Reinforcement Learning Project

Rainbow: Combining Improvements in Deep Reinforcement Learning

Done by :
Mohamed Issa
Riham Mansri
Yousra Leouafi

Supervised by :
Erwan Le Pennec

Academic Year : 2021-2022

Contents

Introduction

Reinforcement learning is an artificial intelligence approach based on the idea of making an individual learn through its interactions with the environment. This principle contrasts in general with the classical machine learning and artificial intelligence algorithms, where the learning is conducted by a knowledgeable teacher or reasoning on modeling the environment.

The domains of interest of the reinforcement learning cover a large spectrum of interdisciplinarity varying from control engineering to industry and even psychology.

Reinforcement learning is simply learning what to do which means what is the best action to be taken in a given situation in order to maximize a scalar reward value. As opposed to the other machine learning techniques, the learner is not told which action to take, on the contrary, it should find out which actions yield the most reward by trying them.

The deep reinforcement learning research community has made many advances in Deep Q -learning algorithm. Each of these extensions has contributed to evolving DQN algorithm. The idea we will be discussing behind this article [9], is to leverage the state-of-the-art by combining several extensions to the DQN in order to construct a powerful learner.

Throughout this report, we will explain the construction of the Rainbow model that combines some improvements of the DQN. We start by defining some technical concepts of reinforcement learning. Then we will explain each improvement separately and study the effect of the ablation of each component on the model.

Chapter 1

Background

1.1 Agents and environment

Before we dive into the details of the **Deep Q-learning** algorithm its implementation and its extension, we'll provide the definition of some of the main concepts of reinforcement learning.

- **The agent :** is the main character of each iteration of a reinforcement learning algorithm. It navigates through the environment, deploys each action and learns which actions are good to use within the constraints of the model.
- **The actions :** are the movements and interactions that the agent can make within the environment. There is usually a predetermined list of them.
- **The environment :** it typically takes the agent and its actions as input and returns a reward or a punishment depending on the implementation of the given model.
- **The state :** is the current situation of the agent at that very moment.
  Usually states and actions are paired together in order to visualize which actions can change the current state and the state that will be the result of that action (reward/punishment).

Reinforcement learning involves no agent supervision. It is the problem of an agent going in an environment and learning which actions are able to maximize a **scalar** reward function.
For the agent action selection is based on the policy which is a probability distribution for all actions give a state. The agent aims to optimize this policy so that it can choose the best actions for each state and thus maximize the reward function.
The process can better be explained by the following graph :

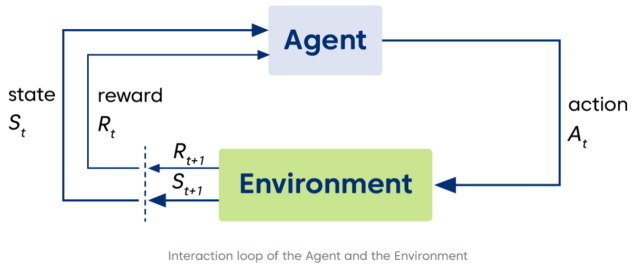


Figure 1.1: Interaction between the agent and the environment [3]

The interaction between the agent and the environment can be formalized through a Markov decision process that consists of 4 components **(S, A, T, r)**:

- **S :** a finite set of states.
- **A :** finite set of actions.
- **T(s, a, s')** $= P(S_{t+1} = s' | S_t = s, A_t = a)$: the transition function
- **r(s,a)** $= E(R_{t+1} | S_t = s, A_t = a)$: the reward function

The policy may be learned directly, or it may be constructed as a function of some other learned quantities such as the probability of taking the highest action (the greedy action) with probability (1-ϵ), or act uniformly at random.

1.2 Deep Reinforcement learning and DQN

In large scale cases, when the dimensionality of states or actions is high, one useful method to learn policies $\pi(s, a)$ or values $q(s, a)$ is to deploy Deep Learning architecture to represent these quantities into a low dimensional space. The parameters of the neural network are trained used gradient decent, minimizing a loss function to learn the Q value. One successful application was demonstrated in [10] by using convolutional neural networks to approximate action values for a given state.

However, when applying deep learning to approximate a value function does not guarantee convergence. In fact, weights are susceptible to oscillate and diverge due to high correlation between states and actions. This problem may be alleviated by using two techniques to make training more stable and efficient:

- **Experience Replay,**
- **Target Network**.

1.2.1 Experience Replay

One major requirement for SGD optimization is the independence and the identically distribution of the data. However, when the agent interacts with its environment, a

high correlation may exist within the experience sequence. By this, a naive Q-learning algorithm may be skewed because of the effect of correlation.

Preventing this issue is done using a buffer of the past experience which is used to sample training data from it instead of using the most recent experience. This buffer contains the transitions $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ for one fixed size. The value that it contains are appended in a deque structure. The technique of sampling from the replay buffer is what we call **experience replay**. In addition to breaking the harmful correlation problem, this technique enables us to learn multiple times from experience and to recall rare occurences.

1.2.2 Target Network

When performing an update of the Neural Networks's parameters to optimize Q(s,a) we can indirectly alter the value produced for Q(s',a') that is one step away from Q(s,a). This can entails into instability in the training.

A useful trick to tackle this problem is to make a copy of the neural network and use it for $Q(s', a')$ in the Bellman equation. The predicted Q values of this second network called the target network are used to backpropagate and train the main network. The parameters of the network are optimized by minimizing the following loss function:

$$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))), \tag{1.1}$$

the gradient of the loss is computed with respect to the parameters θ. And the $\bar{\theta}$ parameter denotes

1.2.3 DQN: Deep Q-Networks algorithm

DQN (Deep Q-Networks algorithm), implies two main phases. The first one is sampling the environment by performing actions and store the observed experiences in the replay buffer. The second is selecting randomly small batch of tuples from this memory and learn from that batch using SGD. The main algorithm using the last two techniques is detailed in the figure 1.2. In the beginning, we create the main network and the target one and initialize the memory buffer D. The agent is also initialized to interact with its environment.

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**

    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
    Choose an action a from state s using policy ϵ-greedy(Q)
    Agent takes action a, observe reward r, and next state s'
    Store transition $(s, a, r, s', done)$ in the experience replay memory D

    if *enough experiences in* D **then**
        /* Learn phase
        Sample a random *minibatch* of N transitions from D
        for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
            if $done_i$ **then**
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
        Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every C steps, copy weights from Q to $\hat{Q}$
    end
end

Figure 1.2: DQN algorithm

Chapter 2

Rainbow Agent Components

2.1 Double Q-learning

Double Q-learning is among the 6 extensions to Deep Q-learning that were mentioned in the article. This extension helps overcome the **overestimation bias** problem that affects the conventional Q-learning.
Overestimation bias occurs when estimated values $Q_theta(s, a)$ are bigger than the genuine values $Q(s, a)$, causing the agent to overestimate future rewards.
This could result in poor policy performance and the spread of estimation errors all along the learning process.
The original Double Q-learning algorithm addresses this overestimation by using two independent estimates Q_a and Q_b thus its name. The way it works is that we use one estimate to determine the maximizing action with which we update the other estimate and so on.

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s', a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s', a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s', a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Figure 2.1: Original Deep Q-learning algorithm [5]

The algorithm was then updated by decoupling the selection of the action from its evaluation. This is done using the loss:

$$(R_{t+1} + \gamma_{t+1} * q_{\bar{\theta}}(S_{t+1}, \arg\max_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2$$

This helped reduce the risk of overestimation.

2.2 Prioritized replay :

2.2.1 Motivation

The main goal of Replay Memory in Reinforcement Learning is to enable agents to remember the past experiences in order to use it after. It plays an important role in deep Q-Learning allowing them to reuse earlier experience and speed up learning breaking unwanted temporal correlations. One of the Rainbow agent components is the Prioritized Experience Replay, which, unlike prior works where the experience transitions were uniformly (have the same probability) sampled from a replay memory, the experience are sampled based on how valuable they are. In (Schaul et al. 2015) [12], they introduce the 'Blind Cliffwalk' problem (Figure 2.1) where the agent should guess the correct action to make it to the reward state by the end of the chain. If the action isn't correct, the agent doesn't get any reward and the agent returns back to the beginning of the chain. The problem here is difficult as the agent should learn two things; first how to gain the reward and second how to return (remember) back to the reward.

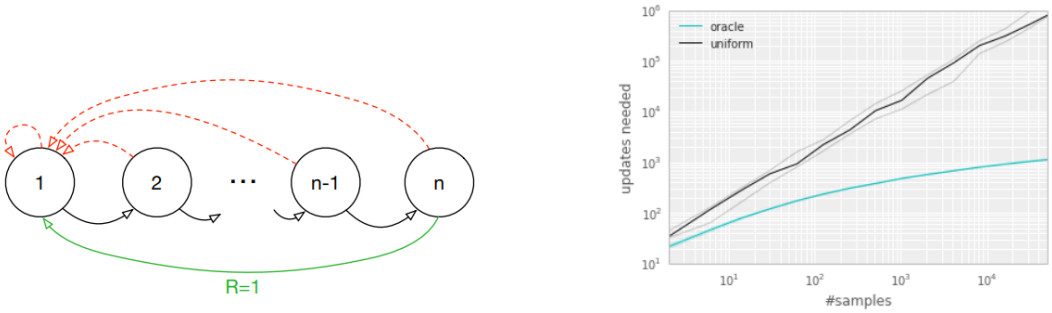


Figure 2.2: **Left** Illustration of the Blind Cliffwalk Problem. **Right**: Median number of the learning steps required to learn the value function. [12]

2.2.2 Stochastic Prioritization with TD-Error

The importance of each transition is measured using the magnitude of a transition's TD-error which shows how much a value and it's next-step bootstrap are far (Andre et al., 1998) and hence give us an idea on how much a transition is 'unexpected' or not. However, greedy prioritization has several issues such as updating only the transitions that are replayed (to avoid expensive sweeps)and hence transitions that have slow errors may not be replayed for a long time. Another issues are sensitivity to noise spikes and focusing on a little subset of non-diversified experiences which leads to over-fitting. In order to avoid these problems, the authors introduce the stochastic sampling method that combines between the greedy prioritization and the uniform random sampling. The probability of sampling transition i is defined as

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

where $p_i > 0$ is the priority of transition i. The exponent α determines the prioritization is used, with $\alpha = 0$ corresponding to the uniform case. α is actually used to reintroduce some randomness. The implementation of the algorithm is as follows (Figure 2.3)

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

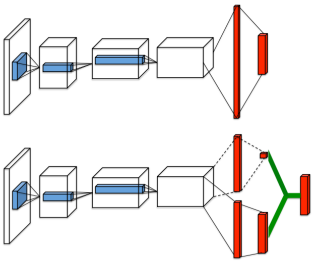
Figure 2.3: Double DQN with proportional prioritization [12]

2.3 Dueling networks :

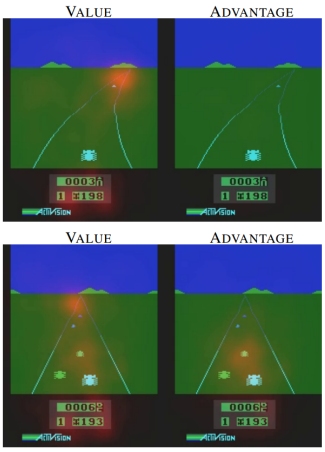
Another important component of the Rainbow agent is the Dueling Networks that were designed for value based Reinforcement Learning and were introduced by (Wang et al. 2016) [13]. This architecture separates the representation of the state values and the action advantages. Actually, the Q-values refers to how good it is to be at a certain state and also while taking an action at that state Q(s,a). Therefore, it can be decomposed as the sum of :

- $V(s)$: the value of being at that state
- $A(s, a)$: the advantage of taking that action in that state

We get $Q(s, a) = A(s, a) + V(s)$. In this type of architecture, the goal is to separate between the estimator of these elements using two new streams: one for the value function and the other for the advantage function. The two streams have a convolutional feature learning module in common (Figure 2.4 (a)). The interesting thing about this dueling architecture is it can learn which state is valuable without leaning the effect of each action at each state. As it can be seen on the Figure 2.4 (b) , the value function only focuses on two things (orange blur): the horizon where



(a) Q-network (Mnih et al., 2015) **top** vs the dueling Q-network (Wang et al. 2016) **bottom**



(b) Atari game Enduro, for a trained dueling architecture.

Figure 2.4: Dueling Q-Network architecture with the 2 streams and an illustration of the value stream and advantage stream an how they learn to pay attention [13]

the new cars appears and on the score while the advantage function doesn't really pay attention when there is no car in front because the action choice making isn't important. It only pays attention to the front car to avoid collisions because here it's crucial to survive.

Let θ denote the parameters of the convolutional layers of the Figure 2.4(a), and α and β are the parameters of the two streams of fully-connected layers. For the aggregation layer, the Q values for each action at that state is definied by :

$$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + \left(A(s,a,\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a',\theta,\alpha)\right)$$

This architecture helps accelerating the training because the value state can be computed without calculating the $Q(s,a)$ for each action and state, and also it enables to find more reliable Q values for each action using each stream alone.

2.4 Multi-step learning:

One of the simplest algorithms in Reinforcement Learning is 1-step Q-Learning. For each pair of state s_t and action a_t, leading to the transition s_{t+1} and a reward r_{t+1}. The update in the algorithm consists in:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\Big] \tag{2.1}$$

with α being the step-size parameter. The policy with respect to the optimal Q value is called the *greedy policy*. It corresponds for a given state s to select the action a with the highest value Q(s,a).

Another alternative is considering multi-step algorithm for Q learning which is introduced to alleviate the bias problem of the estimation as indicated in [8] by taking into consideration longer trajectories with more observations. In DQN, the optimisation problem consists of considering the truncated n-step return from one step S_t:

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

and then train the model to optimize the loss:

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, a_t))^2 \tag{2.2}$$

With a suitable value of n, the multi-step targets algorithm is shown to lead to faster learning.

2.5 Distributional RL [1]

Another extension to the DQN that aims to solving one of its issues is the distributional RL.
In distributional RL, we learn to approximate the distribution of returns instead of the expected average return. In fact, in Q-learning, we usually maximize the expected return as we want to minimize the error :

$$E_{s,a}[(r(s,a) + \gamma E_{s'}[\max_{a'} Q(s', a')] - Q(s,a))^2]$$

where $r(s,a)$ is the expected immediate reward.
The idea of this method, is that, instead of working with expectations and trying to minimize an error formalized by an expectation, we will work directly with the full explicit distribution of the return.
The way this works is that we define a random variable $Z(s,a)$ that represents the return obtained by starting from state s, performing action a and then following the policy in hand. We then have : $Q(s,a) = E[Z(s,a)]$.
By defining this variable, instead of trying to minimize the expectation that we mentioned above, we attempt to minimize an error defined by distributions and which represents the distribution between full distributions :

$$\sup_{s,a} dist(R(s,a) + \gamma Z(s', a^*), Z(s,a))$$

$$s' \sim p(.|s,a)$$

2.6 Noisy Nets

2.6.1 Exploration/Exploitation dilemma

One of the main problems in Reinforcement learning is how the agent can keep a balance between the exploration and exploitation.The two classic approaches for this kind of problem are :

- **Epsilon-Greedy** : where the agent takes a random step instead of the action with some probability epsilon.
- **Entropy Regularisation** : used in policy gradient methods so that the agent avoid getting stuck in local optimums and hence improve exploration and fine-tuning policies.

2.6.2 Noisy Nets for RL

In the paper of DeepMind (Fortunato et al. 2017) [7], they propose a noisy linear layer that combines both deterministic and noisy stream. NoisyNets are neural networks with weights and biases that are perturbed by a noise function.

Let $y = f(x)$ be a neural network where *theta* is the vector of noisy parameters which takes input and output . Let $\theta = \mu + \Sigma\epsilon$ where $\zeta = (\mu, \Sigma)$ is a set of learnable parameters and ϵ is a vector of zero-mean noise with fixed statistics and $\odot$ represents element-wise multiplication. The loss of the neural network is $Ł(\zeta) = E[L(\theta)]$, the optimisation occurs with respect to the set of parameters ζ.

A linear layer of a neural network with p inputs and q outputs is represented by $y = wx + b$ where $w \in \mathbf{R}^{p\times q}$ is the weight matrix and $b \in \mathbf{R}^q$ the bias. The corresponding noisy linear layer is defined as :

$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b$$

As we can see, $(\mu^w + \sigma \odot \epsilon)$ and $(\mu^b + \sigma^b \odot \epsilon^b)$ replace respectively w and b in the linear neural network equation. The parameters μ^w,μ^b,σ^w and σ^b are learnable but ϵ^w andϵ^b are random noise variable. In the article, the author introduces two options of the noise distribution.

- **Independent Gaussian noise**: the noise of each weight and each bias is independent and hence we get $pq + q$ noise variables
- **Factorised Gaussian Noise** : for each entry $\epsilon^w_{i,j}$ (resp. $\epsilon^b_{i,j}$) of the random matrix ϵ^w (resp. ϵ^b), by factorising $\epsilon^w_{i,j}$ we can write $\epsilon^w_{i,j} = f(\epsilon_i)f(\epsilon_j)$ where f is a re al-valued function (resp. $\epsilon^b_j = f(\epsilon_j)$, for example $f(x) = sgn(x)\sqrt{|x|}$), we get $p + q$ variables in total.

Computing the gradient of the loss, we get $\nabla Ł(\zeta) = \nabla E[L(\theta)] = E(\nabla_{\mu,\Sigma} L(\mu + \Sigma\epsilon))$. By using the Monte Carlo Approximation and taking each sample ζ at each step, we get $\nabla Ł(\zeta) = \nabla_{\mu,\Sigma} L(\mu + \Sigma\epsilon)$.

Noisy networks can be adapted to the Deep Q-Networks (DQN) and Dueling, the NoisyNet-DQN loss and NoisyNet-Dueling can also be computed (Fortunato et al. 2017) [7]. The following algorithm shows how this two models can be modified and adapted to noisy networks. (Figure 2.5)

C.1 NOISYNET-DQN AND NOISYNET-DUELING

Algorithm 1: NoisyNet-DQN / NoisyNet-Dueling

Input : Env Environment; ε set of random variables of the network
Input : DUELING Boolean; "true" for NoisyNet-Dueling and "false" for NoisyNet-DQN
Input : B empty replay buffer; ζ initial network parameters; ζ^- initial target network parameters
Input : N_B replay buffer size; N_T training batch size; N^- target network replacement frequency
Output : $Q(\cdot, \varepsilon; \zeta)$ action-value function

for *episode* $e \in \{1, \dots, M\}$ **do**
  Initialise state sequence $x_0 \sim Env$
  for $t \in \{1, \dots\}$ **do**
    /* $l[-1]$ is the last element of the list l */
    Set $x \leftarrow x_0$
    Sample a noisy network $\xi \sim \varepsilon$
    Select an action $a \leftarrow \text{argmax}_{b \in A}\, Q(x, b, \xi; \zeta)$
    Sample next state $y \sim P(\cdot|x, a)$, receive reward $r \leftarrow R(x, a)$ and set $x_0 \leftarrow y$
    Add transition (x, a, r, y) to the replay buffer $B[-1] \leftarrow (x, a, r, y)$
    if $|B| > N_B$ **then**
      Delete oldest transition from B
    end
    /* D is a distribution over the replay, it can be uniform or implementing prioritised replay */
    Sample a minibatch of N_T transitions $((x_j, a_j, r_j, y_j) \sim D)_{j=1}^{N_T}$
    /* Construction of the target values. */
    Sample the noisy variable for the online network $\xi \sim \varepsilon$
    Sample the noisy variables for the target network $\xi' \sim \varepsilon$
    if *DUELING* **then**
      Sample the noisy variables for the action selection network $\xi'' \sim \varepsilon$
    for $j \in \{1, \dots, N_T\}$ **do**
      if y_j *is a terminal state* **then**
        $\widehat{Q} \leftarrow r_j$
      if *DUELING* **then**
        $b^*(y_j) = \arg\max_{b \in \mathcal{A}} Q(y_j, b, \xi''; \zeta)$
        $\widehat{Q} \leftarrow r_j + \gamma Q(y_j, b^*(y_j), \xi'; \zeta^-)$
      else
        $\widehat{Q} \leftarrow r_j + \gamma \max_{b \in A} Q(y_j, b, \xi'; \zeta^-)$
      Do a gradient step with loss $(\widehat{Q} - Q(x_j, a_j, \xi; \zeta))^2$
    end
    if $t \equiv 0 \pmod{N^-}$ **then**
      Update the target network: $\zeta^- \leftarrow \zeta$
    end
  end
end

Figure 2.5: NOISYNET-DQN AND NOISYNET-DUELING Algorithm [7]

2.6.3 Results

The DQN was implemented with the NoisyNets (Blue curve below) and was compared to the classical DQL (Orange curve bellow). The results were impressive, as we can see the reward values increases significantly.

Figure 2.6: Partially-trained DQN on Pong with NoisyNets **Blue** vs Classical DQN **Orange**. The first figure representes the raw reward values, the second one is mean for last 100 episodes. [6]

Chapter 3

Experiments and Analysis

Having seen the deep Q-learning algorithm, its implementation, how the use of deep learning techniques allowed to improve its performance and how its different extensions helped tackle different issues related to it, we'll now give the experimental methods and analyse the results of Rainbow whose agent integrates the components of the different extensions in an attempt to obtain an optimal performance.

3.1 Experimental methods

In this part we explain briefly the experimental methods and setup for the rainbow model.

The agents were evaluated on 57 Atari 2600 games. The training and evaluation procedures are the same as in [10]. The score of the agents were normalized per game. Meaning that 0% corresponds to a random agent and 100% to a human expert. In order to track where the improvement in the median came from, the number of games where the agent's performance is above the human performance is considered. Another evaluation was done on the best agent using two different testing regimes:

- *No-ops starts regime:* A random number of no-ops actions is inserted at the beginning of each episode.
- *human starts regime:* Episodes are initialized with randomly sampled points taken from initial portion of human expert trajectories.

The difference between these two regimes indicates the extent to which the agent can over-fit to its own trajectories.

The hyperparameter values are given in the figure 3.1.

3.2 Cost and Computational Power [2]

We'll first start by discussing the cost and the computational power needed for the Rainbow algorithm.
Rainbow's agent might be praised and might be considered as one of the state of

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	$[-10, 10]$

Figure 3.1: Rainbow hyper-parameters from [9]

the art algorithms in Reinforcement Learning, however it should be mentioned that it requires a high computational power that is only accessible for a large research laboratory.
In fact, on a Tesla P100 GPU, training an Atari game takes about 5 days. It is also typical to employ at least five independent runs to report results with confidence boundaries. All in all running experiments takes about 34,200 GPU hours or more clearly 1425 days.
It is also important to point out that a Tesla P100 cost about US$6000.which limits the implementation and exploitation of the algorithm in the research field to large laboratories.

3.3 Performance analysis

As far as its performance is concerned, in order to get a better understanding of the way the integrated agent of Rainbow differs from the other extensions of DQN, two approaches were used.
In one of the approaches, we compare the performance of Rainbow which is represented by the number of games where the algorithm surpasses a certain percentage of the human performance as a function of the number of millions of frames to that of DQN and its variants: DDQN, Prioritized DDQN, Dueling DDQN, Distributional DQN, and Noisy DQN and the multi-step learning method as used in A3C.
In the second approach, we take the Rainbow algorithm, plot its performance and the performance of different "ablations" which are the Rainbow from which we've removed one of the components. This allows us to get a better idea of the exact contribution of each one of the methods.

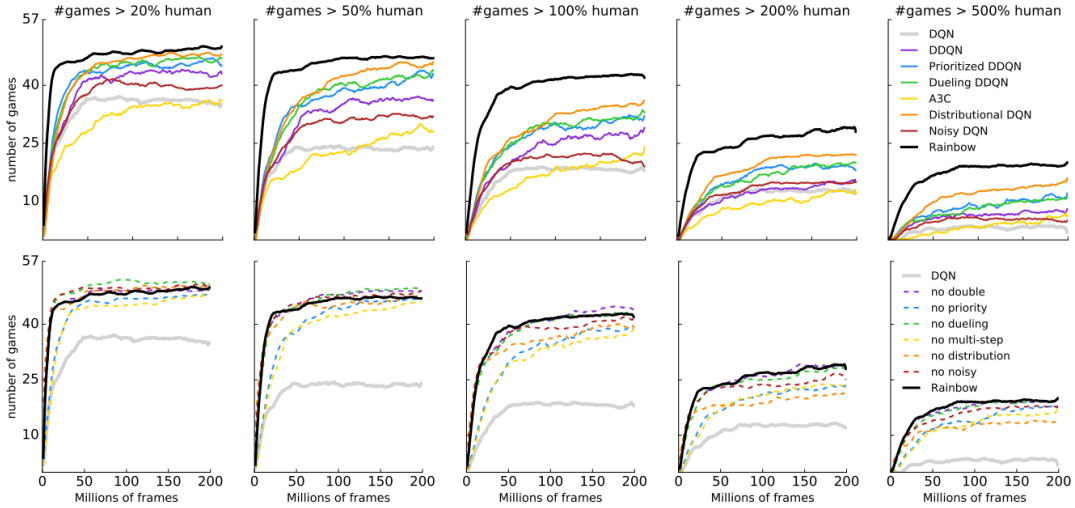


Figure 3.2: First row:Rainbow vs baselines.Second raw:Rainbow vs its ablation [9]

The first approach clearly shows that Rainbow outperforms DQN and its different variants. In fact, in the experiments where good results were already obtained, Rainbow helped get them to an even better performance and in ones where the other methods weren't able to match human performance, it was able to get better results and get closer to achieving equal performances or even better.

In the second approach, known as **"ablation studies"**, the evolution of the performance of different versions of Rainbow where at each time a component is removed was tracked. Comparing the different thresholds of human normalized performance showed which components are the ones that result in the increases of Rainbow's performance the most.
These observations helped conclude that prioritized replay and multi-step learning contribute the most to the rise of the integrated agent's performance given that removing them results in a considerable decrease of Rainbow's performance. Distributional Q-learning is also a component whose removal decreased the performance. However, for the rest of the components, different results were observed. For Noisy Nets, even though they helped increase the performance when they were included, their removal had a positive outcome for certain games. The case of the dueling network is similar to this, where depending on the game, different results were obtained which also the case of double Q-learning.

Chapter 4

Discussion

It is undeniable that introduction of DQN in general in Reinforcement learning and of Rainbow in particular sort of revolutionized the field and caused an important increase in research papers given the effect that it had on the performance of the algorithm when applied to the Atari 2600 benchmark.

However, the integrated agents as we've seen doesn't include all the possible components which leaves an area for further research to be conducted on whether adding other components or changing certain ones could enhance the performance even more.
An example of a method that is promising is **Hierarchical RL** that has shown good results when applied to the Atari games and that was paired with DQN to give h-DQN which has shown to be successful and that has helped solve scaling issues.

Among the other components that exist and that weren't integrated in Rainbow's agent are **Bootstrapped DQN** that explores in a computationally and statistically efficient manner through use of randomized value functions, **intrinsic motivation** that allows the agent to explore when rewards are sparse, something that other methods might fail to do, and **count-based exploration** that are known to give near optimal performance when applied to small discrete Markov decision processes.

This demonstrates that even though outstanding results where obtained with DQN and Rainbow, there is always a possibility to improve both on performance and computation time.

Conclusion

To conclude, we have seen that the Rainbow is based on the DQN algorithm and combines several other algorithms: DQN (Deep Q-Network),DDQN (Double Deep Q-Network),Multi-Step Learning,Prioritized Experience Replay,Dueling Q-Network,Distributional RL and Noisy Networks and we've seen each of these algorithms in details. We have also seen that the results of the Rainbow is significantly better than all the baselines,and the prioritized replay and multi-step learning are contributes the most in better performances. Combining however all the components produces the best overall agent and the best results. Actually, the Rainbow agent made a great improvement in the deep reinforcement learning community and achieved high performances comparing to the previous algorithmes.

A new paper, **Revisiting Rainbow: Promoting more Insightful and Inclusive Deep Reinforcement Learning Research 2020** by Johan S. Obando-Ceron, Pablo Samuel Castro [11] presents some new insights into the algorithms used by Rainbow. They analyse the interaction between the algorithms used in the Rainbow Agent and the network architecture and propose a revisited Rainbow Algorithm. One of there main results is that distributional RL, when added alone to DQN, can lower the performances. They suggest that it should be combined to another algorithmic component (in the classic control environments) or when used on a CNN (as in the MinAtar games).

Bibliography

[1] https://mtomassoli.github.io/2017/12/08/distributional_rl/.

[2] https://psc-g.github.io/posts/research/rl/revisiting_rainbow/#the-cost-of-rainbow.

[3] https://medium.com/sciforce/reinforcement-learning-and-\asynchronous-actor-critic-agent-a3c-algorithm-explained-f0f3146a14ab.

[4] https://medium.com/swlh/states-observation-and-action\-spaces-in-reinforcement-learning-569a30a8d2a1.

[5] https://towardsdatascience.com/double-deep-q-networks-905dd8325412.

[6] https://shmuma.medium.com/summary-noisy-\networks-for-exploration-c8ba6e2759c7.

[7] 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, 2019.

[8] J. Fernando Hernandez-Garcia and Richard S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the dqn target, 2019.

[9] 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.

[10] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[11] Johan S. Obando-Ceron and Pablo Samuel Castro. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research, 2021.

[12] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[13] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.