Hugo DEBES
Rita-Mathilda KABRO
first_name.last_name@polytechnique.edu

Rainbow: A Paper Analysis

Reinforcement Learning

02/18/23

Abstract

This document is a study of the paper **Rainbow: Combining Improvements in Deep Reinforcement Learning** [6] written in 2017 by Hessel et al. from DeepMind. It aims at analysing their work and contributions to the reinforcement learning field while presenting our implementation.

Contents

Introduction

Reinforcement Learning has been one of the most active research fields attracting many scientists. From the founding book of Sutton and Barto [12] to groundbreaking articles from several companies such as DeepMind or OpenAI, it has been complicated to keep a state-of-the-art model that includes each proposed improvement. This paper's goal is to combine a pre-selected list of improvements to a base model in order to surpass the state-of-the-art model at that time. In the first part, we'll analyze the background and previous works that led to the implementation of the DQN algorithm [9]. Then, we'll discuss one by one the selected improvements to this model by the researchers weighing the pros and the cons of each method. Finally, we'll present the implementation of the Rainbow network which is a combination of all the methods.

Before jumping into the article, we wanted to ask ourselves what were the motivation of the authors. Why did they want to write this paper? Why could their approach be a success or a failure ?

Combining multiple models has long been a possible and successful approach when it comes to Artificial Intelligence. We have seen different models being combined in other fields like in protein classification. For example, Gligorijevic et al. [4] proposed a model combining sequence-based LSTM with graph-based GCONV to accurately predict where the protein was effective. In our case, the changes are more precise as they also touched the model itself by altering its loss for example. It seemed intuitive that changing a part of the network with a refined one could lead to an overall better result. For example, refining the loss function by adding penalization terms helped deep learning methods to train better and find a more robust solution. The preselection work made by the authors resided in finding sufficient improvements to the model while being a local change. The paper would have been unreadable if different changes overstepped too much on other parts of the network and lacked independence. Finally, the selected improvements were coded using the same framework which is PyTorch [10] which eased the experiments.

On the other hand, the complexification of a model can also lead to disappointing results. The reasons could be multi-faceted as the training could be harder or the computational cost of the algorithm preventive. Also, the researchers could not have excluded that the combination of methods would be a vain effort compared to more innovative methods replacing the current standard model. For example, the apparition of the Transformer[14] architecture out-shined new development made on recurring models in favour of the attention mechanism. The ability to parallelize more than classic RNN caught the attention of many researchers opening new development capabilities.

As we will see, Hessel et al. manage to greatly improve the DQN model. If the approach had not been successful, the discovered dependency between parts of the model that we thought were interchangeable could have been as interesting to the Reinforcement Learning field.

1 Background

1.1 Markov Decision Process

Reinforcement Learning is a field of Artificial Intelligence aiming to solve problems based on a sequence of decisions. It is not a supervised model as we might not have immediate feedback mandatory to adjust the weights. RL can not be categorized as an unsupervised field since we have a defined goal. Knowing this specificity, RL researchers needed to find a new approach and model the different parts. The representation is as followed: an Agent evolves in an Environment characterized by a State S_t at a discrete time t. The Agent performs an action A_t that will trigger a new state S_{t+1} of the Environment and a Reward R_{t+1}. This discrete-time stochastic control process is called a Markov Decision Process.

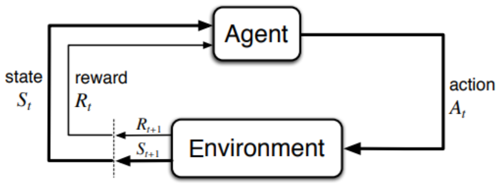


Figure 1: Markov Decision Process (MDP)

To select the action to perform, the Agent follows a policy π. Its goal is then to find the optimal policy to maximize its expected return over a discounted setting. This setting, intuitively, will decrease the reward in the function of the time but introduces a hyper-parameter $\gamma \in [0, 1]$ to the model. The Agent's goal can be written as

$$max_\pi E_\pi[G_t] = max_\pi E_\pi[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k]$$

. The policy π, a probability distribution over actions given a state, cannot be computed using a close form. It is common to define a state-value function, the expected return knowing the state $v_\pi = E[G_t|S_t = s]$, and an action-value function, the expected return knowing a state and an action, $q_\pi = E[G_t|S_t = s, A_t = a]$. A nice property tells us that finding the optimal state-value or action-value function will give us the optimal policy

$$v_*(s) = max_\pi v_\pi(s), q_*(s, a) = max_\pi q_\pi(s, a)$$

.
This optimality problem can be solved using a framework called Generalized Policy Iteration (GPI). Union of policy evaluation and policy improvement, they alternate to converge to the solution. The policy evaluation assesses the current policy to determine its value function v_π while the policy improvement selects the action to maximize the estimated value function.

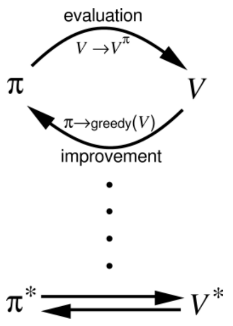


Figure 2: Generalized Policy Iteration (GPI)

1.2 Monte-Carlo Methods and Q-Learning

• From Monte-Carlo Evaluation...

The Monte-Carlo Method is model-free which means it will not be using a model to plan actions before they are taken, specifically the transition probability $p(s', r|s, a)$. The idea is to come up with the optimal Q-Table.

This object stores the action-value function for a given policy. The rows represent each state and the columns each possible action while the intersection is the expected return.

During the evaluation, the goal is using trajectories under a policy π, to estimate q_π. It uses a trick where under this specific phase of the GPI, we can see a q_π estimation as a v_π estimation to simplify the model to a Markov Reward Process (MRP). The M trajectories sample of this process is averaged to approximate the state-value function. The return is averaged over all times by the number of times the state s was visited

$$v_\pi(s) = E_\pi[G_t|S_t = s] \approx \frac{1}{C(s)} \sum_{m=1}^{M} \sum_{\tau=0}^{T_m-1} 1[s_\tau^m = s] g_\tau^m = V_\pi(s)$$

where T_m represents the number of steps for the trajectory m, the indicator $1[s_\tau^m = s]$ selects the states whose values were computed and g_τ^m the returns at time τ. In other words we sum, all the returns in the data that followed a state s divided by $C(s)$ the number of times s was visited. As it is more convenient to work with an update rule on an episode-by-episode basis, we have

$$V(s_t^m) \leftarrow V(s_t^m) + \alpha(g_t^m - V(s_t^m))$$

. Lastly, $C(s)$ was replaced by a constant α called the learning rate as it did not impact the convergence and is easier to compute. This is called the constant α-MC.

Let's recall that we will update the policy in the process so the Markov Reward Process cannot hold at every phase. Fortunately, the state value algorithm is equivalent to the action value function. We can now update the Q-Table the action

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha(g_t^m - Q(s_t^m, a_t^m))$$

- **.. to Q-Learning**

However, the training can be relatively slow in case of long trajectories because we update only after the episode is done. One way to overcome this flaw is the n-step temporal difference. The idea is to update the Q-Table for each n step of the trajectory. The future states that are yet to be seen can be approximated using the state-action function which is by definition the expected return for a given pair. The target g_t^m becomes

$$g_{t:t+n}^m = r_{t+1}^m + \gamma r_{t+2}^m + ... + \gamma^{n-1} r_{t+n}^m + \gamma^n Q(s_{t+n}^m, a_{t+n}^m)$$

Finally, the standard Q-Learning algorithm used a slightly different version with a maximum to go from an on-policy method to an off-policy method

$$g_{t:t+n}^m = r_{t+1}^m + \gamma r_{t+2}^m + ... + \gamma^{n-1} r_{t+n}^m + \gamma^n max_a Q(s_{t+n}^m, a)$$

- **On vs Off Policy and taking an action**

In an On-policy method, the behaviour policy $b(a|s)$ that is used to generate the data is the same as the target policy $\pi(a|s)$ that is being improved and evaluated. Algorithms, where the behaviour and the target policies are different, are called Off-policy methods. They are convenient as we could reuse previously made trajectories to improve a different policy.

$$\begin{cases} \text{On-Policy Methods} & b = \pi \\ \text{Off-Policy Methods} & b \neq \pi \end{cases}$$

The evaluation updates the value of the Q-Table and the control part of the MC method tells us how to use it to select actions. The researchers faced a new tradeoff between Exploration and Exploitation. To discover optimal policies, we must explore all state-action pairs, but to get high returns, we must exploit known high-value pairs. One way to deal with it is to use an ϵ-greedy policy of Q.

$$A \leftarrow \begin{cases} argmax_a Q(a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

The refinement of the policy will drive the exploration of the model in its environment by selecting sub-optimal routes according to its current view. By taking different trajectories, the agent could discover new routes improving the return and closing the gap to the optimal policy.

1.3 Deep Q-Learning

Deep Q-Learning (DQN) is a Reinforcement Learning technique where the action-value function (Q-function) is estimated using a deep neural network. The network receives the state of an environment and outputs a Q-value for each possible action. The Q-value represents the expected reward of taking a particular action in a specific state. By continually updating Q-values through interaction with the environment, the DQN algorithm learns the policy that maximizes the total reward.

The mathematics behind the DQN can be broken down into two main components: the Bellman equation and the Q-network.

The Bellman equation defines the relationship between the expected future rewards and the current state-action value (Q-value). It states that the Q-value for a state-action pair (s, a) is equal to the immediate reward received after taking action a in state s, plus the expected future reward, discounted by a factor (γ):

$$Q(s_t^m, a_t^m) = r + \gamma \max Q(s_{t+1}^m, a_{t+1}^m)$$

where r is the immediate reward, γ is the discount factor, s_{t+1}^m is the next state, and a_{t+1}^m is the next action.

Why did researchers decide to integrate deep learning into reinforcement learning? As we were saying before, the agent will interact with the environment. In the case of visual environments like Atari games, the environments are complex and therefore the computation of the number of Q-value (one for each state-action pair) is very high, so it's important to integrate some deep neural network.

The Q-network is a deep neural network that approximates the Q-function. The input to the network is the state of the environment and the output is a Q-value for each possible action. The network is trained using a variant of supervised learning, where the target Q-value for each state-action pair is computed using the Bellman equation. The network is updated to minimize the mean squared error between the target Q-value and the predicted Q-value.

Here is an aspect of the architecture of the DQN model :

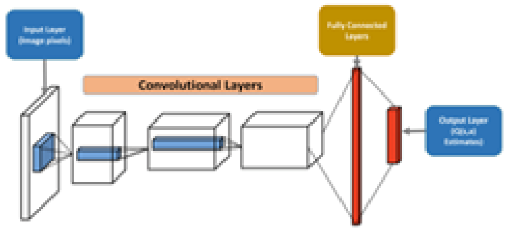


Figure 3: Architecture of a DQN

The architecture of the DQN consists of three main components: an input layer, several hidden layers, and an output layer.

The input layer receives the state representation of the environment. DQN is usually used in visual environments, the state representation is only the raw pixels.

The hidden layers consist of multiple usual neural networks layers that perform non-linear transformations on the input data. Because the environment is an image, those layers are composed of convolutional layers. These layers use activation functions such as ReLU, Tanh, or Sigmoid to introduce non-linearities into the model. The number of hidden layers, the number of neurons in each layer, and the choice of activation functions can all be tuned to improve the performance of the model.

The output layer consists of a single neuron for each possible action in the environment. The output of each neuron represents the predicted Q-value for a specific action in the current state. The action with the highest Q-value is selected as the best action to take in the current state.

The architecture of the model is trained through interaction with the environment, where the target Q-value for each state-action pair is computed using the Bellman equation. The parameters of the model are updated to minimize the mean squared error between the target Q-value and the predicted Q-value:

$$L = (r + \gamma \max Q(s_{t+1}^m, a_{t+1}^m; \theta) - Q(s_t^m, a_t^m; \theta))^2$$

where L is the loss, r is the immediate reward, γ is the discount factor, s_t^m is the current state, a_t^m is the action taken in the current state, s_{t+1}^m is the next state, a_{t+1}^m is the action taken in the next state, θ represents the parameters of the Q-network, and $\max Q(s_{t+1}^m, a_{t+1}^m; \theta)$ represents the maximum Q-value for the next state s_{t+1}^m and all possible actions a_{t+1}^m as estimated by the Q-network with parameters θ. The goal of the DQN algorithm is to minimize this loss function over many iterations of training.

When reading the article about Deep-Q-Learning [9] we denoted the impact of combining deep neural networks and Q-Learning. They apply the DQN approach to some Atari games, and their goal was to train the model by only providing the video input (raw pixels). Then they compared their results with some of the best-performing methods from the RL literature (Sarsa and Contigency). By comparing the average scores per game, the DQN approach outperforms the other learning methods.

2 Selected Improvements

In this section, we will describe each solution selected by the authors to improve the Deep Q-Learning algorithm.

2.1 Double DQN

In DQN, the target Q-values are calculated using the same network that produces the predictions, leading to over-optimistic estimates and unstable learning. Double DQN (DDQN) is used instead of regular DQN to address the issue of over-estimation of Q-values which can lead to poor performance.

DDQN separates the process of selecting the action and evaluating its value, which helps to reduce over-optimism and improve the stability of the learning process. In DDQN, the selection of the action is done by the current network, while its value is evaluated using a separate, fixed target network. This decoupling helps to produce more accurate estimates of the Q-values and results in more stable and reliable learning.

In DDQN, the target Q-values used to train the Q-network are computed using two separate Q-networks. One Q-network is used to select the best action in a given state (the ”online” network), and the other Q-network (the ”target” network) is used to estimate the Q-values for each possible action. The target network is updated periodically to ensure stability in the training process.

The mathematics behind the DDQN algorithm is very similar to the regular DQN algorithm. The key difference lies in the calculation of the target Q-value. In DQN, the target Q-value is calculated using the following formula:

$$Q(s_t^m, a_t^m) = r + \gamma \max Q(s_{t+1}^m, a_{t+1}^m)$$

where r is the immediate reward, γ is the discount factor, s_{t+1}^m is the next state, and a_{t+1}^m is the action that produces the highest Q-value in the next state s_{t+1}^m. The problem with this approach is that it can lead to the overestimation of the Q-values, as the network can sometimes produce high Q-values for suboptimal actions.

DDQN [13] solves this problem by using two separate networks: a target network and a main network. The main network is used to select the action, while the target network is used to calculate the target Q-value. The target Q-value is calculated as follows:

$$Q(s_t^m, a_t^m) = r + \gamma Q(s_{t+1}^m, \arg\max\left(Q(s_{t+1}^m, a_{t+1}^m; \theta), \theta'\right))$$

where θ is the main network's parameters and θ’ is the target network's parameters. This means that the target Q-value is calculated using the Q-values produced by the main network, but the action that is selected to maximize the target Q-value is chosen using the target network. This ensures that the target Q-value is not overestimated, as it is based on the Q-values produced by a separate, fixed network.

Here is an aspect of the DDQN's architecture:

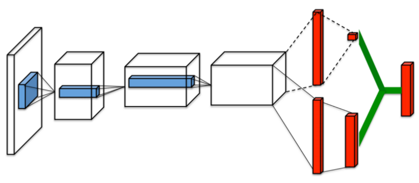


Figure 4: Architecture of a DDQN

In the paper [5], the searchers try the DDQN algorithm on several Atari games and compare them with the simple DQN algorithm. First, they compare the bias, let's consider a state s in which all the true optimal action values are equal at $Q_*(s,a) = V_*(s)$. They try to calculate the bias:

- **For the DQN** : $\max_a (Q(s,a)) - V_*(s)$
- **For the DDQN** : $Q(s, argmax_a(Q(s,a))) - V_*(s)$

After doing those calculations, they noted that the DDQN was nearly unbiased, whereas the DQN was biased.

On 57 Atari games, they tested the DQN and the DDQN algorithms, the scores were very different. The DDQN outperformed the DQN in all Atari games without any exceptions. The Double DQN algorithm appears to be more stable and reliable in difficult evaluations, indicating that it can adapt and that the solutions obtained are not just exploiting the determinism of the environment. This is a positive aspect as it suggests that the algorithm is making progress towards finding general solutions that are more robust and not just relying on a predetermined sequence of steps.

2.2 Prioritized Experience Replay

In the simplest Q-Learning algorithm described in the Background section, the agent updates its parameters while observing a stream of trajectories. One can even imagine discarding the data right after they have been processed by the learning procedure to save memory. Nevertheless, this approach suffers from several issues as explained in [11]:

- The updates are strongly correlated as they are part of the same episode. It breaks the i.i.d hypothesis used by many algorithms.
- The updates are focused on common episodes and forget rapidly rare events which can lead to a lack of generalization.

This issue wasn't unknown by the authors of [9] where they used experience replay to train the value function with a deep neural network. Experience replay proposed by Lin et all [7], stores past experiences in the form of a quadruple (s_t, a, s_{t+1}, r) where s corresponds to the current state and after the action a leading to a reward r. The training loop can include them again in the learning procedure. They showed that the network usually converges faster if the laws of the environment do not change too rapidly making the past inefficient. This adjustment controlled the first issue of temporal correlations and was used on the DQN algorithm. More precisely, they used a sliding window replay memory and sampled uniform transition.

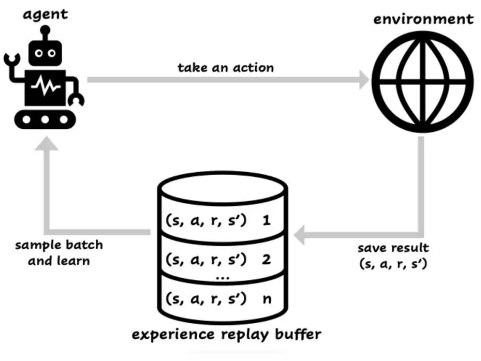


Figure 5: Experience Replay

The authors of [11] propose to replace the uniform sampling by prioritizing experiences where the agent has more to learn. The agent should be able to focus on more surprising events making the training faster. The key contribution of this paper is the criterion they proposed to assess the probability of each transition to be chosen from the pool of experiences, in other words, from which experiences the agent has more to learn. They used the magnitude of a transition's Temporal Difference error δ. It is a measure of how unexpected the outcome is.

$$p_t = R_{t+1} + \gamma_{t+1} max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t)$$

Temporal difference estimates the value function to update parameters before the episode is completed. The TD error specifies how different the estimation is from the new value. Intuitively, the higher the more unexpected the transition is which could lead to interested learning. One of the advantages is this criterion is already computed by Q-Learning or SARSA. Regrettably, the TD error is a poor estimate when rewards are noisy for example, mention the authors. Plus, transitions with low TD error may not be replayed for a long time and create a diversity issue.

Secondly, they refined the criterion by introducing stochastic Prioritization in opposition to the greedy TD-error. The idea is to introduce a priority between transitions while making sure of a non-zero probability for the low scores. The probability of sampling is written as followed

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where p_i is the priority of the transition i, α determines how much prioritization is used. There exist several ways to compute p_i based on the TD error and we will use

$$p_i = |\delta_i| + \epsilon$$

where ϵ prevents zero probabilities. Finally, the last step is to anneal the bias induced by this method as we do not select uniformly the transitions like the original DQN algorithm. The authors used importance sampling on the weights

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta$$

. By nature, the unbiased property of the updates is fundamental when approaching convergence. The importance of sampling correction is therefore increased over time by raising β to 1 at the end of training.

2.3 Dueling Network

The Dueling Network architecture is a variation of the Deep Q-Network (DQN) architecture used in reinforcement learning. It separates the network into two streams, one for estimating the value of being in a state (V-stream), and another for estimating the advantage of each action (A-stream). The final Q-values are then calculated by combining the value and advantage estimates. This architecture allows for faster and more efficient learning by explicitly representing the relative importance of each action in a state. In the traditional DQN architecture, the value function for a particular state-action pair is approximated by a single scalar output from the neural network. However, in the Dueling Network architecture, the value and advantage functions are separately estimated, allowing the network to focus on the relative importance of different actions in a state, leading to improve the convergence and the robustness.

In a standard Q-network, the Q-value for each state-action pair is estimated by a single neural network, where the input is the state of the environment and the output is a Q-value for each possible action.

In contrast, the dueling network architecture separates the estimation of the state values and the state-dependent action advantages. The state values represent how valuable it is to be in a certain state, regardless of the action taken. The state-dependent action advantages represent how much better or worse a certain action is compared to the average of all actions in a certain state. The outputs of the two streams are combined by a special layer that outputs the Q-values for each state-action pair.

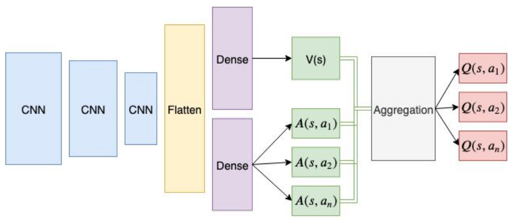


Figure 6: Architecture of a Dueling Network

The final Q-values are computed as the sum of the estimated state values and the estimated state-dependent action advantages, weighted by a learned parameter. This allows the network to focus on the relative values of different actions in a certain state, rather than the absolute values of all actions in all states.

$$Q(s_t^m, a_t^m; \theta, \alpha, \beta) = V(s_t^m; \theta, \beta) + (A(s_t^m, a_t^m; \theta, \alpha) - \max_{a_{t+1}^m \in |A|} (A(s_t^m, a_{t+1}^m; \theta, \alpha)))$$

An alternative way is to replace the max operator with an average:

$$Q(s_t^m, a_t^m; \theta, \alpha, \beta) = V(s_t^m; \theta, \beta) + (A(s_t^m, a_t^m; \theta, \alpha) - \frac{1}{|A|} \sum_{a_{t+1}^m} A(s_t^m, a_{t+1}^m; \theta, \alpha))$$

In the paper, [15] the searchers compare the results on simple environments and Atari games. First, they compare the error on the policy with 5, 10, and 20 actions. The dueling network outperforms a DQN with the performance gap increasing with the number of actions. The squared error decreases a lot faster. Then they compare the scores of 57 Atari games, again the dueling network outperforms the DQN, with a lot more improvement on the baseline.

The dueling architecture offers a key advantage in its ability to learn the state-value function. Unlike the DQN where only the value for one action is updated during Q value updates, the dueling network updates the value stream V with every update, leading to a more accurate approximation of the state values, which are crucial for temporal difference-based methods to be effective. This frequent updating of the value stream allows for better allocation of resources to V.

2.4 Noisy DQN

The Noisy DQN is a variant of the DQN that adds noise to the weights of the neural network to enable exploration during training. Exploration is an important aspect of reinforcement learning, as it allows the agent to discover new actions that might lead to higher rewards. In DQN, exploration is achieved by selecting actions based on an ϵ-greedy strategy, where the agent selects the action with the highest Q-value with probability 1-ϵ, and selects a random action with probability ϵ.

In Noisy DQN, exploration is achieved by adding noise to the weights of the neural network, which enables the agent to select actions that are not based solely on the current state. The noise is generated using a factorized Gaussian distribution that is parameterized by two vectors, one for the mean and one for the standard deviation. The noise is then added to the weights of the network using a special layer called the factorized Gaussian noise layer, which scales the noise by the standard deviation vector and adds it to the mean vector.

During training, the noise is annealed to zero over time, so that the agent becomes more reliant on the Q-values produced by the neural network. The final Q-values produced by the Noisy DQN are similar to those produced by DQN, but the addition of noise allows for more diverse exploration during training.

The Noisy DQN is designed to add noise to the network weights, making it explore the environment more effectively. By adding noise to the network, the Noisy DQN can sample more diverse actions, which helps the agent explore new parts of the state-action space that it might not otherwise have found. This allows the agent to learn a more robust and accurate policy, improving its performance in environments with a high level of stochasticity or when the optimal policy is not well defined. Compared to a DQN, which uses a fixed set of weights, the Noisy DQN can adapt its exploration strategy to the current state of the environment. This can make the Noisy DQN more effective in complex and dynamic environments, where the optimal policy may change over time or be influenced by many factors.

The noisy network is represented by the following function: $f(w, z, x)$ where w represents the vector of trainable weights in the network, z is a vector of noise parameters that are initialized randomly and fixed during the training process, and x is the input to the network. The output of the noisy network is given by:

$$f(w, z, x) = \phi(w + (z \odot w) * x)$$

where ϕ is the activation function, $\odot$ represents element-wise multiplication, and $z \odot w$ is a vector of noise parameters that are learned during training.

The noise parameters $z \odot w$ are updated using gradient descent during the training process and are optimized to minimize the loss function. The loss function used in the Noisy DQN algorithm is similar to the one used in the DQN algorithm, and is based on the Bellman equation:

$$L(w) = E[(r + \gamma \max (Q'(s_{t+1}^m, a_{t+1}^m; w \odot -)) - Q(s_t^m, a_t^m; w))^2]$$

where r is the reward received by the agent, γ is the discount factor, s_t^m is the current state, a_t^m is the action taken by the agent, s_{t+1}^m is the next state, a_{t+1}^m is the next action, $Q(s_t^m, a_t^m; w)$ is the Q-value of the current state-action pair, $Q'(s_{t+1}^m, a_{t+1}^m; w \odot -)$ is the target Q-value of the next state-action pair, and $w \odot -$ represents a set of fixed weights that are used to compute the target Q-value.

In the paper [3], the searchers compare the performances on 57 Atari games of some algorithms (DQN, Dueling network, and A3C) with and without a noisy layer. For the three algorithms, the noisy layer adds an improvement at most 48%. Having weights with greater uncertainty introduces more variability in the decisions made by the policy, which has the potential for exploratory actions

2.5 Learning from multi-step bootstrap targets in A3C

Asynchronous Advantage Actor-Critic (A3C) is a reinforcement learning algorithm that uses the concepts of both actor-critic and asynchronous methods.

Asynchronous methods are a class of algorithms where multiple agents (workers) simultaneously interact with the environment and update a shared model. In these methods, each worker collects their own experience in parallel and updates the shared model asynchronously, without waiting for other workers to finish. This allows the agents to process more data in parallel, which can significantly speed up the learning process and improve the final performance. In asynchronous methods, the order in which updates are applied to the shared model is not fixed and may vary from iteration to iteration, hence the name "asynchronous". A3C is one of the popular asynchronous reinforcement learning algorithms that utilize this approach.

The A3C algorithm combines an actor network that determines the actions to take and a critic network that estimates the value function. In A3C, the actor and critic networks are updated asynchronously, allowing multiple instances of the algorithm to run in parallel, which can improve sample efficiency and stability compared to a single-threaded implementation. A3C uses the advantage function to improve the stability and convergence of the algorithm and is well suited for use in complex environments where the optimal policy is difficult to determine. The A3C algorithm is a popular choice in deep reinforcement learning because of its simplicity, scalability, and the ability to learn directly from high-dimensional observations.

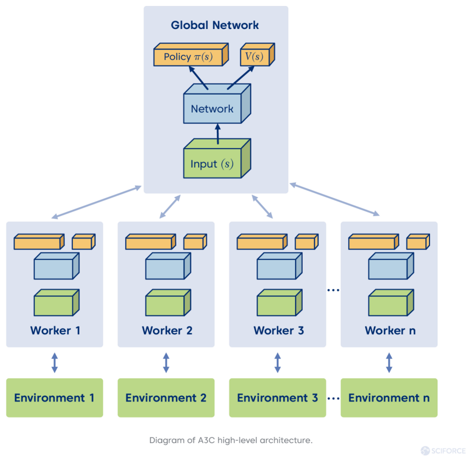


Figure 7: Architecture of a A3C

The A3C is designed to tackle the issue of sample inefficiency in traditional reinforcement learning methods such as DQN that learns from individual experiences. This architecture has been shown to work well for environments with high-dimensional state spaces (like Atari games), where function approximation is essential for effective learning.

The mathematics behind the A3C model is based on the Actor-Critic framework, which combines both the value-based and policy-based approaches. In A3C, the Actor is responsible for selecting actions, while the Critic evaluates the state-value function.

In A3C, the value function is estimated by the Critic, which predicts the expected cumulative reward of a given state-action pair. The value function is then used to improve the policy through policy gradient methods, where the gradient of the expected cumulative reward with respect to the policy parameters is estimated and used to update the policy. It involves estimating the gradient of the expected cumulative reward with respect to the policy parameters, using the estimated value function as a baseline. The gradient is estimated using the following formula:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log(\pi(a_t^m|s_t^m, \theta)) * Q(s_t^m, a_t^m)]$$

where $J(\theta)$ is the expected cumulative reward, $\pi(a_t^m|s_t^m, \theta)$ is the policy represented by the parameters θ, and $Q(s_t^m, a_t^m)$ is the state-value function estimated by the Critic. The parameters θ are then updated in the direction of the gradient, to maximize the expected cumulative reward.

In the paper [8], the searchers compared the performances of the A3C and other algorithms (like DQN, DDQN, Dueling network . . .). First, they tried to compare the score with the training time. They noted that the A3C algorithm succeeded faster to higher scores. Indeed, the A3C is way faster to train because it's an asynchronous method. This paper shows that using parallel actor learners to update a shared model had a stabilizing effect on the learning process.

2.6 Distributional Q-Learning

One of the major principles of reinforcement Learning is to maximize the expected return. However, the return might contain more information for the agent than a single average value. The authors [2] aim at removing the notion of value in favour of a distributional perspective. They demonstrate the benefits of learning an approximate distribution over an approximate expectation.

In classic reinforcement learning, we define the return of the model in a discounted setting $G_t^\gamma = \gamma^t R_t$ and maximizes its expected value. However, it is possible to work directly with the distribution by defining the return distribution or random return $Z^\pi(s_0) = \sum_{t=0} \gamma^t R(s_t, a_t)$. It is composed of 3 random variables

- $R(s_t, a_t) \sim P(r_t|s_t, a_t)$
- $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$
- $a_t \sim pi(a_t|s_t)$

Based on the random return, we can write new objectives which used different statistical operators like the variance or standard deviation. For example, we could maximise the mean-variance objective $E[Z] - \lambda V[Z]$.

During the policy evaluation, the expected return is replaced by the entire distribution:

- $Q(s, a) = E_P[R(s, a)] + \gamma E_{P,\pi}[Q(s', a')]$
- $Z(s, a) = R(s, a) + \gamma Z(s', a')$

One important thing the authors mentioned is how different the equation reacts when we deal with distribution rather than value. The following figure demonstrates how the multiplication by γ will shrink the support leading to a probable increase in probability. Then, adding the reward which is supposed constant for this example will move the support to the left.

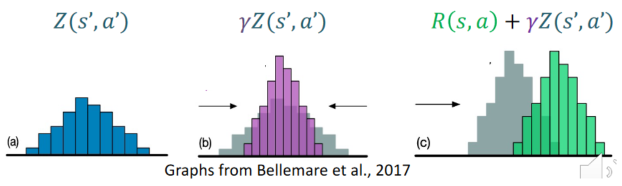


Figure 8: Distribution operations

The authors wrote the Bellman optimality equations in the function of the return distribution

$$Z(s, a) = R(s, a) + \gamma Z(s', argmax_{a'} E[Z(s', a')])$$

One of the drawbacks of such method is the algorithm will not converge to a unique distribution. We have a set of optimal return distributions and we might need to go through a sub-optimal path to find one. However, this drawback and the computational cost added by computed not only an expected value were largely overcome by the results the researchers got on the Atari Games benchmark. They introduced the categorical DQN that can be seen as computing a weighted ensemble of returns. Like an ensemble method, the errors in different returns/probabilities may cancel each other giving a more accurate estimate of the expectation. Formally, they proposed to model such distributions with probability masses placed on a discrete support vector z with N_{atoms} defined by $z_i = V_{min} + (i-1)\frac{V_{max}-V_{min}}{N-1}$ for $i \in \{1, ..., N_{atoms}\}$. The metric used for computing the difference between distributions is the Kullback-Leibler Divergence. The loss function becomes the cross entropy term of the KL divergence.

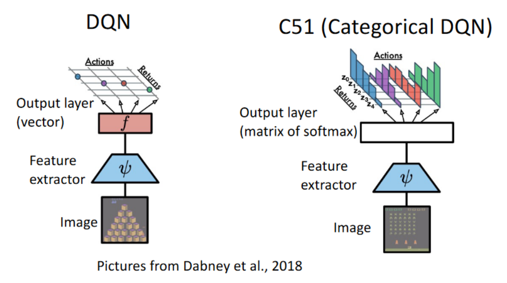


Figure 9: Comparison between the DQN and the Categorical DQN

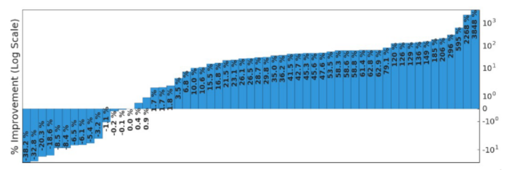


Figure 10: Double DQN vs Categorical DQN on Atari Games

This paper was a major improvement compared to the standard DQN as the agent outperformed in a vast majority of games. In the rainbow comparison, this method brings the biggest boost to performance.

3 Rainbow: the combined Agent

The DQN paved the way for many algorithms and ideas to solve Reinforcement Learning problems in complex environment. As we have seen, a lot of improvements were made to enhance it.

- **Double Q-Learning** : This addresses the problem of overestimation of Q-values by using two separate Q-functions to estimate the value of each action.
- **Prioritized Experience Replay** : This improves the sampling of experience data by assigning priorities to the transitions in the replay buffer and sampling more frequently from the transitions with higher priority.
- **Dueling Networks** : This separates the Q-value estimation into state-value and action-advantage streams, which allows the agent to learn which states are valuable without having to learn the effect of each action in each state.
- **Multi-step Learning** : This allows the agent to learn from the effect of multiple consecutive actions, rather than just the effect of the immediately subsequent action.
- **Distributional RL** : This involves representing the Q-value function as a probability distribution, rather than a single expected value. This can help account for the inherent uncertainty in the estimation of Q-values.
- **Noisy Networks** : This adds random noise to the weights of the neural network, which can encourage exploration and prevent the agent from becoming overconfident in its predictions.

Those methods are good and achieve impressive performances. However, it focuses only on one aspect of the DQN. Each of the algorithms selected in the Rainbow algorithm addresses a different aspect of the DQN algorithm that can lead to instability, slow learning, or suboptimal performance. One of the main question for the authors was the the practical implementation of each method into a single agent.

First, they replace the 1-step loss with a multi-step variant :

$$D_{KL}(\phi_z d_t^{(n)} || d_t)$$

where ϕ_z is the projection into z.

This loss is the result of the combination of the **distributional RL** and **the multi-step learning**.

Then they combined the multi-step distributional loss with **the double Q learning** by using the greedy action selected according to the online network and evaluating the action using the target network.

They also combined the **proportional prioritized replay** in the distributional settings by minimizing the KL loss instead of the usual TD error. They advocate that the KL Loss as priority might be more robust to noisy stochastic environments.

Then they use the **dueling network architecture** as the principal architecture of the Rainbow Algorithm by defining a value stream and advantage stream in N_{atoms}. To estimate the return's distributions, the streams are aggregated for each atom and passed through a softmax layer.
Finally, they replace all linear layers by their **noisy equivalents**:

$$y = (b + Wx) + (b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^w)x)$$

4 Dataset

4.1 The Atari Benchmark

To assess the performance of a model, and compare it with the state-of-the-art, the researchers need to work on a similar playground. This playground needs to be challenging to offer near-term advancements while being accessible to the community. In 2013, Bellemare et al. [1] released an arcade learning platform dedicated to evaluating Reinforcement Learning agents. It provides an interface to dozens of Atari 2600 Games, each of them with a different level of complexity with a wide range of genres.

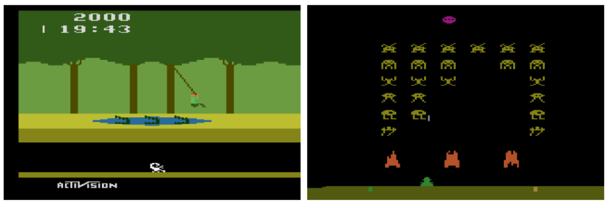


Figure 11: Screenshots of PITFALL! and Space Invaders

Games have been a well-known playground for RL where we need a proper definition for the environment, rewards, and actions. They are easy to model, interact with and allowed the researchers to focus mainly on their architecture. The Atari 2600, released in 1977, is interesting in many aspects for RL. First, the actions are limited with only 18 possibilities for the agent either using the joystick or a single button. Secondly, the image definition is only 160 pixels wide by 210 pixels high making it outdated for the current TV standard but usual for a neural network. The games themselves are simple to understand for a human, making the supervision of the agent easy.

Also, most of them resulted from arcade games where the goal of the game-maker was to develop fast-paced, rewarding games. The games would end early to encourage the player to play again at the cost of a couple of cents. The quick die and retry make the exploration easier as the agent will not spend too much time stuck on an episode. Plus, the emphasis on the score settles nicely with maximizing the expected return.

As we have seen, most of the improvements we treated were assessed against the Atari Benchmark. For the implementation part, we decided to focus on the Breakout game. The agent controls a bar and can fire a ball that will hit some bricks. Once hit, the brick disappears and gave the agent a reward that depends on the colour of the brick. On the other hand, the agent needs to move to the left or right to catch the ball. If he misses it, he loses one of his 5 starting balls. When he ran out of balls, the episode is done. The agent has then 4 different actions [Do nothing, Fire a ball, Move Left, Move Right].

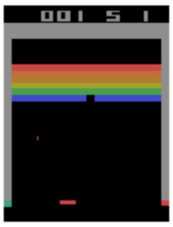


Figure 12: Screenshots of Atari Breakout

4.2 Gymnasium: A Practical Atari Environment

Gymnasium, previously known as gym, is an open-source python package aiming to provide a standard environment for researchers to develop Reinforcement Learning algorithms. Its goal is to democratize this field by providing easy access and a bug-free complex game environment. Initiated by OpenAi and currently maintained by the Farama Foundation, Gymnasium supports each Atari Games with clear documentation.

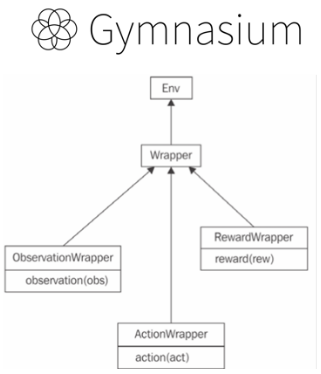


Figure 13: Environment and Wrappers of Gymnasium

Gymnasium is built on two-main principles to let us develop Deep Q Network and more advanced RL solutions. First, a basic environment is proposed using the function *gymnasium.make(env_id)*. Once created, we can simply interact with the agent in that environment with built-in functions such as *step* to perform an action. On top of this simple environment, the second key of the gymnasium package is the usage of wrappers. They are utility classes that can alter the way we interact with the environment or the agent. We used them when preprocessing the data as described in the next section. Lastly, the wrappers are useful to monitor the performance of the agent to retrieve each reward for example. We used the *RecordEpisodeStatistics* wrapper to obtain detailed statistics and the built-in *render()* function to watch our agent plays.

Lastly, this package can be easily combined with the state-of-the-art deep learning framework such as PyTorch or TensorFlow. For this project, we used PyTorch.

4.3 Data Preprocessing

As mentioned, the size of the Atari Games is 210x160x3 where the last columns correspond to the colours channel RGB. First of all, the colour of the image is irrelevant to the learning process. One step is to switch to a grey-scale image reducing the number of channels of the input size from 3 to 1. Then, it is common to reduce the image definition in a trade-off between precision and speed of the training. We down-sample the image to a lower definition and crop it to roughly capture the playing area. It leads to image of 84x84x1. In gymnasium, we use a wrapper called *AtariPreprocessing* that is standard for our environment. Plus, we scaled the value of the pixels from [0,255] to [0,1] which is better when implementing deep learning.

The researchers noted how it is impossible to represent the environment by only one image. In 12, we can not determine if the ball is moving up or moving down. We need to stack a sequence of images in order to hint the current flow of the environment to the agent. Commonly, 4 images are stacked together.

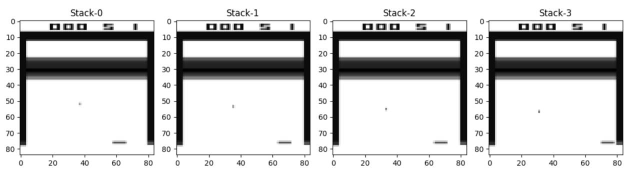


Figure 14: Stacking of an history of 4 frames

Lastly, we noted another hyper-parameter called frame-skipping also set to 4. It means the agent will only consider one-fourth of the actual screenshots of the game to then form the sequences that are overlapping. This operation was made using the wrapper *FrameStack*.

Before diving into our implementation, we will present the results the authors of the paper obtained after developing their own.

5 Rainbow Experiments

All the agents are evaluated on 57 Atari games. They compute the average scores of the agents evaluated during the training. Note that the scores are normalized such that 0% corresponds to a random agent and 100% the average score of a human expert.

There is a lot of hyper-parameters in all the rainbow's components, it was important to select them properly to have the best result after training. But because there is too many hyper-parameters the authors performed limited tuning instead of an exhaustive search.

They decided to start with the values used in the several papers introducing the method and tuned some of them by manual coordinate descent.

Parameter	Value
Min history to start learning	$80K$ frames
Adam learning rate	0.0000625
Exploration ϵ	0.0
Noisy Nets σ_0	0.5
Target Network Period	$32K$ frames
Adam ϵ	1.5×10^{-4}
Prioritization type	proportional
Prioritization exponent ω	0.5
Prioritization importance sampling β	$0.4 \rightarrow 1.0$
Multi-step returns n	3
Distributional atoms	51
Distributional min/max values	$[-10, 10]$

Table 1: Rainbow Hyper-parameters delivered by the searchers

For example, in the DQN, there is no learning during the first 200K frames to ensure sufficiently uncorrelated updates, but for the Rainbow algorithm, because they are using the prioritized replay, they can start the learning at 80K frames. They also choose to work with the Adam optimizer because it seems less sensitive to the learning rate and therefore not exhaustive tuning needed for this parameter. Let's note that all the hyper parameters are the same for all the Atari games, and that's what's important to keep in mind, that the rainbow algorithm is universal to all Atari games.

Now let's compare Rainbow's performance to the other components (15). We can see that the Rainbow algorithm largely outperforms the others as it is the only one beating 200% of scoring. Also, the Rainbow reaches a human-expert level extremely quickly with less than 20M frames. The other agents, if they ever reached it, needs around 75M frames which is far longer to train.

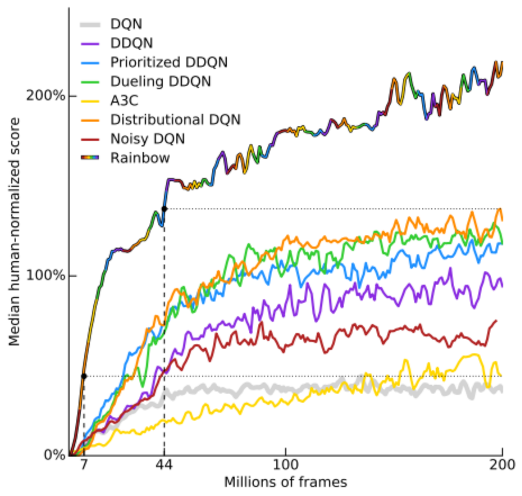


Figure 15: Median human normalized performance across 57 Atari games

They also did some ablation studies to identify the impact of each component of the rainbow on the performance (16). For this, they removed one component at the time and note the median normalized score.

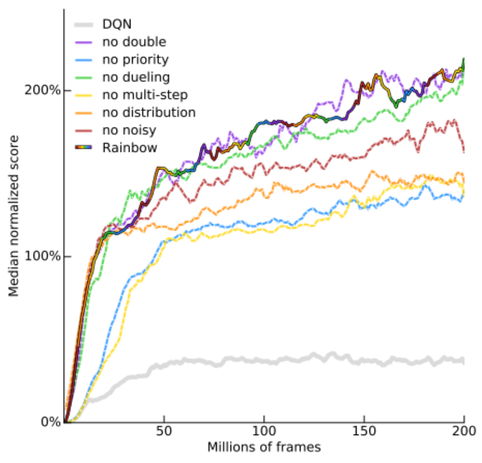


Figure 16: Median human normalized performance across 57 Atari games after ablation

We can detect two types of problem. The first one it's at the early stage of learning. We can see that the priority and the multi-step have a huge impact on the rainbow's agent. The removal of those two components causes a drop in the performances and at the early stage of the learning process. The second problem that we can denote is after a certain time in the learning process. Without the distribution or the noisy layer, the rainbow's agent performed as well as the full rainbow algorithm at the beginning, but after some frames, they started to stop learning.

This ablation study permit us to better understand the impact of each component for different types of games and therefore maybe adapt it for other applications. For example, as the distribution tend to have a higher impact on the advanced stage of the learning process, the performances on games that are on human level need it to have optimal performances. In the paper, they denote the influence of the components on each game, we cannot go through every performance because there is 57 games, and also because the impact of the different components is different for each of the games.

6 DQN-Rainbow Implementation

6.1 DQN Architecture

The code can be found Here

As a starter, we wanted to implement the Convolutional DQN architecture proposed in [9] by Mnih et al. in 2015. This model is the base of all the improvements selected by the researchers. Having no previous experience in Reinforcement Learning, its implementation has been a challenge for our team. We coded 4 different files that we will now present.

The *utils.py* file contains utility functions for the project to run smoothly. We create our environment alongside all the required wrappers in *make_env()*. The *make_replay_buffer()* function make the agent randomly play for a defined number of steps in the environment to create a pool of transition. It is necessary to reduce correlation before the agent can start its learning. Finally, we implemented two simple functions *load_state* and *save_state* to train our agent in multiple sessions. As we will see, the algorithm needs a massive set of data to perform well.

We then develop the model responsible for the agent behaviour in the *model.py* file. It is a class that inherits from the PyTorch Module composed of 4 functions. The *__init__()* and *forward()* are common in this framework to define the network and its forward propagation. The Network is composed of a succession of 3 convolutional layers followed by a flatten layer and lastly 2 linear layers. The ReLu was used as an activation function throughout the entire network. Contrary to a classical Deep Learning architecture we added the function *compute_loss()* that computes the targets and the smooth L1 Loss between it and the Q-values of the online network. Lastly, the *act()* function applies an $\epsilon-$Greedy Policy to choose an action.

The third file is the training loop of our model called *main()*. It is a loop that instantiates the environment, the target, and the online network while checking for previous training parameters. Then, we launch a new episode, compute the epsilon based on its decay, and choose an action using the online net. Then, we add the transition to our replay buffer and update the weight parameters after computing the loss between the target and online net on a random batch of transitions. After the TARGET_UPDATE_FREQ steps, we update the parameters of the target net with the online net. The fourth file enables us to render the Breakout game and watch our agent plays to assess its performance.

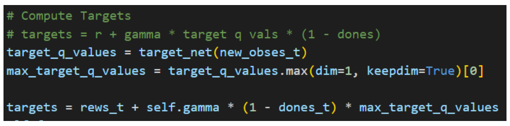
Hyper-Parameters	Value	Description
ENV_ID	”ALE/Breakout-v5”	Breakout environment
GAMMA	0.99	Discount Factor
BATCH_SIZE	32	Size of the training batch
BUFFER_SIZE	$1e6$	Size of the memory
MIN_REPLAY_SIZE	50 000	Size of the Random Memory to start learning
EPSILON_START	1.0	Start of Epsilon ($\epsilon-$Greedy Policy)
EPSILON_END	0.1	End of Epsilon ($\epsilon-$Greedy Policy)
EPSILON_DECAY	$5e4$	Decay Rate of Epsilon (start to finish)
TARGET_UPDATE	10 000	Number of step before updating the target
LR	$2.5e-4$	Learning rate of the Optimizer

Table 2: Hyper-Parameters of the DQN

6.2 Rainbow Improvements

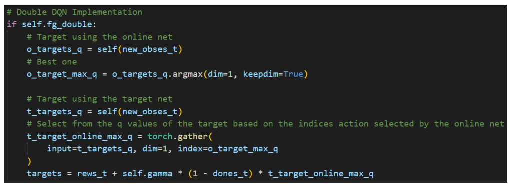
After implementing the classic DQN model, we looked at the improvements selected to create our Rainbow model. We implemented the paper one by one and respected some boolean variable to add or remove such implementations. For example, the variable *fg_double* enable us to activate or not the Double DQN.

• Double DQN
This innovation was the first that we worked on as we only need to modify the *compute_loss()* function. To avoid the model to overestimate the Q-values, we change the computation of the targets of our network. Rather than only using the target network, we select the Q-values of the target network based on the indices selected by the online net.



```
# Compute Targets
# targets = r + gamma * target q vals * (1 - dones)
target_q_values = target_net(new_obses_t)
max_target_q_values = target_q_values.max(dim=1, keepdim=True)[0]

targets = rews_t + self.gamma * (1 - dones_t) * max_target_q_values
```

Figure 17: Original DQN Implementation



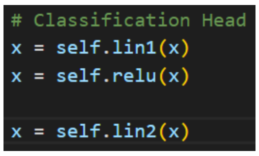
```
# Double DQN Implementation
if self.fg_double:
    # Target using the online net
    o_targets_q = self(new_obses_t)
    # Best one
    o_target_max_q = o_targets_q.argmax(dim=1, keepdim=True)

    # Target using the target net
    t_targets_q = self(new_obses_t)
    # Select from the q values of the target based on the indices action selected by the online net
    t_target_online_max_q = torch.gather(
        input=t_targets_q, dim=1, index=o_target_max_q
    )
    targets = rews_t + self.gamma * (1 - dones_t) * t_target_online_max_q
```

Figure 18: Double DQN Implementation

• Dueling Network
The Dueling Network is an architectural change of our agent. It consists in replacing the original classification head of the Neural Network into 3 distinct parts to create a state-value stream and action-advantage stream:

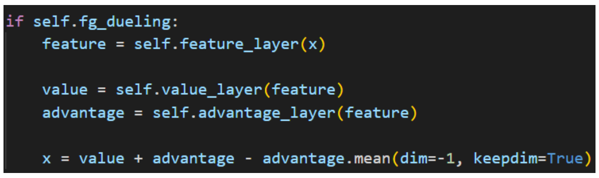
- The Feature Layer
- The Advantage Layer
- The Value Layer

We modified accordingly the *forward()* pass of our network.



```
# Classification Head
x = self.lin1(x)
x = self.relu(x)

x = self.lin2(x)
```

Figure 19: Original DQN Implementation



```
if self.fg_dueling:
    feature = self.feature_layer(x)

    value = self.value_layer(feature)
    advantage = self.advantage_layer(feature)

    x = value + advantage - advantage.mean(dim=-1, keepdim=True)
```

Figure 20: Dueling Network Implementation

• Prioritized Experience Replay Buffer
Creating a priority of the samples inside our memory buffer proved to be more challenging that we thought. The reason behind it is a practical limitation of complexity to compute the sum of priorities and find its minimum. A naive approach with a list would result in an $\mathcal{O}(n)$ algorithm which is expensive with a volume of millions. To overcome it, the researchers used two binary trees to reach a complexity of $\mathcal{O}(n \log n)$.

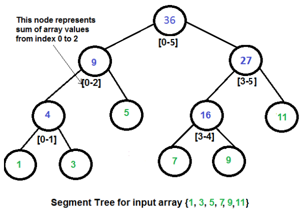

This node represents sum of array values from index 0 to 2
36
[0-5]
9
[0-2]
27
[3-5]
4
5
16
11
[0-1]
[3-4]
1
3
7
9
Segment Tree for input array {1, 3, 5, 7, 9,11}

Figure 21: Sum Binary Tree

We used a class *PrioritizedReplayBuffer* that can be found in a newly created file *per.py*. The code is adapted from the "lablml.ai" platform that provides many educational content about papers with a permissive MIT license. It builds the different trees and implement the formula from the papers to compute the samples probability. We modified the main loop to replace the old memory buffer by the prioritised one. It adds some hyper-parameters to the model

Hyper-Parameters	Value	Description
BETA	0.2	Regularization parameter
BUFFER_CAPACITY	2^{14}	Size of the buffer
ALPHA	0.6	Prioritization importance

Table 3: Hyper-Parameters of the Prioritized Experience Replay buffer

• Rest of the improvements
We unfortunately did not succeed to improve the rest of the selected papers and leave it as a further work to obtain the full Rainbow Network.

6.3 Results

After training for several hours our model on Breakout, we obtain the following results.

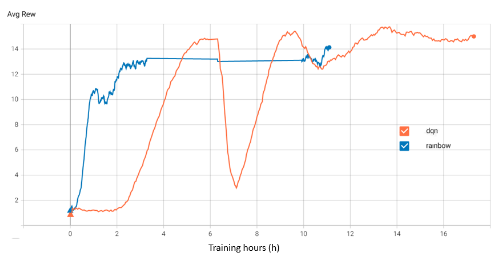


Figure 22: Performance of the DQN and of our Rainbow agent in function of the time

In the figure 22, the orange curve represents the DQN the blue curve represents our implementation of the Rainbow agent (DDQN+Dueling+PER). The first observation that we can make is that our rainbow agent learn a lot more faster than the simple DQN. Indeed the Rainbow agent reach the plateau faster only with 3 hours of training. It seems logical, because the DDQN associated with the dueling network in our agent separates the estimation of the state value function and the action advantage function. By doing so, it reduces the number of computations needed to estimate the Q-values for all actions. The shared feature representation of the state is computed only once and is then used to compute both the value and advantage streams. Additionally, the dueling architecture has been shown to be more sample efficient than the DQN, requiring fewer samples to achieve the same level of performance.

The drop of performance in the DQN (orange) is due to a training in multiple sessions and an unfortunate reset of the ϵ-greedy policy.

As a surprise, they reached the same plateau around an average value of 13. In the original paper, the Rainbow performs significantly better than the single algorithms. Of course, it can be due to not having a full Rainbow but it could also be related to the lack of training. The learning process can be time consuming to make our model generalize and we did not own GPU which speeds the training. In comparison with several articles, the models were trained on millions of steps where here we had less than a million step.

Also, one of our difficulties was a recent redesign of the Gymnasium package making many of the articles irrelevant. The package is easier to understand, but the vast previous project from the community are outdated, making it difficult to debug and find the answer to practical interrogations.

Conclusion and further research

• What have we covered ?

To conclude, we have covered many topics related to Deep Reinforcement Learning by analysing this article. We particularly liked how it made us understand the core of the DQN [9] and how the researchers improved it over the years.

We learned about various techniques to improve the architecture like the Double DQN [13], and Duelling Network [15]. We discovered a new way of exploring and modifying the network accordingly with the Noisy Layers [3]. We also learned how important it was to remove the correlation between a variable and favoured the training on the surprising event with the Prioritized Experience Replay [11]. Also, we have seen how to optimise the training phase with the multi-step learning [8]. Lastly, we studied the improvement gained by adopting a distributional perspective on the agent rather than a single value estimation in the categorical DQN [2]. This was a broad article but we are conscious that many more methods exist and could be interesting to study in a follow-up.

• How to combine several models?

In this paper, the researchers followed a scientific method to combine different propositions into a coherent algorithm while preserving a possible computational complexity. They showed its performance in comparison with the state-of-the-art methods on a benchmark dataset. Their methodology will be useful for our further work.

• Discover New package - Get our hands dirty on RL

We focus a lot of our attention on the practical side of such models. This made us discover a new package Gymnasium and get our hands dirty on an unknown field. This package is far from being completed and we look forward to seeing the next releases of the maintainer. We liked the implementation of a convolutional DQN even though it remained a challenge. Then, we tried to implement the different improvements selected by the researchers.

• A demanding field data wise

This project made us realize how demanding Reinforcement Learning is when it comes to data. To be trained, millions of episodes can be mandatory which can be a difficulty for complex environments. We were surprised and it raised questions on how to optimize the training and the cost of such models.

• Further research

As previously mentioned, we did not implement the full Rainbow algorithm and it is a logical follow-up of this report. It can be explained by our lack of practical experience, but we remain confident the last weeks of work will help us reach this goal.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[2] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning, 2017.

[3] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2017.

[4] Vladimir Gligorijevic, P. Douglas Renfrew, Tomasz Kosciolek, Julia Koehler Leman, Daniel Berenberg, Tommi Vatanen, Chris Chandler, Bryn C. Taylor, Ian M. Fisk, Hera Vlamakis, Ramnik J. Xavier, Rob Knight, Kyunghyun Cho, and Richard Bonneau. Structure-based protein function prediction using graph convolutional networks. *bioRxiv*, 2020.

[5] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[6] 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, 2017.

[7] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3–4):293–321, may 1992.

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[11] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[13] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.