In Order To Finish The Mission

Deep Q-Learning has been successfully applied to a wide variety of tasks in the past several years. However, the architecture of the vanilla Deep Q-Network is not suited to deal with partially observable environments such as 3D video games. For this, recurrent layers had been added to the Deep Q-Network in order to allow it to handle past dependencies. We here use Minecraft for its customization advantages and design two very simple missions that can be frames as Partially Observable Markov Decision Process. We compare on these missions the Deep Q-Network and the Deep Recurrent Q-Network in order to see if the latter, which is trickier and longer to train, is always the best architecture when the agent has to deal with partial observability.

Deep Reinforcement Learning has been highly active since the successfull work of Mnih et al. (2013) on Atari 2600 games. From that moment, a lot of methods have been used on a wide range of environments in order to make an agent reach an objective (Justesen et al., 2017). These environments can be framed as Markov Decision Processes (MDPs) defined by the tuple fragmentsS,A,P,Rwhere at each timestep ttitalic_t of the environment, the agent observes a state sSs\inSitalic_s italic_S, takes the action aAa\inAitalic_a italic_A, ends up in a new state sP(s,a)similar-tosuperscripts^\prime\sim P(s,a)italic_s start_POSTSUPERSCRIPT end_POSTSUPERSCRIPT italic_P (italic_s , italic_a) and receives a reward rRr\inRitalic_r italic_R. In order to find the policy \piitalic_ (the choice of the action from the state) maximizing the sum of rewards, one can use the Q-Learning algorithm (Watkins and Dayan, 1992) to estimate the expected sum of rewards from a state ssitalic_s and an action aaitalic_a. This expected sum of reward is called the Q-value and is noted :

Q(s,a)=[Rt+1+Q(St+1,At+1)|St=s,At=A].delimited-[]formulae-sequencesubscript1conditionalsubscript1subscript1subscriptsubscriptQ(s,a)=\mathbbE[R_t+1+\gamma Q(S_t+1,A_t+1)|S_t=s,A_t=A].italic_Q (italic_s , italic_a) = blackboard_E [italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT + italic_ italic_Q (italic_S start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT) | italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_s , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_A] .
This is actually a discounted sum with [0,1]01\gamma\in[0,1]italic_ [0 , 1]. This discount factor allows us to handle how important future rewards are, but also prevents the algorithm from an infinite loop. Although useful, finding the Q-value with the Q-Learning algorithm for all the state-action pairs is simply intractable on complex environments involving wide states and actions spaces. To solve this issue, we can approximate this Q-value with a neural network (called Q-Network) which takes the state in input and outputs the approximated Q-value for each possible actions. This technique is called Deep Q-Learning (Mnih et al., 2013) and has moved the state of the art on several Atari games to a human-level performance (Mnih et al., 2015). In the context of vision-based environments, these Deep Q-Networks usually take in input the raw image of the current state and use a convolutional architecture

to extract feature maps. These feature maps then go to fully-connected layers with the last one outputting the approximated Q-values. The fact of taking only the current observation as an image in input and choose the action knowing only that information makes the assumption that the observation holds all the knowledge about the current state of the environment. This is unfortunately not always the case. Let's take for instance the Atari game Pong. The image gives us the information about the location of the ball but doesn't give us neither the information about the trajectory nor the speed. This is also the case for 3D environments such as Doom (a first-person shooter game). These environments where an observation gives us only a partial information about the current state of the game can be framed as Partially Observable Markov Decision Processes (POMDPs) defined by the tuple fragmentsS,A,P,R,,Owhere SSitalic_S, AAitalic_A, PPitalic_P and RRitalic_R are the same as the MDP. The main difference is that the agent no longer observes the whole current state but instead receives an observation oo\in\Omegaitalic_o roman_ from the probability distribution oO(s,a)similar-too\sim O(s,a)italic_o italic_O (italic_s , italic_a). In order to find the best policy \piitalic_ for this POMDP, we can still use the Q-Learning algorithm and approximate the Q-value with a neural network. However, the Deep Q-Networks aforementioned are not meant to deal with this partial observability as the network needs more than one observation to understand the current state. In practice, Mnih et al. (2013) used the last four frames as input of their DQN to deal with partially observable environments such as Pong. But the use of stacked frames has drawbacks. The number of stacked frames is fixed and our network can't handle older things than this fixed size. Some complex environments may need the agent to remember events that occurred tens of frames before. In order to fix this memory issue of the DQN, Hausknecht and Stone (2015) proposed to add a recurrent layer after the convolutional part of the original DQN. They used LSTM cells (Hochreiter and Schmidhuber, 1997) to handle long term dependencies and called this architecture the Deep Recurrent Q-Network (DRQN). They showed that the DRQN could outperform the vanilla DQN on some of the Atari 2600 games. Lample and Chaplot (2017) also showed that the DRQN architecture could achieve good results on complex 3D environments such as Doom.

In this paper, we want to check if using a Deep Recurrent Q-Network, which is trickier and longer to train than the classical DQN, is always the best choice when we are in a POMDP, even if the mission to solve is simple. We thus chose a 3D environment called Minecraft, which is a sandbox video game where an agent can live, build, and explore a free world. With the help of the Project Malmö111https://www.microsoft.com/en-us/research/project/project-malmo/ (Johnson et al., 2016) and Gym-Minecraft222https://github.com/tambetm/gym-minecraft, we were able to design missions for an agent that can be solved with Deep Reinforcement Learning. This environment has the advantage of being highly customizable. In order to see if a modification of the network architecture is always needed to deal with the partial observability, we compare the performances of three models :

- A Deep Q-Network constituted of a convolutional neural network that takes in input only the current frame.

- The Mnih et al. (2013) Deep Q-network which is similar to the previous one but takes the last four frames in input.

- A Deep Recurrent Q-Network with a convolutional part taking the current frame as input, then followed by an LSTM before the feedforward layers.

We designed two simple environments resolvable by a vanilla DQN :

- The Basic : the agent is in a 7x7x7 blocks closed room where a goal has been placed randomly on one of the axis (see figure 1).

- The Cliff Walking : the agent has to walk through a pathway of blocks surrounded by lava to reach is goal.

We measure the performances of the models on both the training and evaluation phases. Our implementations can be found on our GitHub repository333https://github.com/vincentberaud/Minecraft-Reinforcement-Learning.

Some works have already tried to apply Reinforcement Learning on Minecraft. Alaniz (2018) used Minecraft to make their agent place a cube at the right place. They chose to use a model-based approach where a convolutional neural network (called the Transition Model) takes as input the last four frames and outputs the next frame and the associated reward. They then used the Monte Carlo Tree Search algorithm to do planning and choose the action that would maximize the expected sum of rewards based on the Transition Model predictions. Their results showed that their method seems to learn faster than the DQN but also seems to be less efficient as the number of training steps increases.

Oh et al. (2016) had an approach closer to ours. They used Minecraft to make an agent resolve a maze where there are two ending paths with an indicator (at the beginning of the maze) showing the right one to choose. Their agent uses a Deep Recurrent Q-Network but with an external memory and a feedback connection. They called their architecture the Feedback Recurrent Memory Q-Network (FRMQN). Their method outperformed the DRQN and the DQN on their environment.

Finally, Matiisen et al. (2017) made their agent find a goal in a room with hierarchical tasks by using a transfer learning technique between a neural network (Teacher) with a partial view (POMDP) and an agent which is another neural network (Student) performing tasks more and more complex. They called their approach Teacher-Student Curriculum Learning (TSCL) and used the Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017) to learn the policy. Our use of Gym-Minecraft is inspired by their work and our basic mission is the

first mission their agent had to accomplish.

We choose here to highlight the behaviour of the DRQN versus two DQNs versions (one handling some partial observability and not the other one) on simpler environments than the ones just evoked in order to have a baseline of the need of the DRQN when it comes to partial observability.

3 Methods

3.1 Q-Learning

In Reinforcement Learning, an agent has to learn the policy \piitalic_ that maximizes the sum of discounted future rewards. This sum is called the return and is noted :

Gt=Rt+1+Rt+2+2Rt+3+…=k=0kRt+k+1.subscriptsubscript1subscript2superscript2subscript3…superscriptsubscript0superscriptsubscript1G_t=R_t+1+\gamma R_t+2+\gamma^2R_t+3+...=\sum_k=0^\infty\gamma^% kR_t+k+1.italic_G start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT + italic_ italic_R start_POSTSUBSCRIPT italic_t + 2 end_POSTSUBSCRIPT + italic_ start_POSTSUPERSCRIPT 2 end_POSTSUPERSCRIPT italic_R start_POSTSUBSCRIPT italic_t + 3 end_POSTSUBSCRIPT + … = start_POSTSUBSCRIPT italic_k = 0 end_POSTSUBSCRIPT start_POSTSUPERSCRIPT end_POSTSUPERSCRIPT italic_ start_POSTSUPERSCRIPT italic_k end_POSTSUPERSCRIPT italic_R start_POSTSUBSCRIPT italic_t + italic_k + 1 end_POSTSUBSCRIPT .

Knowing this, we can say that the goodness of a state (how rewarding it is to be in this state) is the return of following the policy \piitalic_ starting from that state ssitalic_s . This is called the value and is written :

V(s)=[Gt|St=s].subscriptsubscriptdelimited-[]conditionalsubscriptsubscriptV(s)_\pi=\mathbbE_\pi[G_t|S_t=s].italic_V (italic_s) start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT = blackboard_E start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT [italic_G start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT | italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_s] .

We can now be even more precised and write how good it is of being in the state ssitalic_s and choose the action aaitalic_a from that state. This is called the Q-value and can thus be written :

Q(s,a)=[Gt|St=s,At=a]=[Rt+1+Gt+1|St=s,At=a]=[Rt+1+Q(St+1,At+1|St=s,At=a]fragmentssubscriptfragments(,)subscriptfragments[subscriptconditionalsubscript,subscript]subscriptfragments[subscript1subscript1conditionalsubscript,subscript]subscriptfragments[subscript1fragments(subscript1,subscript1|subscript,subscript]\beginsplitQ(s,a)_\pi&=\mathbbE_\pi[G_t|S_t=s,A_t=a]\\ &=\mathbbE_\pi[R_t+1+\gamma G_t+1|S_t=s,A_t=a]\\ &=\mathbbE_\pi[R_t+1+\gamma

Q(S_t+1,A_t+1|S_t=s,A_t=a]\endsplitstart_ROW start_CELL italic_Q (italic_s , italic_a) start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT end_CELL start_CELL = blackboard_E start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT [italic_G start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT | italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_s , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_a] end_CELL end_ROW start_ROW start_CELL end_CELL start_CELL = blackboard_E start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT [italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT + italic_ italic_G start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT | italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_s , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_a] end_CELL end_ROW start_ROW start_CELL end_CELL start_CELL = blackboard_E start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT [italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT + italic_ italic_Q (italic_S start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT | italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_s , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_a] end_CELL end_ROW (1)
In order to find the optimal policy (noted *subscript\pi_*italic_ start_POSTSUBSCRIPT * end_POSTSUBSCRIPT), we want to find the optimal Q-value :

Q*(s,a)=maxQ(s,a)=Q*(s,a)subscriptsubscriptsubscriptsubscriptsubscript\beginsplitQ_\pi_*(s,a)&=\max_\piQ_\pi(s,a)\\ &=Q_*(s,a)\endsplitstart_ROW start_CELL italic_Q start_POSTSUBSCRIPT italic_ start_POSTSUBSCRIPT * end_POSTSUBSCRIPT end_POSTSUBSCRIPT (italic_s , italic_a) end_CELL start_CELL = roman_max start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT italic_Q start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT (italic_s , italic_a) end_CELL end_ROW start_ROW start_CELL end_CELL start_CELL = italic_Q start_POSTSUBSCRIPT * end_POSTSUBSCRIPT (italic_s , italic_a) end_CELL end_ROW (2)

To do so, we can use the Q-Learning (Watkins and Dayan, 1992) algorithm. Q-Learning is a model-free off-policy method that estimates the Q-value using Temporal Difference (TD) Learning (Sutton, 1988). The key idea behind it, is that it updates iteratively the estimated Q-value towards the true Q-value :

Q(St,At)(1-)Q(St,At)+GtQ(St,At)+(Gt-Q(St,At))Q(St,At)+(Rt+1+Q(St+1,At+1)-Q(St,At))subscriptsubscript1subscriptsubscriptsubscriptsubscriptsubscriptsubscriptsubscriptsubscriptsubscriptsubscriptsubscriptsubscript1subscript1subscript1subscriptsubscript\beginsplitQ(S_t,A_t)&\leftarrow(1-\alpha)Q(S_t,A_t)+\alpha G_t\\ &\leftarrow Q(S_t,A_t)+\alpha(G_t-Q(S_t,A_t))\\ &\leftarrow Q(S_t,A_t)+\alpha(R_t+1+\gamma Q(S_t+1,A_t+1)-Q(S_t,A_%t))\endsplitstart_ROW start_CELL italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT) end_CELL start_CELL (1 - italic_) italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT)

+ italic_ italic_G start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT end_CELL end_ROW start_ROW start_CELL end_CELL start_CELL italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT) + italic_ (italic_G start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT - italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT)) end_CELL end_ROW start_ROW start_CELL end_CELL start_CELL italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT) + italic_ (italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT + italic_ italic_Q (italic_S start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT) - italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT)) end_CELL end_ROW (3)
where \alphaitalic_ is the learning rate parameter.

So we first need to pick an action aaitalic_a following the current policy using At=argmaxaAQ(St,a)subscriptsubscriptsubscriptA_t=arg\,max_a\in AQ(S_t,a)italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_a italic_r italic_g italic_m italic_a italic_x start_POSTSUBSCRIPT italic_a italic_A end_POSTSUBSCRIPT italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_a). By taking the action AtsubscriptA_titalic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT, we observe a reward Rt+1subscript1R_t+1italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT and get to the next state St+1subscript1S_t+1italic_S start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT. We can then update our Q-value using :

Q(St,At)Q(St,At)+(Rt+1+maxaAQ(St+1,a)-Q(St,At))subscriptsubscriptsubscriptsubscriptsubscript1subscriptsubscript1subscriptsubscriptQ(S_t,A_t)\leftarrow Q(S_t,A_t)+\alpha(R_t+1+\gamma\,max_a\in AQ(S% _t+1,a)-Q(S_t,A_t))italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT) italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT) + italic_ (italic_R start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT + italic_ italic_m italic_a italic_x start_POSTSUBSCRIPT italic_a italic_A end_POSTSUBSCRIPT italic_Q (italic_S start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_a) - italic_Q (italic_S start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_A start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT))

italic-\epsilonitalic_-greedy policy

In Reinforcement Learning, the exploration vs exploitation dilemma is crucial. Indeed, during the training, our agent needs not to get stuck in his current policy and continue exploring new actions but it also needs to use more his policy as its confidence grows. The italic-

\epsilonitalic_-greedy method offers a solution to this problem by picking a random action instead of using the current policy if a uniformly picked number is lower than an italic-\epsilonitalic_ number. At the beginning of the training, italic-\epsilonitalic_ is set to 1 so our agent does only exploration. As the training advances (and our agent gets more confident), we reduce italic-\epsilonitalic_ so that our agent uses more his policy. Q-Learning is an off-policy method because it uses maxQaA(St+1,a)subscriptsubscript1\,maxQ_a\in A(S_t+1,a)italic_m italic_a italic_x italic_Q start_POSTSUBSCRIPT italic_a italic_A end_POSTSUBSCRIPT (italic_S start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_a) in his Q-value update no matter if the current italic-\epsilonitalic_-greedy would have picked a random action or not. We use an italic-\epsilonitalic_-greedy policy during the training of our models with italic-\epsilonitalic_ starting from 1 and decreasing linearly to 0.1.

3.2 Deep Q-Learning

As previously mentioned, calculating the Q-value of every state-action pair gets computationally infeasible as the complexity of the environment grows (and so the state and action spaces). To tackle this problem, we can use a machine learning model to act as a function approximator of the Q-value. In the context of Deep Q-Learning, we use a neural network whose weights and biased are denoted \thetaitalic_ in the new Q-value Q(s,a;)Q(s,a;\theta)italic_Q (italic_s , italic_a ; italic_).

Though several attempts had already been made, Mnih et al. (2013) introduced the Deep Q-Learning and reached human-level performances on several Atari 2600 games. Their architecture used a two layers convolutional neural network that takes as input the last four frames (in greyscale) stacked. They then used one feedforward hidden layer before the last layer outputting the Q-values. We use a similar architecture which can be seen in figure 2. One can notice that these recent work are based on the first association of deep architectures and Q-learning (Lange and Riedmiller, 2010) which followed the work of Riedmiller (2005) who previously established a multi-layer perceptron dedicated to Q-learning, .

Experience replay

One of the key concepts brought by Mnih et al. (2013) work is the use of experience replay. Each step is stored as an experience et=(st,at,rt,st+1)subscriptsubscriptsubscriptsubscriptsubscript1e_t=(s_t,a_t,r_t,s_t+1)italic_e start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = (italic_s start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_a start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_r start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_s start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT) in a replay memory. Mini-batches of experiences are then randomly sampled and used to train the DQN. This allows the network to see multiple times each experience and removes the correlation between these samples. It was shown that this mechanism improves the stability of the

training.

Double Deep Q-Network

The DQN uses the loss function L()=(s,a,r,s)U(D)[(y-Q(s,a)2]fragmentsLfragments()subscriptsimilar-tosuperscriptfragments[fragments(ysubscriptsuperscriptfragments(s,a)2]L(\theta)=\mathbbE_(s,a,r,s^\prime)\sim U(D)[(y-Q_\theta(s,a)^2]italic_L (italic_) = blackboard_E start_POSTSUBSCRIPT (italic_s , italic_a , italic_r , italic_s start_POSTSUPERSCRIPT end_POSTSUPERSCRIPT) italic_U (italic_D) end_POSTSUBSCRIPT [(italic_y - italic_Q start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT (italic_s , italic_a) start_POSTSUPERSCRIPT 2 end_POSTSUPERSCRIPT] to update its weights \thetaitalic_ using s,a,r,ssuperscripts,a,r,s^\primeitalic_s , italic_a , italic_r , italic_s start_POSTSUPERSCRIPT end_POSTSUPERSCRIPT sampled from the uniform distribution U(D)U(D)italic_U (italic_D) over the experience replay memory. In order to calculate our target yyitalic_y, standard Q-Learning with function approximator would use y=r+maxaQ(st+1,a)subscriptsubscript1y=r+\gamma\max_aQ(s_t+1,a)italic_y = italic_r + italic_ roman_max start_POSTSUBSCRIPT italic_a end_POSTSUBSCRIPT italic_Q (italic_s start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_a). This method suffers from instability and to solve this, the extended version of the DQN (Mnih et al., 2015) introduced a target network with weights -superscript\theta^-italic_ start_POSTSUPERSCRIPT - end_POSTSUPERSCRIPT. This network is the same as the the original one except that its parameters are kept fixed and updated every \tauitalic_ steps to t-=tsubscriptsuperscriptsubscript\theta^-_t=\theta_titalic_ start_POSTSUPERSCRIPT - end_POSTSUPERSCRIPT start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = italic_ start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT. Hasselt et al. (2016) improved this idea by adding the concept of Double Q-Learning (Hasselt, 2010). This technique is known to reduce the fact that the Q-Learning algorithm tends to overestimate Q-values. The calculus of yyitalic_y now uses the original network to choose the next action but uses the target network to estimate the Q-value :

y=r+Q-(st+1,maxaQ(st+1,a))subscriptsuperscriptsubscript1subscriptsubscriptsubscript1y=r+\gamma Q_\theta^-(s_t+1,\,max_aQ_\theta(s_t+1,a))italic_y = italic_r + italic_ italic_Q start_POSTSUBSCRIPT italic_ start_POSTSUPERSCRIPT - end_POSTSUPERSCRIPT end_POSTSUBSCRIPT (italic_s start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_m italic_a italic_x start_POSTSUBSCRIPT italic_a end_POSTSUBSCRIPT italic_Q start_POSTSUBSCRIPT italic_ end_POSTSUBSCRIPT (italic_s start_POSTSUBSCRIPT italic_t + 1 end_POSTSUBSCRIPT , italic_a))

3.3 Deep Recurrent Q-Learning

The Deep Q-Network has shown great results but the main idea behind its architecture makes the assumption that the observation received contains all the information about the

current state of the environment. In the context of 3D video game such as Minecraft, the agent only sees a fraction of the whole environment. Mnih et al. (2015) bypassed this issue by using the last four frames as input of their DQN and thus allowing the model to access more information than just the current observation. This solution works well for short dependencies needs-four in the case of Mnih et al. (2015)'s DQN-but can't work in environments where the agent needs to remember older information.

To deal with such environments, Hausknecht and Stone (2015) modified the DQN architecture by adding a recurrent layer between the convolutional and the feedforward layers. Their model, called Deep Recurrent Q-Network (DRQN), now estimates Q(ot,at,ht-1)subscriptsubscriptsubscript1Q(o_t,a_t,h_t-1)italic_Q (italic_o start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_a start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_h start_POSTSUBSCRIPT italic_t - 1 end_POSTSUBSCRIPT) instead of Q(st,at)subscriptsubscriptQ(s_t,a_t)italic_Q (italic_s start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_a start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT) where otsubscripto_titalic_o start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT is the current observation (note that it's no longer stsubscripts_titalic_s start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT which was considered as the whole current state) and ht-1subscript1h_t-1italic_h start_POSTSUBSCRIPT italic_t - 1 end_POSTSUBSCRIPT is the hidden state of the agent at the previous step. They used LSTM (Hochreiter and Schmidhuber, 1997) cells in the recurrent layer with ht=subscriptabsenth_t=italic_h start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT = LSTM(ot,ht-1)subscriptsubscript1(o_t,h_t-1)(italic_o start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_h start_POSTSUBSCRIPT italic_t - 1 end_POSTSUBSCRIPT) and thus estimate Q(ht,at)subscriptsubscriptQ(h_t,a_t)italic_Q (italic_h start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT , italic_a start_POSTSUBSCRIPT italic_t end_POSTSUBSCRIPT). Our DRQN architecture is built on this method (see figure 3).

4.1 Models

We chose to compare the performances of three Deep Q-Networks. They are all constituted of the same basis : A 6x6x32 convolutional layer, another 6x6x36 convolutional layer and a last 4x4x64 convolutional layer. Each convolutional layer is followed by a non-linearity function. The three networks also share the same architecture for the last two layers : A feedforward hidden layer with 512 neurons, and the output layer with four neurons with linear activation corresponding to the estimated Q-values. We used ReLu has both non-linearity function for the convolutional layers and activation function for the feedforward layer.

We chose to downscale the frames observed from the environment to greyscale images in order to simplify and accelerate the training. We call our first tested network the Simple DQN (shown in figure 4). It takes the frame observed in input, pass it through the convolutional layers, flattens the feature maps and then gives it to the feedforward layers. This model has thus no structure intended to deal with the partial observability of our environment. Our

second model, the DQN, is closer to the model introduced by Mnih et al. (2013). It has the same structure as the previous one but takes the last four frames stacked as input in order to handle the short term dependencies of our partially observable environment (see figure 2). The last model, which we call the DRQN, has the same structure as the Simple DQN but has a recurrent layer before the feedforward hidden layer (see figure 3). This recurrent layer is built with an LSTM cell with 256 units. The LSTM hidden state flows through the game episodes and is reset to zero at the beginning of every episode. We used four frames sequences for each mini-batch sample in order to train our DRQN.

Note that all the models use the Double Deep Q-Learning method. We use an experience replay buffer from which we sample our mini-batches (of size 32 for the DQNs and 32/4 for the DRQN). We also use an italic-\epsilonitalic_-greedy policy which starts from 1 and decreases linearly to 0.1 during the training. Moreover, we use a pre-training phase where our agent only plays randomly in order to fill the experience replay buffer. The details of our implementations can be found in appendix A. For the training, we used a computer with a processor Intel Core I7 6700, 16 Gb of RAM and a graphic card Nvidia GeForce GTX 1060 with 6 Gb. We used Tensorflow444https://www.tensorflow.org/ on the GPU of our graphic card.

4.2 Environments

We decided to compare our models performances on two simple missions. The first one is an easy version of the found-the-goal problem, where our agent is in a 7x7x7 room and has to find a block of gold (you can see a screenshot in figure 1). Both the block and the agent spawn randomly on the yyitalic_y axis and have a fixed position on the xxitalic_x axis. In order to finish the mission, the agent must touch it. We restricted the maximum number of steps allowed to the agent to reach the goal to 40. A reward of 1 is given for the goal found and -1 if the maximum number of steps is reached. The agent also receives a -0.01 reward for each step taken.

The other environment is the classical cliff walking problem where our agent starts from a point A and has to reach a point B without falling from the cliff. In our environment, our agent spawns on 8x3 pathway surrounded by lava. The objective is a block of diamond placed at the end of the pathway. The agent receives a reward of -0.01 for each step, 1 if it has touched the goal and -1 if it has drowned in lava or hasn’t reached the goal after 70 steps. The length of 8 was chosen empirically. Indeed, it appeared during our tests that with a length longer than 8, the random actions taken during the pre-training or with italic-\epsilonitalic_-greedy policy couldn’t get close enough to the objective. This caused our agents to never learn how to reach the goal and get stuck at 0% of win. We believe our exploration strategy is not the best for this environment and are thinking about trying other methods (e.g. Thompson Sampling). Refer to section 6 for more information. Also note that this is a very simple cliff walking problem and that there’s no randomness in the generation of the cliff. As aforementioned, this work aims to check if the DRQN is always the best architecture on POMDPs when they are very simple. We thus made our cliff walking very

simple in order to see if the DRQN could learn faster than the Simple DQN or the DQN.

We restricted our agent's possible actions to :

- Move one step forward
- Move one step backward
- Turn head 90°°\lx@math@degree° to right
- Turn head 90°°\lx@math@degree° to left

4.3 Results

To measure the performance of each model on the two environments, we monitored the training and then evaluated the trained models. The training performances rely mostly on three indicators measured every 50 episodes : the percent of win on these last 50 episodes, the mean of the number of steps that the agent took to make these wins and the mean of the accumulated rewards per episode. We also plotted the model's loss and the approximated Q-value (the network output) versus the target Q-value (the target from which the loss is calculated). These last two metrics can be found in appendix B.

Concerning the evaluation, we evaluated our trained models at three checkpoints of their training (in terms of episodes) because we thought it would give an interesting information about how fast the model learns. We made each evaluation three times on 100 episodes (with no seed fixed for the placement of the goal of the Basic mission) and kept the mean of the metrics to have more accurate results. We measured the percent of win and the mean of the number of steps by win. Moreover, every 50 episodes, we printed for each step the frame received by the agent, the Q-values estimated and the action chosen. It allowed us to better understand how our agents were behaving. These visualizations can be found in the appendix C.

4.3.1 Basic Mission

We chose to evaluate our models on the Basic mission to have a baseline of the performances on a very easy task where our agent can't die. We saved our models after 5000, 10 000 and 15 000 episodes (which means around 288 000 steps for the Simple DQN, 276 000 for the DQN and 303 000 for the DRQN at the end of the training). In terms of training time, the Simple DQN was, as expected, the fastest (23 hours to reach 15 000 episodes). The DQN was the longest because of our implementation of its input which needs to be padded with zeros when the episode has not seen four frames yet and also because of

the complexity of its experience replay buffer. It took 1 day and 8 hours for the DQN to be trained, which is roughly 39% longer than the Simple DQN. The DRQN was a bit longer than the Simple DQN (1 day and 1 hour) because of its experience replay buffer too and because of the recurrent layer that adds complexity to the training.

During the training, the DQN quickly achieved a mean of 100% of victory (after 6500 episodes) and then kept this result. Though almost as fast, the Simple DQN which reached the mean 100% of victory after 8500 episodes, kept oscillating around 100% and started to get worse at 12 500 episodes. On the other hand, the DRQN never reached that mean of 100%. It had a similar behaviour as the two other models but floored around 95% after 8000 episodes and its performance started to get sensibly deteriorated at 12 000 episodes. The three models had a similar behaviour concerning the number of steps by win. They main difference is that the DQN kept getting better after 10 000 episodes whereas the Simple DQN and the DRQN floored or became worse. All these results can be found in figure 5.

We then evaluated each model after 5000, 10 000 and 15 000 episodes. The results obtained are gathered in table 1. The results showed that the DQN was already able to perform in 100% of the episodes after only 5000 of training episodes. The DQN obtained an average of 7.6 steps to win after 15 000 episodes of training, which is clearly better than the two other models. The simple DQN reached the 100% performance after 10 000 episodes but, as seen in the training metrics, floored at 9 steps per win. Finally, the evaluation of the DRQN confirms what we had seen during its training. Indeed the 5000 episodes of training version of the DRQN performs better than the Simple DQN, but it also floored after 10 000 episodes and clearly worsened after 15 000 episodes (only 90% of win and a mean of 12.5 steps).

4.3.2 Cliff Walking Mission

After using the Basic environment, we wanted a little harder environment where our agent could die. We thus chose the Cliff Walking, with no randomness in the generation, where the task would still be easy to resolve even with the Simple DQN but maybe harder than the Basic. We trained our three networks on 25 000 episodes and saved them after 10 000, 20 000 and 25 000 episodes. After the end of training, the DRQN had seen 280 000 frames, the Simple DQN 290 000, and the DQN 236 000. The whole training of the Simple DQN (25 000 episodes) lasted 1 day and 4 hours. It was roughly the same for both the DQN (1 day and 3 hours) and the DRQN (1 day and 3 hours).

As shown fin figure 6, the Simple DQN was the fastest to learn. It reached a mean 90% of win after only 10 000 episodes and then stayed around 95%. The DRQN needed more episodes to reach his best training performance (around 15 000 episodes to reach 90%). It floored around 90% and then started to worsen after 25 000 episodes. We were expecting the DQN to have the same behaviour as the Simple DQN but were surprised to see that it was the one struggling the most. It was converging like the two others when it stopped after 10 000 episodes and then got worse.

In the same way as the Basic environment, we evaluated our models trained on the Cliff Walking mission at three checkpoints : after 10 000, 20 000 and 25 000 episodes. As expected with the shape of the training graphs in figure 6, the Simple DQN was able to complete all the evaluations after only 10 000 episodes of training. The DQN results were non consistent and the means of these results were getting worse with more trained versions. The best DRQN version (after 25 000 episodes) succeeded 99.3% of win but was still under the Simple DQN results.

5 Conclusion

In this paper, we wanted to better understand the need of architecture modifications of the DQN in order to better deal with the partial observability of simple POMDPs. We thus tested models with no modification, stacked frames as input to handle short term past dependencies, and recurrent layers to handle long term dependencies. We used the 3D video game Minecraft, which can be framed as a POMDP, to design two simple and classical missions that could be solved by a vanilla DQN.

Our results on the Basic mission showed that the fact of stacking frames made the DQN much more efficient and faster than the Simple DQN on this mission. These results also showed that the DRQN could be tricky to train, longer, and not necessarily better. The DRQN showed better results on the Cliff Walking but still wasn’t the best model. We were surprised that the Simple DQN was the fastest and most efficient model on that mission. The struggle of the DQN doesn’t seem normal to us and requires further investigations to better understand this behaviour.

These results on two very simple missions in a partially observable environment show that adding a recurrent layer to a DQN and thus making it harder and longer to train is not always the best option. The DRQN even showed worse performances than at least one of the other two models on both the Basic and the Cliff Walking missions. We believe that, when it comes to partially observable environments, one should wonder if the agent really needs to remember old informations (it’s for instance the case in hard exploratory problems such as the Obstacle Tower (Juliani et al., 2019)). If it is case, the vanilla DQN is clearly not suited to deal with this and needs to be modified. The DRQN could be one solution but, as aforementioned, other exist. If it is not the case, as it were in our two missions, the DQN can be enough depending on the complexity of the environment.

6 Future Work

This work acts as a baseline for us and opens perspectives. First, we believe our results can be improved. Secondly, we would like to test and improve the idea of DRQN on much harder partially observable environments. So, our future work will be focused on the four methods below.

Dueling

One famous way to improve the DQN, is the Wang et al. (2016) Dueling DQN technique. This architecture is very close to our work since the only change is that we compute the Value function V(s)V(s)italic_V (italic_s) and the Advantage function A(a)A(a)italic_A (italic_a), both components of the Q-value, using two separate neural networks. By decoupling these estimations we could allow our agent to better understand them separately rather than understanding a total fusion of them. Indeed, calculating the Value and the Advantage together may be a problem if the current action doesn't influence a lot the state value or if we have many similar-valued actions. We could expect better empirical results implementing this technique.

Thompson Sampling

Azizzadenesheli et al. (2018) demonstrated the naive aspect of the italic-\epsilonitalic_-greedy exploration and proposed a Bayesian alternative using a Bayesian Deep-Q Network (BDQN). The idea is to use the Thompson Sampling technique through Gaussian sampling. Using Bayesian properties also allow uncertainty measurements for the Q-Values that may be interesting depending on the context. MINECRAFT et al. (2018) used Bayesian linear regression and Thompson Sampling which resulted in as efficient exploration giving higher rewards faster. As our agent is evolving in a POMDP, the exploration is very important, we could implement the Thompson Sampling hoping for a more efficient exploration.

Policy Gradient

The recent methods obtaining mostly the best results in Reinforcement Learning are the policy gradient algorithms. Unlike the off-policy methods, policy gradient techniques are policy-based which means that they try to find directly the best policy that maximizes the sum of rewards. The main current methods are the Trust Region Policy Optimization (TRPO) (Schulman et al., 2015) and the Proximal Policy Optimization (PPO) (Schulman et al., 2017). The latter is the state-of-the-art algorithm as it's simpler and more efficient than its parent (TRPO). These algorithms are mainly used for 3D locomotion (Todorov et al., 2012) because they suit continuous actions and high dimensions. Implementing one of this policy-based algorithm coupled with the two ideas aforementioned may improve significantly our results.

Recurrent Neural Networks

As aforementioned in section 2, different approaches exist when it comes to add a recurrent part to the classical DQN. Indeed, Oh et al. (2016) showed that an approach with an external memory combined with a feedback connection could outperform both the DQN and the DRQN in a Minecraft environment. Chen et al. (2016) and Sorokin et al. (2015) also tried to add an Attention mechanism (Mnih et al., 2014) to help the recurrent layer to focus on interesting information of the past. They obtained promising results and we think there are still investigations to be done in this direction. Although the LSTM has been promising in his

ability to handle long term dependencies, we believe different approaches could improve the performances of the DRQN we used in this paper.