Deep Q-Network

Prof. Capobianco, Reinforcement Learning

Sveva Pepe - 1743997

1 Introduction

The project consists in testing the algorithm on an environment that is assigned to us and analyze the data that it is working in that environment.
In this project, we use Reinforcement Learning, which is a technique that allows the agent to learn a certain behavior depending on the environment submit to it with methods based on the concept of reward.
In my case, I was given the DQN algorithm to test on 'Assault' environment of OpenAi Gym. [1]
The algorithm was provided to us using the "stable baselines" library.

2 Markov Decision Process

To explain what the DQN algorithm is about, we need to introduce the concept of Markov decision process (MDP). A Markov Decision Process is used to model the interaction between the agent and the controlled environment. The components of a MDP include:

$$< S, A, T, R >$$

- S is the state space, s ∈ S
- A is the set of actions, a ∈ A;
- $T = P(s'|s, a)$ = probability that action a in state s at time t will lead to state s' at time t+1 → P($s_{t+1} = s'|s_t = s, a_t = a$)
- R is the reward function. $R(s, a, s')$ represents the reward when applying the action a in the state s which leads to the state s'.
- $\gamma \in [0, 1]$ is the discount factor, in particular, it informs the agent of how much it should care about rewards now to rewards in the future. If ($\gamma = 0$), that means the agent is short-sighted, in other words, it only cares about the first reward. If ($\gamma = 1$), that means the agent is far-sighted, i.e. it cares about all future rewards. What we care about is the total rewards that we're going to get.

At each step, the process is in some states and the decision-maker can choose any action available in the state s. The process responds to the next time randomly moving into a new state and giving the decision-maker a reward corresponding $R(s, a, s')$. The probability that the process moves on to its new one state s' is influenced by the chosen action. In particular, it is given by the state transition function $P(s'|a, s)$. Therefore, the next state s depends on the current state and the action 'a' of the decision-maker.
The agent's behavior depends on the policy he learns. A policy π is a distribution over actions given states:

$$\pi : S \to A$$
$$\pi(a|s) = P(a_t = a|s_t = s)$$

The MDP policies depend on the current state, not the history, which means that whenever the agent arrives in a certain state, he will take the action that decided before, for all the different temporal phases.
Then there is another factor that we need to consider: value function.
A value function, $U^{\pi}(s)$, tells us how good it is to be in state s if we decide to follow policy π, that is, expectations when we sample all actions according to policy π,

$$U^{\pi}(s) = E[R] = E[\sum_{t=0}^{\infty} \gamma^t r(s_t)] = E[R_t]$$

Expected utility can be used to compare policies, we can say that when we find a policy with the highest expected utility (starting in s) this means that is an optimal policy.

[1] Open AI is a toolkit regarding reinforcement learning. Gym's library contains several environments, and each environment communicates with the algorithm used by the user through observations, actions and rewards.

3 Q-Learning

Q-learning is a model-free and off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state.
Model-free means that it is an algorithm that makes several attempts and checks whether it has produced the desired effect. If so, the attempt constitutes a solution to the problem, otherwise continue with a different attempt.
In addition, this algorithm is off-policy, it learns the actions regardless of the policy used. Q-learning is based on Bellman equation:

$$Q^*(s,a) = E_{s'}[r + \gamma \max_{a'} Q^*(s',a')|s,a]$$

where E is the expected value of next state s', r is the immediate reward that we observe after the execution of action a and γ is discount factor.

The agent can choose the action to be performed in two ways: exploration or exploitation.
Exploitation consists in going to select an action in order to maximize $Q^*(s,a)$ instead exploration means that an action is chosen randomly, sometimes selecting a random action allows the agent to explore new states that might otherwise be unselected during the exploitation process.
Usually, we try to balance both methods using an ϵ probability.
For example, in the ϵ-greedy strategy we go to select a random action with ϵ probability, and to select the best action with $1 - \epsilon$ probability. ϵ is decremented with each iteration. So, in this case, an exploration is done first and then an exploitation.

4 Deep Q-Network

Deep Q-Network consists of Deep Q-learning with the use of a neural network to estimate the value of the Q-value function. We have that the input is the state and the output is the Q-value of all the possible actions generated.

4.1 Deep Q-Learning

Deep Q-learning is an algorithm base on Q-learning but with the experience replay technique in which we store the agent's experiences at each time stage $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $D = e_1, ..., e_N$, grouped in many episodes in replay memory. During the internal cycle of the algorithm, we apply Q-learning updates to samples of experience randomly extracted from the stored samples.
After performing experience replay, the agent selects and performs an action according to ϵ-greedy strategy. You can find the explanation of the strategy in the previous section. 3
Since the use of arbitrary length histories as input for a neural network can be difficult, our Q function instead works on a fixed-length representation of histories produced by a function ϕ.

This approach has several advantages as each step of the experience replay is used in many updates of weights of the neural network and this favours data efficiency.
Furthermore, in this way, by randomizing the samples we break the various links between the samples (a problem that occurred when we were going to consider the consecutive samples directly) and we reduce the variance of the updates.
Finally, being an off-policy approach we don't end up stuck in a local minimum since we won't have that the current parameters determine the next sample of data on which the parameters are trained, as in the case of the on-policy.

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 loss function
  end for
end for

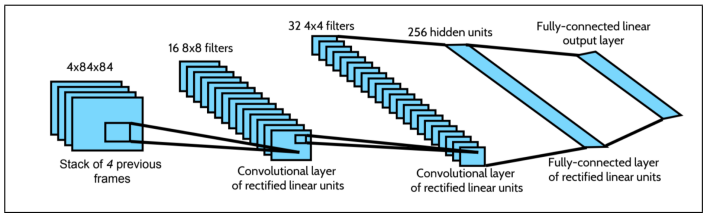
where loss function is:

$$L_i(\theta_i) = E_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta_i))^2]$$

where $y_i = E_{s'}[r + \gamma \max_{a'} Q^*(s', a'; \theta_{i-1})|s, a]$ is the target of iteration i and in particular, targets depend of network weights θ. $\rho(s, a)$ is a probability distribution over sequences s and actions a.

The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. For more information on Q-learning, see the section 3.

4.2 DQN with Atari Games



The input to the neural network consists is an $84x84x4$ image produced by function ϕ, seen in Deep-Q learning 4.1.
The first hidden layer convolves 16 of $8x8$ filters with stride 4 with the input image and as activation function, a ReLu function. [2]
The second hidden layer convolves 32 of $4x4$ filters with stride 2 and as activation function, a ReLu function.
The final hidden layer is fully-connected and consists of 256 units with ReLu function.
The output layer is a fully-connected linear layer with a single output for each valid action. In the final layer, you are approximating a real state action value for each action. You output to a linear layer as your Q value estimate can generally take on any real value. And then you add a mean squares error loss with the linear layer output. This implies having a linear activation function [3] in the output layer.

5 Environment

Assault is an environment in which the agent must fire bullets at an enemy mother ship which deploys smaller ships to attack the agent.
In the environment "Assault-v0" the observation consists in the image, the k frames received or it can be the ram; instead in "Assault-ram-v0" is executed the observation is the ram.
The agent acts following the commands of ATARI 2600: NOOP, FIRE, RIGHT, LEFT, RIGHTFIRE, LEFTFIRE, UPFIRE.
The reward starts from zero with each episode of the game and increases every time a spacecraft is hit, so the reward depends on the action done by the agent. When the agent dies, it returns to the initial state in which the reward is set to zero.
Our aim is to train the game using DQN algorithm, in particular, to modify hyperparameter in order to obtain quite significant results.

5.1 Hyperparameters

- **policy**: `CNN, LnCNN, Mlp and LnMlp`
  Policies are used to select the action to be performed, in particular, CNN [4], it is usually used when observations are given by the images of the game and LnCNN is a CNN with layer normalization.
  Instead Mlp[5] takes the concatenation of all observations from the environment as input for predicting actions. LnMlp is a Mlp with layer normalization.
- **gamma**: discount factor, see section 2.
- **learning rate**: the learning rate is the factor that controls how quickly or slowly the neural network learns that particular problem.
  If your learning rate is set too low, the training will proceed very slowly as you are making various updates to your network weights. However, if the set learning rate is too high, it can cause undesirable divergent behaviour. In general, you want to find a learning rate that is low enough that the network converges to something useful, but high enough that you don't have to spend years training it.
- **buffer_size**: size of the replay buffer.

[2] ReLU stands for rectified linear unit, and is a type of activation function. Mathematically, it is defined as $y = \max(0, x)$.
[3] Linear function is mathematically deifined as $f(x) = ax$
[4] Convolutional Neural Network
[5] Multi-layer Perceptron

- **exploration_initial_eps**: it identifies initial value of random action probability. By decreasing this probability, I decrease the probability of having to select a random action (exploration phase).
- **exploration_final_eps**: it identifies the final value ofrandom action probability.
  In particular, this affects my ϵ-greedy strategy. Because by increasing this probability, I decrease the probability of having to select the best action (exploitation phase).
- **train_freq**: update the model every "train_freq" steps. For example if you choose train_freq=2, means that you model update its parameters every two times.
- **learning_starts**: how many steps of the model to collect transitions for before learning starts.
- **target_network_update_freq**: update the target network every "target_network_update_freq" steps. Basically "y" is updated, y is seen in the section 4.1.
- **prioritized_replay**: if this variable is sett True prioritized replay buffer will be used. Prioritized replay consists of prioritized sampling that will weigh the samples so that "important" ones are drawn more frequently for training.
  Proportional priority is used, which is based on stochastic prioritization. the probability of sampling transition i as $P_i(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where $p_i > 0$ is the priority of transition i and exponent α determines how much prioritization is used. It is called proportional prioritization where $p_i = |\delta_i| + \epsilon$, where ϵ, is a small positive constant that prevents states from revisiting once their error is zero.
- **prioritized_replay_alpha**: alpha parameter for prioritized replay buffer. It determines how much prioritization is used, with alpha=0 corresponding to the uniform case.

6 Assault-v0

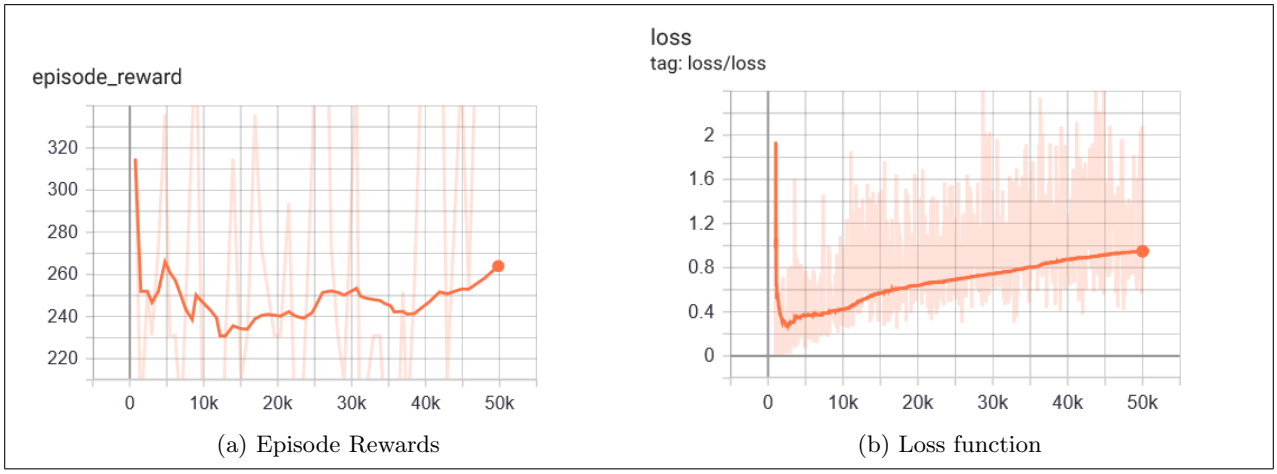
I will show 5 models in which improvements have been made in each of them to make sure that the last one can be the best after several attempts.

6.1 Train

The train is made using the learn function in which I get the agent a number of episodes, in which in each episode it is composed of "total_timesteps", which will depend on the duration of the game and this train helps the agent to build his own policy and it will allow it to act better within the environment.
In this environment for all the different models used, the number of "total_timesteps" is equal to 50000.

Model 1

First model is a basic model, the model I started from. The policy used is CNN because I am working with images. The learning rate is decreased to make the neural network learn slower but more efficiently, it is equal to $2.5e^{-4}$. Instead, exploration_final_eps has been changed, a parameter that influences exploitation. I increased it to see the result, which must have been negative.



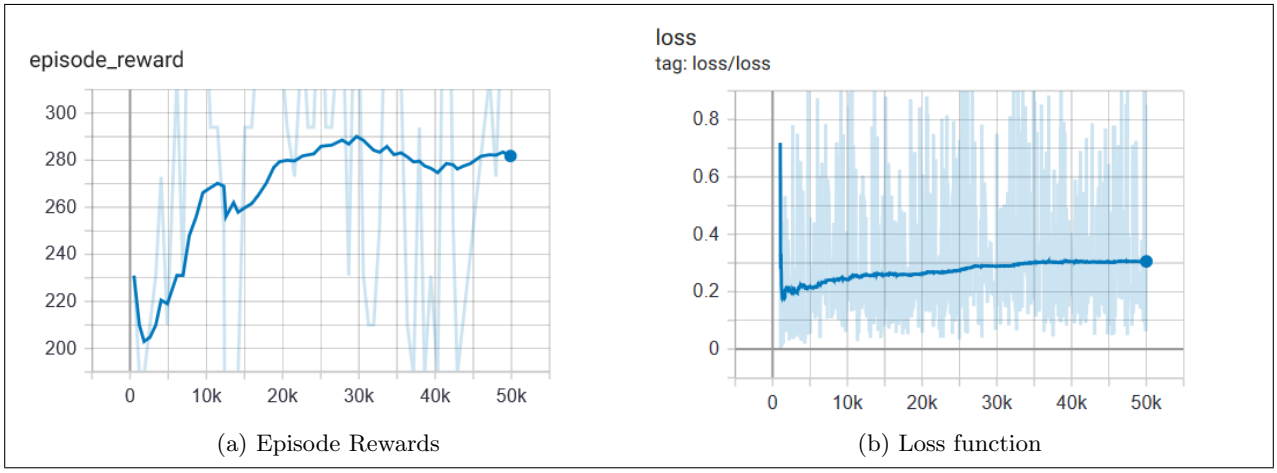
(a) Episode Rewards (b) Loss function

These are two "smoothed" graphs showing the trends of the reward values and the loss function during the training phase. In particular, we can observe how the trend of the rewards is not entirely stable, even if towards the end it tends to grow. As we can see, as a first model the result is not bad. But the problem is in the loss.
The DQN is trained by minimizing the loss function, but with this configuration, the loss tends to increase during the

timesteps, this means that the model used is not good, and more particularly the exploration_final_eps must not be increased because in this way I decrease the probability of selecting the best action in the ϵ-greedy strategy.

Model 2

In the second model only gamma and learning rate hyperparameters are considered. I preferred to leave the default value of epsilon_final_eps (0.02) to see the behaviour of the model, to then later understand if it was necessary to modify it or not. Gamma is taken 0.89 instead of the default of 0.99. Learning rate is $2.5e^{-4}$, equal to the previous model.

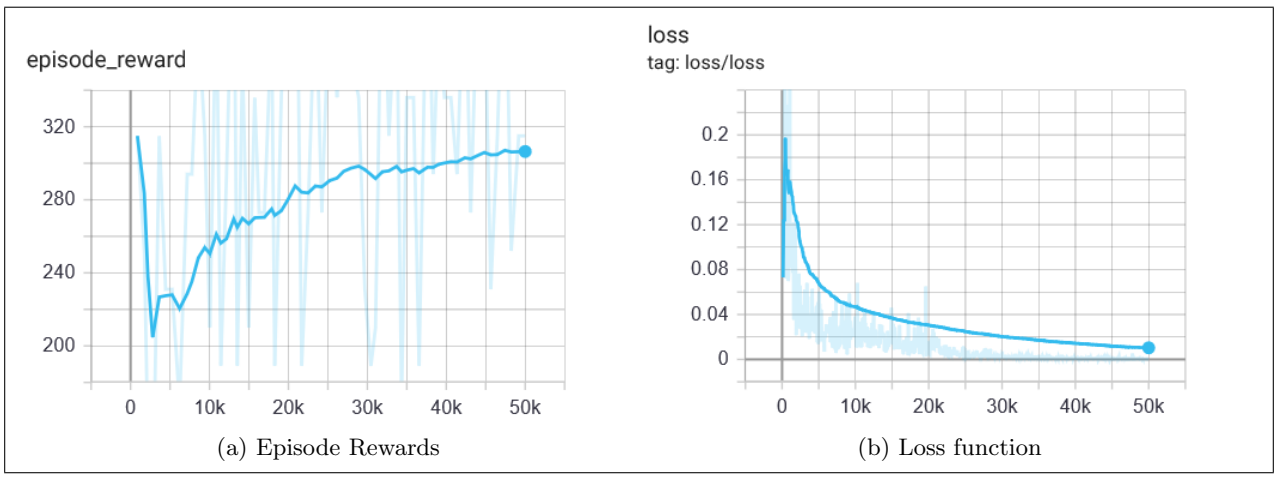


(a) Episode Rewards (b) Loss function

As we can see from the graphs, the reward trend is much higher than that of the previous model and also the loss function is better, even if it is not yet close to zero.
We must further improve the model so as not to have large swings in the reward of each episode, and also that the value of the loss function is as small as possible.

Model 3

In the third model,I changed the policy, from CNN to Ln CNN, in order to have a layer normalization that allows me to have a more stable trend. I established the default learning rate value ($5e^{-4}$) but decreasing the learning_start from 1000 (the default one) to 100 to make the agent learn faster. Gamma is set to 0.89 because in the previous model we had seen that there were too many different oscillations, so trying to decrease the value, I tried to see if there were any changes of any kind.
Furthermore, the exploration_final_eps decreased, equal to 0.01, to favuor the exploitation phase. The fundamental point made in this model was to try to change the approach by working on the replay buffer part.
First of all, I increased the buffer_size, size of the replay buffer, to store as much data as possible learned by the agent. After that, we modify the parameter train_freq and target_network_update_freq to increase the modification steps of the neural network, to avoid making continuous changes. They will be 4 and 1000 respectively.
Consider choosing samples in a prioritized and uneven way to choose the best from the replay buffer in order to try to get better results. prioritized_replay parameter set to True. Finally, the prioritized_replay_alpha parameter is increased, setting it equal to 0.6, favouring prioritization.

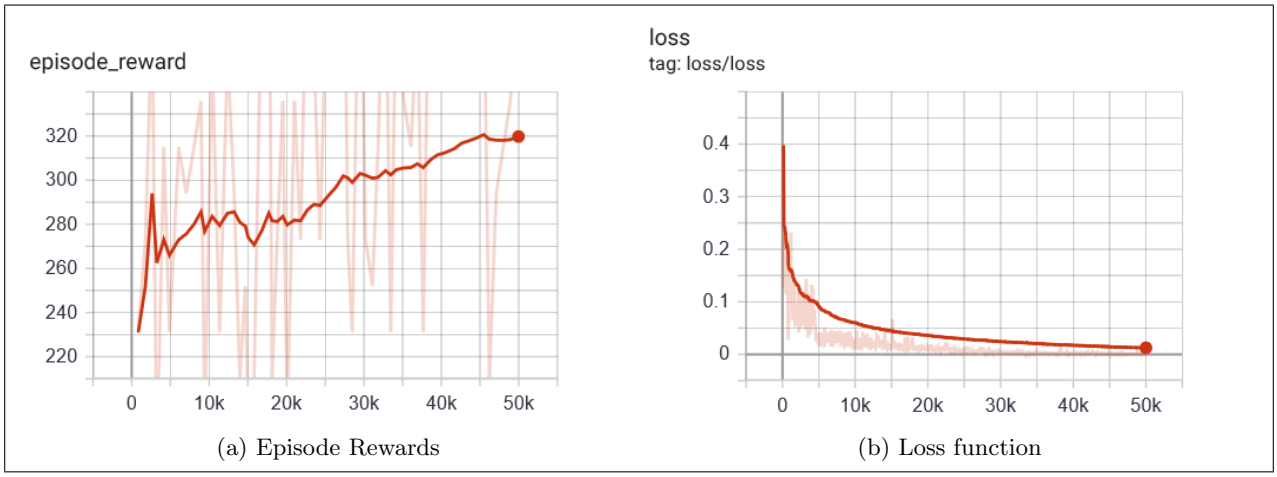


(a) Episode Rewards (b) Loss function

As shown above, this model turns out to be clearly superior to the two previous ones seen before, this is because if we observe the loss function, as the timesteps used for the train increase, it gets closer and closer to zero. In addition, rewards also begin to have a more stable trend than previous versions.

Model 4

In the fourth model, small changes have been made compared to the third, because all in all the model is quite good. I changed the learning rate, setting it back to $2.5e^{-4}$ and getting some good improvements. Gamma has been increased because since the replay buffer is bigger now, it is set to 0.99.

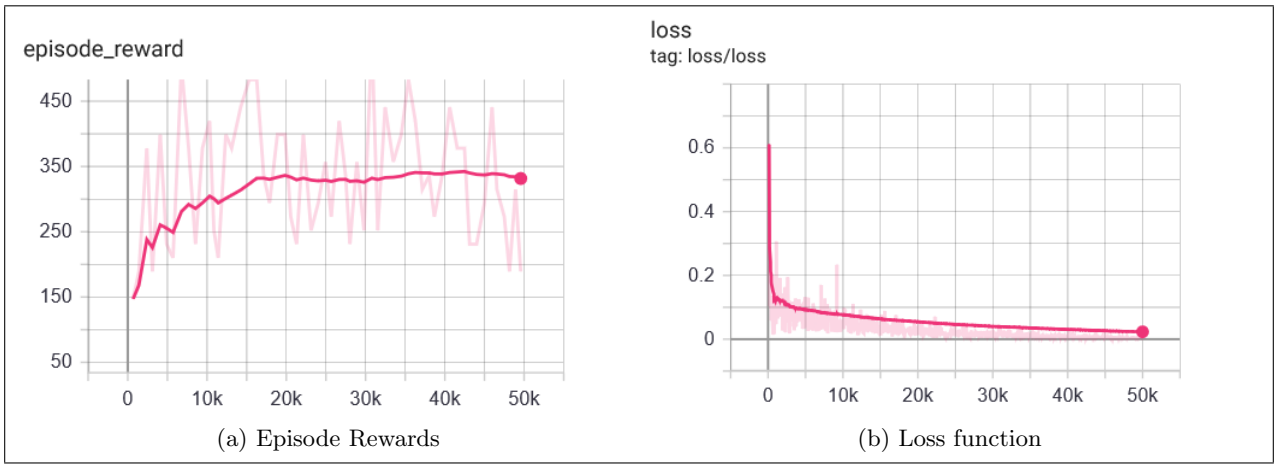


(a) Episode Rewards (b) Loss function

Rewards start to rise right from the start,when the number of train cycles increases, the trend of rewards goes up and the loss function is close to zero.
The behaviour does not undergo great variations during the whole train, even if the model is not yet the definitive one.

Model 5

In the latest model, no major changes are made. The gamma value is changed, because perhaps too high, it is reduced to 0.9 and the value of parameter prioritized_replay_alpha is reduced to 0.4, which probably produced the different initial peaks of the various rewards of each episode during training.



(a) Episode Rewards (b) Loss function

The behaviour we get is pretty good because the reward trend is starting to be much more stable than previous versions. The loss function also takes values close to zero with increasing train cycles. This means that the following model is an excellent result to help the agent learn to obtain rather high rewards, minimizing the loss function as much as possible.

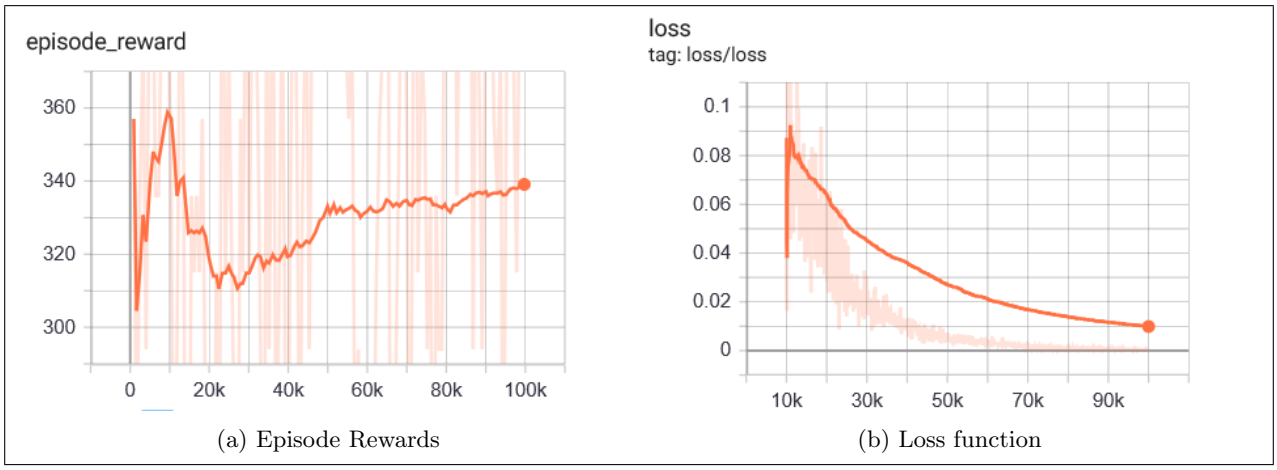
7 Assault-ram-v0

7.1 Train

Also in this case, as in the previous one, the learn function is used to train the 5 models that we will analyze.
The number of cycles of training will be 100000.

Model 1

In this first model, an Mlp policy is considered because we are working on the ram and no longer on the images. The learning rate is decreased from $5e^{-4}$ to $2.5e^{-4}$ to make the neural network learn more slowly trying to get good performance. Also, gamma is decreased from 0.99 to 0.95 and the exploration_final_eps is set equal to 0.005 to favour the exploitation instead of the exploration trying to find the best action having the probability $1 - \epsilon$.

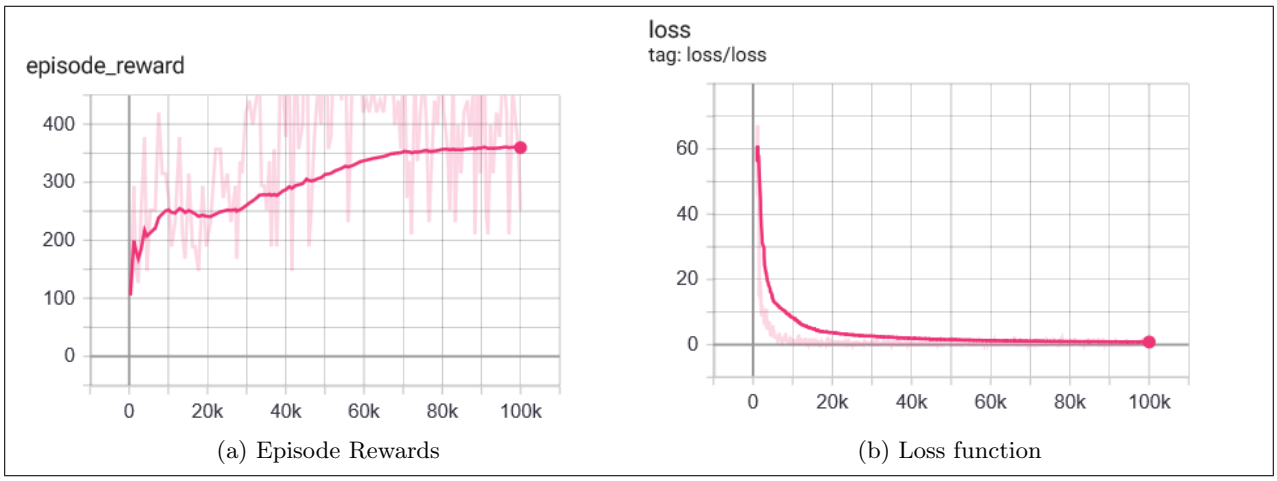


(a) Episode Rewards (b) Loss function

As we can see, the model is not that great, in fact, the rewards are staggered; even if after about 40000 train cycles the rewards begin to rise, the trend of reward of each episode increases as the number of "total_timesteps" increases.
The loss function must be as small as possible for the model to be good. In this case the general trend is not bad, indeed, but it is not the best since it initially has several peaks before converging towards zero.

Model 2

To improve the performance of the previous model, the only changes that have been made are the following: exploration_final_eps is set to 0.01 because thanks to it the rewards of the proceeding model underwent several variations during the training phase.
Furthermore, the size of the replay buffer is increased to facilitate the storage of more information learned during the various cycles, therefore buffer_size = 100000. The rest remains unchanged from the previous model.

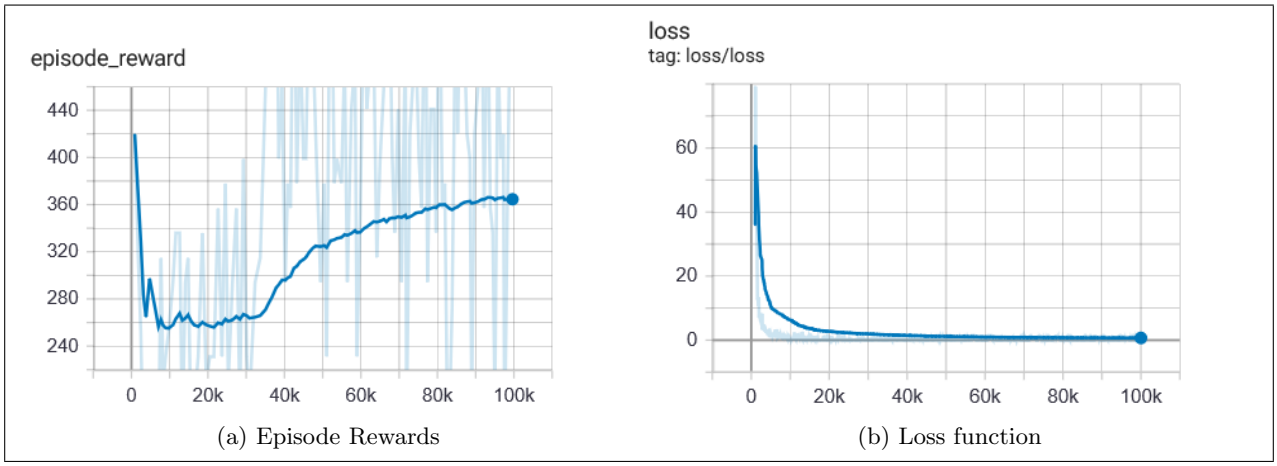


(a) Episode Rewards (b) Loss function

We can see from the figures how there has been an improvement, especially as regards the trend of the rewards which is more stable than the previous one. The loss function is also close to zero.

Model 3

In the third model, I reduced the hyperparameters to be taken into consideration to see if the trend was the best or worst of the previous one. I have not taken into consideration the exploration_fraction_eps and the buffer_size, this implies that they will assume their default values: 0.02 and 1000.
The learning rate is equal to the previous versions and gamma has been significantly decreased down to the value of 0.9.

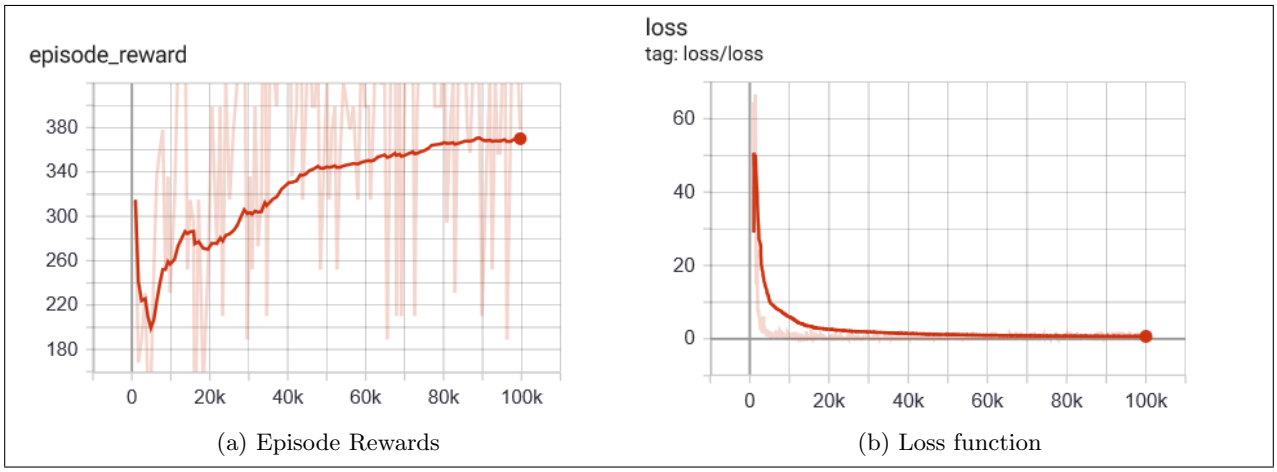


(a) Episode Rewards (b) Loss function

The trend is not very uniform at the beginning, indeed, from this point of view, the previous model is better, due to the low value of the exploration_final_eps. But after all, after the 40000 cycles, the model significantly increases the value of the rewards until it reaches the rewards of the previous model after 100000 train cycles.
The loss function is almost identical to the previous one, in fact, the important thing is that it is as close as possible to zero.

Model 4

Given that the previous model does not consider it very good, I decided to take the hyperparameters of the first two models and modify them further. The learning rate is taken as $5e^{-4}$ to make the network learn faster. Gamma instead I decided to leave it equal to 0.9 because in model 3 they are still quite good rewards.
The exploration_final_eps I consider it equal to 0.01 because in the second model the trend of the rewards was quite uniform. I increase the buffer_size to 100000 as in the second model. And finally, I consider the exploration_initial_eps takes it equal to 0.85, which will be the probability of choosing a random action (exploration).



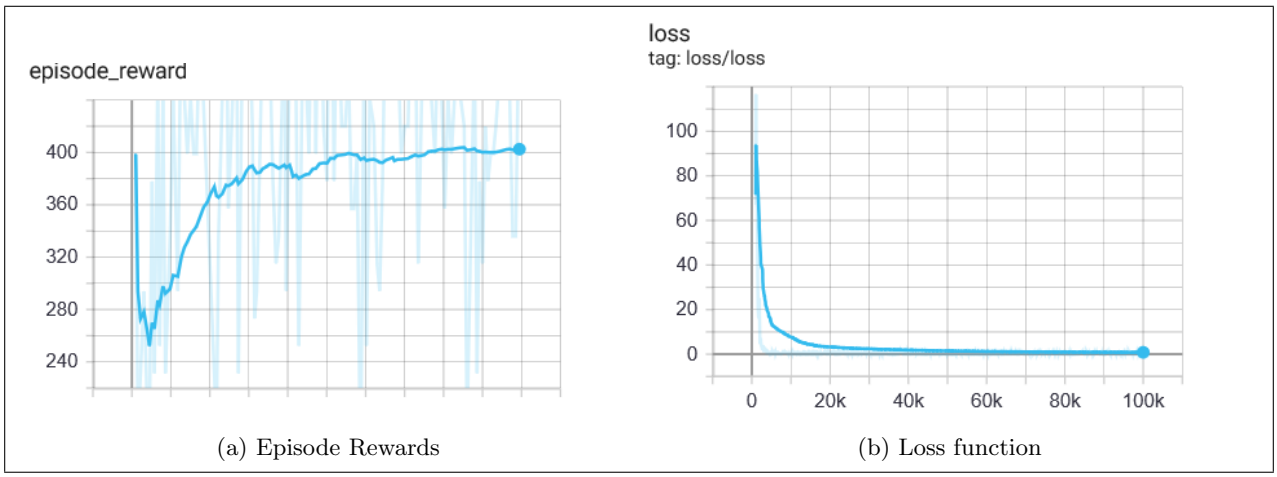
(a) Episode Rewards (b) Loss function

The rewards are higher than the previous models and moreover, except for the first 20000 cycles, the trend is growing up to a reward value of 380. The loss function is excellent, close to zero. This means that we are going in the right direction.

Model 5

In the last model I changed the policy, I preferred the normalized one, LnMlp, because normalization tends to stabilize the trend and therefore to have pretty good rewards right away. I lowered the learning rate to $1e^{-4}$ because

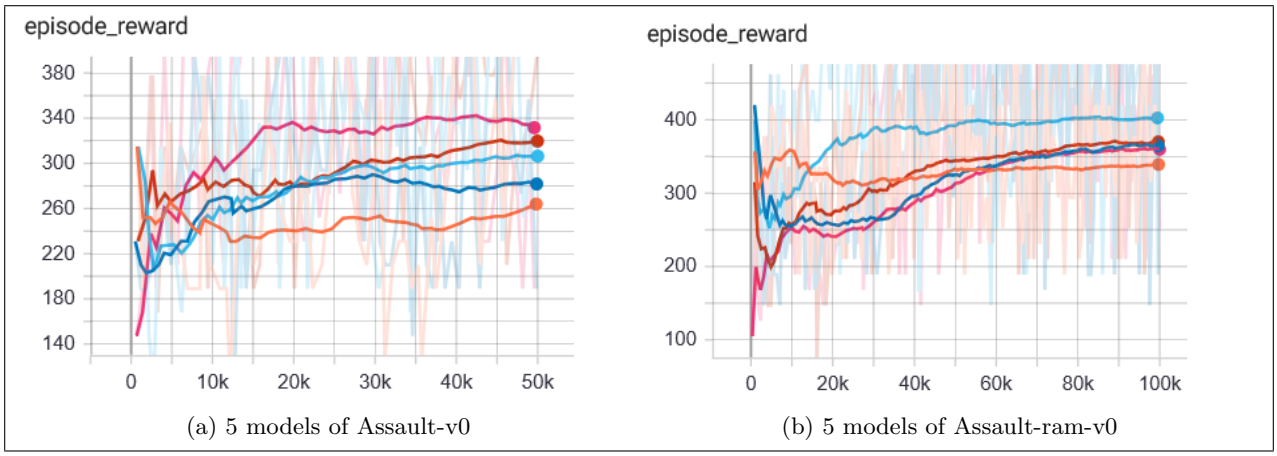
after several experiments I saw that by making the neural network learn more slowly I got better results of each episode as the timesteps increased. Gamma is taken 0.95 to care for many future rewards. The exploration_final_eps is set to 0.01 like the previous model. Finally, in the latter model, the prioritized replay buffer is considered, with prioritized_replay_alpha equal to 0.3 because after several experiments having a low alpha value improve the results.



(a) Episode Rewards (b) Loss function

We can see that compared to the previous models, the initial rewards are already good and eventually stabilize, stability is given by the normalization of the policy. The loss function is already very close to zero from the first moment.

8 Conclusions



(a) 5 models of Assault-v0 (b) 5 models of Assault-ram-v0

As can be seen from the two graphs, the performances obtained for both environments are not very high, but working on this RL project made me very curious about this stuff. Furthermore, it allowed me to deepen my knowledge, interact pratically on this type of topic.