Deep Convolutional Q-Learning applied to Doom

Mehdi Boubnan
CentraleSupelec - ENS Paris-Saclay
mehdi.boubnan@student.ecp.fr

Ayman Chaouki
CentraleSupelec - ENS Paris-Saclay
ayman.chaouki@student.ecp.fr

Abstract

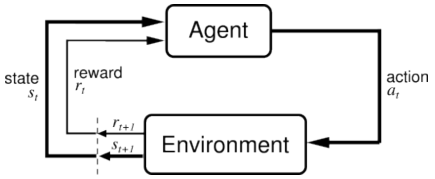
There are two main approaches to reinforcement learning, value based methods that aim to find an optimal Q-function, and policy based ones that directly look for the optimal policy. However most of reinforcement learning problems (like learning to play games) have large or even continuous states spaces, which makes constructing Q-values table impossible, thus the need for approximate reinforcement learning. In this work we present the Deep reinforcement learning methods we've used to train an agent to play in a Doom environment.

1 Introduction

Deep reinforcement learning (DRL) allowed for some major breakthroughs in reinforcement learning, like beating world champions in Go with AlphaGo Zero which was a very challenging task due to the immensity of the state-space. In fact, DRL relies on neural networks for the approximation task, thus enabling good prediction (of the state-action value function or the policy) given unstructured data like images as input state.
In this work, we will try to train agents to play in some specific scenarios of Doom. To do so, we will use the VizDoom package as an environment to get access to states, actions and rewards. We will try different types of algorithms, from Deep Q-learning (value based) to more advanced algorithms like Asynchronous-Advantage-Actor-Critic *A3C* (hybrid between policy and value based), and curiosity (with an A3C backbone).
We will start by explaining each one of the algorithms we've used and how they aim at solving the RL problem, then we will test them on some specific scenarios of VizDoom (the basic, deadly corridor and defend the center). This way we can criticise the algorithms and try to interpret their successes, failures and behaviours depending on the scenario.

2 Background

In a reinforcement learning problem, there is an agent interacting with an environment through actions $a_t \in \mathcal{A}$ he takes in specific states s_t and the environment provides feedback to the agent in the form of a reward r_t and another state s_{t+1}.



The action a_t taken by an agent in a state s_t is dictated by a policy π which can be deterministic ($\pi(s_t) = a_t$) or stochastic (return the probability $\pi(a_t|s_t)$ of taking action a_t in state s_t instead of a fixed action a_t).

In this framework we work under the reward hypothesis, that is any goal can be formulated as searching for a policy (called optimal policy) that maximizes a state value function $\mathcal{V}^{\pi}$ which is an expectation of the cumulative rewards. In our case, the state value function maps each state s to the expected discounted cumulative reward in a finite time horizon with final state.

$$\mathcal{V}^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^{t} r\left(s_t, \pi\left(s_t\right)\right) | s_0 = s; \pi\right]$$

Where T is the first random time when we reach the terminal state or the time limit T_{max}, and $0 \leq \gamma < 1$ (typically $\gamma = 0.99$ in order to give less importance to rewards that are far in the future due to a greater uncertainty).
Note that in order to unroll a trajectory, the necessary variables at each time-step t are s_t, a_t, r_t, s_{t+1} that can all be provided by VizDoom environment.
In practice, we use the state-action value function in the Bellman equation form

$$\begin{aligned}\mathcal{Q}^{\pi}\left(s, a\right) &= \mathbb{E}\left[\sum_{t=0}^{T} \gamma^{t} r\left(s_t, \pi\left(s_t\right)\right) | s_0 = s, \pi\left(s_0\right) = a; \pi\right] \\ &= r\left(s_0, a\right) + \gamma \mathcal{Q}^{\pi}\left(s_1, \pi\left(s_1\right)\right) \\ &= \mathcal{T}^{\pi} \mathcal{Q}^{\pi}\left(s, a\right)\end{aligned}$$

Where $\mathcal{T}^{\pi}$ is the Bellman operator.
We define the optimal Bellman operator as follows:

$$\mathcal{T}^{*} \mathcal{Q}\left(s, a\right) = r\left(s, a\right) + \gamma \underset{a'}{max} \mathcal{Q}\left(s', a'\right)$$

Where s' is the state the agent got by applying action a in state s.
The optimal Bellman operator $\mathcal{T}^{*}$ is a contraction for the l_2-norm, using the Banach fixed point theorem we have that $\mathcal{T}^{*}$ admits a unique fixed point that is (by inspection) the optimal state-value function $\mathcal{Q}^{*}$
Dynamic programming can be used to look for the optimal policy and value function through value based or policy based methods when transition probabilities and the reward function are available and it also requires an exact representation of value functions and policies which is impossible when the state space is huge or even continuous, thus the need for approximate reinforcement learning.

3 Deep reinforcement learning

Approximate reinforcement learning tries to approximate state-action value functions in some "smaller" space, in deep reinforcement learning the approach is to rely on neural networks to perform the approximation, this is especially convenient for games (Atari, FPS ...) since the input states are frames. Here, we will test some value based and hybrid (policy and value based) algorithms in Doom environments for the following scenarios: basic, deadly corridor and defend the center; each with its own settings (possible actions and reward reshaping). More about the experimental settings in the Experiments section.

3.1 Value based algorithms

3.1.1 Deep Q-learning

In each Doom scenario the environment provides us with frames (images) at each time step, these will constitute our states. Deep Q-learning then uses a neural network to return $\mathcal{Q}$-values for each action taken in that state. The architecture of the network contains a series of convolutional neural networks followed by fully connected ones, and some non-linear activation (generally relu) is used along the forward pass [5] .

States : In some environments, the agent must have a knowledge about motion in order to perform well (aiming at a moving monster for example); the network architecture in DQN, when used with single frames as states, does not reveal any notion of motion, therefore we can use stacked frames as states to solve this problem.
Note that this concerns only the DQN model as there are models like Deep recurrent Q-learning that

include the motion information using only single frames as states.

Loss function : Given a transition (s, a, r, s'), the forward pass of the neural network outputs $\{\mathcal{Q}(s,a); a \in \mathcal{A}\}$, we compute the target $\mathcal{Q}$-value $\mathcal{Q}_{target}\left(s,a\right) = r\left(s,a\right) + \gamma \underset{a'}{max} \mathcal{Q}\left(s', a'\right)$ for action a. Since $\mathcal{Q}^*$ is the fixed point of $\mathcal{T}^*$ the optimal Bellman operator, we assume that the function approximating it in the space generated by the neural network (all functions f such that there exists a set of weights Θ making $f(s)$ equal to a forward pass through the network with Θ for all possible states s) will be close to a fixed point of $\mathcal{T}^*$, thus minimizing the TD error $\left(\mathcal{Q}_{target}\left(s,a\right) - \mathcal{Q}\left(s,a\right)\right)^2$. Concretely this loss is minimized as a mean-squared error over a batch of transitions gathered through playing the game.
In practice, we decouple experience gathering from learning through a replay buffer in order to decorrelate experiences and avoid overwriting previously learned weights.

Replay buffer : Experiences are described as the transitions (s, a, r, s'), when we unroll an episode, at each time step an action a is taken with respect to some ϵ-greedy policy in a current state s resulting in a reward r and a next state s', this transition is then stored in a replay buffer. When we have enough experiences (some number higher than the batch size) we perform training.

Training : Random experiences are sampled from the replay buffer in order to perform training by back-propagating the TD error.

Instability : DQN presents the problem of being very unstable as the loss oscillates a lot during training, this is due to the fact that actions to be taken in the next state and target Q-values are estimated using the same network which weights gets updated each time, in other words each time we try to learn a different function. We can alleviate this problem by using another network (same architecture, but delayed weights) to estimate the target Q-values while the original one chooses the action to be taken and then updating the weights of the target network with those of our network each τ steps. This is called Double Q-learning [8], in fact we used it with some other DQN enhancements as we will see in the next sections.

3.1.2 Double Dueling Deep Q-learning

The novelty of dueling deep Q-learning is the introduction of a new network architecture composed of two streams, one estimating the state-value function $V(s)$ *thus outputting a scalar* while the other stream estimates the action advantage function $\{A\left(s,a\right); a \in \mathcal{A}\}$ *thus outputting a vector of size the number of possible actions*; the outputs of the two streams are then aggregated (in a certain fashion) to output the Q-values $\{\mathcal{Q}\left(s,a\right); a \in \mathcal{A}\}$. The value stream allows us to estimate states values as there are some states indifferent to all possible actions (no action leads to an improved expected cumulative reward) while the advantage stream estimates the advantage of taking an action at a specific state. We define the advantage as follows $\mathcal{Q}\left(s,a\right) = A\left(s,a\right) + V\left(s\right)$.

Aggregation : Aggregating the two streams is not as straightforward as just taking the sum $\mathcal{Q}\left(s,a\right) = A\left(s,a\right) + V\left(s\right)$ because in this case we run into the problem of identifiability as $V(s)$ and $A(s,a)$ cannot be recovered from this formula; instead, we can consider the module used in [9], where the estimator of the advantage function is in fact the difference between the output of the advantage stream and its mean, therefore we have

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

As specified before, we will implement the double dueling deep Q-learning algorithm which makes use of the dueling architecture in a double Q-learning fashion, in fact we will directly use it with a prioritized experience replay (that we will explain in the next section) instead of the uniform one.

3.1.3 Prioritized experience replay

The uniform experience replay we've seen before considers all experiences in the replay buffer of equal importance, however some experiences help the network learn better than others and are therefore more relevant. Unfortunately such experiences are often rarely explored and a uniform

sampling setup from the replay buffer makes them also rarely used by the network to update its weights. [7] proposes a novel sampling method -the prioritized experience replay- to achieve more frequent backward pass with relevant experiences.
But how do we measure experience importance?
We use the magnitude of the TD error δ yielded by a transition to construct the probability of sampling it, intuitively $|\delta|$ can be interpreted as how unexpected the transition is for the neural network. There are two ways of formulating the priority of an experience i in the replay buffer:

Proportional prioritization : $p_i = |\delta_i| + \epsilon$ where ϵ is a small positive number introduced just to ensure sampling experiences even when $\delta_i = 0$

Rank-based prioritization : $p_i = \frac{1}{rank(i)}$ where $rank(i)$ is the rank of experience i when the replay buffer is sorted according to $|\delta|$
Sampling probabilities are then formulated as follows $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ where α is a prioritization strength coefficient ($\alpha = 0$ corresponds to the uniform sampling).
However, changing the sampling distribution in such a manner introduces bias towards the frequently sampled experiences, in order to avoid this bias introduction we use importance sampling weights $w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$, which means that for an experience i the network will be get a backward pass on $w_i\delta_i$ instead of δ_i with β a coefficient annealed from a value near 0 to 1 over the episodes, this allows the frequently used experiences to influence a lot more the network weights updates in the beginning of training and we slowly compensate for their frequency of usage till full compensation when $\beta = 1$.

3.1.4 Deep recurrent Q-learning

In this section we briefly present an alternative to the DQN architecture that makes use of recurrent neural networks and specifically LSTM (in most implementations). With DRQN, the LSTM module put right after the convolutional layers can model well sequences thus enabling us to forget about representing states as stacked frames because single frames are now enough, this allows us for example to process efficiently RGB frames.
Since we sample batches of experiences from the replay buffer instead of unrolling whole episodes, the hidden units of the LSTM must be initialized to 0 at the beginning of each update, this makes it harder for the LSTM to learn functions that span longer time scales than that of a backward-pass [2]

3.2 Hybrid algorithms

3.2.1 A3C : Asynchronous Advantage Actor Critic

The Asynchronous Advantage Actor Critic (A3C) [4] is an algorithm released by Google's DeepMind. This simple, fast and robust algorithm has outperformed most other algorithms such as DQN and its enhanced versions. The idea behind it was to merge between value based methods and policy based ones, taking advantage of both their benefits. Let's take a look at each word composing the name of the algorithm to understand its mechanics :

Actor-Critic : The backbone network behind the algorithm tries to estimate a stochastic policy $\pi(a|s)$ (a set of action probability outputs for an input state) described as an "Actor" and a value function $V(s)$ as a "Critic" that measures how good the action taken is. The Actor begins by playing randomly, updating its behavior using the feedback from the Critic. On the other hand, the Critic updates itself to provide more accurate feedback to the Actor, leading to a satisfying result. We'll see more about the network settings in the Experiments section.

Asynchronous : In DQN, one single agent trains using a replay. For the A3C algorithm, different workers trains simultaneously, each on its proper environment with its own copy of the global network, and therefore its own set of network parameters, and share what they've learned with each other asynchronously. After a defined number of steps done by a worker or when a terminal state is reached, the worker updates the global network, copies it, and continues its training in its own environment, while other workers may be training with the old version of the global network if they haven't finished the steps needed for the update.

Advantage : As stated in the Dueling Deep Q-Learning, the advantage is defined as $A(s_t, a_t) = \mathcal{Q}(s_t, a_t) - V(s_t)$. The value function V is outputted by the network, and the $\mathcal{Q}$ function is approximated as follows for an enrolled episode of lenght k :

$$\mathcal{Q}(s_t, a_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k})$$

where k is the number of steps taken by the worker before the update (upper-bounded by the maximum number of steps n_{max} defined by the algorithm). Remember that k is equal to n_{max} if the worker hasn't reached a terminal state during these k steps. The advantage has therefore the following expression :

$$\mathcal{A}(s_t, a_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}) - V(s_t)$$

Now that we've seen how A3C works, we will see how the updates are made to the network.

A3C loss : The loss of the actor-critic network is defined as follows :

$$\mathcal{L}_{A3C}(s_t, a_t, r_t, s_{t+1}) = \mathcal{L}_{value} + \mathcal{L}_{policy} + \lambda \mathcal{L}_{entropy}$$

The value is updated using the assumption that the estimation is close to the fixed point of the Bellman Operator $\mathcal{T}^*$:

$$\mathcal{L}_{value}(s_t, r_t, s_{t+1}) = \frac{1}{2}||r_t + \gamma V(s_{t+1}) - V(s_t)||^2$$

The policy is updated using the following loss :

$$\mathcal{L}_{policy}(s_t, a_t) = -log\pi(a_t|s_t)\mathcal{A}(s_t, a_t)$$

Finally, we add an additional loss to encourage exploration on the state-action space, pushing the actor to be sure about the correct action by penalizing the negative entropy using the following loss :

$$\mathcal{L}_{entropy}(s_t, a_t) = \pi(a_t|s_t)log\pi(a_t|s_t)$$

3.2.2 Curiosity-Driven learning

In every method we've seen so far, the agent collects extrinsic reward received from the environment. The idea behind Curiosity-Driven learning is to make the agent learn by himself, building his own *intrinsic* reward by exploring the environment. This idea was introduced by [6] and was applied to Doom. It essentially tries to overcome a big problem in Reinforcement Learning : facing an environment with *sparse rewards*. Indeed, not having a direct feedback from the environment on every action an agent takes or having rewards far in the future makes the learning difficult because the agent does not know which actions taken in which states lead to such reward (Montezuma game for example). Thus, feeding the agent with a new intrinsic reward based on the exploration of new states pushes it to always try to discover unpredictable states and learn useful skills in the process.

In order to model a curious behaviour, an intuition is to base the intrinsic reward on the agent's ability to predict the next state, the more it can the less relevant is the state and therefore the lower the reward it should get. One may be tempted to build a predictive model of the next state based on the current one and the action taken and consider the reward as the prediction error; the problem with this approach is it's sensitivity to irrelevant states (states that are inherently unpredictable and of no use to the agent), a solution to this problem is to avoid using the raw sensory space (images) and learn instead an embedding function ϕ of the states where inherently unpredictable next states from some state will have the same representation. According to [6], this can be done with the co-training of the inverse dynamics and forward dynamics models.

Inverse dynamics model : Given an experience (s, a, r, s'), try predict action a using $\phi(s)$ and $\phi(s')$.

Forward dynamics model : Given an experience (s, a, r, s'), try predict next "state" $\phi(s')$ using a and $\phi(s)$.

This co-training of the two models above lead to robust state representation ϕ with respect to unpredictable states.

4 Experiments

ViZDoom [3] is an AI Research Platform for Reinforcement Learning for the game Doom. We applied the above methods on different scenarios of Doom environment using this platform (see our github `https://github.com/Swirler/Deep-Reinforcement-Learning-applied-to-DOOM`). The scenarios we used are the following : basic, deadly corridor and defend the center. See ViZDoom github for more details on the scenarios and the game rewards.

Value based parameters : All value based algorithms used the same convolutional layers, conv 16 filters with kernel size 5 and stride 2, conv 32 filters with kernel size 5 and stride 2, conv 32 filters with kernel size 5 and stride 2, then the DQN feeds the output of conv layers to a 128 fully-connected layer before outputting the Q-values, while dueling architecture does the same for each stream and DRQN feeds the output of conv layers to a 512 hidden units LSTM before outputting Q-values. All these algorithms were trained with Adam optimizer using a learning rate of 10^{-4}

A3C parameters : For the A3C algorithm, the actor-critic network used was composed of two convolutional layers, the first one with 16 filters with a kernel size of 8 and a stride of 4, and the second one with 32 filters with a kernel size of 4 and a stride of 2, followed by a fully connected layer of 256 units, and an LSTM network. Two fully connected layers are added on top, one with one unit to estimate the value function, and the other one with a number of units equal to the action space size to estimate the policy. For the training, we used an Adam optimizer shared between workers with a learning rate of 10^{-4}, and clipped the gradients norms by 40.

Curiosity : The curiosity module was trained with an Adam optimizer with a learning rate of 10^{-3}. The gradients norms were also clipped by 40. However, we didn't clip the intrinsic reward, which confused the A3C backbone in the training phase.

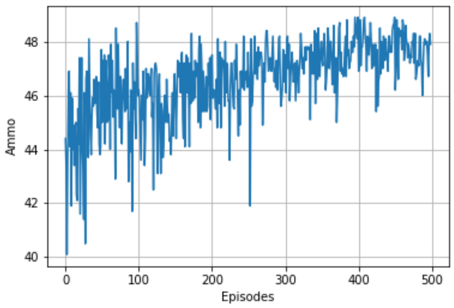
Confidence intervals : When building confidence intervals (for promising experiments), we repeat the experiments 3 times.

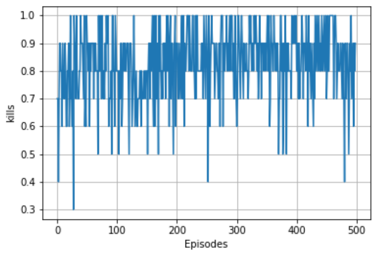
All details about training can be found on our github [1].

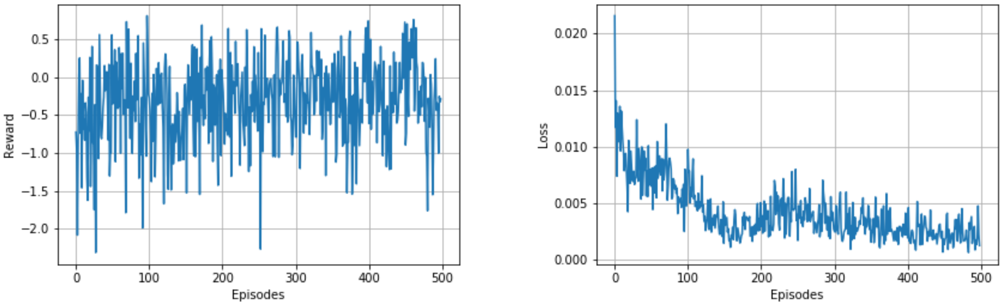
4.1 Basic scenario

For this scenario, the map is a rectangle where the agent is in the center of one of the two longer walls, and a monster spawns randomly along the other wall. The only reward he takes from the environment is **+100** if he kills the monster and **-1** if not.

Deep dueling Q-learning results :



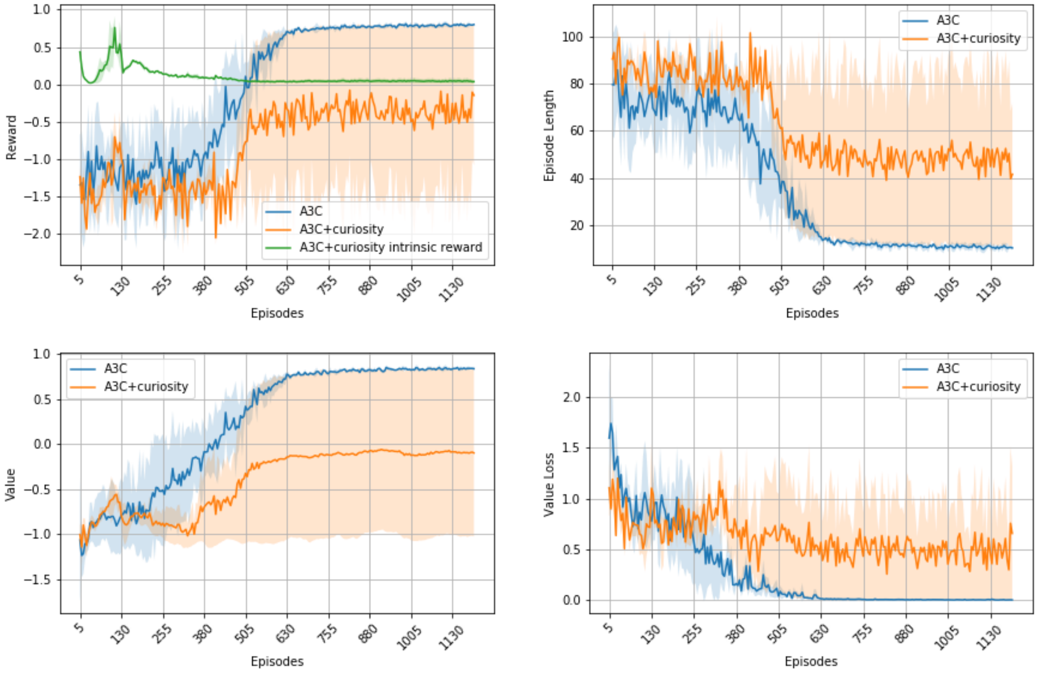




Discussion : Value based algorithms failed to yield satisfying results in the simple basic scenario of Doom (mainly due to the huge oscillations of the loss), only deep dueling Q-learning converged to a suboptimal policy after 5000 episodes and it required a lot of tuning of the memory size, learning rate ... Therefore in the next experiments we will focus on the A3C and curiosity.

A3C results with and without using curiosity :

Reward reshaping : Dividing the reward by 100 to keep the oscillations in $[-1, 1]$.

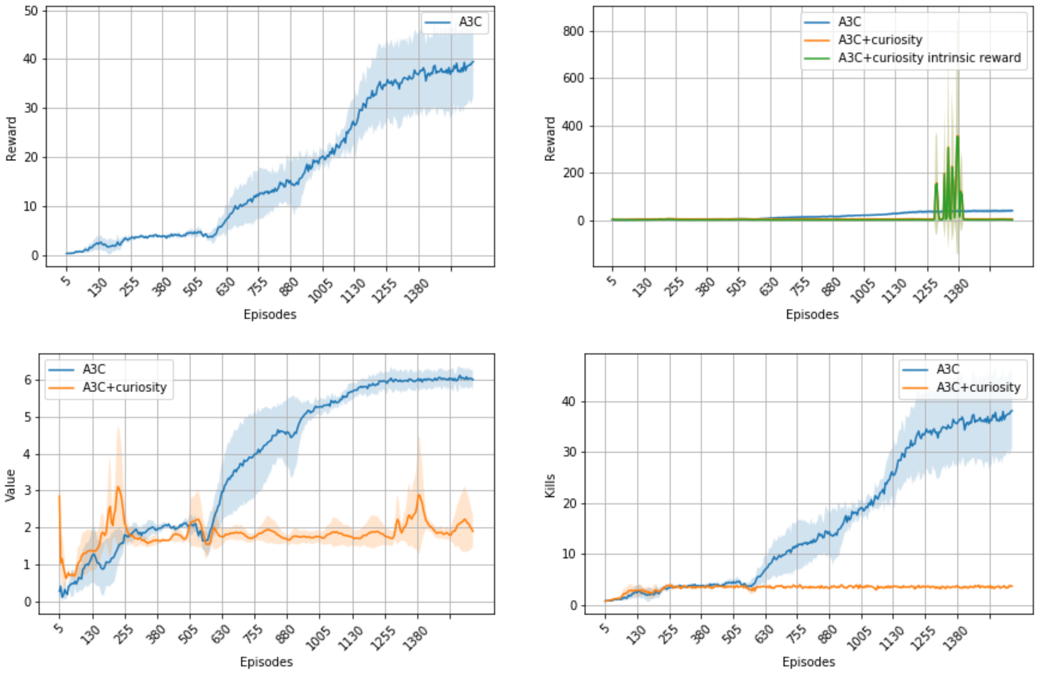


Discussion : A3C algorithm converged perfectly in the basic scenario as it has learned to target the monster immediately when the episode begins. This was not the case when adding the curiosity module, and it is due to the confusion of A3C agent caused by the magnitude of the intrinsic reward as seen in the Reward plot. The difference of magnitude lets some irrelevant experiences disrupt the learning process, this perturbation leads to high oscillations of both reward and value function (see the std on the plot).

4.2 Defend the center

For this scenario, the map is a large circle where the agent is in the middle, and monsters spawns along the wall. The only reward he takes from the environment is **+1** if he kills a monster and **-1** if he's dead.

Reward reshaping : Adding an additive cost with regards to the amount of ammo used. For every shot he takes, the agent receives a reward of **-0.5**.

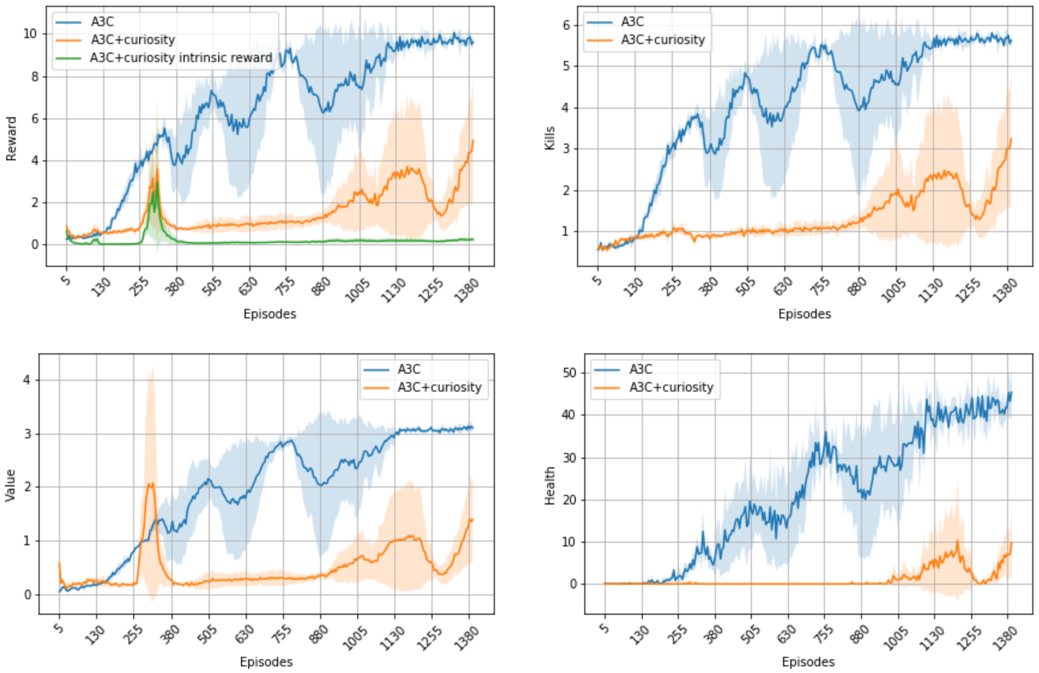


Discussion : A3C performs very well in this scenario and again including the curiosity model decreased its performance, this can be justified in this scenario by the high variance of states since monsters appear and move randomly in a circular map making the states embedding learning difficult as seen in the green curve of the reward that shows very high spikes of the intrinsic module. There is maybe a need to clip or readjust the intrinsic rewards magnitude in order to be of use for the A3C backbone.

4.3 Deadly Corridor

For this scenario, the map is a corridor where the agent is spawned in one end of the corridor, and a green vest is placed on the other end. Three pairs of monsters are placed on both sides of the corridor. The reward he takes from the environment can be positive and negative. It is proportional to the change of distance between the agent and the vest.

Reward reshaping : We divide the game reward by 5 and add a reward of **+100** on killing a monster. We also penalize using ammo by adding a reward of **-0.5** every time the agent shoots. A final reward of **-5** is added whenever the agent looses health. This final reward is finally divided by 100 to keep the oscillations in $[-1, 1]$.



Discussion : This scenario was challenging, we made A3C algorithm converge with lots of efforts put in the reward reshaping and actions composition. This difficulty arises from the early death of the agent as soon as he starts an episode (because of the high and soon damage taken from the first corridor monsters) making the exploration a bit hard. A3C did converge but it is not robust as seen when the curiosity module was added, notice in the top left plot that the small perturbation introduced by the intrinsic reward made the cumulative reward diverge from that of A3C alone. This suggests that a small perturbation may have put the network weights in some topology difficult to lead to a convergence (it needed a lot of episodes in order to restart increasing).

5 Future work

The curiosity module did not lead to good results in our scenarios, and this is maybe due to the difficulty of tuning it (need for a good embedding function ϕ and avoid high perturbation of the network weights. An idea is to clip the intrinsic reward and use a well tuned coefficient for a weighted average between intrinsic and extrinsic rewards.

Plus these scenarios were not well suited for good exploratory algorithms like curiosity-driven, so in order to test the potential of the curiosity module we can try it on another Doom scenario better suited for it, this scenario is my way home where [6] did get very good results even in the complete absence of extrinsic reward.

References

[1] Mehdi Boubnan and Ayman Chaouki. Deep reinforcement learning applied to doom. `https://github.com/Swirler/Deep-Reinforcement-Learning-applied-to-DOOM`, 2018.

[2] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527*, 7(1), 2015.

[3] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 341–348, Santorini, Greece, Sep 2016. IEEE. The best paper award.

[4] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

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

[6] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

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

[8] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.