INFO8010: Project report

Baptiste Debes,[1] **Louis Nelissen**,[2] and **Pierre-François Weyders**[3]

[1] *bdebes@student.uliege.be (s162012)*
[2] *louis.nelissen@student.uliege.be (s160708)*
[3] *pfweyders@student.uliege.be (s160729)*

I. INTRODUCTION

In this paper we detail how we applied deep neural networks and reinforcement learning to a simplified version of the game agar.io. *Agar.io* is a Massively Multiplayer browser game with a simple gameplay: the player is a cell that must eat food (called "pellets") to grow while avoiding enemies. The player moves around on a bounded 2D grid in any direction using its mouse. Eating food makes the cell grow and its score increases. It also makes its speed decrease.

We first explain the mechanics and workings of our own implementation of the game (referred to henceforth as *JAVAgario*), and discuss related works. We then go over our chosen architectures and their results. We finally discuss our results.

A. Game engine

We will now make a brief description of how the game works. A more detailed description of the game engine, state space and dynamics of the game can be found in the first part of the Annexes section.

The game consists in an agent moving in a square board containing pellets. The game is rendered on a grey-scale image. The agent's goal is to eat all pellets in the most effective, i.e. fastest, way.

1. Action space

The action space is continuous, between $[-1; 1]$, but can be descretized: $a_x, a_y \in \{-1, -0.5, 0, 0.5, 1\}$. The number of pellets is constant over all simulations and each simulation terminates after 250 steps (movements) or early if all pellets have been eaten.

2. Environment

The agent's environment is partially observable for two reasons.

1. It does not have access to the raw dynamics of the game (the variables describing its environment) but instead takes in one image at a time. This means the agent cannot derives certain indicators such as its speed vector.
2. Its input is only partial with respect to the whole environment. We said that the agent's view is limited or *scoped*.

3. Reward function

Reward engineering is an important and complex step for many Reinforcement Learning problems and so it was for this game. We choose to give a reward equivalent to the number of eaten pellets and -1 before the agent eat any pellets. This stimulates the agent to eat as much as possible and thus as quickly as possible.

II. RELATED WORK

Applying Deep RL to games seems to have become a very fertile research topic. In [7], the authors have shown the capabilities of deep learning based reinforcement learning techniques to reach and surpass human performance in a wide range of Atari games by only being fed raw pixels. Results has been improved extensively since then by individual improvements or combinations. One famous example is *Rainbow* from [4] which is a combination of several tweaks whose purpose is to stabilize and ease the learning process. More recently, Agent57 has been introduced in [2] and is claimed to have reached human level performance on the whole 57 games of the Atari benchmark.

Agar.io has already been considered as a learning environment in [1]. They were able to reach the performances of a greedy pre-programmed agent by simplifying the inputs using a semantic grid. They also optimized the training by using a prioritized experience replay. Sampling more often transitions who may lead to a larger learning experience. The original environment has continuous action space which can either be descretized or kept continuous.

In [6], Deep Deterministic Policy Gradient - a parametric version of the same algorithm - was introduced which kicked off the deep learning reinforcement learning approach for continous action spaces. DDPG is an actor-critic method which outputs a deterministic policy approximator.

III. METHODS

A. Theoretical developments

Reinforcement learning problems are classically modeled as agents interacting with their environment and learning from the result of its interaction through a reward function. The purpose of the learning process is to derive a policy which maps states to actions, such that this policy is able to collect the maximum total reward. The total reward will be definde later on. The policy can either be deterministic or stochastic. Among the big family of reinforcement learning techniques we have explored only the so-called model-free branch. Hence, the optimal policy is found by optimizing a set of parametric approximators modeling either a value function (V or Q) and/or the optimal policy directly. In this model-free branch we explored firstly a value-based technique. This method is the parametric extension of the Q-learning algorithm introduced in [11] Hence, we aim at finding an approximator $Q_\theta(s,a)$ of the optimal value-action function $Q^*(s,a)$. The optimal policy is then derived greedily by taking

$$a_{\text{opt }\theta} = arg\max_{a'} Q_\theta(s,a') \tag{1}$$

We recall that this function is solution of the Bellman equation. In an MDP setup, it is

$$Q^*(s,a) = E\left[r|s,a\right] + \gamma \sum_{a'} P(s'|s,a)Q^*(s',a') \tag{2}$$

with γ being called the discount factor. This factor scales how future rewards are valuable with respect to earlier rewards. A model-based approach would seek to approximate the reward expectation as well as the transition distribution. Q-learning finds an approximation of $Q^*(s,a)$ by making a one sample approximation of 2 which is

$$Q^*(s,a) \approx r + \gamma \max_{a'}(Q^*(s',a')) \tag{3}$$

The difference between 2 and 3 is called the *temporal difference* and it can provide a loss whose direction of decrease could improve the approximator $Q_\theta(s,a)$. Hence, the loss is defined as

$$L^{TD}(\theta) = \frac{1}{2}\left[Q_\theta(s,a) - r - \gamma \max_{a'} Q_{\bar{\theta}}(s',a')\right]^2 \tag{4}$$

The Deep Q-learning algorithm simply minimizes this loss by sampling somehow transitions from the environment. The minimization is performed by taking a step in the descent direction. Historically, this approach seems to have really taken over after the introduction of two improvements namely the Experience Replay Buffer and the Double Q-learning. Experience replay buffers alleviate a sampling issue. Stochastic gradient descent requires samples to be sampled uniformly. When DQN updates the model parameters sequentially, samples are correlated since transitions might lead to nearby states. Replay buffer store a finite number of five-tuples (state, action, reward, next-state, terminal) which are sampled uniformly to constitute batches for the loss estimation. The replay buffer also allows to re-use samples, which leads to a better data efficiency. The double network mitigate divergence by keeping a second set of parameters $\bar{\theta}$ and updating it as time goes by.

A second family of model-free methods is called Policy Gradient methods. In this context, one seeks an approximator $\pi_\theta(a|s)$ of the optimal policy $\pi^*(a|s)$ that can either be stochastic or deterministic. As said before, the purpose of reinforcement learning is to find a policy that maximizes the total reward. This total reward is defined as the expected discounted sum of reward also called value function. For a parametric policy and n the step number it is

$$V_\theta(s_0) = \sum_n \gamma^n E_{\theta,\, s_0}\left[r_n|s_n,\, a_n\right] \tag{5}$$

In [9], the authors introduced REINFORCE as well as a one sample approximation for the gradient of the value function **for a stochastic policy**

$$\nabla V_\theta(s_0) \approx \gamma^n G_n \nabla log(\pi_\theta(a_n|s_n)) \tag{6}$$

where G_n is the **onward** sum of discounted reward of a length T trajectory sampled from the environment.

$$G_n = \sum_{t=0}^{T-n} \gamma^t r_{n+t} \tag{7}$$

This is the base for the Vanilla Policy Gradient Method. This methods gives higher probability to action that leaded to good onward cumulative reward. The next step is to define a combination of both Policy Gradient and Q-learning. This combination is called actor critic. The actor is the policy being optimized. The critic is a Q function approximator. In Advantage Actor Critic or A2C[12], one computes the Advantage which is $A(s,a) = Q(s,a) - V_\theta(s)$. It is computed for a given iteration n by

$$\begin{aligned} A(s_n,a_n) = r_n + \gamma \max_{a_{n+1}} Q(s_{n+1},a_{n+1}) \\ - \sum_a \pi_\theta(a|s_n)Q(s_n,a) \end{aligned} \tag{8}$$

This signal when compared to G_n is much more meaningful. It is an estimated discrepancy between what what the agent could win onward after performing action a_n versus what it could have won in average. We recall that the reward is the only numeric signal that will shape the parameters of the models. Thanks to advantage, a more efficient use of this signal is performed. In the case of Advantage Actor Critic, the loss to be minimized with respect to the policy network will be

$$L^{PG}(\theta) = E\left[-log(\pi_\theta(a_n|s_n)A(s_n,a_n)\right] \tag{9}$$

Layer Type	Filter Shape	Input Shape
Conv2d + ReLU	$[8 \times 8]$s4	$[64 \times 64 \times 1]$
Batch Normalization	/	$[15 \times 15 \times 32]$
Conv2d + ReLU	$[4 \times 4]$s2	$[15 \times 15 \times 32]$
Batch Normalization	/	$[6 \times 6 \times 64]$
Conv2d + ReLU	$[3 \times 3]$s1	$[6 \times 6 \times 64]$
Batch Normalization	/	$[4 \times 4 \times 64]$
Flatten	/	$[4 \times 4 \times 64]$

TABLE I: Convolutional network used in all networks for DDPG, A2C and DDQN

which will be estimated on batches.

Until now, no assumption have been made of the action space. Deep Q-learning and Advantage Actor Critic both perform max operators over the action space. If this action space is continuous, a new optimization problem has to be performed at every iteration of the algorithm. This is not suitable. Hence, the use of these methods is restricted to **discrete** action spaces. Consequently, we introduce briefly a continuous space actor-critic method called Deep Deterministic Policy Gradient. It takes advantage of a formulation of the value function gradient which is

$$\nabla V_\theta(s_0) \propto E\left[\nabla_\theta \pi_\theta(s) \nabla_a Q_\tau(s,a)\Big|_{a=\pi_\theta(s)}\right] \tag{10}$$

which can be one-sample estimated. In order to use auto-differentiation we must specify an equivalent loss. It would be

$$L^{\text{DDPG}}(\theta) = E\left[Q_\tau(s,a)\Big|_{s=s_n, a_n=\pi_\theta(s_n)}\right] \tag{11}$$

The value-function loss is similar to DQN.

B. Network architectures

We note that all the following networks we implemented use an Adam optimizer [5]. The hyperparameters of the respective networks can be found in the Annex.

1. *A2C*

The first architecture we implemented follows the A2C model[12]. The pseudo code for this algorithm can be found at ALGORITHM 1 as adapted from [8]. It uses 4 nets: the Critic, the Actor and one target network for each of these networks.

All the network follows the standard schema represented in *(a)* of FIGURE 1: a combination of convolutional layers, as detailed in TABLE I and dense layers. The Critic and its target network use the dense layers as

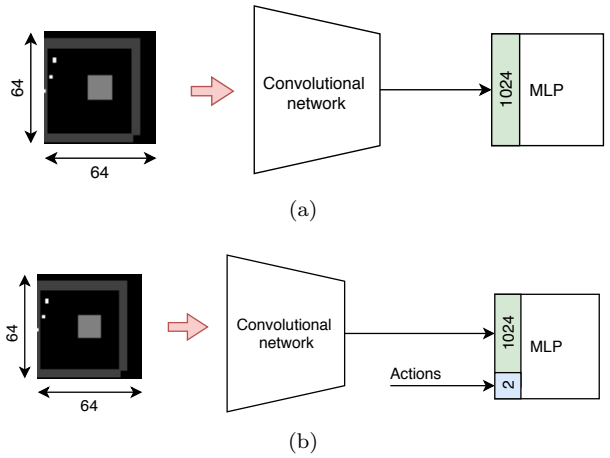

FIG. 1: A general overview of our networks' architecture (a), and the particular case of DDPG's Critic network (b)

Layer Type	Filter Shape	Input Shape
Dense + ReLU	$[1024 \times 512]$	[1024]
Batch Normalization	/	[512]
Dense	$[512 \times 512]$	[512]

TABLE II: Multilayer perceptron used in A2C's Critic network

described in TABLE II, while the Actor and its target network use the ones in TABLE III. These are almost the same aside from the fact that the Actor model has an additional Softmax activation layer at the end of the network.

2. *DDPG*

The second architecture we implemented follows the DDPG model[6]. The pseudo code for this algorithm can be found at ALGORITHM 2. It uses 4 nets: the Critic, the Actor and one target network for each of these networks.

While the Actor and its target network follow the standard schema of *(a)* in FIGURE 1, the Critic instead uses a variant represented in *(b)*. Its peculiarity is that it the dense layers take as input the output of the convolutional

Layer Type	Filter Shape	Input Shape
Dense + ReLU	$[1024 \times 512]$	[1024]
Batch Normalization	/	[512]
Dense	$[512 \times 512]$	[512]
Softmax	$[25 \times 1]$	[25]

TABLE III: Multilayer perceptron used in A2C's Actor network

Algorithm 1: A2C

Initialize π_θ to anything
Loop forever (for each episode)
  Initialize s_0 and set $n \leftarrow 0$
  Loop while s is not terminal (for each time step n)
    Select a_n
    Execute a_n, observe s_{n+1}, r_n
    $\delta \leftarrow r_n + \gamma \max_{a_{n+1}} Q_w(s_{n+1}, a_{n+1}) - Q_w(s_n, a_n)$
    $A(s_n, a_n) \leftarrow r_n + \gamma \max_{a_{n+1}} Q_w(s_{n+1}, a_{n+1}) - \sum_a \pi_\theta(a \mid s_n) Q_w(s_n, a)$
    Update $Q : w \leftarrow w + \alpha_w \gamma^n \delta \nabla_w Q_w(s_n, a_n)$
    Update $\pi : \theta \leftarrow \theta + \alpha_\theta \gamma^n A(s_n, a_n) \nabla \log \pi_\theta(a_n \mid s_n)$
    $n \leftarrow n + 1$

Layer Type	**Filter Shape**	**Input Shape**
Dense + ReLU	[1026 × 512]	[1024+2]
Batch Normalization	/	[512]
Dense	[512 × 512]	[512]

TABLE IV: Multilayer perceptron used in DDPG's Critic network

Layer Type	**Filter Shape**	**Input Shape**
Dense + ReLU	[1024 × 512]	[1024]
Batch Normalization	/	[512]
Dense	[512 × 512]	[512]
TanH	[2 × 2]	[2]

TABLE V: Multilayer perceptron used in DDPG's Actor network

layers, but also the action take by the actor, described in 2D using 2 values.

The Critic and its target network use the dense layers as described in TABLE IV, while the Actor and its target network use the ones in TABLE V. Once again they are almost the same aside from the fact that the Actor model has an additional activation layer (hyperbolic tangent in this case) at the end of the network.

3. DDQN

The third and final architecture we implemented follows the DDQN model[10]. The pseudo code for this algorithm can be found at ALGORITHM 3. It uses 2 nets: a value network called the *Q net* and a target network.

Both these networks follow the standard schema of *(a)* in FIGURE 1. They use the convolutional layers as described in TABLE I and the dense layers as described in TABLE VI.

Layer Type	**Filter Shape**	**Input Shape**
Dense + ReLU	[1024 × 512]	[1024]
Batch Normalization	/	[512]
Dense	[512 × 512]	[512]

TABLE VI: Multilayer perceptron used in DDQN's Q network

C. Training and methodology

The algorithms were trained on random environments. The pellets as well as the starting position were draw uniform at random. Twenty pellets were put on the maps. The map size is constant and relatively small with respect to the agent. This small size is a departure from the original game whose maps are much larger.

In order to assess the performance of the algorithms, two baseline have been implemented. The first one is a *greedy* bot that directs itself in the direction of the closest visible pellet. The second one is a random agent which samples the x and y input uniformly between -1 and 1. The *greedy* bot was provided with the same input as the RL methods.

The RL algorithms were trained on episodes of 2500 steps after which the episodes is terminated. Model and bots validation are terminated after 200 steps. The agents were provided an image of 64 by 64 pixels in greyscale. On this image the pellets, the player as well as the walls can be seen in different shades of grey as can be seen in FIGURE 2

(a) (b)

FIG. 2: 2 examples of inputs

The algorithms were trained during 350 000 iterations

Algorithm 2: DDPG algorithm

Randomly initialize critic network $Q\left(s, a \mid \theta^{Q}\right)$ and actor $\mu\left(s \mid \theta^{\mu}\right)$ with weights θ^{Q} and θ^{μ}.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^{Q}, \theta^{\mu'} \leftarrow \theta^{\mu}$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $t = 1, T$ **do**
    Select action $a_t = \mu\left(s_t \mid \theta^{\mu}\right) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'\left(s_{i+1}, \mu'\left(s_{i+1} \mid \theta^{\mu'}\right) \mid \theta^{Q'}\right)$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i \left(y_i - Q\left(s_i, a_i \mid \theta^{Q}\right)\right)^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_i \nabla_a Q\left(s, a \mid \theta^{Q}\right)\Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu\left(s \mid \theta^{\mu}\right)$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

  end for
end for

Algorithm 3: DDQN algorithm

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}, \tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi\left(a_t, s_t\right)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R\left(s_t, a_t\right)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  end for
  for each update step **do**
    Sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:

$$Q^*\left(s_t, a_t\right) \approx r_t + \gamma Q_\theta\left(s_{t+1}, \text{argmax}_{a'} Q_{\theta'}\left(s_{t+1}, a'\right)\right)$$

    Perform gradient descent step on $\left(Q^*\left(s_t, a_t\right) - Q_\theta\left(s_t, a_t\right)\right)^2$
    Update target network parameters:

$$\theta' \leftarrow \tau * \theta + (1-\tau) * \theta'$$

  end for
end for

of their main loop. Which corresponds to 350 000 frames generated. Frames were stored in a uniform replay buffer of size 65 000. Since the batch size is equal to 64, 22 400 000 frames were drawn uniformly from the buffer.

Each method has been tested on 100 randomly generated environments. Metrics were averaged over these groups of 100 episodes. It is noteworthy to mention that this process has not been repeated. Consequently, these results are only able to assess the performances of **one** iteration of the training process. The variability is only due to the randomness in the environment generation.

Each network, in every method, was doubled by a target network. The weights of these target networks were update *softly* using Polyak Averaging as an exponentially decay average

$$\bar{\theta} \leftarrow \tau \times \theta + (1-\tau) \times \bar{\theta} \quad \text{with} \quad \tau \in [0, 1] \tag{12}$$

For Deep Q-learning and A2C, actions were discretized in 5 by 5 in $D = \{-1, -0.5, 0, 0.5, 1\}$ such that $a_x, a_y \in D \times D$. DDPG was able to output actions in $[-1, 1] \times [-1, 1]$

IV. RESULTS

Results are reported according to two metrics. The first is the number of pellets remaining (not eaten) of the 20 at start. Hence, lower is better. The second is the total non-discounted reward experienced by the agent during the 200 steps of the validation phase. The average number of remaining pellets results can be seen in FIGURE 3. The average total reward is displayed in FIGURE 4. As mentioned before, these results only assess the performance of the algorithms on one run of each training process. Hence, one must stay cautious regarding the interpretation.

As a first observation, using these limited data, is that every RL agent performance lies between the random agent and the greedy agent. None of the RL agents have been able to beat the greedy one on any of the metrics. The RL agents' performances are quite close to eachother. For Deep Q-learning and A2C the average of one is very often within the standard deviation interval of the other. DDPG performed less well than the other two RL agents and ends up with results very close to the random agent's. It suggests it has not been able to learn much from its experiences. It seems quite consistent to note that A2C has the best score for every number of iterations on both metrics. For some unknown reason, performance has not improved dramatically as training progressed. This will be discussed in the next section.

Results have show the ability of the A2C and DQN agent to learn from their environments. Indeed, total reward has departed from the random score and the remaining number of pallets has sometimes reached levels similar to the greedy agent. After the 200kth iteration of the A2C learning, the remaining number of pellets was close to that of the greedy agent but quite far when one considers the total reward. This might suggest the agent has developed a far from optimal routing where it ate most of the pellets but not taking the shortest path. Though it is to note that the greedy agent does not follow the shortest path either.

V. DISCUSSION

As discussed in the previous section. Results of the RL agents are not very impressive with respect to the pre-programmed greedy agent. It should be remembered that this agent has been given the same input as the RL agents. Consequently, these agents could have played at least as good as the greedy one. Here is gathered some of the reasons that might explain this discrepancy and what could be done in order to reduce it.

- **Lack of training** In [2], the authors have trained their agents for several millions of iterations of the learning algorithms. The problem setting was more complex the one discussed here.
- **Network architectures** The approach of using CNN and then a MLP is a very classic one. The actor-critic methods have two networks whose purpose is firstly to process the input image. These CNNs could have been shared. This weight sharing is sometimes suggested but instabilities problem are often pointed out. An encoder-decoder architecture could have been used in place of the CNN. Its learning process could have been disconnected from the learning agent. The encoder could outputs an embedding of the image that could be used by a MLP. In our setting, the CNN is optimized through a very weak signal that is the reward function.
- **Simplicity of the RL algorithms** The three algorithms we have implemented are very far from being state-of-the art. Many RL-related improvements could have been performed as suggested in the Rainbow paper [4]. A prioritized replay buffer as well as dueling networks could have been simple to implement improvements.
- **Complexity of the input** A very similar problem has been tackled by simplifying the input in [2]. They used a feature engineered semantic grid. This grid of size 11×11 gathered the number of pellets and the walls in the view of the agent. Hence, these features do not have to be learned from raw pixel.
- **Size of the replay buffer** The size of the replay buffer (around 50 000) is very limited when compared to common figures in the literature which are several order of magnitudes larger. With such a small number, it is not obvious whether the experiences are sampled *uniformly*. Since the states are 64×64 greyscale images, storing 1 000 000 of these images would take $1\ Byte \times 64 \times 64 \times 10^6 \approx 4GB$.
- **Partial observability** This subject could make the learning agents outperforms the greedy agent. As shown in [3], the partial observability can be tackled by maintaining a representation of what the agent has experienced from its environment as *hidden vectors*. This representation could be queried and updated as new inputs are provided. This would imply the use of LSTM or GRU networks. Hence, some form of planning and speed estimation could be made possible.

VI. APPENDICES

A. The Game Engine

Since the game engine technical requirements were unknown when we began this work, we adopted an implementation that is as agnostic as possible. The game engine is accessible via the network and can host the games remotely. Each simulation is encapsulated in

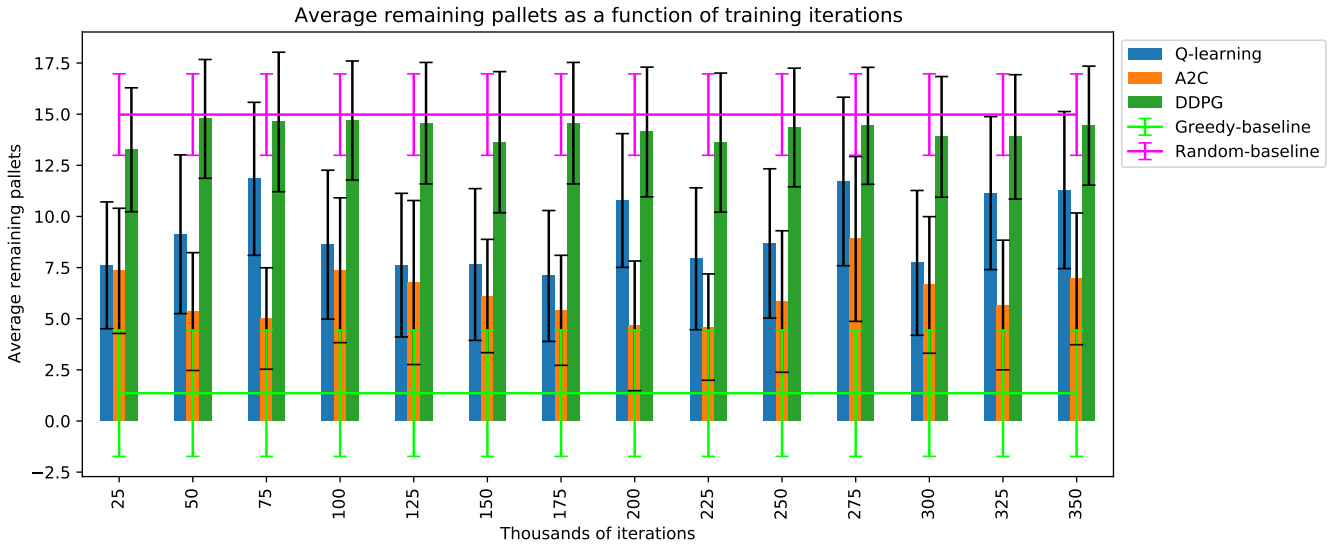

FIG. 3: Average remaining pallets as a function of training iterations for our different algorithms as well as two baselines. These results were obtained over one run of each training process. The lower the better.

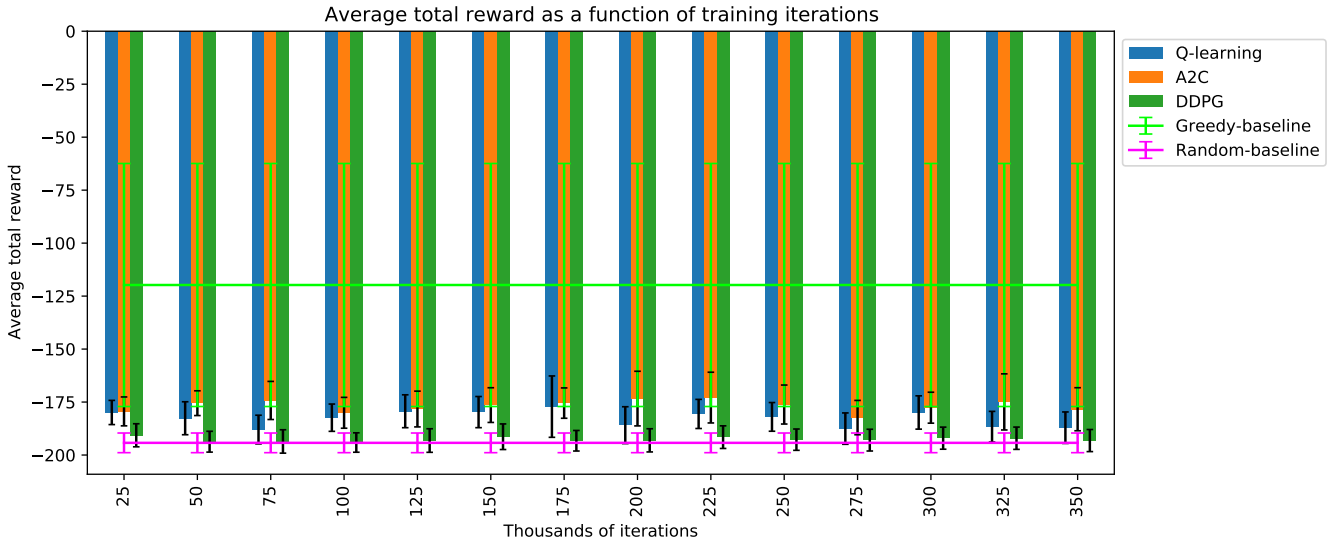

FIG. 4: Average total reward as a function of training iterations for our different algorithms as well as two baselines. These results were obtained over one run of each training process. The higher the better.

an individual thread making the accesses non-blocking. This allows to obtain a higher state-actions throughput with respect to a sequential architecture. Hence, the game engine is actually made of many smaller game servers. The game engine waits for actions to be received before updating a given game state. Updating asynchronously would have made the dynamics depend on the speed of the computer hosting the game server and lead to inconsistent learning.

The following sections describe the implementation of our version of the game. Note that both pellets and Agar are square shaped and no adversarial component is considered in the game.

1. *System state space*

The system state space is the real world description. We can divide this description into subparts.

1. The position of the agent:

$$(x, y) \in \mathbb{R}^2 \tag{13}$$

2. Mass of the agent: m which determines the size it takes. The mass is proportional to the area $= L^2$ thus $m = \alpha L^2$ and then $L = \sqrt{\frac{m}{\alpha}}$.

3. The speed of the agent:

$$(v, u) \in \mathbb{R}^2 \tag{14}$$

4. Position of food:

$$(x_{f_i}, y_{f_i}) \in \mathbb{R}^2 \ \ \forall \ \ in \ \ \{0, ..., n_f\} \tag{15}$$

5. Mass of food:

$$m_{f_i} \in \mathbb{R}^2 \ \ \forall \ \ in \ \ \{0, ..., n_f\} \tag{16}$$

2. Action space

We consider several methods to handle the challenge that exploit either a continuous or discrete action space. The continuous action space takes values between $[-1; 1]$ in both axis direction and the discrete action space is reduced to a set of 25 possible actions: $a_x, a_y \in \{-1, -0.5, 0, 0.5, 1\}$.

3. Dynamics

For the agent, lets define $\mathbf{p}$ as the position vector and $\mathbf{s}$ being the velocity vector.

1.

$$\dot{\mathbf{p}} = \mathbf{s} \tag{17}$$

$$\dot{\mathbf{s}} = \begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} = \mathbf{f}(t, x, y) \tag{18}$$

We also use a simple acceleration/drag model ; the drag implies that the cell speed is limited by its terminal speed as a free fall. The coefficient C_f depends on the mass of the object.

$$\mathbf{f}(t, x, y) = \frac{1}{m(t)} \left(\begin{pmatrix} a_x \\ a_y \end{pmatrix} - C_f \left\| \begin{pmatrix} u \\ v \end{pmatrix} \right\| \begin{pmatrix} u \\ v \end{pmatrix} \right) \tag{19}$$

The mass of the cell decays with time, represented by this differential equation:

$$\frac{dm(t)}{dt} = \alpha_m m(t) \tag{20}$$

Each time the agent overlaps a pellet, it gains the food's mass.

$$\Delta m_{\text{eat}} = m_f \tag{21}$$

4. Terminal states

The goal for the agent is to eat all pellets of the board as far as possible. The number of pellets remains constant over all simulations. Instead of dying when crossing the maps borders, the action space of the agent is adjusted so that it may not leave the board. Each simulation terminates when all pellets are eaten or because the number of steps done by the agent exceeds 2500.

B. Parameters

Parameter	**Value**	**Description**
n_pellets	20	Number of pellets spawned on game board
state_size	64	Size of game board
max_steps	2500	Maximum number of steps before training simulation is terminated

TABLE VII: Parameters used by JAVAgario in all simulations

Pseudocode	**Code**	**Value**
size($\mathcal{D}$)	batch_size	64
/	policy_net_learning_rate	0.0001
/	q_net_learning_rate	0.0001
τ	tau	0.001
γ	discount_factor	0.99
/	hidden_size	512
/	n_frames_state	1
/	state_size	64
/	max_buffer_size	100000

TABLE VIII: Hyperparameters for the A2C networks

Pseudocode	**Code**	**Value**
size($\mathcal{D}$)	batch_size	64
/	learning_rate_critric	0.0005
τ	tau	0.001
/	learning_rate_actor	0.00001
γ	discount_factor	0.99
/	hidden_size	512
/	n_frames_state	1
/	state_size	64
/	max_buffer_size	20000

TABLE IX: Hyperparameters for the DDPG networks

Pseudocode	**Code**	**Value**
size($\mathcal{D}$)	batch_size	64
/	q_net_learning_rate	0.005
τ	tau	0.001
γ	discount_factor	0.99
/	hidden_size	512
/	n_frames_state	1
/	state_size	64
/	max_buffer_size	65000

TABLE X: Hyperparameters for the DDQN networks

[1] Ansó, N., Wiehe, A., Drugan, M., and Wiering, M. (2019). Deep reinforcement learning for pellet eating in agar.io.

[2] Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D., and Blundell, C. (2020). Agent57: Outperforming the atari human benchmark.

[3] Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps.

[4] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298.

[5] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

[6] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N. M. O., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971.

[7] Mnih, V. et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[8] Poupart, P. (2018). Lecture notes in reinforcement learning.

[9] Sutton, R., Mcallester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Adv. Neural Inf. Process. Syst*, 12.

[10] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning.

[11] Watkins, C.J., D. P. (1992). Q-learning.

[12] Yuhuai Wu, Elman Mansimov, S. L. A. R. and Schulman, J. (2017). Openai baselines: Acktr & a2c. *OpenAI Blog*.