Deep reinforcement learning for video games

Mahdi KALLEL: mahdi.kallel@ip-paris.fr | Yassine NAJI : yassine.naji@ip-paris.fr

Abstract

The aim of this work is to provide a baseline and a benchmark for standard reinforcement learning algorithms for the video game "Snake".

In the first section we present the game and detail it's implementation.

In the following sections we present two standard RL algorithms : Deep Q Learning and Advantage Actor Critic.

1. Introduction:

Reinforcement learning methods have proven that they can outperform the humans in many games (Go, chess , Atari games, etc). The theory behind reinforcement learning dates back several decades. However, it was difficult to apply it to complex tasks that require high dimensional representations because of the lack of computational power. The emergence of deep learning, and the advancements in hardware accelerators (GPU, TPU ..) has allowed to well approximate Q functions with large number of states or even with continuous states representation, and therefore, solve complex task like video games in particular.

2. Snake Game

The snake game is a good example to evaluate how the agent catches long term dependencies between actions and future rewards. As we want to get the full control of the environment characteristics, we implemented a snake game environment based on the Library PyGame ([4] helped us for using it). So we were able to manage the state representation, the reward distribution, the action space, and other parameters:

Snake environment parameters:

- **H, W** (default 80x80): size of the environment
- **Reward** (default 1): the value of the reward given for 'eating the apple'. By default the reward of a collision with a wall or with the agent itself is $-reward$.
- **Early Stopping Factor** (default 10): the game finishes if the snake don't find the apple within : $ES\ factor \times len(snake)$
- **Use Images** (default True) : Represents states as RGB images, otherwise the state will be represented by a vector of 11 element (check the section 2.2 for more details)
- **Use Gray Scale** (default False) : Represents states with gray scale images.

2.1. Reward distribution:

the behavior of the agent depends heavily on the reward distribution, We may consider that doing a 'non-wrong' action (i.e an action that will not finishes the game) deserves giving a small reward (for example $\frac{1}{10}$ of the reward of eating apple). This configuration helps the agent to understand quickly the actions to take to prevent collisions. However, the drawback is that the agent learns to loop over itself infinitely, and sometimes it avoid for the target in order to preserve it size and therefore get infinite reward. Now if we penalize the agent (by $\frac{1}{10}$ of the reward of collision in order to incite him to reach the apple in a quick way), one drawback is that during the first training steps, the agent don't reach the apple with high probability, so he will get the total reward : $n_{steps} \times \frac{r_{collision}}{10} + r_{collision} < r_{collision}$ so it's more advantageous for the agent to produce a collision as soon as possible. To prevent theses behaviors we chose to give 0 reward for taking a non wrong action and to add the **Early stopping factor** (in other to finish the episode if the agent don't find the target quickly enough, the agent receives the collision reward in that case.

We chose the values : $r_{apple} = 1$ and $r_{collision} = -1$ for normalization purposes, in fact, we noticed that high rewards values lead to exploding gradients.

2.2. States representations

We choose to represent our states in two ways:

- **Discrete representation**: we represent a state by a boolean vector ($S = \{0,1\}^{11}$), the first 3 elements indicates if there will be a collision if the agent go straight, left or right. The next 4 elements represent the direction of the snake. The last 4 elements represents the food relative position :

$$S_{target} = (CS, CR, CL, DR, DL, DU, DD, FR, FL, FU, FD) \tag{1}$$

Where :

- CS = 1 if their will be a collision by going straight, 0 otherwise
- CR = 1 if their will be a collision by going right, 0 otherwise
- CL = 1 if their will be a collision by going left, 0 otherwise
- DR =1 if the current direction is right, 0 otherwise
- DL =1 if the current direction is left, 0 otherwise
- DU =1 if the current direction is up, 0 otherwise
- DD =1 if the current direction is down, 0 otherwise
- FR =1 if the food is located right, 0 otherwise
- FL =1 if the food is located left, 0 otherwise
- FU =1 if the food is located is up, 0 otherwise
- FD =1 if the food is located is down, 0 otherwise

- **Image representation**: Embedding states into a discreet space is not always feasible, especially when the state space is complex, also we want that our agent learns from scratch as will do a new player. So we represent the states by the current game display. The latter captures all the necessary information for taking the optimal action. The objective is to embed those images into a low dimension space so that the decision making is easier.

Following figure 1 we see that training on image data requires is a harder task than from discreet representation, therefore we chose to tune the networks parameters using discreet representation, we will discuss in the section 4.3.1 how we improved the performance of the agent while using image representation.

Figure 1. Actor critic score during training

3. Architectures

3.1. Deep Q network

3.1.1 Standard DQN

A classical approach to train an agent to play a video game is to use deep Q network with experience replay [5]. the objective is to approximate the optimal state action function Q^* which the formula is given by the Bellman equation:

$$Q^*(s,a) = r(s,a) + \gamma \max_{a'} Q^*(s',a') \tag{2}$$

Where s' designates the state resulting of taking the action a from the state s.

Following a greedy strategy, for a given state the agent select the action a^* which maximizes the Q function a that state :

$$a^* = \max_a Q^*(s,a)$$

- **Q function estimation :** In order to learn Q^*, at each training step we compute the TD error for certain action-states pairs, sampled from the replay buffer. The TD error is defined by:

$$\delta_t = r + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a) \tag{3}$$

Then we update the Q function as follows:

$$Q_{t+1}(s,a) \leftarrow (1-\alpha) Q_t(s,a) + \alpha \delta_t \tag{4}$$

the quantity $r + \gamma \max_{a'} Q_t(s',a')$ can be interpreted as a target in a context of a supervised learning task.

- **Periodically Updated Target :** Our target depends on the the function that we want to learn which makes the learning much more harder. In order to ensure a certain stability during training, we freeze the Q function that we use of for computing the targets for some training steps, typically of our experiments we fixed the number of steps before updating to 200, since it's offers a good compromise between stability and convergence speed.

  We are training the model to have good estimates of current states, this is enforced using the following loss term:

$$\mathbb{L}(\theta) = r + \gamma \max_{a'} Q(s',a',\theta^-) - Q(s,a,\theta)$$

  where θ^- stand for the frozen parameters.

- **Replay buffer :**During training, all the transition steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are stored in a replay memory $D_t = e_1, \ldots, e_N$. D_t has experience tuples over

many episodes. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.

- **Exploration strategy**

  We'd like the RL agent to find the best solution as fast as possible. However, in the meantime, committing to solutions too quickly without exploring the environment enough can lead to local minima or total failure. Therefore we need to fix an exploration strategy for the agent to try new actions even if they're not necessarily the most promising.

  In the case of DQN we are using ϵ-greedy strategy with a decreasing ϵ in order to encourage the exploration in the beginning of training, then we decrease exploration as long as our model gets better so we can trust it more. One drawback of sampling on policy (ϵ-greedy for example) is that the model encounter mostly states action pairs which it thinks that they lead to the best rewards. This can cause sub-optimal convergence. In particular, we notice this phenomenon in Snake game when we used images for states representation and a small buffer (of size 10^4) , since the event of 'eating the apple' is rare compared to others, using a small buffer leads to a very limited number of experiences with positive reward, as a consequence, the snake learns only to avoid collisions but not to target the apple. In order to reduce this sampling bias, we introduce the concept of replay buffer which stores experiences in memory and we sample from it in the training phase. For our experiences, we used a buffer of size 10^6 which is standard value for our task. The following figure summarizes DQN algorithm with experience replay:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2. DQN with Experience replay pseudo code [5]

We implemented the following architecture :

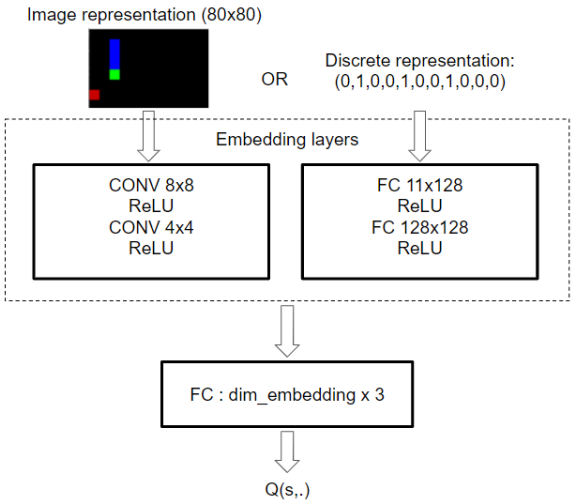

Figure 3. DQN network

3.1.2 Double Q learning

In the vanilla Q learning algorithm we take the maximum estimate of state action pairs for planning.

One of the problems with DQN is that it is prone to overestimation due to the use of the max operator in the Q-learning update (check back on Equation 1). This is a real problem especially early in the training process when all action-value estimates contain some amount of random noise. These over-estimations can result in noisy actions being selected more often than the true optimal action in any given state, which in term leads to sub-optimal policies. And since Q-learning involves bootstrapping (learning estimates from estimates) such overestimation can be problematic.

The solution involves using two separate Q-value estimators, each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates. [7]

3.1.3 Dueling Q learning:

The Dueling DQN architecture [6] trades on the idea that the evaluation of the Q function implicitly calculates two quantities: V(s), the value of being in the current state and A(s,a), the advantage of taking action "a" in state s.

$$Q(s,a) = A(s,a) + V(s)$$

By decoupling the estimation, intuitively our DDQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state (since it's also calculating V(s)). As a consequence, by decoupling we're able to calculate V(s). This is particularly useful for

states where their actions do not affect the environment in a relevant way. The following figures shows the implemented architecture.

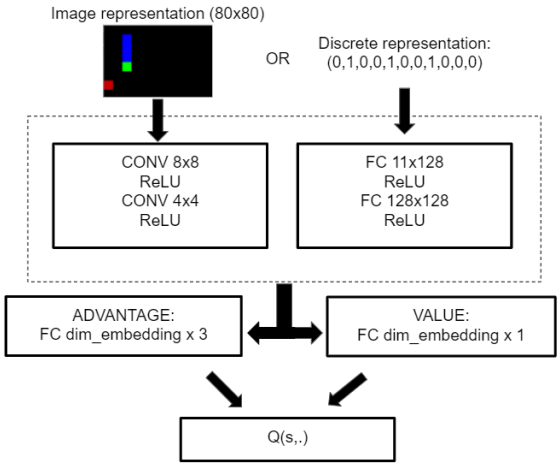

Figure 4. Dueling DQN

3.2. Experiments :

- **Setup:** In order to compare the performances of the different algorithms on the Snake game (using discreet representation), we trained each algorithm for 10^5 steps using the following default parameters (unless stated so): Adam optimizer with a learning rate of 10^{-4}, a factor $\gamma = 0.99$ (in order to capture the long term rewards), a batch size of 256 and a buffer of size 10^6. The following figures shows the averaged scores (over 50 simulations).

- **Convergence:**In figure 5 we notice that DQN algorithms behaves similarly and learn more quickly than Actor critic at the beginning . Dueling DQN seems to perform slightly better than the other DQN algorithms. Double DQN algorithms seems to be less stable than others and under perform over others (especially for Standard double DQN). However, DQN algorithms seem to struggle to exceed a score of 8, unlike the actor critic algorithms which learns more slowly at the beginning, but it surpass all other networks.

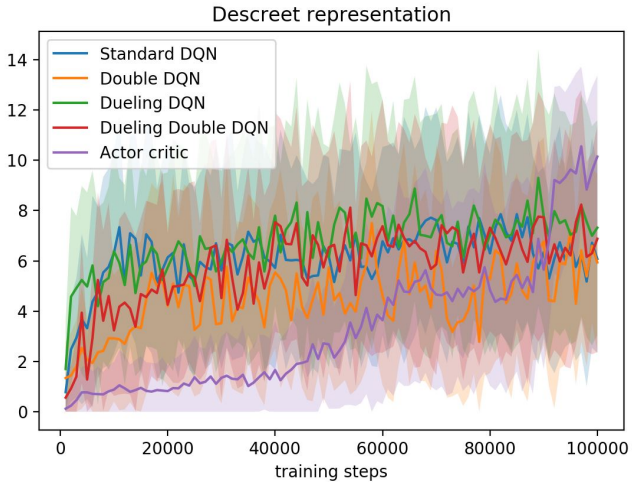

Figure 5. Comparison of DQN algorithms

- **Batch size:** Counter intuitively, we found that the batch size6 had almost no effect on the training. One would expect that with more samples the model can have a broader view of possible states and thus perform updates that generalize well but the model stagnated at around a reward of 8.

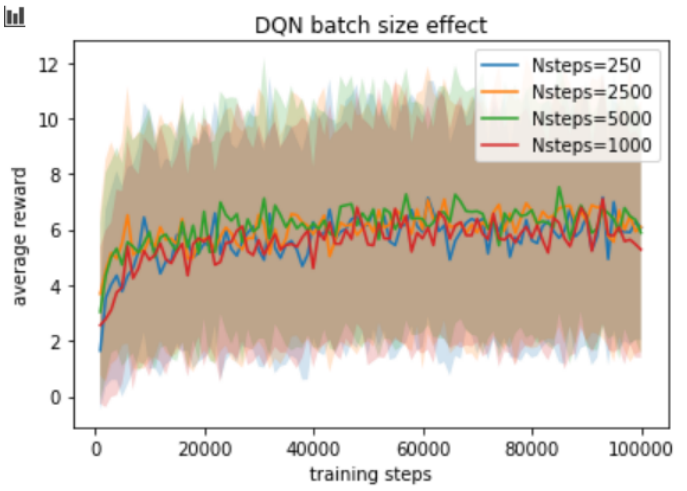

Figure 6. DQN effect of batch size

- **Learning rate:** Also, one interesting result is that learning rate had almost no effect on the speed of the convergence 7, nor the result of the model (except in the extreme case of lr = 0.1).

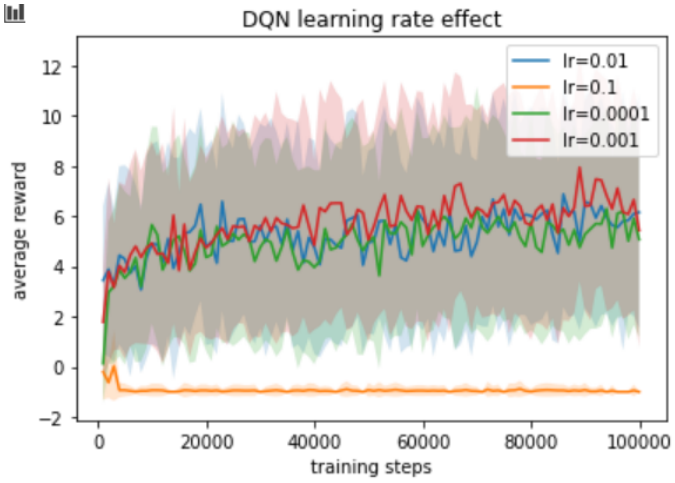

Figure 7. DQN effect of the learning rate

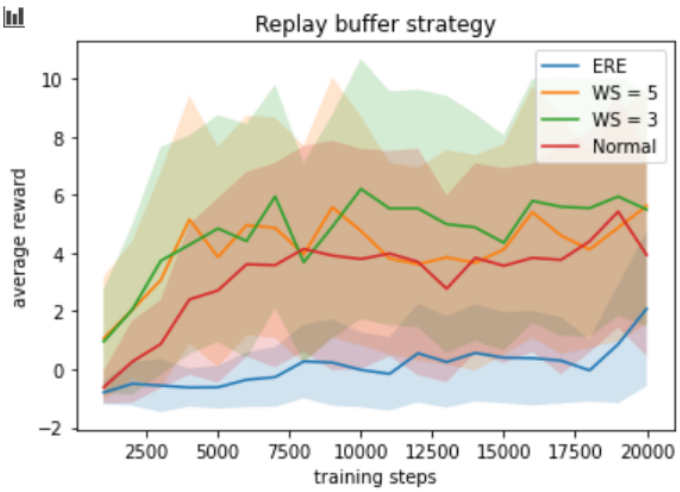

Figure 8. DQN effect of the buffer strategy

- **Replay buffer strategy:** Training is usually done by sampling B_{size} samples at random from the whole buffer. In our context we figured this strategy can lead to poor results as the images with rewards are quite rare.

  For efficient training, we would like to ensure that each batch contains a certain number of samples with rewards. Therefore we try to enforce this property using what we call "Weighted sampling" strategy. This enforces that a batch contains a fixed proportion of rewarding samples.

  Another strategy we found is "Emphasizing Recent Experience", in this setup we do not sample from the entire experience buffer, but rather from last N_{ERE} steps to form a decreasing window over the last experiences.

  The intuition behind it is that very old experiences are irrelevant to our current training process. In it's newer versions the model tends to visit states different that in the beginning. By fixing such a decrease window we focus more on the states that our more relevant to our current policy. In figure 8 We see that "Emphasize recent experience does not improve our model, maybe this strategy becomes more useful when training for a longer period. On the other hand, "Weight sampling" seems to yield better results.

4. Advantage Actor Critic

4.1. Theory

The Actor Critic methods [2] are an improvement of the policy gradient algorithm (REINFORCE) [3]. Recall the Policy Gradient formula:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_t|s_t)) G_t \right)$$

Where $G_t = \sum_{k=t}^{T} r_k$ is the collected rewards through a trajectory.

Policy gradient suffers from large variance due to Monte Carlo sampling in order to calculate G_t and also from exploding gradient if the rewards are not normalized.

Let's modify the policy gradient formula :

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_\tau \left(\sum_{t=1}^{T-1} \nabla_\theta \log(\pi_\theta(a_t|s_t)) G_t \right) \\
&= \sum_{t=1}^{T-1} \mathbb{E}_\tau \left(\mathbb{E} \left(\nabla_\theta \log(\pi_\theta(a_t|s_t)) G_t | t \right) \right) \\
&= \sum_{t=1}^{T-1} \mathbb{E}_\tau \left(\nabla_\theta \log(\pi_\theta(a_t|s_t)) \mathbb{E} \left(G_t | t \right) \right) \\
&= \mathbb{E}_\tau \left(\sum_{t=1}^{T-1} \nabla_\theta \log(\pi_\theta(a_t|s_t)) Q(s_t, a_t) \right) \\
&= \mathbb{E}_\tau \left(\sum_{t=1}^{T-1} \nabla_\theta \log(\pi_\theta(a_t|s_t)) (r_{t+1} + \gamma V(s_{t+1})) \right)
\end{aligned}$$

The actor critic method aims to learn jointly the value function (critic) which is used to update the policy (actor).

To guaranty that the gradient doesn't blow up due to large values of the term $r_{t+1} + \gamma V(s_{t+1})$, we introduce the advantage function $A(s_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$. So the gradient of the objective function is :

$$\mathbb{E}_\tau \left(\sum_{t=1}^{T-1} \nabla_\theta \log(\pi_\theta(a_t|s_t)) A(s_t) \right)$$

Algorithm 1 Advantage actor-critic - pseudocode

// Assume parameter vectors θ and θ_v
Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  $t_{start} = t$
  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}
    $t \leftarrow t+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v) & \text{for non-terminal } s_t \text{ //Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a_i|s_i;\theta)(R - V(s_i;\theta_v)) + \beta_e \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
    Accumulate gradients wrt θ_v: $d\theta_v \leftarrow d\theta_v + \beta_v(R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v)$
  end for
  Perform update of θ using $d\theta$ and of θ_v using $d\theta_v$.
  $E \leftarrow E + 1$
until $E > E_{max}$

Figure 9. ACN algorithm

4.2. Implementation

For our task we implemented a variant of Advantage Actor critic. We fixed $T = \min(T_{finish}, 20)$, and we calculate $A(s_t)$ as the following:

$$A(s_t) = V(s_T) + \sum_{k=t+1}^{T-1} \gamma^{k-1} r_k - V(s_t) \qquad (5)$$

We notice that their is a bias variance trade off in the choice of T value, a large T reduces the bias and increases the variance (similarly to REINFORCE) and vice versa. For snake game, since we have long terms dependencies it's seems relevant to set a relatively high value for T.

The architecture we used is composed of a common branch, which is used for features extraction. In case of image representation of states, we use 2 convolution layers with ReLU activation, the first receive a 80×80 RGB image which represents the current state, and apply convolutions with a kernel size of (8×8) and strides of 4 and outputs 16 channels. The second apply (4×4) convolutions with strides of 2. However for discreet states representation we use a simply 2 fully connected layers with 128 hidden units and ReLU activation. The actor branch is composed of a fully connected layers that outputs a probability distribution over actions (log probabilities exactly). The critic is also a fully connected layer that outputs the value estimation of the current state.

We summaries the architecture in the following graph :

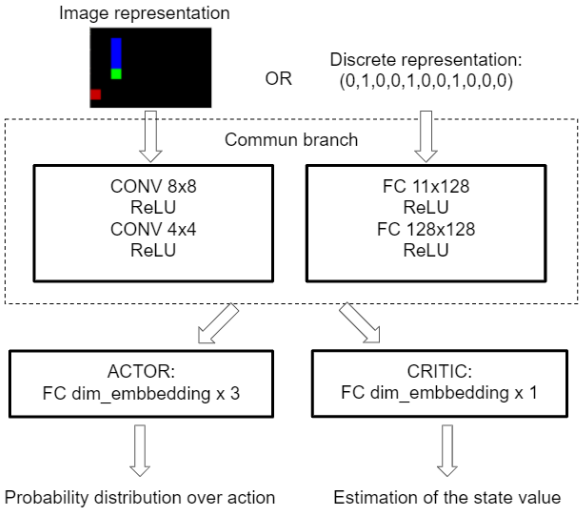

Figure 10. Actor critic architecture for snake game

Computing the loss :

In order to compute the loss, we start an episode, we predict a policy and a value of the state, we take a greedy action with respect to that policy then we store the reward and next state. We do the same procedure for the next state until we reach T steps. Then we compute advantages by using the formula 5.Then we compute the actor and the critic losses:

$$L_{critic} = \sum_{t=1}^{T-1} A(s_t)^2 \qquad (6)$$

$$L_{actor} = -\sum_{t=1}^{T-1} \nabla_\theta \log(\pi_\theta(a_t|s_t)) A(s_t) \qquad (7)$$

So we minimize the global loss :

$$L = L_{actor} + L_{critic}$$

Training the network once using discreet states and the other using image representation for 10^5 steps using Adam optimizer with a learning rate of 10^{-4} gives the following results :

As expected the network learns much more quickly using discreet representation, than with image representation. The maximum score that we obtained after 1 million training steps using image representation is 5, this score is passed only after 30000 iterations using discreet representation, to reach 15 after only 60000 iteration. This show the important of having a relevant representation of states. In fact, using Image representation, the network seeks to maximize directly the policy, but their is no guaranty that it learns a good state representation that will allow it to take the right action(s) decisions on unseen states. Our aim in the next section is to design a new loss that takes into consideration the quality of states embedding after the convolutional layers.

4.3. Experiments:

- **Number of steps:** This parameter represents the number of steps the agent takes in the environment before updating it's estimates. In the case of bootstrapping we choose to have a bootstrap estimate of the last state (if the episode is not finished). On the other hand we can choose not to bootstrap and wait for the end of the episode. This is probably the most important parameter to tune, it depends on the training game. The idea is that we need to update over significant periods in the game (eg: between eating two apples). From figure 11 we can see the impact of this parameter as tuning it changes radically the performance.

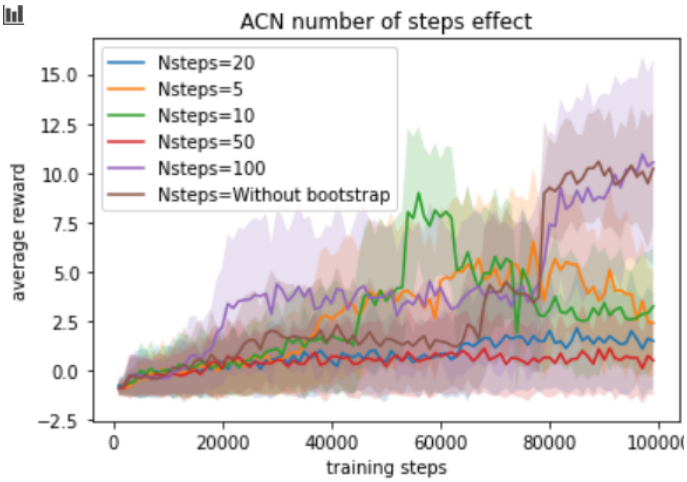

Figure 11. ACN effect of number of steps before update.

- **Gradient clipping:** A heuristic method for variance reduction in Actor critic estimates is to clip the gradient values. From figure 12 we see that this allows for faster convergence.

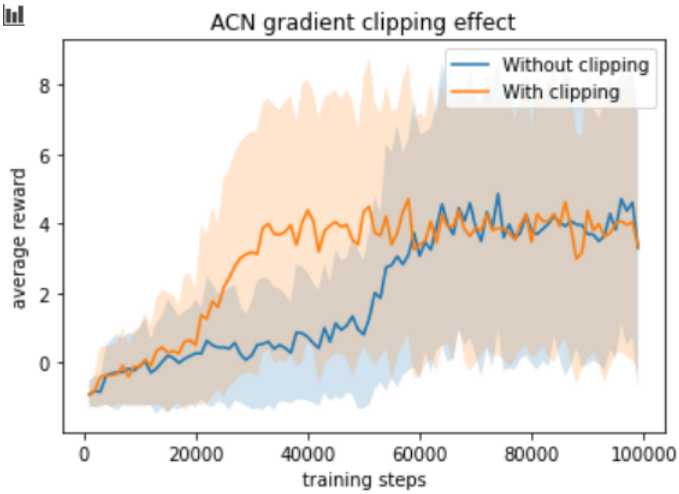

Figure 12. Effect of gradient clipping on ACN

- **Training time:** We compare in the figure 13 networks with respect to the training time for the same number of training steps (10^5), we notice that in case of actor critic, it strongly depends on the number of steps that the agent takes before bootstrapping.

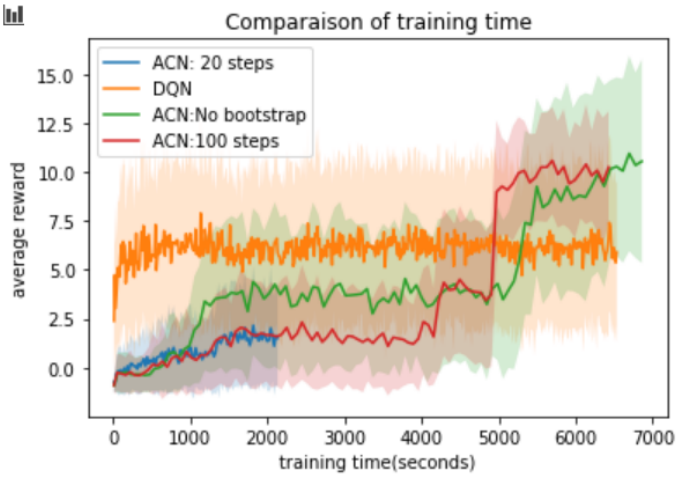

Figure 13. Training time

4.3.1 Designing a new loss function for the Actor Critic model:

Consider the case when we input raw display images (as states) to our Actor Critic model. In order to penalize bad states internal representations in case we introduce the embedding loss L_{emb} between the current state representation embedding and the target embedding that we assume for this case is the discreet representation $S_{target} \in \mathbb{R}^{11}$ defined in 1.

We chose this representation because it's encompasses all the necessary information about the state and also allows use to preform multi-label classification since the vector is binary. Thus we can compute L_{emb} as a the binary cross entropy for each vector component (to be more rigorous, we can apply Softmax to food location and also to direction components since their is only one possible direction). For seek of simplicity we used binary cross entropy loss component wise which is implemented by the `PyTorch` function `nn.BCEWithLogitsLoss()`. The new objective function is given by:

$$L = L_{actor} + L_{critic} + L_{embedding} \tag{8}$$

We summarized the new architecture in the following diagram:

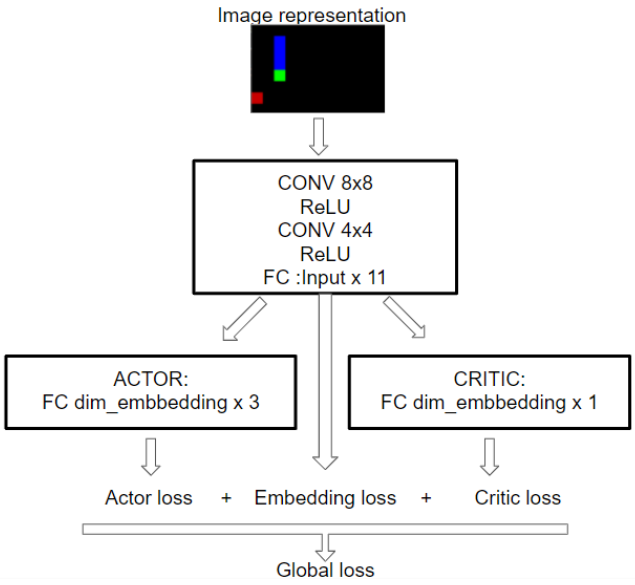

Figure 14. The architecture using the embedding loss

We trained the network for 1 millions training steps using a learning rate of 10^{-4}, with 100 agent steps before bootstrapping (we chose to maintain bootstrapping because otherwise the training will take much longer), and using gradient clipping. We got the following results:

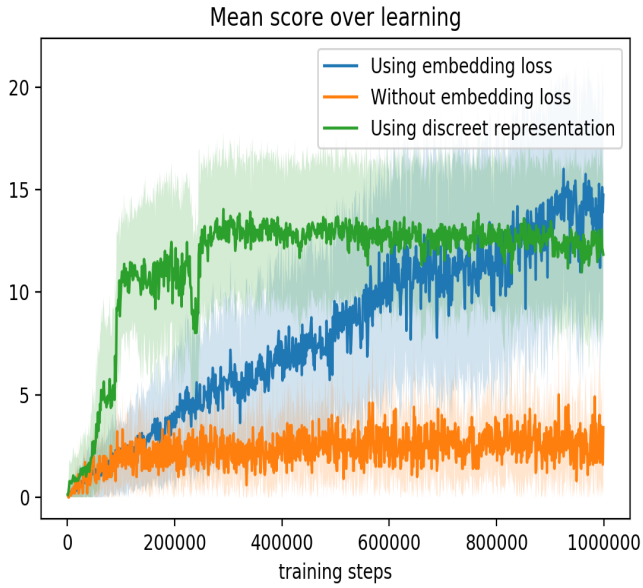

Figure 15. Mean scores

The training with the new loss function takes considerably much more time than other training methods. For comparison, training for 1 million steps using discreet representation took from us 5h one CPU and using image representation without embedding loss took about 10 hours. However using the new loss the training finishes after 2 days !

Nevertheless it's over preform all the previous training methods, even using discreet representation that we consider the embedding loss with respect to. This can be explained by the fact that the embedding layers don't learn exactly the same discreet representation as indicated by the target, because the back propagation through the embedding layers is done with respect to the global loss. The embedding loss seems to play the role of a regularizer which prevent the model don't overfit on the states which the agent have seen.

5. Conclusion:

In this work, we studied two reinforcement learning algorithms, Deep Q learning and Actor Critic. We presented an overview of the theory and intuition behind these algorithms. We studied the impact of various parameters on the convergence of these algorithms. We also evaluated the impact of state representation by introducing the embedding loss. We reached the best performance using the Actor Critic architecture trained with the latter loss.

Through our experiments we conclude that Actor critic methods are better suited for the snake game than the Deep Q Network.

References

[1] **Github of our project:** https://github.com/YassineNJ/RL-Snake

[2] **Distributional Advantage Actor-Critic**, Shangda Li, Selina Bing, Steven Yang

[3] **Sample Efficient Reinforcement Learning with REINFORCE**, Junzi Zhang1, Jongho Kim2, Brendan O'Donoghue3, Stephen Boyd 5

5

[4] https://github.com/python-engineer/python-fun 1

[5] **Playing Atari with Deep Reinforcement Learning**, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller 2, 3

[6] **Dueling Network Architectures for Deep Reinforcement Learning**, Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas

[7] **Deep Reinforcement Learning with Double Q-learning** , Hado van Hasselt, Arthur Guez, David Silver 3