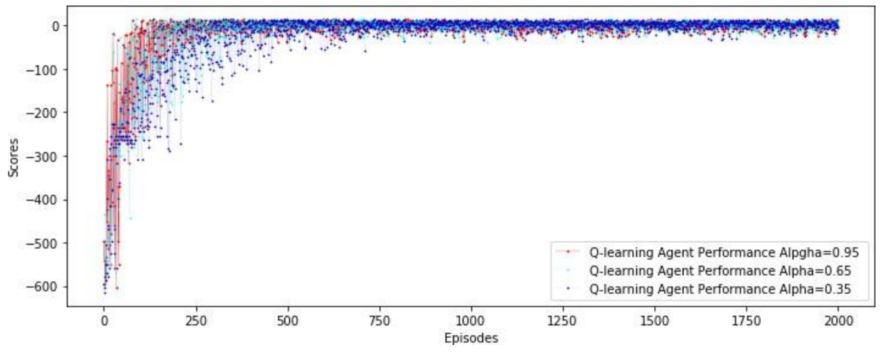


**Figure 4:** Q-learning Agent Performance with α sensitivity for $\gamma = 0.65$, $\varepsilon = 0.1$, Episodes=10k

**Source.** Own elaboration

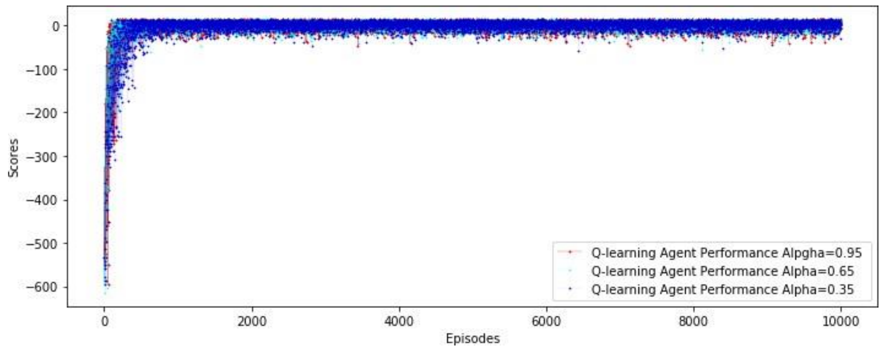


**Figure 5:** Q-learning Agent Performance with score over time = [-0.5,-1.01,-2.8] for decreasing α

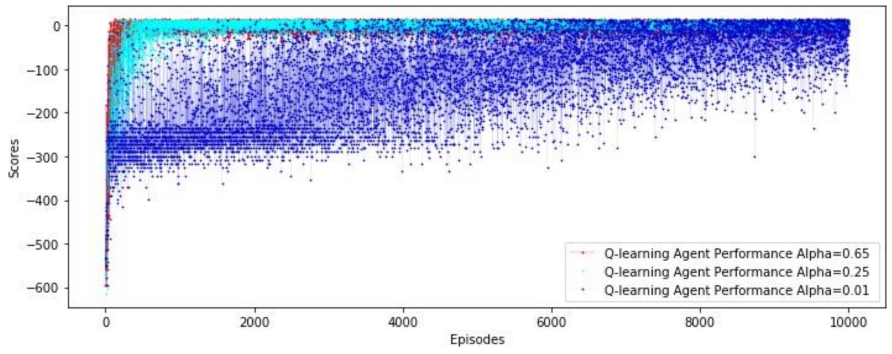


**Figure 6:** Q-learning Agent Performance with score over time = [-1.01, -4.5 ,-118.8] for decreasing $\alpha$

We also provide a heat map to help to have a view in the bivariate variation of the Alpha & Gamma sensitivity. (Reported as score over time and step value for params are 10% Epsilon = 0.1 & Episodes =10k)

**Conclusion Learning/Gamma rate effect:** Clearly strength of learning rate is readable, even in computation time. Ideally the hyper parameter α must be time decreasing as the agent continues to gain a larger and larger knowledge base.

**Source.** Own elaboration

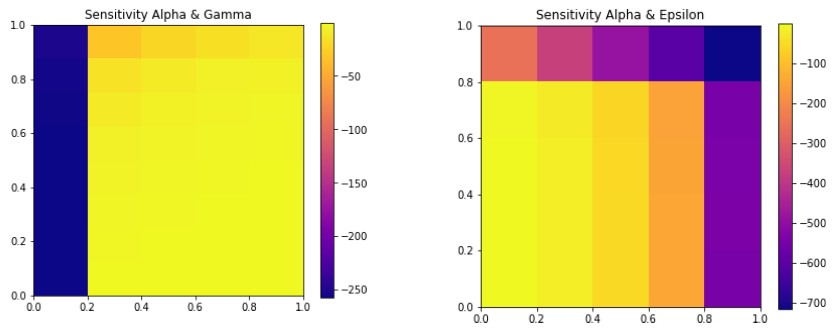


**Figure 7:** Alpha & Gamma sensitivity and Alpha & Epsilon sensitivity for Epsilon = 0.1 and 10k episodes

**Greedy Binary probability effect**: Most of the score sensitivity is carried by the choice of the binary probability $\epsilon$, we then decided to custom a bit this parameter in aim to give a first impulse of exploratory mindset to fill the Q-table and then decided to strongly decay this parameter in aim to allow exploitation policy phase.

**Source.** Own elaboration

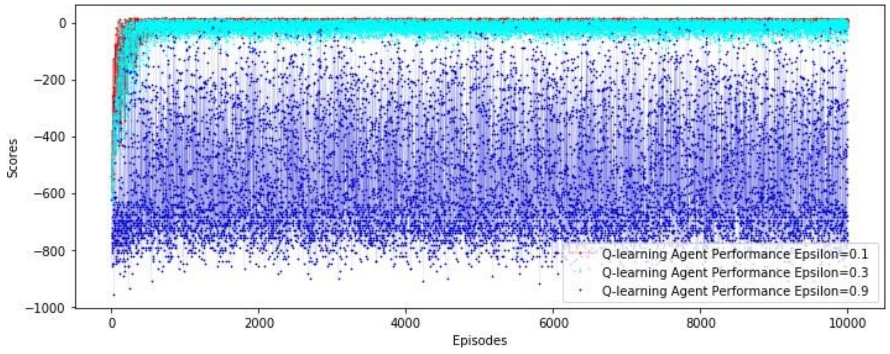


**Figure 8:** Q-Learning Agent Performance with Epsilon effect for $\alpha = 0.3, \gamma = 0.9$

Here we report the score of our adaptive strategy:

**Source.** Own elaboration

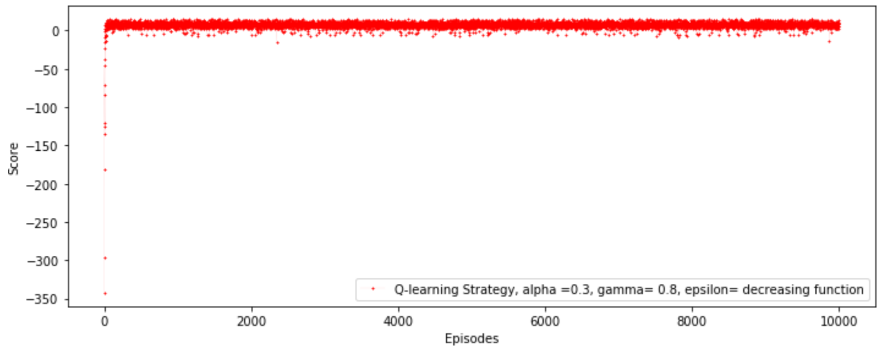


**Figure 9:** Q-Learning with score overtime = 7.5, exploration probability = 0.005, decay rate= 0.1

We are now ready to approach the Deep-Q-Network Algorithm as we became more familiar with the Q-Learning Algorithm.

## Deep q-network algorithms

As we previously mentioned, the major issue with Q-Learning is, once the number of states in the environment are very high, it becomes not efficient to implement them with Q table as the size would become very, very large. State of the art techniques uses **Deep neural networks** instead of the Q-table (we call this technology combination as **Deep Reinforcement Learning**). The neural network framework takes in state information's and actions to the input layer and learns to output the right action over the time.

Deep learning techniques (like Convolution Neural Networks) are also used to interpret the pixels on the screen and extract information out of the game (like scores), and then letting the agent control the game. We will then at this stage fluently use the **DQN** notation for Deep Q Network, we will also emphasis in this section about 7 different complementary improvements for the DQN algorithm and we will try to understand how to combine them genuinely using the Rainbow repartition.

To reach our target, we will try to be more familiar with the Neural Networks notions and the DQN, in aim to be more pertinent in our way to aboard the strength and limitation of all our Rainbows colors.

**Neural Networks basics and Deep Neural Networks.** The Neural Networks democratization has been impulse by a combination of several factors: Big Data emergence, GPU extensive use and some very flexible model that could allow to divert the curse of high dimensionality.

A neural network in his simplest form could be seen as a nonlinear application with respect to his parameter set $\theta$ that associate to an entry $x$ an output $y = f(x, \theta)$.

We can deal with multidimensional problem and the functional $f$ has a particular form. Then we can understand that a neural network could be a regression or a classifier, the set of parameters $\theta$ would be estimated from a learning sample. The cost function is not convex leading to local optimums, back propagation of the gradient would allow us to retrieve local minimizer without any greedy computational cost.

**The perceptron or Single Artificial Neuron.** Let focus on the concept of a single perceptron or artificial neuron, in term of architecture, a neural network is considered as a partition of several artificial neuron $\left(f(x_j)\right)_{j\in J}$ in term of functionality, we consider all the basic element wise:

- Given inputs are represented by the $d$–dimensional vector $x = (x_k)_{k\in[1,d]}$. One can figure out that this vector can be seen as the set of $d$ neural termination linked to the neuron $x$:
- Intensity of excitatory or inhibitory signal transmission is represented by the weighted vector of connexion strength directed to the $j$-th artificial neuron, let denote the weight connexion family by $w_j = (w_{j,k})_{k\in[1,d]}$.
- A bias neuron associated $b_j$.
- An activation function φ that could be into this kind of function family (Opening Ions gate function):

  - **Identity** function $\phi(x) = x$.
  - **Sigmoid** or so-called **Logistic** function: $\phi(x) = \frac{1}{1+exp(-x)}$ (homogeneous to a probability repartition function).
  - The **hyperbolic tangent function:** $\phi(x) = tanh(x)$.
  - The **hard threshold function or Binary signal activation** $\phi(x) = 1_{x>a}$.
  - The **Rectified Linear Unit (ReLU) activation function**, or positive part function: $\phi(x) = max(x, 0) + \delta\, min(x, 0)$.

- Output the functional $y_j = f(x_j)$ that would be the exit effective transmitted signal.

$$y_j = \phi(\langle x, w_j\rangle + b_j) \qquad (6)$$

For multilayer network, ReLU is favored as its gradient is not null for small value of $x$ contrary of the sigmoid. The ReLU function also has a sparsification effect. The ReLU function and its derivative are null for negative values, and no information can be obtained in this case for such values, then a small add-on would allow to give values to negative values $\phi(x, \delta)$ with $\delta$ positive and small enough.

**Neural Network or Multilayer perceptron.** A Neural Network is a structure composed of several hidden layers of neurons where the output of a given neuron of a layer becomes at its turn the input of a neuron of the next layer.

We are speaking about **Recurrent Neural Network** when the output of a neuron can also be the input of a neuron of the same layer or of neuron of previous layers. On the last layer called output layer, one may also apply a different activation function as for the hidden layers depending on the type of problems we have at hand: regression or classification threat.

The architectural parameters of the network are: The Cardinal of hidden layers *L*; the Cardinal of neurons into a layer and let describe all the links and the functionals engaged in a such network, as mentioned upper let consider different functional for the interlayer activation and the output one.

## Overview of the Network

- Initialization and input configuration, let set $h^0(x) = x$.
- Hidden Layers: For $k = 1, \dots, L$ and an activation function interlayer $\Phi$

$$a^k(x) = b^k + h^{k-1}(x) \qquad (7)$$
$$h^k(x) = \phi(a^k(x)) \qquad (8)$$

- Output Layer: For $k = L + 1$ and an output activation function $\psi$

$$a^{L+1}(x) = b^{L+1} + W^{L+1}h^L(x) \qquad (9)$$
$$h^{L+1}(x) = \psi(a^{L+1}(x)) = f(x,\theta) \qquad (10)$$

We are now able to speak about Deep Neural Network, as the qualitative Deep just reminds us that the amount of hidden layers could be important. Once the architecture of the network has been chosen, our objective is to estimate the parameter set $\theta = (W^k, b^k)$. As usual, the estimation is obtained by minimizing a loss function with a gradient descent algorithm. We first must choose the loss function: let select the opposite of log likelihood and use an adapted **Stochastic Gradient Descent** within **Back propagation Algorithm** for the penalized problem. Let understand how we can combine the Q-learning algorithm within the Deep Neural Network technology, **the association of these two concepts give birth to the AI**.

**Introduction to Deep Q-Network.** In this section as we previously presented the Deep Neural Network, we will see how we could map this precious technology to the Q functions.

In high dimensional Q-matrix framework, the optimal Q-function given from the Q-table could not be retrieved, to divert such issue, one could approximate or learn it from the different interactions within the environment as classical Q-learning algorithm.

The “Deep” function is a way that would allow to approximate the Q-function using extensive deep neural networks which are universal approximation. Deep Q-Networks is the common name for Deep neural networks that approximate the Q-values. In short, our goal now is to approximate the optimal Q-function noted $Q^*$,using the well know Bellman equation and supervised learning method to approximate $Q^*$.

Our Neural Network input are the state and the action, the output is the Q-value related to this pair, this naive but heuristic vision has a limitation, we need to call the Q function as many times as there are actions. So, one would decide only to take the state as input and output the Q-values for all the possible actions.

Deep Q-learning is a form of the classic Q-learning model. At every step, the state is evaluated and will give a Q-value which approximate the reward of each possible action. Machine learning is applied to makes a supervised model with input state and output action. In this case, a deep neural net is used as a machine learning model, hence the name Deep Q-learning.

To have a better understanding the selection of the Deep Q-learning, let introduce the notion of **Tabular Q-learning** that deals non efficiently with all frameworks:

In **tabular Q learning**, when the number of elements in the state or action space is enormous or the state or action space is continuous, we often express the Q function as a parametric function $Q(s, a, \theta)$ using the parameters $\theta$ and then update the parameters according to the gradient method.

$$\theta \leftarrow \theta + \alpha\left(\boldsymbol{Target}_Q - Q(s, a, \theta)\right)\nabla_\theta Q(s, a, \theta) \qquad (11)$$

The Target Value is obtained by the Bellman equation:

$$\boldsymbol{Target}_Q = r(s, a, s') + \gamma \max_{a} Q(s', a', \theta) \qquad (12)$$

The Method is homogeneous of a **Bootstraping** one where the Q function approximate is regressed toward the target value, which depends on itself.

His main **drawback** is: The target value changes automatically when the learning network is updated. Therefore, when a non-linear function, such as a neural network is used for approximation, this learning process becomes unstable due to dynamical changes in the target, and in the worst case, **the Q function diverges** (Sutton and Barto, 1998).

This **instability has several causes** one well knows is the **correlations present in the sequence of observations**, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values Q and the target values $r(s,a,s') + \gamma \max_{a} Q(s',a',\theta)$**.** We will emphasis within this issue solved by the **Experience Replay Memory Concept** in the next sections.

**DQN architecture efficiency and filling Network procedure.** The Deep Q-Network architecture could manage with drawback easily, to prevent Q-function from any kind of divergence, DQN introduces the target network that will be a duplication of the Q-function that is used to calculate the Q-target Value.

DQN use the following kind of rule:

$$\theta \leftarrow \theta + \alpha \left( \boldsymbol{Target}_{DQN} - Q(s,a,\theta) \right) \nabla_{\theta} Q(s,a,\theta). \qquad (13)$$

The trick is that the Target Value from DQN is obtained exploiting the target network:

$$\boldsymbol{Target}_{DQN} = r(s,a,s') + \gamma \max_{a} \boldsymbol{TNet}(s',a'). \qquad (14)$$

$\boldsymbol{TNet}(s',a')$ is the target network, one may recognize a Time Differential error computation as learning procedure, one can then remark in term of homogeneity that:

$$\nabla\theta = \alpha \left[ r(s,a,s') + \gamma \max_{a} \boldsymbol{TNet}(s',a') - Q(s,a,\theta) \right] \nabla_{\theta} Q(s,a,\theta) \qquad (15)$$

Let explain the nature of each quantity:

- $\boldsymbol{Target}_{DQN} = r(s,a,s') + \gamma \max_{a} \boldsymbol{TNet}(s',a')$ is the **maximum possible Q-value** for the next state.
- $Q(s,a,\theta)$ is the **current predicted Q-value**.

- $\nabla_{\theta} Q(s, a, \theta)$ is the **Gradient** of our current predicted Q-value.
- $\left[r(s, a, s') + \gamma \max_{a} \boldsymbol{TNet}(s', a') - Q(s, a, \theta)\right]$ is the **TD error.**

Naturally, we want to know how to upgrade such a network.

- Periodic update where the target network is synchronized with the current Q function at every C learning step when the following condition is satisfied:

$$\boldsymbol{Totalsteps\ ModC} = 0, \boldsymbol{Target}_{DQN} \leftarrow Q \qquad (16)$$

where **Totalsteps** represents the total number of updates applied to the Q function up to the present time. This method is based on Neural Fitted Q Iteration (Riedmiller, 2005), which is a batch Q learning method that employs a neural network.

- The symmetric update, in which the target network is updated symmetrically as the learning network, and this is introduced in double Q-Learning (van Hasselt, 2010; van Hasselt et al., 2016).

In addition, one can remark that two processes occur during this algorithm, we distinguished them during just upper.

We must now proceed to update with the completed with the repetition of experiences and then calibration of the learning procedure through a gradient descent for a given function Loss. Using replay memory buffer is not free from all kinds of risk, one need to divert a severe drawback that is induced by the correlation between samples, the random selection would help to reduce these phenomena.

DQN utilizes the **Experience Replay** concept that would avoid the Agent to forget the previous experiences as there is strong correlation between actions and environment, to illustrate it at each time one receives a tuple $(s_t, a_t, r_t, s_{t+1})$ we then need to reduce correlation between experiences. Specifically, when the agent selects an action an at a state $s$ and receives a reward $r$ and the state then transits to $s'$, this data point $(s, a, r, s')$ is stored in replay memory *D* and used for mini-batch learning.

In mini-batch learning, the parameters are updated based on a certain number of data points randomly selected from the replay memory *D*, and this procedure is repeated several times.

This makes it possible to prevent stagnation of learning because of correlation between data points while maintaining the one-step Markov property. Now incorporating the Experience replay in our TD error computation will lead to the passage to the conditional expectation with respect of uniform pick of the experienced sample tuple $(s, a, r, s')$ from the replay buffer *D*.

$$\theta \leftarrow \theta + \alpha E_{(s,a,r,s')\sim U(D)}\left[\left(\boldsymbol{Target}_{DQN} - Q(s,a,\theta)\right)\nabla_\theta Q(s,a,\theta)\right]. \qquad (17)$$

The Learning procedure at each step in then given by the Loss representation:

$$L(\theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(\boldsymbol{Target}_{DQN}(\theta_{i-}) - Q(s,a,\theta)\right)^2\right]. \qquad (18)$$

Here we reported the **Mean Square Error** for the Loss, one can choose the **cross entropy** or **Mean Absolute Error**: $\theta_i$ represent the set of parameters for each iteration, whereas $\theta_{i-}$ are the one used to compute the target at this iteration, we remind that the target network is only updated at every periodic step *C* and are held fixed between individual updates.

**Stability of the DQN**: The learning process of DQN is more stable than that of Q-Learning because the update rule of DQN introduces a delay between the time when $Q(s,a,\theta)$ is updated and the time when **TNet**(*s*, *a*) is updated. Although the use of the target network is critical for stable learning, it hinders fast learning due to this delay.

**DQN hyper-parameters:** The Batch size *B*; The Target Network Frequency *K*; the probability of exploration function $\varepsilon(t)$; the neural network $Q^*(s,a,\theta)$; SGD optimizer hyper parameters **ADAM.**

## Dqn algorithm

We report here the pseudo code for the Vanilla DQN:

- **Hyper Parameters setting procedure** (iterative posterior calibration would allow to understand their sensitive as we have done previously on Q-learning section).
- **Initialization:**
  - Arbitrary setting of the weights $\theta_-$
  - Initialize $\theta^- \leftarrow \theta$
- **On each interaction step:**

- Select an action $a$ randomly within the exploration probability $\varepsilon(t)$, else $a = argmax_a Q^*(s, a, \theta)$.
- Observe the experience tuple $(s, a, r', s')$.
- Add the observed experience into the experience replay buffer.
- Sample batch of size $B$ from the experience replay.
- For each Transition $T$ from the batch compute the Target Functional:

$$y(T) = r(s') + \gamma \max_{a'} Q^*(s, a, \theta^-) \qquad (19)$$

- Compute the Loss:

$$L = \frac{1}{B}\sum_T \left(Q^*(s, a, \theta) - y(T)\right)^2 \qquad (20)$$

- Make a step of Stochastic gradient descent using the Gradient $\frac{\partial L}{\partial \theta}$.
- If $t \bmod K = 0,\ \theta^- \leftarrow \theta$.

**Convolution Networks:** Here is the road map to understand the mechanism of the Convolutional Network: Let consider an image as input, then we rescale it in term of important features using preprocessing procedure and we stack the preprocessed frame in stack of frames.

**State Frame ⇒ Preprocessed Frame ⇒ Preprocessed stack of Frames**

**Preprocessing:** Let consider that our Deep Q network takes a stack of several frames as an input, each frame would be a collection of images encoded by a pixel. Preprocessing is a major computational advantage as it allows to reduce the complexity of the states set. This can be done by keeping just the relevant element that composes the image, as an illustration colors, RGB could be removed, considering a gray scale when color does not add important information, this is an important saving as we reduced our RBG channel to one where the preprocessing could be done within color environment; could be size reduction and preprocessing could be not important features removing.

**Stacked Frames through Convolution Layers:** The frames would be processed by a certain number of convolution layers. These layers would allow to detect and exploit any spatial relationships between images, using the stack structure, one may also be able to exploit spatial properties across those frames. Each convolutional layer will fire the ELU as of activation function, this kind of choice of function has made its proof for Convolution layers.

As illustration for the Pacman environment, we report the architecture of a DQN to learn the function $Q(s,a)$ within Pacman images as input.

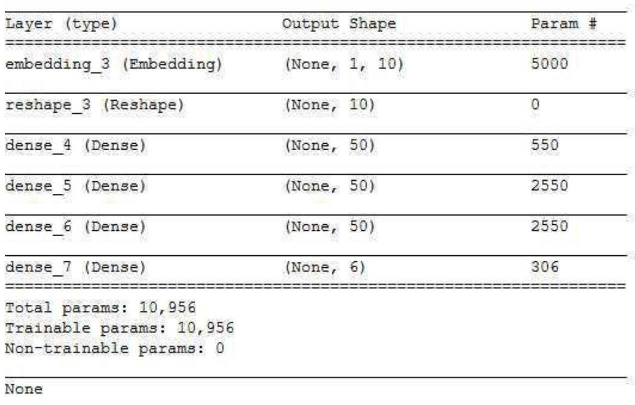
This architecture uses:

- As **input an image** coming from our Pacman environment.
- The **equivalent preprocessed image** that represents in synthetic way all the necessary information issued from the initial image.
- **3 Convolutional Neuron Network Layers (CNN)** that allows to exploit the local connectivity of the image.
- One flattening layer a fully connected Layer.
- **Output the Q value** for all the actions issued from our environment.

**Application of DQN in Taxi-V3.** Let come back on our Taxi-V3 environment, the target would be to compare the Q-learning method vs the DQN one for this game. We completed the Embedding Layers within 4 Dense Layers within the last one that must have the same size of the action set. Console exit for the Deep Neural Network show us:

and the Fitting Output **shows** an ETA status bar within the metrics and the time elapsed.

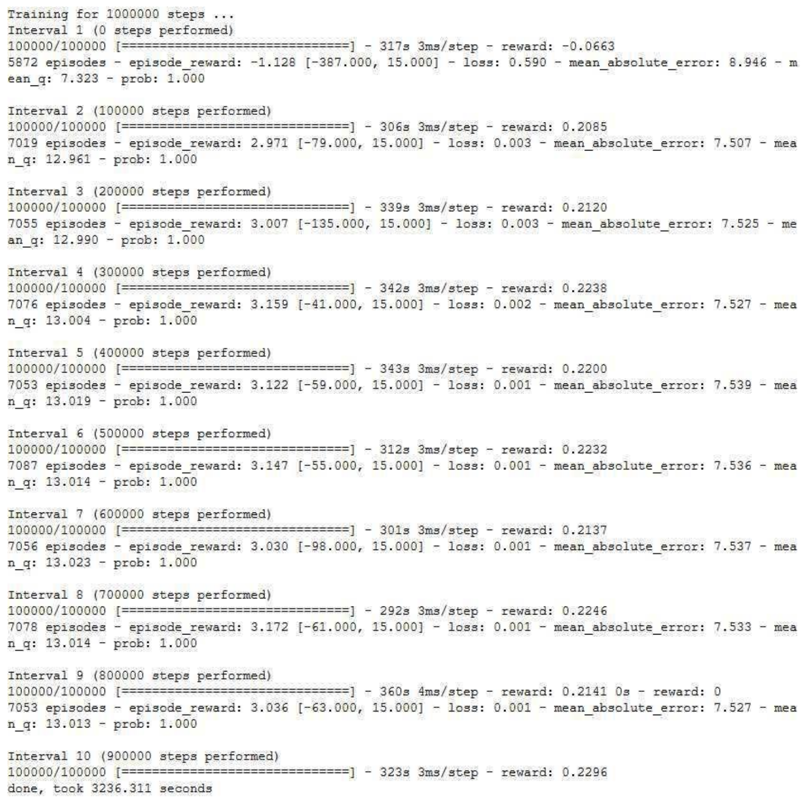
**Source.** Own elaboration



```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 1, 10)             5000
_________________________________________________________________
reshape_3 (Reshape)          (None, 10)                0
_________________________________________________________________
dense_4 (Dense)              (None, 50)                550
_________________________________________________________________
dense_5 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_6 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_7 (Dense)              (None, 6)                 306
=================================================================
Total params: 10,956
Trainable params: 10,956
Non-trainable params: 0
_________________________________________________________________
None
```

**Figure 10:** Console exits for DNN

**Source.** Own elaboration



```
Training for 1000000 steps ...
Interval 1 (0 steps performed)
100000/100000 [==============================] - 317s 3ms/step - reward: -0.0663
5872 episodes - episode_reward: -1.128 [-387.000, 15.000] - loss: 0.590 - mean_absolute_error: 8.946 - m
ean_q: 7.323 - prob: 1.000

Interval 2 (100000 steps performed)
100000/100000 [==============================] - 306s 3ms/step - reward: 0.2085
7019 episodes - episode_reward: 2.971 [-79.000, 15.000] - loss: 0.003 - mean_absolute_error: 7.507 - mea
n_q: 12.961 - prob: 1.000

Interval 3 (200000 steps performed)
100000/100000 [==============================] - 339s 3ms/step - reward: 0.2120
7055 episodes - episode_reward: 3.007 [-135.000, 15.000] - loss: 0.003 - mean_absolute_error: 7.525 - me
an_q: 12.990 - prob: 1.000

Interval 4 (300000 steps performed)
100000/100000 [==============================] - 342s 3ms/step - reward: 0.2238
7076 episodes - episode_reward: 3.159 [-41.000, 15.000] - loss: 0.002 - mean_absolute_error: 7.527 - mea
n_q: 13.004 - prob: 1.000

Interval 5 (400000 steps performed)
100000/100000 [==============================] - 343s 3ms/step - reward: 0.2200
7053 episodes - episode_reward: 3.122 [-59.000, 15.000] - loss: 0.001 - mean_absolute_error: 7.539 - mea
n_q: 13.019 - prob: 1.000

Interval 6 (500000 steps performed)
100000/100000 [==============================] - 312s 3ms/step - reward: 0.2232
7087 episodes - episode_reward: 3.147 [-55.000, 15.000] - loss: 0.001 - mean_absolute_error: 7.536 - mea
n_q: 13.014 - prob: 1.000

Interval 7 (600000 steps performed)
100000/100000 [==============================] - 301s 3ms/step - reward: 0.2137
7056 episodes - episode_reward: 3.030 [-98.000, 15.000] - loss: 0.001 - mean_absolute_error: 7.537 - mea
n_q: 13.023 - prob: 1.000

Interval 8 (700000 steps performed)
100000/100000 [==============================] - 292s 3ms/step - reward: 0.2246
7078 episodes - episode_reward: 3.172 [-61.000, 15.000] - loss: 0.001 - mean_absolute_error: 7.533 - mea
n_q: 13.014 - prob: 1.000

Interval 9 (800000 steps performed)
100000/100000 [==============================] - 360s 4ms/step - reward: 0.2141 0s - reward: 0
7053 episodes - episode_reward: 3.036 [-63.000, 15.000] - loss: 0.001 - mean_absolute_error: 7.527 - mea
n_q: 13.013 - prob: 1.000

Interval 10 (900000 steps performed)
100000/100000 [==============================] - 323s 3ms/step - reward: 0.2296
done, took 3236.311 seconds
```

**Figure 11:** Intervals exits for training of 1000000 steps

**Remark:** One can remark that we don't have a very negative reward at the beginning of the training session as we get previously into the Q-learning application. Scores are comparable within long time episodes.

**Q-Learning**: The training process from Q-learning begin with a massive negative reward score since the algorithm knows just a few elements of the environment, the Q-table are clearly not optimized at this stage and has a clear need to be filled in through several trials.

**DQN**: clearly does not need a period of rapid average reward growth during its training session, while the algorithm also received a negative reward score at the very first stage of its training phase, it does not require many episodes to improve its reward scores.

We report here the Test phase Moving Averaged Reward for the DQN over 50k episodes.

**Source.** Own elaboration

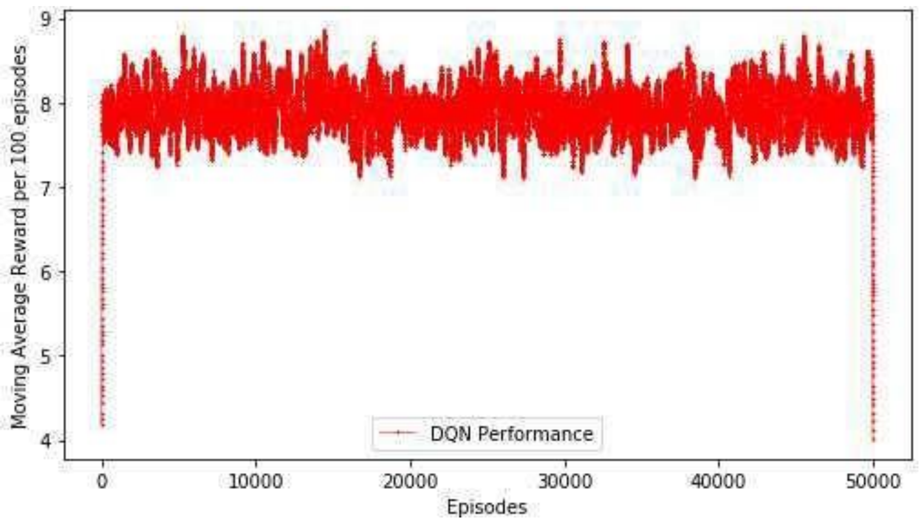


**Figure 12:** The Test phase Moving Averaged Reward for the DQN over 50k episodes

**Source.** Own elaboration

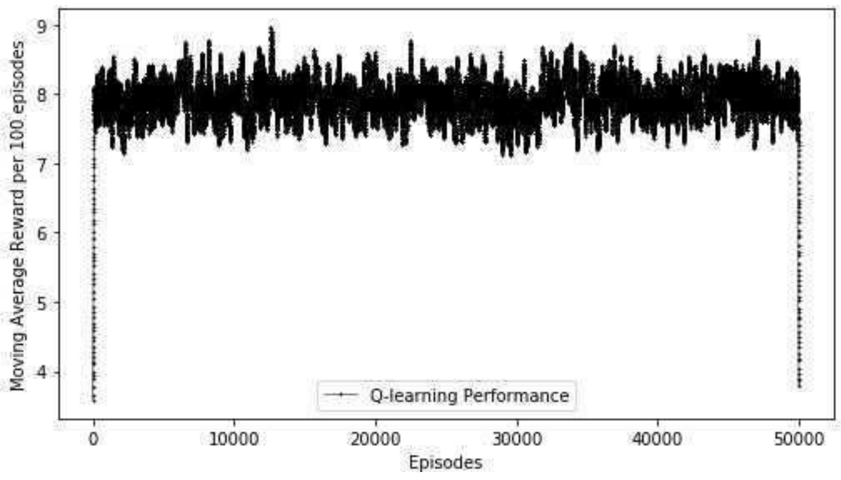
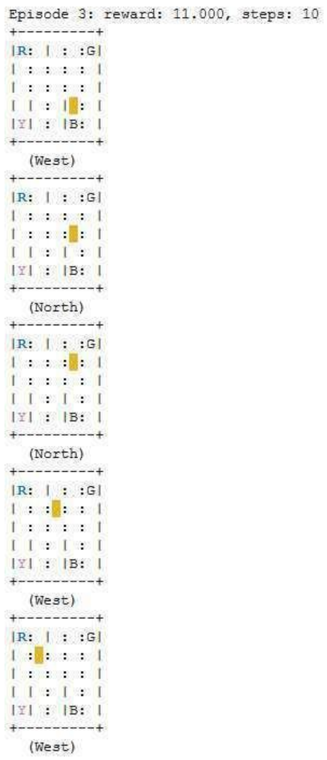


**Figure 13:** The Test phase Moving Averaged Reward for the DQN over 100k episodes

Q-learning and DQN tends to have the same potential in test phase with close Maximum rewards reached.

Test gaming based on the trained agent:

**Source.** Own elaboration



```
Episode 3: reward: 11.000, steps: 10
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
```

**Figure 14:** Results of test gaming based on the trained agent

**Distributional RL.** The common reinforcement learning algorithms tend to use the average of the Q value as target, that is bit arbitrary in some sense, the Q values are not dense around the mean in general. Q values diversity could be tracked with the learning procedure of the whole distribution.

The concept of Distributional RL is to learn a distribution of Q-values different that the average one. The Approximation of this learned distribution noted $d_t$ is define on discrete support grid $z\_$.

$$z_i = V_{min} + \frac{i}{N_{atom} - 1}(V_{max} - V_{min}) \tag{21}$$

with a probability mass $p_\theta^i(s_t, a_t)$ on each atom $i$ of the grid such that the approximation of the distribution could be represented within this scalar product projection:

$$d_t = (z, p_\theta(s_t, a_t)) \tag{22}$$

At this stage the calibration procedure is to find the set of parameters $\theta$ that would closely match the actual distribution return.

**Learning procedure of the distribution:** The distributional variant of the Q learning could be seen as follow updating of the Bellman equation within the distribution $Z^*$ instead of the average:

$$Z^*(s,a) = r(s,a) + \gamma Z^*(s',a') \qquad (23)$$

Leading to target distribution:

$$d_t' = \left(r_{t+1} + \gamma_{t+1} z, p_{\theta^-}\left(s_{t+1}, \overline{a^*}_{t+1}\right)\right) \qquad (24)$$

where $\overline{a^*}_{t+1} = argmax_a Q_{\theta'}(s_{t+1}, a)$ is the greedy action with respect to the average action value is state $s_{t+1}$.

Loss for calibration procedure is expressed then is term of Kullback-Leibler divergence:

$$L = D_{KL}(\Phi_z \, d'_t || \, d_t) \qquad (25)$$

where $\Phi_z$ is the projection of the target distribution onto the grid support $z$.

**Global Learning procedure for the network:** As for vanilla DQN we can use the frozen parameter copy $\theta^-$ to construct the distribution, then the parametrized distribution can be expressed as a neural network with enriched output dimension associated to each action value associated to each corresponding atoms value, so output exit is $N_{atoms} * N_{actions}$. Soft max is then exercised for each action dimension to insure that the output action distribution is well normalized.

**Hyper parameters:** The grid builder one; the support parameters $V_{min}, V_{max}, N_{atom}$ and the optimizer routine one.

## Rainbow agent

In the previous section we presented all the current known improvement of the Vanilla DQN algorithm, most of this improvement has been done in the TD error enhancement. Each improvement method of the DQN described upper is a color of our rainbow, one may then try to combine all these previous improvement to one Algorithm.

The result is that the Combination beats all the previous methods by a comfortable advance.

Rainbow agent is then the combination of these 7 technologies: DQN; Double DQN; PER Dueling Network; Noisy Network; Distributional RL and the multi-step Q-Learning.

This combination method demonstrates an impressive performance on Atari 2600 benchmark.

**Source.** Hessel et *al.* (2018)

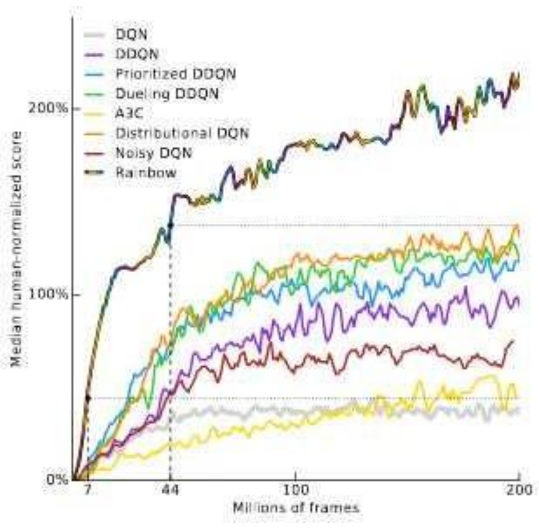


**Figure 15:** Median human-normalized performance

We will present the efficiency of the Rainbow agent within sonic environment game.

In videos games, Rainbow agent is a revolution, some contest in retro game environment has emerged to highlight the performance of the Rainbow agent. Sonic is the most known and Rainbow agent is clearly not beaten and we emphasize that, however, as rainbow agent is a combination, integration would not be so simple because there is no independence of each component and we need to understand first what the **rainbow recipes** is and then using the **Ablation method,** rank by importance the component of our combination that integrated the Rainbow agent.

Here is how it can be put into operation Rainbow:

- First take the Distributional RL enhancement and compose it within the Multi-step Learning leading to this Loss representation with the KL divergence:

$$d_t(n) = \left(r_t^{(n)} + \gamma_t^{(n)} z, Q_{\theta'}(s_{t+n}, a_{t+n}^*)\right) \tag{26}$$

$$\boldsymbol{ResultingLoss} = D_{KL}\left(D_{KL}\left(\phi_z d_t^{(n)} || d_t\right)\right) \tag{27}$$

- We obtain a Multi-step distributional Loss, we then use the Double Q-Learning.
- We take our new object and we map it into the PER, by changing the TD error within the KL divergence obtained in 2, this gives us a KL loss with priority:

$$p_t^{KL} \propto \left(D_{KL}\left(\Phi_z d_t^{(n)} || d_t\right)\right)^{\omega} \quad (28)$$

Using a KL divergence is a dissimilarity measure for two distributions provide us a way for weighting in Prioritized Replay in term of relevant loss of information.

We do have our plate, what about our own to cook it? in other term how to deal this the architecture?

- For the network architecture: Use the Dueling Network within the SoftMax Layer and change the Linear Network by the Noisy one.
- Then enjoy your Rainbow integrated Agent.

**Rainbow DQN scalping:** The cooking is not straightforward, we will try to be more explicative particularly for the **Prioritized Replay** as this is the most important component, even in practice this technology would be our leverage.

**Rainbow is core based on an application of experience replay in distributional setting:** The measure of **experience importance** must be genuinely provided; the major idea is inherited from the Vanilla DQN where the priority is just loss this experience:

$$L = D_{KL}(\phi_z \, \mathrm{d}^{(n)}{}_t || \, \mathrm{d}_t) \quad (29)$$

To combine noisy networks with double DQN heuristic, it is proposed to re sample noise on each forward pass through its copy for target computation. This decision implies that action selection, action evaluation and network utilization are independent and stochastic steps.

**Combination with the Distributional RL and the dueling DQN:** The main idea is to model the advantage $A^*(s,a,\theta)$ within a distributional setting. The Rainbow network has two heads: The value stream $V(s,\theta)$ and the advantage stream $A(s,a,\theta)$.

These streams are integrated in respect of the equation (2.29) & (2.30) in the Dueling network architecture (Wang, Z. 2015) with a particularity **softmax is applied across atoms dimension to guarantee that the output would be a categorical distribution**.

$$p_\theta^i \propto exp\left(V(s,\theta)_i + A(s,a,\theta)_i - \frac{1}{|A|}\sum_a A(s,a,\theta)_i\right) \qquad (30)$$

**Rainbow Algorithm:** We present here in pseudo code the way to build our special Agent.

- **Hyper parameters setting**
- **Initialization:**

1. Initialize the weight $\theta$ of the neural net arbitrary
2. Initialize$\theta^{-} \leftarrow \theta$
3. Precompute the support grid

$$z_i = V_{min} + \frac{i}{N_{atom}-1}(V_{max} - V_{min}) \qquad (31)$$

1. **On each interaction step:**

1 Select the action a within this criterion (**Double DQN within noisy Network**):

$$a = argmax_a \sum_i z_i p_\theta^i(s', a, \epsilon) = argmax_a Q_{\theta^-}(s', a, \epsilon) \qquad (32)$$

$$\epsilon \sim N(0, I) \qquad (33)$$

2 Observe the experience tuple $(s, a, r', s')$ and construct the $s$ -step transition T: **(Multi-step n -learning).**

$$T = \left(s, a, \sum_{j=0}^{n} \gamma^{(j)} r^{(j+1)}, s^{(n)}\right) \qquad (34)$$

Add it to the experience replay with priority using this criterion: **(Distribution RL & PER).**

$$p_t^{KL} \propto \left(D_{KL}\left(\phi_z d_T^{(n)} || d_t\right)\right)^{\omega} \qquad (35)$$

$$\max_T p_T^{KL} \qquad (36)$$

3 Sample batch of size B from the experience replay buffer using the probability

$$P(T) \propto p_t^{KL} \qquad (37)$$

4 Compute weights for the batch and for each transition $T = (s, a, \bar{r}, \bar{s})$ from the batch compute the target:

$$\epsilon_1, \epsilon_2 \sim N(0, I) \qquad (66)$$

$$Proba\left(y(T) = \bar{r} + \gamma^{(n)} z_i\right) = p_\theta^i\left(\bar{s}, argmax_{\bar{a}} \sum_i z_i p_\theta^i(\bar{s}, \bar{a}, \epsilon_1), \epsilon_2\right) \qquad (38)$$

5 Project $y(T)$ on the support $\{z_0, \ldots, z_{N_{atom}-1}\}$ and update the transition priorities:

$$p_T \leftarrow D_{KL}(\phi_z d_T^{(n)} || d_T) \qquad (39)$$

6 Compute the Loss:

$$L = \frac{1}{B}\sum_T w(T) p_T \qquad (40)$$

7 if step $t$ mod $K = 0, \theta^- \leftarrow \theta$.

## Results and performance evaluation of the Rainbow agent

**Playground:** The evaluation area for all the agents would be the Atari 2600 games from the arcade learning environment.

**Score Metric:** The average score of the agents is evaluated during the training phase each 1M steps, episodes are truncated at 108K frames equivalent of 30minutes of simulated play. Agent score is rescaled per games in respect of human expert performance, meaning 100% is a human expert and 0% would be a pure random agent, we will call this score per game as Normalized score.

Normalized score can be aggregated across the game set, we then use the median human normalized performance across all the game. But this is not enough are this is not a real relevant measure as some agent could clearly outperform the human benchmark across some games. Then we decide to consider the median human tracker at particular environment steps rather as metric split in two regimes: no ops start regime and human starts regime.

**Baselines:** Vanilla DQN is the benchmark, we report the results for the Rainbow integrated vs All its margins.

**Source.** Hessel et *al.* (2018)

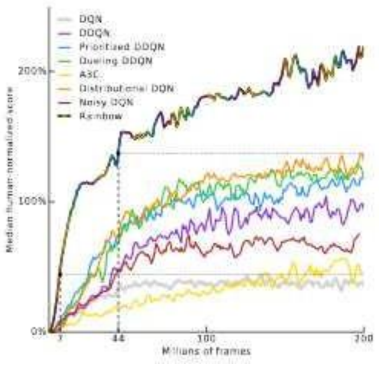


**Figure 16:** Median human-normalized performance

**Source.** Hessel et *al.* (2018)

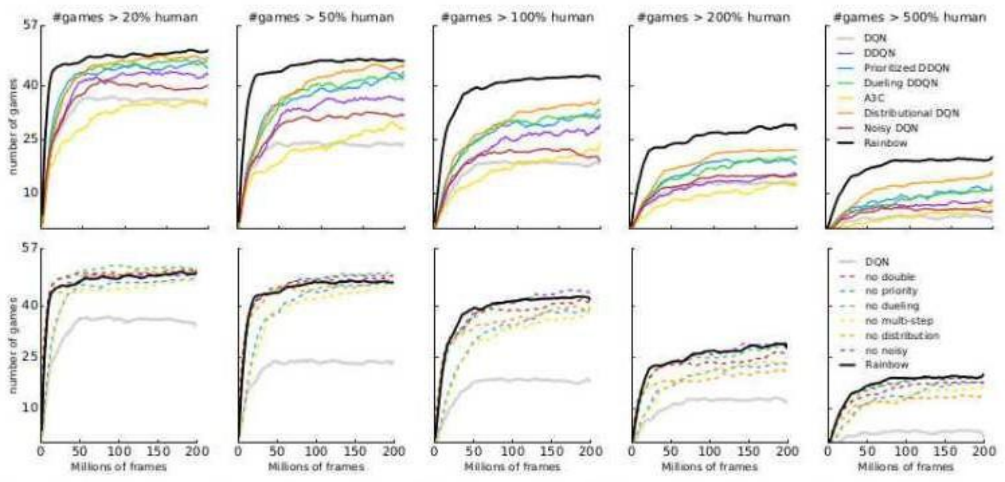


**Figure 17:** Each plot shows, for several agents, the number of games where they have achieved at least a given fraction of human performance, as a function of time

## Learning Curve for all games

**Source.** Hessel et*al.*(2018)

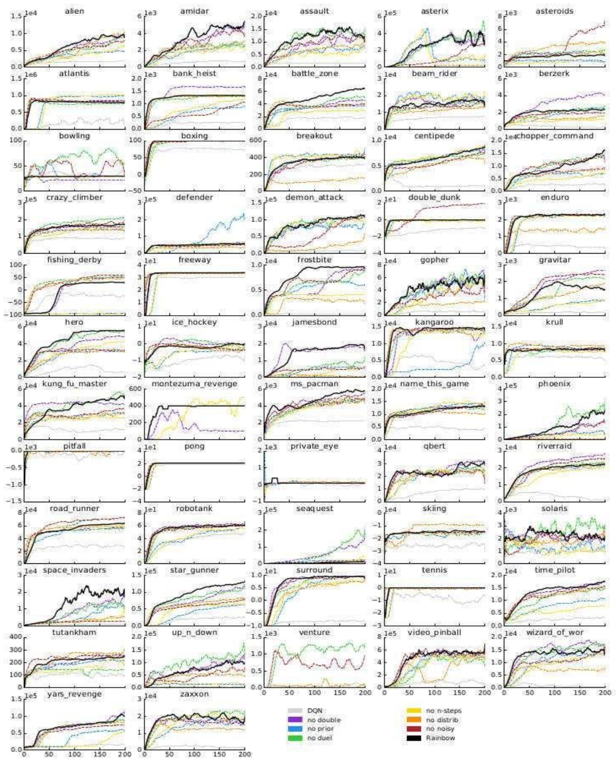


**Figure 18:** Learning curves for Rainbow and its ablations, for each individual games

**Results:** Rainbow beats all other methods used on their own, both in learning speed and maximum skill level. It performs far better than the classic DQN approach. Even all kind of ablated still outperform the DQN benchmark.

**Ablation results:** As Rainbow agent is real "Michelin recipe", one may need to understand what would be the weight of each ingredient, do to so , we make a ablation of each ingredient and we test the plate flavor.

**Source.** Hessel et *al.* (2018)

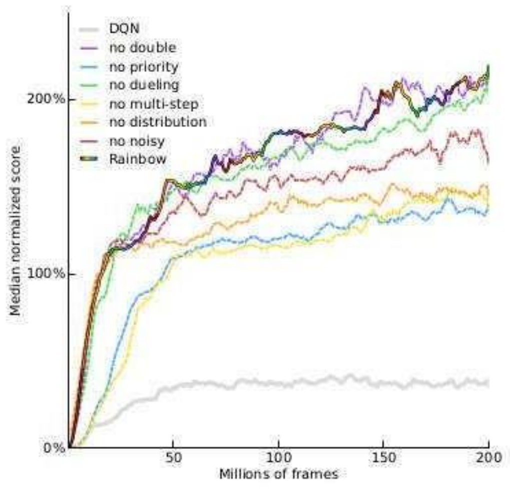
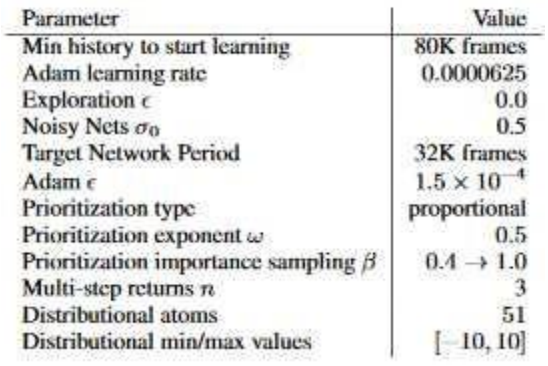


**Figure 19:** Median human-normalized performance across 57 Atari games, as function of time

**Removing the priority replay, multi-step learning or distributional RL significantly worsens the performance:** Clearly **priority replay** this is our main ingredient to provide an outperforming score.

Removing noise nets also harms the performance, although a bit less and removing double Q-learning or dueling networks seems to have no significant effect.

## Rainbow Hyper-parameters calibration



| Parameter | Value |
|---|---|
| Min history to start learning | 80K frames |
| Adam learning rate | 0.0000625 |
| Exploration $\epsilon$ | 0.0 |
| Noisy Nets $\sigma_0$ | 0.5 |
| Target Network Period | 32K frames |
| Adam $\epsilon$ | $1.5 \times 10^{-4}$ |
| Prioritization type | proportional |
| Prioritization exponent $\omega$ | 0.5 |
| Prioritization importance sampling $\beta$ | $0.4 \rightarrow 1.0$ |
| Multi-step returns $n$ | 3 |
| Distributional atoms | 51 |
| Distributional min/max values | $[-10, 10]$ |

**Figura 20:** Rainbow hyper-parameters

*2.* The Batch size *B.*

*3.* The Target Network Update Frequency *K.*

*4.* The multi-step *n.*

5. Degree of prioritized experience replay $\omega$ (Could be seen as persistence of an experience sample).

6. Importance sampling bias correction for the prioritized replay $\beta(t)$.

7. The neural network, the SGD optimizer parameters **ADAM.**

*8.* The support grid parameters $V_{min}, V_{max}, N_{atom}$.

The hyper parameters calibration procedure is carrying all the pattern of each component in term of efficiency, ablation could give an idea of some in term of importance weighting. Naive method resulting from a pure combination of all the rainbow would be rubbish in term of sensitivity understanding, we then decide to perform just a sample of them for tuning procedure.

Iterative coordinate manual sensitivity discovering is done on the sampled set of hyperparameters:

1. For Vanilla DQN, 200k of frames size is empirical enough to reduce the correlation in the updates, combining with prioritized replay we can relax then the batch size to 80k.

2. Exploration policy calibration is driven by our noisy nets hyperparameters, we use the initial $\sigma_0 = 0.5$. If one chooses not to use the noisy net (for ablation test as example) the greedy policy

will be selected within a time decay $\varepsilon(t)$ function that would vanish at 0.01 in the first 250k frames.

3. Adam optimizer parameter use $\alpha, \varepsilon$ value: $\varepsilon = 1.5 \times 10^{-4}$; $\{\alpha_{DQN}/2, \alpha_{DQN}/4, \alpha_{DQN}/6\}$ where $\alpha_{DQN}$ is the default DQN ADAM default parameters, for rainbow $\alpha_{DQN}/4$ is picked.

4. For replay prioritization, the priority parameter is set to $\omega = 0.5$ and then linear increase of the sampling parameter $\beta$ is tested on the range [0.4, 1] on the training phase. $\omega$ is robust in term of efficiency, this is a pretty good news.

5. Multi-step parameter is the most sensitive one, best performer is selected at $n = 3$.

6. Hyper parameters are robust when set across all the games, so Rainbow is a genuine single agent.

**Recent proof of concept: Sonic and rainbow agent baseline for OPENAI contest**

As a real proof of concept during the Last two years, many contest has emerged, based on ranking the most important score in sonic level Rooms. Most of the participant used the Rainbow agent and tuned the hyperparameters to be present in the top ranks: PER hyper parameters are leveraging inputs and the Target Network Update Frequency $K$ set at 8k instead of the 32K tends to help also. From experience, the prioritization is the most important component, then to improve the agent performance, one may need to focus on it hyperparameters setting and thing about how to use the rainbow as a new baseline by combining other technology to this new baseline.

In OpenAI contest rainbow is provided to the participant as baseline as well **PPO (Proximal Policy Optimization).**

**Participants experience in Sonic**: Generally speaking, the fact that it is off-policy makes it more sample efficient (theoretically speaking) than on-policy algorithms such as PPO. Hence, it is more suited to tackle problems with less training examples. On the flip side, as with other DQN methods, Rainbow comes with many hyperparameters so it is notoriously hard to tune and not as stable as PPO.

**A PPO fan Participant testify:** "After the contest ended, **I realized that a score of 6000 can be achieved simply by tuning 2 hyperparameters of the baseline Rainbow model, with no pretraining involved**. (As benchmark, human score is close to 7000)

The lesson to take is that for the sake of the competition always try to go for the low hanging fruits first. It’s not that hyperparameters tuning is easy but it is certainly more straightforward as compared to implementing more complex methods. And I am quite surprised that a well-tuned Rainbow can achieve such a high score. I did try to tune PPO hyperparameters (e.g., minibatch size, number of epochs, entropy constant, reward scaling factor, etc.) but could not find a better combination than the default ones.

## Acknowledgement

I would like to thank my colleague Jonathan Lévy, who kindly shared with me his experience and knowledge of Data Science as well as in helping to compress certain training algorithms.

## Referencias

1. **FORTUNATO, M., AZAR, M. G., PIOT, B., MENICK, J., OSBAND, I., GRAVES, A., ... & LEGG, S.** (2017). Noisy networks for exploration. arXiv preprint arXiv:1706.10295..
2. **GRIGSBY, J.** (2018). “Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning.” Retrieved (https://towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814).
3. **HESSEL, M., MODAYIL, J., VAN HASSELT, H., SCHAUL, T., OSTROVSKI, G., DABNEY, W., ... & SILVER, D.** (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.
4. **SIMONINI, T.** “Deep Q Learning with Atari© Space Invaders©”. Retrieved (https://simoninithomas.github.io/Deep_reinforcement_learning_Course/).
5. **SUTTON, R. S.** (1988). Learning to predict by the methods of temporal differences. *Machine learning*, *3*(1), 9-44.
6. **SUTTON, R. S., AND BARTO, A. G.** (1998). Reinforcement Learning: An Introduction. The MIT press, Cambridge MA.
7. **TALKINGCOMICSSITE.** (2020). “Megadrive Programming: Hello World.” Retrieved (https://8bitheaven.home.blog/2020/02/01/megadrive-programming-hello-world/).
8. **TURNER, A. J., & MILLER, J. F.** (2015). Introducing a cross platform open source cartesian genetic programming library. Genetic Programming and Evolvable Machines, 16(1), 83-91.

9. **VAN HASSELT, H., GUEZ, A., & SILVER, D.** (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

10. **WANG, Z., SCHAUL, T., HESSEL, M., HASSELT, H., LANCTOT, M., & FREITAS, N.** (2016, June). Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003). PMLR.

11. **ZHANG, T.** (2021). "Deep Q-network." Retrieved (https://github.com/moduIo/Deep-Q-network).