Deep Learning within the Context of Doom

Saran Ahluwalia
North Carolina State University
ssahluwa@ncsu.edu

Abstract

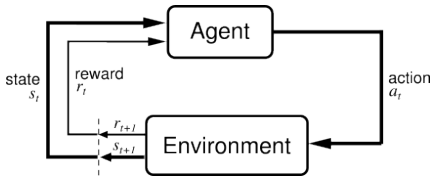
There are two main approaches to reinforcement learning: value based methods that aim to find an optimal Q-function, and policy based ones that directly look for the optimal policy. However most of reinforcement learning problems (such as learning to play games) have large or even continuous states spaces,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

1 Introduction

Deep reinforcement learning (DRL) has exapanded the applications of reinforcement learning to more complex problems. The quintessential example is the defeat of the world champion in Go with AlphaGo Zero. This was a remarkable feat due to the challenge of representing the state-space. In fact, DRL relies on neural networks for the approximation task, thus enabling better prediction (of the state-action value function or the policy) given unstructured data like images as input state.
In this work, we will attempt to train agents to play in specific scenarios of Doom. To do so, we will use the VizDoom package as an environment in order to access states, actions and rewards. We will attempt to use 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).
First, each algorithm will be explored and a justification for their use for solving the RL problem will be extrapolated. In the experiments section we will test each of these aforementioned algorithms on specific scenarios of VizDoom (the basic, deadly corridor and defend the center). Finally, we will include a critical analysis of the algorithms and attempt 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).
Under this framework, the reward hypothesis is any goal that can be formulated as searching for a policy - the optimal policy - that maximizes a state value function $\mathcal{V}^\pi$. This state value function 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 within a 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

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

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

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

Here s' is the state the agent obtained 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. Thus 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. However, such an exact representation of value functions and policies is impossible when the state space is expansive or even continuous, thus the need for approximate reinforcement learning.

3 Deep reinforcement learning

Approximate reinforcement learning attempts to approximate state-action value functions in a "smaller" space. In deep reinforcement learning the approach is to rely on neural networks to perform the approximation. The use of neural networks is especially convenient for games (Atari, FPS etc.) because 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 of the aforementioned scenarios maintain its own settings (possible actions and reward reshaping). These details will be elaborated on during the discussion of 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 frames will constitute our states. Deep Q-learning leverages 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, along with a non-linear activation. The non-linear activivation, 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). However, the network architecture in DQN, when used with single frames as states, does not reveal any notion of motion. In order to ameliorate this, we can use stacked frames as states to solve this problem.
Note that this concerns only the DQN model as there are other models, such as 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 $\mathcal{Q}_{target}(s, a) = r(s, a) + \gamma \underset{a'}{max} \mathcal{Q}(s', a')$ for action a. Since $\mathcal{Q}^*$ is the fixed point of $\mathcal{T}^*$ the optimal Bellman operator, we assume that the function approximating 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 $(\mathcal{Q}_{target}(s, a) - \mathcal{Q}(s, a))^2$. 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 an episode is unrolled, 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 state-action pair of 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 for estimating the state-value function $V(s)$ *thus outputting a scalar*; the second provides a stream if estimates of the the action advantage function $\{A(s, a); a \in \mathcal{A}\}$ *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 $\{\mathcal{Q}(s, a); a \in \mathcal{A}\}$. 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 $\mathcal{Q}(s, a) = A(s, a) + V(s)$.

Aggregation : Aggregating the two streams is not as straightforward as just taking the sum $\mathcal{Q}(s, a) = A(s, a) + V(s)$ because in this case we run into the problem of identifiability due to the fact that $V(s)$ and $A(s, a)$ cannot be recovered. 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

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

As specified previously, we will implement the double dueling deep Q-learning algorithm which makes use of the dueling architecture in a double Q-learning fashion. Through this mechanism we will directly use a prioritized experience replay (that we will explain in the next section) instead of uniform replay.

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 there are a small minutiae of experiences that abet the network in learning 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, there is the introduction of importance sampling weights $w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$, which signifies that for an experience i the network will be receive 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. This alternative leverages a recurrent neural networks - specifically LSTM. With DRQN, the LSTM module is placed right after the convolutional layers in order to better model sequences. Thus, this LSTM network enables us to abstain from representing states as stacked frames. Here, single frames are now enough. The direct benefit of this architecture is that we can now more efficiently process RGB frames.
The one caveat of this model is that as 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. Thus, the LSTM has difficulty in learning 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. For the purpose of this analysis, we decompose each word of this network in order to derive it's representation.

Actor-Critic : The base network of this algorithm attempted 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. A discussion of network tuning will be extrapolated on 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. Each of these agents share what they've learned with each other asynchronously. After a defined number of steps performed 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). Recall that k is equal to n_{max} if the worker has not 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, goading the actor to be certaing 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 its own *intrinsic* reward by exploring the environment. This idea was introduced by [6] and was applied to Doom. It essentially tries to overcome a larger 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 encourages the network 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 the model can do so, the more state loses relevacny. The direct consequence is a lower reward. 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. This is misguided as the agent is sensitive to irrelevant states. That is, 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 a Research Platform for Reinforcement Learning for the game Doom. We applied the above methods on different scenarios of Doom environment using this platform (please refer to github `https://github.com/ahlusar1989/CS234ReinforcementLearning/tree/master/DoomDQN`). The scenarios we used are the following : basic, deadly corridor and defend the center. Please 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.

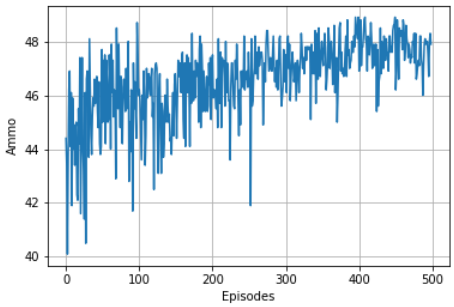
Confidence intervals : When building confidence intervals (for promising experiments), we repeat the experiments 3 times.

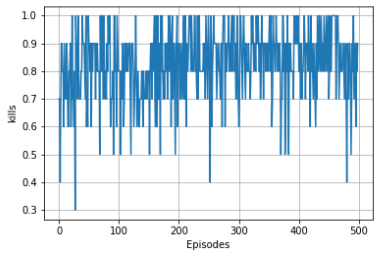
All details regarding training can be found on Github [1].

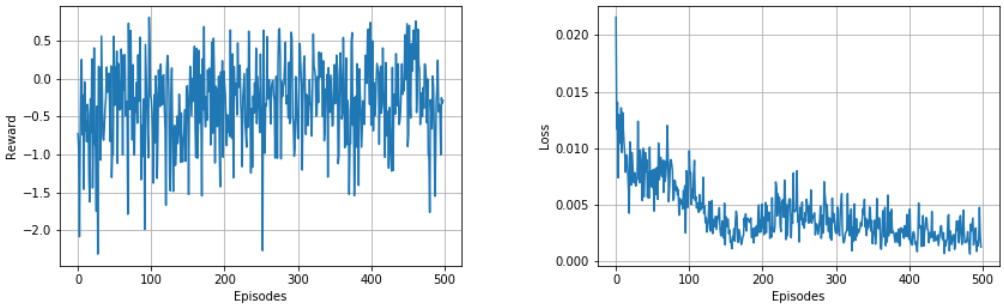
4.1 Basic scenario

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 obtained from the environment is **+100** if the agent kills the monster and **-1** if not.

Deep dueling Q-learning results :



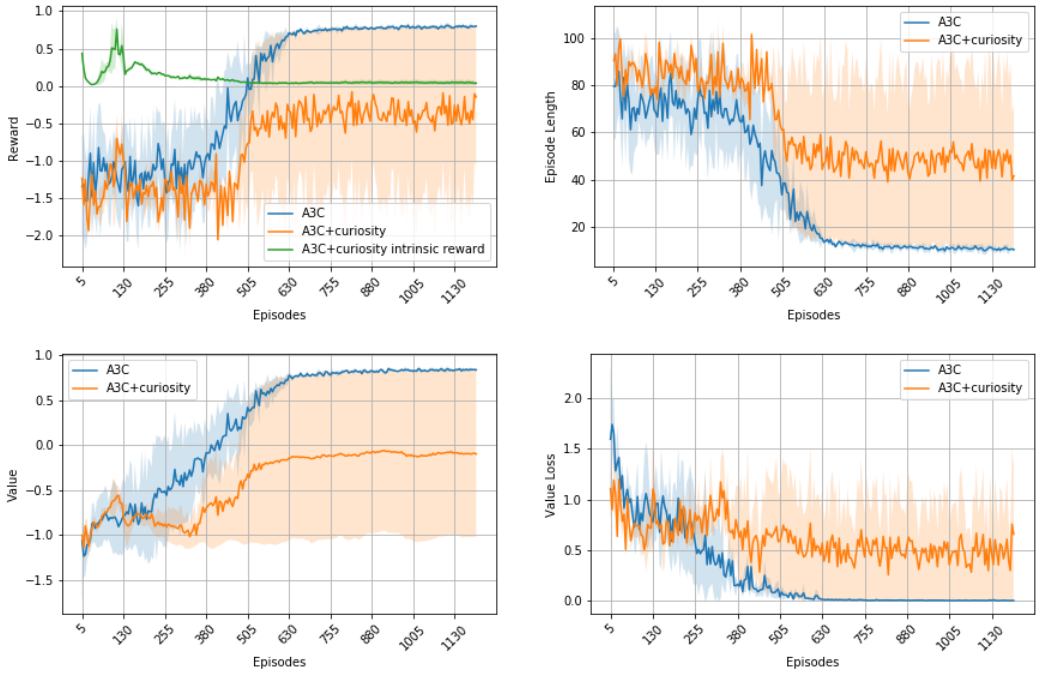




Discussion : Value based algorithms failed to yield satisfying results in the simple basic scenario of Doom (mainly due to the huge oscillations of the loss). Only deep dueling Q-learning converged to a suboptimal policy after 5000 episodes. This required much tuning of the memory size, learning rate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

A3C results with and without using curiosity :

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

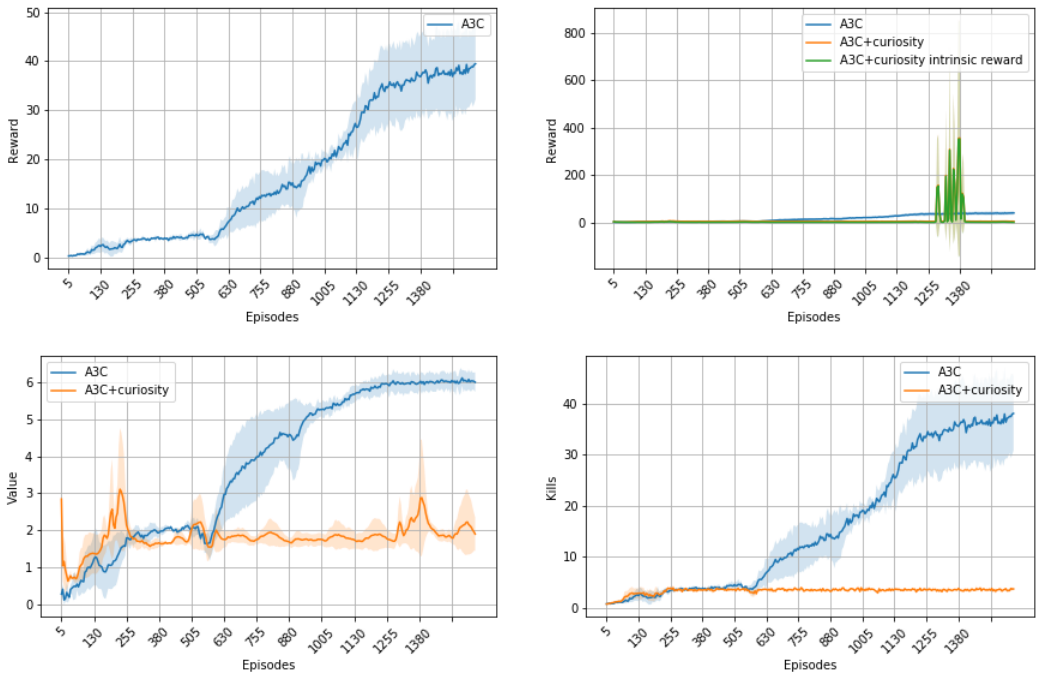


Discussion : A3C algorithm converged perfectly in the basic scenario. The A3C learned to target the monster immediately when the episode begins. This was not the case when adding the curiosity module. This is most likely due to the confusion of A3C agent caused by the magnitude of the intrinsic reward 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**.

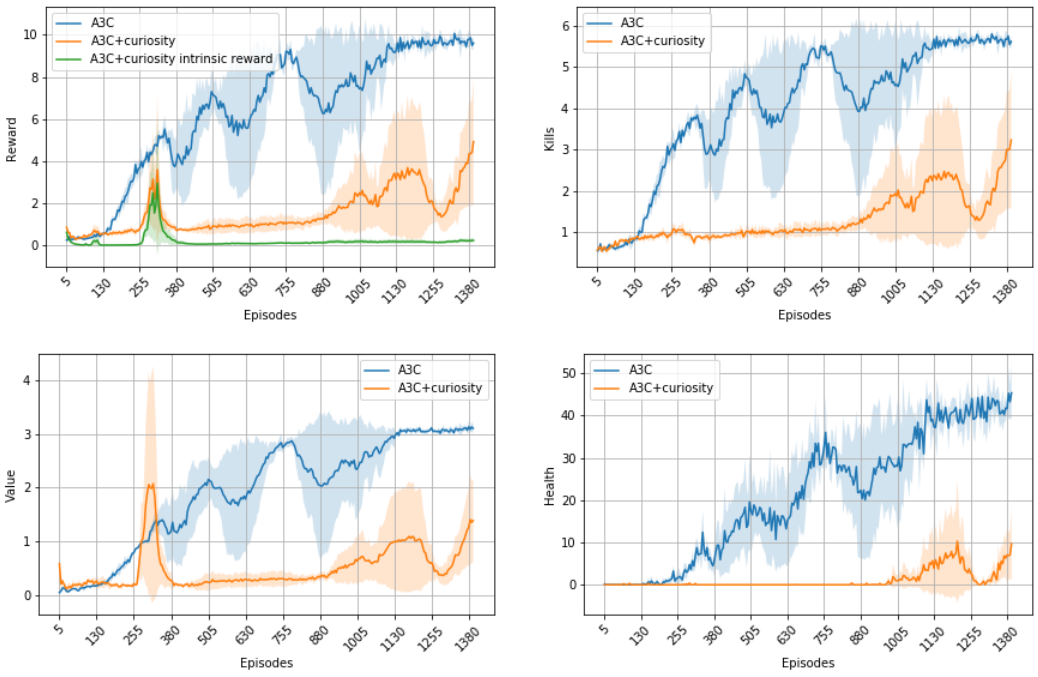


Discussion : A3C performs very well in this scenario. Including the curiosity model decreased performance markedly. This is notable in the figure by high variance of states, as each monster appears to move randomly in a circular map. This makes the states embedding learning difficult as seen in the green curve of the reward. Note that there are very sporadic oscillations of the intrinsic module. There may be a need to clip or readjust the intrinsic rewards magnitude in order to be of use for the A3C base network.

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 the agent obtains from the environment can be positive and negative. This reward 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 due to the fact that we made the A3C algorithm converge with more emphasis placed on the reward reshaping and actions composition. This difficulty arises from the early death of the agent as soon as it starts an episode (because of the high probability of damage incurred in the first corridor's adversaries). This makes the exploration more intractable. A3C did converge; however, this was not robust - particularly with the addition of the curiosity module. Notice in the top left plot that the small perturbation introduced by the intrinsic reward caused the cumulative reward to diverge from that of A3C. This phenomenon suggests that a small perturbation may have put the network weights in some topology difficult to lead to a convergence.

5 Future work

The curiosity module did not lead to improved results in the experimental scenarios. This is due to the difficulty of tuning the module; this requires the need for a better embedding function ϕ in order to avoid high perturbation of the network weights. One idea is to clip the intrinsic reward and use a well tuned coefficient for a weighted average between intrinsic and extrinsic rewards.
Another detriment is that the experimental scenarios used here were not well suited for a exploratory algorithm, such as curiosity-driven. Hence, in order to test the potential of the curiosity module one could attempt to experiment with another Doom scenario better suited for exploration. This scenario is described here [6]. In this paper, the results produced were a significant improvement - even in the complete absence of extrinsic reward.

References

[1] Saran Ahluwalia. Deep reinforcement learning applied to doom. `https://github.com/ahlusar1989/CS234ReinforcementLearning/tree/master/DoomDQN`, 2019.

[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.