Exploring Deep Reinforcement Learning with Super Mario Bros

Amine SADEQ
CentraleSupelec, ENS Paris-Saclay
Paris, France
amine.sadeq@student.ecp.fr

Otmane SAKHI
CentraleSupelec, ENS Paris-Saclay
Paris, France
otmane.sakhi@student.ecp.fr

Abstract

In this paper, we try to compare different deep reinforcement learning algorithms by applying them to the 1985 NES game Super Mario Bros according to different reward scenarios from dense to sparse rewards. The idea of this project is to see in what extent engineering the reward is a crucial part in designing an agent and how the recent breakthroughs in deep reinforcement learning which goes from using more advanced learning algorithms like A3C or PPO, or even introducing a new form of intrinsic reward creating a curious agent could lessen the importance of reward designing to the point of dropping completely the environment reward for the agent to learn.

1 Introduction

One of the most interesting topics in machine learning is reinforcement learning. The main idea is that an agent tries to learn, by interacting with an uncertain environment and using the experience gathered, to optimize some goal given in the form of cumulative rewards.

After the Deep Learning breakthrough, Deep Reinforcement Learning has become really popular due to its success outperforming humans in Atari Games [4] or beating GO champions with DeepMind's Alpha Go Zero [9]. It is most useful for high-dimensional state spaces where Value and Q iteration algorithms are unusable and approximate algorithms are more suitable making it easier to approximate any form of Value functions and Policies from very complex input data.
In this paper, we will focus on video games and especially on the Super Mario Bros NES game of 1985. Video games represent perfect simulations for challenging real world problems. We can then use them to study and test the performance of new approaches.

Deep learning through the last years, made huge advances especially in extracting features from raw sensory data (images in our case), which can be very beneficial to our use in the deep reinforcement learning framework. However Deep learning is confronted to a lot of challenges in the reinforcement learning point of view. In deep learning, we assume that our data comes from the same distribution while in reinforcement learning the distribution changes while advancing in a game for example. This kind of behaviour tends to make vanilla deep learning algorithms unstable and really hard to converge to acceptable solutions.

Reinforcement Learning in the other hand knew some interesting advancement in the last years from DQN [4] which was able to beat Atari Games to the use of new forms of reward [5] combined with better algorithms which led to great performances in complex games without any reward engineering. In this paper, we will begin by introducing briefly the algorithms we used, and present the experiments we built to test the different approaches.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

2 Deep Reinforcement Learning

Deep Reinforcement Learning falls within the framework of Approximate Reinforcement Learning with a neural network used to approximate our objective function. In the literature, we refer to Value/Q methods when our objective is to approximate the Value/Q function which are mainly used for their simplicity and speed especially when dealing with small finite action spaces. Policy gradient methods define the algorithms that try to approximate the policy instead and can be used in cases where Value based methods fail such as problems with large or continuous action spaces or optimal stochastic policies. Recent state of the art methods tend to combine the best of the two worlds and use Actor-Critic methods which approaches the Value function and the Policy at the same time resulting in better performances. In the next section, we're gonna briefly present you the algorithms we tried to use for our problem, the Deep Q-Network with memory replay [4] algorithm as a first approach as it was introduced as a universal algorithm that dealt efficiently with basic Atari games. the Asynchronous Advantage Actor Critic [3] which is used in the Curiosity Driven Learning paper [5] and on which we based the majority of our work, and Proximal Policy Optimization [8] as it showed promising performances in complex games such as Dota 2 and is used in the paper of Large scale curiosity [1] which demonstrated great performances on the Super Mario Bros game.

2.1 Deep Q-Network with Memory Replay

As we mentioned before, Deep Reinforcement Learning relies on learning the objective function with neural networks, which assume that the data is sampled from i.i.d distributions like the majority of supervised learning models. This is achieved for example by randomizing the data among batches to get independent and identically distributed samples in the same batch. in classical Q-learning, that's rarely the case. as we simulate our trajectory, we simply gather successive data samples which are highly correlated, thus, Memory Replay came to solve this problem by introducing a buffer where we store all the information $e_t = (s_t, a_t, r_t, s_{t+1})$ about the state at each time step t. For instance, we store at most M frames in our case with M the capacity of the buffer and sample a mini-batch from it to train the neural network by doing a Q-learning update as this method falls within the Value based methods. This is done in the hope of forming an input batch which is stable enough for training. As we randomly sample from the replay buffer, the data is more independent of each other and closer to i.i.d. This method boils down to the algorithm DQN algorithm which is described better in [4].

Unfortunately, even if the DQN algorithm with Memory Replay gave good results when applied to most Atari Games, its performance on the Super Mario Bros game wasn't satisfactory mainly due to the fact that Memory Replay isn't justified in a game where the action at a current state relies on all the information from the previous states. Fox example, the agent needs to keep pushing the jump button in 12 successive frames to get past obstacles for example. The use of recurrent policies is thus important in Super Mario Bros and vanilla DQN with Memory Replay is not suitable for such policies. In the experiments section, we will not present the results of this algorithm as they were very poor.

2.2 Asynchronous Advantage Actor Critic (A3C)

The A3C algorithm was used by the [5] throughout all of their experiments. The algorithm can be explained by decomposing it to three parts :

Asynchronous : to deal with the changing distributions in reinforcement learning, A3C uses multiple asynchronous agents. Where each one (thread) interacts with it's own environment independently. As each agent's actions are independent from the others, each one gathering different experiences which the deep learning models uses to learn it's parameters. This way the data selected is much more diverse and leads to small distribution changes.

Advantage : We replace in the policy iteration the discounted rewards by the advantage function, $A = Q(s, a) - V(s)$. This function allows the agent to learn how much it's action are better than a baseline which is an estimate of V.

Actor Critic : Actor critic is a combination of policy iteration and Q learning, it benefits from both approaches. the Network estimates both the value function $V(s)$ and the policy $\pi(s)$. Then the agent will use it's value function estimate (the critic) to update the policy (the actor). Both

gradients will help the network learn faster and better. In order to compute a single update, the algorithm first selects actions using its exploration policy for up to t_{max} steps or until a terminal state is reached. He receives then up to t_{max} rewards from the environment. The algorithm then computes gradients for n-step Q-learning updates for each of the state-action pairs he received. Each n-step update uses the longest possible sequence of returns up to t_{max} updates. It was also found that adding the entropy $H(\pi(s_t)$ of the policy to the objective function improved exploration.

Algorithm 1 Asynchronous Advantage Actor-Critic algorithm

// Assume global shared parameter vectors θ and θ_v and global shared counter T = 0.
// Assume thread-specific parameter vector θ' and θ'_v.
Initialize thread step counter t $\leftarrow 1$.
repeat
  Reset gradients: $d\theta$ gets 0 and $d\theta_v$ gets 0.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$.
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$.
    Receive reward r_t and new state s_{t+1}.
    $t \leftarrow t + 1$.
    $T \leftarrow T + 1$.
  until terminal s_t **or** $t - t_{start} == t_{max}$

  R = $\begin{cases} 0 & \text{for terminal } s_t. \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t. \end{cases}$
  for $i \in \{t-1, ..., t_{start}\}$ **do**.
    $R \leftarrow r_i + \gamma R$.
    Accumulate gradients wrt $\theta' : d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i, \theta')(R - V(s_i, \theta'_v))$
    Accumulate gradients of the entropy wrt $\theta' : d\theta \leftarrow d\theta + \beta \nabla_{\theta'} H(\pi(s_i, \theta'))$
    Accumulate gradients wrt $\theta'_v : d\theta_v \leftarrow d\theta_v + \frac{\partial (R - V(s_i, \theta'_v)^2}{\partial \theta'_v}$
  end for
  Perform Asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

In the Curiosity paper [5], Using only curiosity as a reward with A3C algorithm didn't lead to good performances and it allowed the agent to cross only 30% of the first level. One way of getting better performances is to switch to a more advanced algorithm which tries to deal with stability issues. In the Large Curiosity [1], they used PPO with only curiosity as a reward and have been able to finish the Super Mario Bros game. Their results encouraged us to use this algorithm in our experiments.

2.3 Proximal Policy Optimization (PPO)

The intuition behind the PPO [8] algorithm is that the Policy should not change drastically after the update of our parameters. Compared to algorithms such as ACKTR [10] or TRPO [7] that relies on some heavy approximations to achieve that, PPO implement this intuition in a simple way making it faster with comparable performances to the state of the art methods. with that said, PPO also belongs to the category of policy gradient. It adopts the actor-critic architecture, but modifies how the policy parameters of the actor are updated. In its implementation, we maintain two policy networks. The first one is the current policy that we want to refine $\pi_\theta(a_t|s_t)$. The second is the policy that we last used to collect samples $\pi_{\theta_k}(a_t|s_t)$. With these two policies, we can use importance sampling, we evaluate a new policy with samples collected from an older policy. This clearly improves sample efficiency. But as we refine the current policy, the difference between the current and the old policy is getting larger. The variance of the estimation will increase which will make our results inaccurate. So for every small number of iterations, we synchronise the second network with the refined policy again. To implement the intuition discussed above, we compute the ratio between the new and the old policy $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ with which we write the loss function to

optimize: $L_{\theta_k}(\theta) = \mathbb{E}[\sum_{t=0}^{T} min(r_t(\theta)A_t^{\pi_k}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t^{\pi_k})]$ with $A_t^{\pi_k}$ an estimation of the Advantage. So, if the ratio of both policies falls outside our trust region ($[1-\epsilon, 1+\epsilon]$ in our case), it will be clipped, we then take the minimum of the clipped and unclipped objective. In [8] , they used an ϵ of 0.2 which we will also use for our experiments. The algorithm is simple and can be easily described as follows :

Algorithm 2 Clipped Objective PPO

Input: initial actor-critic parameters θ_0, clipping threshold ϵ.
for iteration $= 0, 1, 2, ...$ **do**.
  for actor $= 1, 2, ..., N$ **do**
    Run policy π_{θ_k} in environment for T timesteps
    Compute Advantage estimates $A_1, ..., A_T$ (with the critic help)
  end for
  Optimize $L_{\theta_k}(\theta)$ wrt θ , with K epochs and minibatch size $M \leq NT$
  $\theta_k \leftarrow \theta$
end for

3 Curious Agents & Intrinsic Reward

In the classical methods used in Reinforcement learning, the agent collects a form of reward received from the environment. The better the reward is engineered, the faster and better our agent can learn. The idea behind Curiosity-Driven learning is to make the agent learn by himself, mimicking the behaviour of humans, which can be modelled as curious agents wanting to discover and explore the unknown. This idea was brought by [5] and was applied to Doom and Mario. Curiosity came to tackle the reward designing part, which is essential in all the approaches we saw earlier. Curiosity helps when facing an environment where sparse rewards and delayed signals are all we get from the environment which makes it difficult for our algorithms to map actions to reward signals. Thus, feeding the agent with a new form of intrinsic reward based on the exploration of unpredictable new states makes it learn useful patterns to be able to discover more. In order to model a curious agent, [5] based the intrinsic reward on the agent's ability to predict the next state, given the current state and the action taken, the more the next state is predictable, the less curious our agent will be, thus the less curiosity reward it will get. As the states in their raw format contain noise and irrelevant information, the curiosity framework introduces a model that can learn an embedding of the states on which the evaluation of our agent will be done. This is achieved throught the introduction of the Intrinsic Curiosity Module (ICM) which is defined by the inverse dynamics and forward dynamics models which we will describe in the next section.

ICM Module

The ICM is composed of two networks : the forward dynamics model and the inverse dynamics model. They both share an encoder implemented as successive convolutions that maps a state which is an images s_t to a vector feature $\phi(s_t)$ in a much smaller dimension.

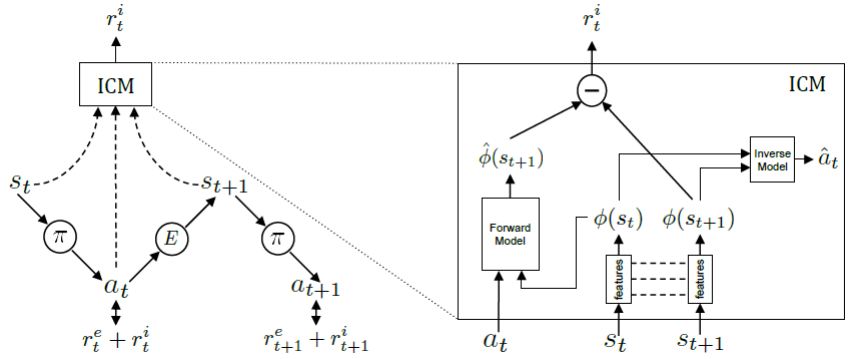


Figure 1: Intrinsic Curiosity Module from the [5] paper.

3.1 Forward Model

In the forward model, we try to predict the next state in the feature space $\phi(s_{t+1})$ from the current state $\phi(s_t)$ and the action that was taken a_t. The loss for this network can be then written as follow :

$$L_F(\hat{\phi}(s_{t+1}), \phi(s_{t+1})) = \frac{1}{2}||f(\phi(s_t), a_t) - \phi(s_{t+1})||_2^2$$

where $f(\phi(s_t), a_t)$ is the output of the forward model. We can then define the intrinsic reward r_t^i as with η a scaling parameter :

$$r_t^i = \eta L_F(\phi(s_t), \phi(s_{t+1})) = \frac{\eta}{2}||f(\phi(s_t), a_t) - \phi(s_{t+1})||_2^2$$

Inverse Model

In the inverse model, we try to predict the action a_t that was taken from the current state $\phi(s_t)$ and the next state $\phi(s_{t+1})$. The loss for the inverse model can be written as :

$$L_I(\hat{a}_t, a_t) = h(g(\phi(s_t), \phi(s_{t+1})), a_t)$$

where $g(\phi(s_t), \phi(s_{t+1}))$ is the output of the model and h is the cross entropy loss.

This model doesn't play any role in the reward returned but it will help improve the encoding to the feature space as the loss is back-propagated through the embedding as well.

4 Experiments

We define a good experimental setup to compare the different algorithms we use when dealing with either sparse of dense rewards. This section will describe the multiple setups we used and the different results we got.

4.1 Environment :

We evaluate all of our approaches on the Super Mario Bros environment [2], specifically on the first level. We compare all of the methods in their ability to make the agent learn how to finish the first level. The action space of the agent consist of 14 discrete actions composed from 7 actions (left, right, up, down, run, jump, no action) and some of there combinations. Episodes are terminated either when the agent dies, finishes the level or runs out of time (2000 steps).

Reward Settings:

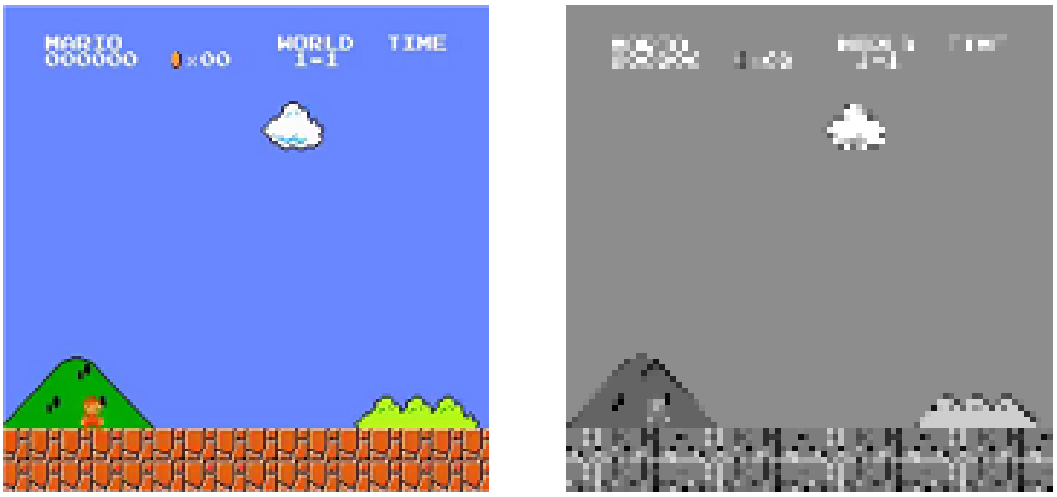
As we mentioned before, we defined two reward settings to test the limitations of each algorithm and quantify the impact of the curiosity module.

Dense Reward: In this setting, the agent gets a reward for each step he makes. the reward is positive if he goes forward (to the direction of the flag) and is negative when he makes a step back. the reward is also computed according to the time left for the agent, he receives negative rewards as the time is ticking. It also gets positive reward from accumulating coins, stepping on enemies or even getting the mushroom. Of course, if the agent dies, he gets a a bigger negative reward and gets the ultimate reward when he finishes the game. All of these signals define the reward given to the agent to learn in this setting. To make the settings unified, we divide the given rewards by a constant to get it in the $[-1, 1]$ so as to use a constant learning rate for all of our experiments. For more details, you can refer to our implementation in our github [6].

Sparse Reward: In the Sparse Reward environment, we give our agent +1 reward when he crosses each 5% of the game, making the mapping between the actions taken and the reward signal much difficult to infer. Our agents gets -1 reward when he dies or runs out of time and gets +5 when he reaches the flag and thus complete the level. This setting makes the job really challenging for our algorithms and gives the curiosity module a good setting where we can test its impact.

4.2 Training Details:

The Super Mario agent was trained using visual inputs that are converted into grayscale, re-sized to 82 and concatenated into 4 frames representation making the input of all of our models of size (4,84 84). This makes the training faster without any major loss of information. In the rest of the paper, we will refer to this processed state as the actual state s_t Following the asynchronous training protocol in A3C, all the agents were trained asynchronously with twenty four workers on CPUs only using an ADAM optimizer with its parameters not shared across the workers. However, for the PPO implementation, we used a twelve CPU machine with an Nvidia Quadro GPU as it was faster for the training phase.



(a) Original environment state

(b) Preprocessed state

Actor Critic : The Actor-Critic architecture we used was heavily inspired from the one used in [5]. The input state s_t is passed through a sequence of four convolution layers with 32 filters each, kernel size of 3x3, stride of 2 and padding of 1. We used the exponential linear unit ELU as activation for each convolution layer. The output of the last convolution layer is fed into a LSTM with 512 units for the A3C implementation and a GRU with 512 units for the PPO implementation. We connect the LSTM output features to two separate fully connected layers to predict the action and the value function.

Intrinsic Curiosity Module (ICM) architecture : The intrinsic curiosity module consists of the forward and the inverse model. We first map the input state s_t into the feature vector through a down sampling layer of pool size 2 followed by series of four convolution layers, with 32 filters each, kernel size 3x3, stride 2 and padding of 1. We used the exponential linear unit ELU as activation for each convolution layer. We get as an output a feature vector of size 288. For the inverse model, both feature vector for s_t and s_{t+1} are concatenated and fed to a fully connected layer of 256 units followed by another fully connected layer with 14 units to predict one of all the possible actions. For the forward model, we concatenate the feature vector of s_t and a one encoding of the action a_t and give it as input to a sequence of two fully connected layers with 256 and 288 units respectively.

Exploration : In the training phase, we select 90% of the workers to sample the action from a multinomial distribution from the outputted probabilities of our actor network, and the rest of the workers execute the greedy policy to balance between exploration and exploitation.

Training parameters : We denote $L_{ac}(s_t, \theta)$ the loss for the actor critic model, $L_F(\hat{\phi}(s_{t+1}), \phi(s_t))$ the loss for the forward model of the ICM module and $L_I(\hat{a}_t, a_t)$ the loss for the inverse model. We can then write the total loss as :

$$L_T = \lambda L_{ac} + (1 - \beta)L_I + \beta L_F$$

where λ represents the importance of the policy gradient loss againt the importance of learning the intrinsic reward, and β represents the importance of the forward loss against the inverse loss.

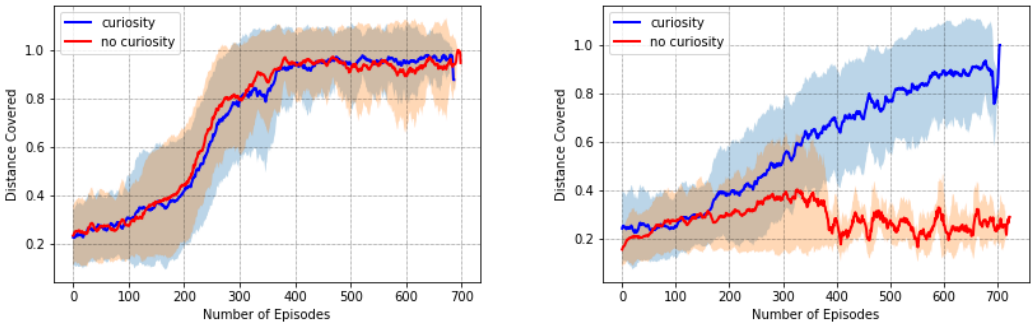
In all of our experiments, we fixed λ to 1.0, β to 0.2 and η to 0.2 as to bound the intrinsic reward into comparable range with the extrinsic reward especially when the agent is still curious. We adopted a learning rate of 1e-4 for the actor critic model and 1e-3 for the ICM module. As for The PPO algorithm, we adopted a batch size of 12 as we didn't have enough workers in the GPU machine.

4.3 Results

To be more demonstrative, we adopted the percentage of the distance covered by our agent before finishing the episode as a metric for comparison between the different models as it shows how our agent progressed through out its learning. Better models can be then defined as the models that led the agent to quickly learn how to reach the final goal.
We begin by demonstrating the impact of the curiosity add-on on both reward settings with the A3C algorithm as a backbone. We thus trained our agent using only the extrinsic reward r_t^e and both the extrinsic and intrinsic reward $r_t = r_t^e + r_t^i$, with the extrinsic reward depending on the setting we're experimenting on. We then try to compare the performances of the learning algorithms and see how can they impact the convergence of the curious agent as well. For this, we adopt the curiosity framework which defines the reward as the sum of intrinsic and extrinsic rewards and see which algorithm between A3C and PPO performs better in the two reward settings we defined.

A3C: Impact of Curiosity in the different rewards settings

We can see from the figure (a) that in the dense reward setting, using curiosity doesn't improve the performance of the algorithm. If the reward is well engineered, there's no need for an intrinsic reward. However in the figure (b), we can see that for the sparse setting, without the intrinsic reward the algorithm can't even finish the game and gets stuck at 30% of the level on average. Adding the curiosity reward helps the agent to explore more and continue learning until finishing the game. One can be tempted to always use the curiosity module as if it doesn't improve the performances, it wouldn't damage them as we have seen in the dense reward. But adding this module has a computational cost even if it can be minimal when using a powerful machine.

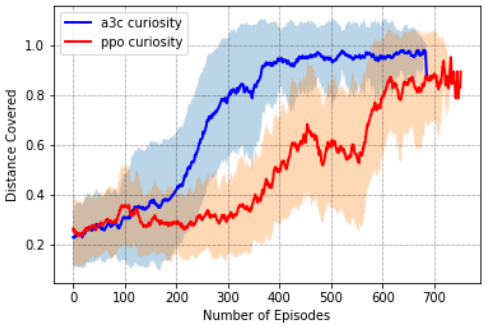


(a) Dense Reward Setting

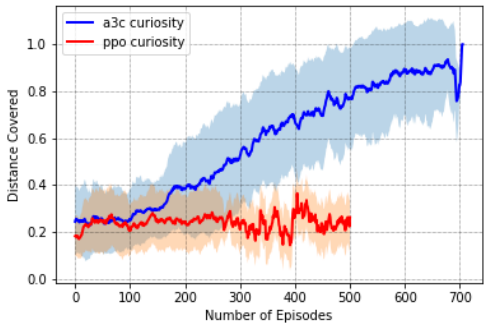
(b) Sparse Reward Setting

PPO vs A3C: Impact of the choice of the learning approach

We can see from the figure (a), that for the dense reward setting, the PPO model is able to finish the game too. However, it's much slower than the A3C model and much more unstable. This may be due to the small number of workers we used for the PPO model. In [1], they used 2048 workers, outperformed A3C and finished all the Super Mario Bros levels. In the figure (b) for the sparse reward setting, we can see that PPO gets also stuck on 30% of the level. this is probably due to the same reasons as before but also because 30% of the game marks the first encounter of the agent with a pitfall which represent a difficult obstacle in the game and once the agent has learned how to overcome it, the rest of the level becomes easy for him. This empirical argument was noticed in the training phase and can also be deduced from the graphs we're showing, as we can see that for both learning algorithms, the learning curve increases after the 30% cap.



(a) Dense Reward Setting : PPO vs A3C



(b) Sparse Reward Setting : PPO vs A3C

5 Conclusion

In this project, we tried to explore some recent deep reinforcement learning approaches and applied them to Super Mario Bros, one of the NES classic but yet challenging game. We demonstrated the impact of the curiosity module especially in the sparse reward settings and the power of the A3C algorithm as an efficient CPU based algorithm. The PPO algorithm however was proven to work well with curiosity in [1] but needs huge computational resources which is not in our disposal.

References

[1] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. In *ICLR*, 2019.

[2] Christian Kauten. Super Mario Bros for OpenAI Gym. `https://github.com/Kautenja/gym-super-mario-bros`, 2018.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[5] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017.

[6] Otmane Sakhi & Amine Sadeq. Super mario bros learning with extrinsic/intrinsic rewards. `https://github.com/sadeqa/Super-Mario-Bros-RL`, 2019.

[7] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[9] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[10] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger B. Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *CoRR*, abs/1708.05144, 2017.